Ganesh Gopalakrishnan
Shaz Qadeer (Eds.)

LNCS 6806

Computer Aided Verification

23rd International Conference, CAV 2011
Snowbird, UT, USA, July 2011
Proceedings

Springer

Lecture Notes in Computer Science 6806

Ganesh Gopalakrishnan
Shaz Qadeer (Eds.)

Computer Aided Verification

23rd International Conference, CAV 2011
Snowbird, UT, USA, July 14-20, 2011
Proceedings

Springer

Volume Editors

Ganesh Gopalakrishnan
University of Utah
School of Computing
50 South Central Campus Dr.
Salt Lake City, UT 84112-9205, USA
E-mail: ganesh@cs.utah.edu

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
E-mail: qadeer@microsoft.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-22109-5 e-ISBN 978-3-642-22110-1
DOI 10.1007/978-3-642-22110-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011930052

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

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

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

The International Conference on Computer-Aided Verification (CAV) is dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. Its scope ranges from theoretical results to concrete applications, with an emphasis on practical verification tools and the underlying algorithms and techniques. This volume contains the proceedings of the 23rd edition of this conference held in Snowbird, Utah, USA, during July 14–20, 2011. The conference included two workshop days, a tutorial day, and four days for the main program.

At CAV 2009, Bob Kurshan approached us with the idea of holding CAV 2011 in Salt Lake City. Encouraged by the enthusiastic support from late Amir Pnueli, we had little hesitation in agreeing to Bob's proposal. While the initial proposal was to organize the conference on the campus of the University of Utah, we eventually decided to hold it at the Snowbird resort near Salt Lake City. Our decision was motivated by the dual desire to showcase the abundant natural beauty of Utah and to provide a collegial atmosphere similar to a Dagstuhl workshop.

We are happy to report that CAV is thriving, as evidenced by the large number of submissions. We received 161 submissions and selected 35 regular and 20 tool papers. We appreciate the diligence of our Program Committee and our external reviewers due to which all (except two) papers received at least four reviews. A big thank you to all our reviewers!

The conference was preceded by the eight affiliated workshops:

– The 4th International Workshop on Numerical Software Verification (NSV 2011), Thursday, 7/14
– 10th International Workshop on Parallel and Distributed Methods in Verifications (PDMC 2011), Thursday, 7/14
– The 4th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2 2011), 7/14-7/15
– Frontiers in Analog Circuit Synthesis and Verification (FAC 2011), 7/14-7/15
– International Workshop on Satisfiability Modulo Theories, including SMT-COMP (SMT 2011), 7/14-7/15
– 18th International SPIN Workshop on Model Checking of Software (SPIN 2011), 7/14-7/15
– Formal Methods for Robotics and Automation (FM-R 2011), 7/15
– Practical Synthesis for Concurrent Systems (PSY 2011), 7/15

In addition to the presentations for the accepted papers, the conference also featured four invited talks and four invited tutorials.

– Invited talks:
  - *Andy Chou* (Coverity Inc.): "Static Analysis Tools in Industry: Notes from the Front Line"

  - *Vigyan Singhal* and Prashant Aggarwal (Oski Technology): “Using Coverage to Deploy Formal Verification in a Simulation World”
  - *Vikram Adve* (University of Illinois at Urbana-Champaign): “Parallel Programming Should Be and Can Be Deterministic-by-default”
  - *Rolf Ernst* (TU Braunschweig): “Formal Performance Analysis in Automotive Systems Design: A Rocky Ride to New Grounds”
- Invited tutorials:
  - *Shuvendu Lahiri* (Microsoft Research): “SMT-Based Modular Analysis of Sequential Systems Code”
  - *Vijay Ganesh* (Massachussetts Institute of Technology): “HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection”
  - *Ranjit Jhala* (University of California at San Diego): “Using Types for Software Verification”
  - *Andre Platzer* (Carnegie Mellon University): “Logic and Compositional Verification of Hybrid Systems”

A big thank you to all our invited speakers!

We thank the members of the CAV Steering Committee —Michael Gordon, Orna Grumberg, Bob Kurshan, and Ken McMillan— for their timely advice on various organizational matters. Neha Rungta, our Workshop Chair, smoothly handled the organization of the workshops. Eric Mercer, our Local Arrangements Chair, set up the registration portal at Brigham Young University. Sandip Ray, our Publicity Chair, helped publicize CAV 2011. We thank Aarti Gupta, past CAV Chair, for her help and advice in running the conference and maintaining its budget.

We thank Geof Sawaya for maintaining the CAV 2011 website. We are grateful to Wendy Adamson for arranging the beautiful Cliff Lodge facility at an affordable price and really making the budget work in our favor. We thank Alfred Hofmann of Springer for publishing the paper and USB proceedings for CAV 2011. We thank Andrei Voronkov and his team for offering us EasyChair which has proven invaluable at every juncture in conducting the work of CAV. We thank the office staff of the School of Computing, University of Utah, especially Karen Feinauer and Chris Coleman, for allowing us to use the school resources for managing CAV activities.

We are especially grateful to our corporate sponsors —Microsoft Research, Coverity, Google, NEC Research, Jasper, IBM, Intel, Fujitsu, and Nvidia— for their donations. We are also grateful to Judith Bishop and Wolfram Schulte of Microsoft Research for their substantial financial backing of CAV. We also thank Lenore Zuck, Nina Amla, and Sol Greenspan who helped with obtaining an NSF travel award.

CAV 2012 will be held in Berkeley, California.

April 2011

Ganesh Gopalakrishnan
Shaz Qadeer

Organization

Program Committee

Azadeh Farzan	University of Toronto, Canada
Jasmin Fisher	Microsoft Research, Cambridge, UK
Cormac Flanagan	University of California at Santa Cruz, USA
Steven German	IBM Yorktown Heights, NY
Dimitra Giannakopoulou	RIACS/NASA Ames, USA
Ganesh Gopalakrishnan	University of Utah, USA
Susanne Graf	Université Joseph Fourier, CNRS, VERIMAG, France
Keijo Heljanko	Helsinki University of Technology, Finland
William Hung	Synopsys Inc., USA
Franjo Ivancic	NEC Labs America, USA
Joost-Pieter Katoen	RWTH Aachen, Germany
Stefan Kowalewski	RWTH Aachen, Germany
Daniel Kroening	Oxford University, UK
Orna Kupferman	Hebrew University, Israel
Robert P. Kurshan	Cadence Design Systems, USA
Akash Lal	Microsoft Research, Bangalore, India
Kim G. Larsen	Aalborg University, Denmark
Ken Mcmillan	Microsoft Research, Redmond, USA
Madan Musuvathi	Microsoft Research, Redmond, USA
Michael Norrish	NICTA, Australia
Madhusudan Parthasarathy	University of Illinois at Urbana-Champaign, USA
Shaz Qadeer	Microsoft Research, Redmond, USA
John Regehr	University of Utah, USA
Andrey Rybalchenko	TU Munich, Germany
Sriram Sankaranarayanan	University of Colorado at Boulder, USA
Roberto Sebastiani	University of Trento, Italy
Sanjit A. Seshia	University of California at Berkeley, USA
Ofer Strichman	Technion, Israel
Murali Talupur	Intel, Santa Clara, USA
Serdar Tasiran	Koc University, Turkey
Ashish Tiwari	SRI International, Menlo Park, USA
Tayssir Touili	LIAFA, CNRS, France and Université Paris Diderot
Viktor Vafeiadis	MPI-SWS, Germany
Bow-Yaw Wang	Academia Sinica, Taiwan

Additional Reviewers

A

Abraham, Erika
Ait Mohamed, Otmane
Alglave, Jade
Andres, Miguel E.
Asarin, Eugene
Atig, Faouzi

B

Baier, Christel
Balakrishnan, Gogul
Barakat, Kamal
Batty, Mark
Bauer, Andreas
Bell, Christian
Bensalem, Saddek
Berdine, Josh
Bhattacharya, Ritwik
Biallas, Sebastian
Biere, Armin
Bingham, Jesse
Boker, Udi
Bonakdarpour, Borzoo
Bouajjani, Ahmed
Bozzano, Marco
Brady, Bryan
Brauer, Jörg
Brihaye, Thomas
Bruttomesso, Roberto
Buchholz, Peter
Burckhardt, Sebastian

C

Cabodi, Gianpiero
Cardelli, Luca
Case, Michael
Chan, Yang
Chaudhuri, Swarat
Chechik, Marsha
Chen, Taolue
Chen, Xiaofang
Chen, Yang
Chen, Yu-Fang
Cook, Byron

D

D'Argenio, Pedro R.
D'Silva, Vijay
Dang, Thao
David, Alexandre
De Moura, Leonardo
De Paula, Flavio M.
De Rougemont, Michel
Distefano, Dino
Donaldson, Alastair
Donzé, Alexandre
Doyen, Laurent
Dragoi, Cezara
Duan, Jianjun
Dubrovin, Jori
Durairaj, Vijay
Dutertre, Bruno

E

Een, Niklas
Elenbogen, Dima
Elmas, Tayfun
Emmer, Moshe
Emmi, Michael
Enea, Constantin

F

Fahrenberg, Uli
Ferrante, Alessandro
Forejt, Vojtech
Franke, Dominik
Freund, Stephen

G

Gan, Xiang
Ganai, Malay
Ganesh, Vijay
Garg, Pranav
Garnier, Florent

Ghorbal, Khalil
Gimbert, Hugo
Girard, Antoine
Godefroid, Patrice
Gotsman, Alexey
Griggio, Alberto
Groce, Alex
Grundy, Jim
Gueckel, Dominique
Gupta, Ashutosh
Gurfinkel, Arie

H

Haemmerlé, Rémy
Haensch, Paul
Haller, Leopold
Hartmanns, Arnd
He, Fei
He, Nannan
Heinen, Jonathan
Heizmann, Matthias
Holcomb, Daniel
Huang, Chung-Yang Ric
Humphrey, Alan

J

Jalbert, Nicholas
Janhunen, Tomi
Jansen, Christina
Janssen, Geert
Jha, Susmit
Jhala, Ranjit
Jiang, Jie-Hong Roland
Jin, Hoonsang
Jin, Naiyong
Jobstmann, Barbara
Jones, Robert
Junttila, Tommi
Jürjens, Jan

K

Kahlon, Vineet
Kaiser, Alexander
Kamin, Volker
Kim, Hyondeuk
Kincaid, Zachary
Kindermann, Roland
King, Andy
Kishinevsky, Michael
Kodakara, Sreekumar
Kotker, Jonathan
Krepska, Elzbieta
Krstic, Sava
Kwiatkowska, Marta
Kähkönen, Kari
Köpf, Boris

L

La Torre, Salvatore
Lahiri, Shuvendu
Launiainen, Tuomas
Leroux, Jerome
Levhari, Yossi
Lewis, Matt
Li, Guodong
Li, Jian-Qi
Li, Wenchao
Logozzo, Francesco
Lvov, Alexey

M

Mador-Haim, Sela
Maeda, Naoto
Majumdar, Rupak
Maler, Oded
Malkis, Alexander
Maoz, Shahar
Mardare, Radu
Mateescu, Maria
Mayr, Richard
Mereacre, Alexandru
Merschen, Daniel
Might, Matthew
Miner, Paul
Mishchenko, Alan
Mitra, Sayan
Mogavero, Fabio
Mover, Sergio
Murano, Aniello

N

Nain, Sumit
Napoli, Margherita
Narasamdya, Iman
Nickovic, Dejan
Nimal, Vincent
Noll, Thomas
Norman, Gethin
Nuzzo, Pierluigi
Nyman, Ulrik

P

Palmer, Robert
Pandav, Sudhindra
Parente, Mimmo
Parker, David
Parlato, Gennaro
Pedersen, Mikkel L.
Pek, Edgar
Peled, Doron
Pike, Lee
Piskac, Ruzica
Piterman, Nir
Platzer, Andre
Popeea, Corneliu

Q

Qian, Kairong
Qiang, Katherine
Qiu, Xiaokang
Quesel, Jan-David
Quinton, Sophie

R

Rajan, Ajitha
Ravi, Kavita
Reinbacher, Thomas
Rezine, Ahmed
Ridgeway, Jeremy
Rinetzky, Noam
Rintanen, Jussi
Rival, Xavier
Rogalewicz, Adam
Rozanov, Mirron
Rozier, Kristin Yvonne
Rungta, Neha
Ryvchin, Vadim

S

Sa'Ar, Yaniv
Sahoo, Debashis
Sangnier, Arnaud
Sanner, Scott
Saxena, Prateek
Schewe, Sven
Schlich, Bastian
Schuppan, Viktor
Segerlind, Nathan
Sen, Koushik
Sepp, Alexander
Serbanuta, Traian
Sevcik, Jaroslav
Sezgin, Ali
Sharma, Subodh
Sheinvald, Sarai
Sighireanu, Mihaela
Sinha, Nishant
Spalazzi, Luca
Srba, Jiri
Srivastava, Saurabh
Stefanescu, Alin
Steffen, Bernhard
Stoelinga, Marielle
Stoller, Scott
Stursberg, Olaf
Szubzda, Grzegorz

T

Tautschnig, Michael
Thrane, Claus
Tiu, Alwen
Tonetta, Stefano
Tsai, Ming-Hsien
Tsay, Yih-Kuen
Tuerk, Thomas

V

Vardi, Moshe
Vaswani, Kapil
Vechev, Martin
Vestergaard, Rene
Vighio, Saleem
Visser, Willem
Viswanathan, Mahesh
Vojnar, Tomas

W

Wahl, Thomas
Wang, Chao
Wang, Farn
Wang, Liqiang
Wang, Zilong
Weissenbacher, Georg
Wickerson, John
Wieringa, Siert
Wolf, Verena

Y

Yahav, Eran
Yang, Yu
Yen, Hsu-Chun
Yorsh, Greta
Yrke Jørgensen, Kenneth
Yu, Andy
Yu, Fang
Yuan, Jun

Z

Zhang, Lijun
Zhang, Ting
Zhao, Lu
Zhou, Min
Zunino, Roberto

Table of Contents

HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection

Vijay Ganesh[1], Adam Kieżun[2]
Shay Artzi[3], Philip J. Guo[4], Pieter Hooimeijer[5], and Michael Ernst[6]

[1] Massachusetts Institute of Technology,
[2] Harvard Medical School
[3] IBM Research,
[4] Stanford University,
[5] University of Virginia,
[6] University of Washington
vganesh@csail.mit.edu, akiezun@gmail.com, artzi@us.ibm.com,
pg@cs.stanford.edu, pieter@cs.virginia.edu, mernst@cs.washington.edu

Abstract. Many automatic testing, analysis, and verification techniques for programs can effectively be reduced to a constraint-generation phase followed by a constraint-solving phase. This separation of concerns often leads to more effective and maintainable software reliability tools. The increasing efficiency of off-the-shelf constraint solvers makes this approach even more compelling. However, there are few effective and sufficiently expressive off-the-shelf solvers for string constraints generated by analysis of string-manipulating programs, and hence researchers end up implementing their own ad-hoc solvers. Thus, there is a clear need for an effective and expressive string-constraint solver that can be easily integrated into a variety of applications.

To fulfill this need, we designed and implemented Hampi, an efficient and easy-to-use string solver. Users of the Hampi string solver specify constraints using membership predicate over regular expressions, context-free grammars, and equality/dis-equality between string terms. These terms are constructed out of string constants, bounded string variables, and typical string operations such as concatenation and substring extraction. Hampi takes such a constraint as input and decides whether it is satisfiable or not. If an input constraint is satisfiable, Hampi generates a satsfying assignment for the string variables that occur in it.

We demonstrate Hampi's expressiveness and efficiency by applying it to program analysis and automated testing: We used Hampi in static and dynamic analyses for finding SQL injection vulnerabilities in Web applications with hundreds of thousands of lines of code. We also used Hampi in the context of automated bug finding in C programs using dynamic systematic testing (also known as concolic testing). Hampi's source code, documentation, and experimental data are available at `http://people.csail.mit.edu/akiezun/hampi`.

1 Introduction

Many automatic testing [4, 9], analysis [12], and verification [14] techniques for programs can be effectively reduced to a constraint-generation phase followed by a constraint solving phase. This separation of concerns often leads to more effective and

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 1–19, 2011.

maintainable tools. Such an approach to analyzing programs is becoming more effective as off-the-shelf constraint solvers for Boolean SAT [20] and other theories [5, 8] continue to become more efficient. Most of these solvers are aimed at propositional logic, linear arithmetic, theories of functions, arrays or bit-vectors [5].

Many programs (e.g., Web applications) take string values as input, manipulate them, and then use them in sensitive operations such as database queries. Analyses of such string-manipulating programs in techniques for automatic testing [6, 9, 2], verifying correctness of program output [21], and finding security faults [25] produce *string constraints*, which are then solved by custom string solvers written by the authors of these analyses. Writing a custom solver for every application is time-consuming and error-prone, and the lack of separation of concerns may lead to systems that are difficult to maintain. Thus, there is a clear need for an effective and sufficiently expressive off-the-shelf string-constraint solver that can be easily integrated into a variety of applications.

To fulfill this need, we designed and implemented Hampi[1], a solver for constraints over bounded string variables. Hampi constraints express membership in bounded regular and context-free languages, substring relation, and equalities/dis-equalities over string terms.

String terms in the Hampi language are constructed out of string constants, bounded string variables, concatenation, and sub-string extraction operations. Regular expressions and context-free grammar terms are constructed out of standard regular expression operations and grammar productions, respectively. Atomic formulas in the Hampi language are equality over string terms, the membership predicate for regular expressions and context-free grammars, and the substring predicate that takes two string terms and asserts that one is a substring of the other. Given a set of constraints, Hampi outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable.

Hampi is designed to be used as a component in testing, analysis, and verification applications. Hampi can also be used to solve the intersection, containment, and equivalence problems for bounded regular and context-free languages.

A key feature of Hampi is bounding of regular and context-free languages. Bounding makes Hampi different from custom string-constraint solvers commonly used in testing and analysis tools [6]. As we demonstrate in our experiments, for many practical applications, bounding the input languages is not a handicap. In fact, it allows for a more expressive input language that enables operations on context-free languages that would be undecidable without bounding. Furthermore, bounding makes the satisfiability problem solved by Hampi more tractable. This difference is analogous to that between model-checking and bounded model-checking [1].

As one example application, Hampi's input language can encode constraints on SQL queries to find possible injection attacks, such as:

> Find a string v of at most 12 characters, such that the SQL query "`SELECT msg FROM messages WHERE topicid=`v" is a syntactically valid SQL statement, and that the query contains the substring "`OR 1=1`".

[1] This paper is an extended version of the HAMPI paper accepted at the International Symposium on Software Testing and Analysis (ISSTA) 2009 conference. A journal version is under submission.

Note that “`OR 1=1`” is a common tautology that can lead to SQL injection attacks. HAMPI either finds a string value that satisfies these constraints or answers that no satisfying value exists. For the above example, the string “`1 OR 1=1`” is a valid solution.

HAMPI Overview: HAMPI takes four steps to solve input string constraints.

1. Normalize the input constraints to a *core form*, which consists of expressions of the form $v \in R$ or $v \notin R$, where v is a bounded string variable, and R is a regular expression.
2. Translate core form string constraints into a quantifier-free logic of bit-vectors. A bit-vector is a bounded, ordered list of bits. The fragment of bit-vector logic that HAMPI uses allows standard Boolean operations, bit comparisons, and extracting sub-vectors.
3. Invoke the STP bit-vector solver [8] on the bit-vector constraints.
4. If STP reports that the constraints are unsatisfiable, then HAMPI reports the same. Otherwise, STP will generate a satisfying assignment in its bit-vector language, so HAMPI decodes this to output an ASCII string solution.

Experimental Results Summary: We ran four experiments to evaluate HAMPI. Our results show that HAMPI is efficient and that its input language can express string constraints that arise from real-world program analysis and automated testing tools.

1. *SQL Injection Vulnerability Detection (static analysis):* We used HAMPI in a static analysis tool [23] for identifying SQL injection vulnerabilities. We applied the analysis tool to 6 PHP Web applications (total lines of code: 339,750). HAMPI solved all constraints generated by the analysis, and solved 99.7% of those constraints in less than 1 second per constraint. All solutions found by HAMPI for these constraints were less than 5 characters long. These experiments bolster our claim that bounding the string constraints is not a handicap.
2. *SQL Injection Attack Generation (dynamic analysis):* We used HAMPI in Ardilla, a dynamic analysis tool for creating SQL injection attacks [17]. We applied Ardilla to 5 PHP Web applications (total lines of code: 14,941). HAMPI successfully replaced a custom-made attack generator and constructed all 23 attacks on those applications that Ardilla originally constructed.
3. *Input Generation for Systematic Testing:* We used HAMPI in Klee [3], a systematic-testing tool for C programs. We applied Klee to 3 programs with structured input formats (total executable lines of code: 4,100). We used HAMPI to generate constraints that specify legal inputs to these programs. HAMPI's constraints eliminated all illegal inputs, improved the line-coverage by up to 2× overall (and up to 5× in parsing code), and discovered 3 new error-revealing inputs.

1.1 Paper Organization

We first introduce HAMPI's capabilities with an example (§2), then present HAMPI's input format and solving algorithm (§3), and present experimental evaluation (§4). We briefly touch upon related work in (§5).

```
1 $my_topicid = $_GET['topicid'];
2
3 $sqlstmt = "SELECT msg FROM messages WHERE topicid='$my_topicid'";
4 $result = mysql_query($sqlstmt);
5
6 //display messages
7 while($row = mysql_fetch_assoc($result)){
8   echo "Message " . $row['msg'];
9 }
```

Fig. 1. Fragment of a PHP program that displays messages stored in a MySQL database. This program is vulnerable to an SQL injection attack. Section 2 discusses the vulnerability.

```
//string variable representing '$my_topicid' from Figure 1
var v:6..12;   // size is between 6 and 12 characters

//simple SQL context-free grammar
cfg SqlSmall := "SELECT " (Letter)+ " FROM " (Letter)+ " WHERE " Cond;
cfg Cond     := Val "=" Val | Cond " OR " Cond";
cfg Val      := (Letter)+ | "'" (LetterOrDigit)* "'" | (Digit)+;
cfg LetterOrDigit := Letter | Digit;
cfg Letter := ['a'-'z'] ;
cfg Digit  := ['0'-'9'] ;

//the SQL query $sqlstmt from line 3 of Figure 1
val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");

//constraint conjuncts
assert q in SqlSmall;
assert q contains "OR '1'='1'";
```

Fig. 2. Hampi input that, when solved, produces an SQL injection attack vector for the vulnerability from Figure 1

2 Example: SQL Injection

SQL injections are a prevalent class of Web-application vulnerabilities. This section illustrates how an automated tool [17, 25] could use Hampi to detect SQL injection vulnerabilities and to produce attack inputs.

Figure 1 shows a fragment of a PHP program that implements a simple Web application: a message board that allows users to read and post messages stored in a MySQL database. Users of the message board fill in an HTML form (not shown here) that communicates the inputs to the server via a specially formatted URL, e.g., `http://www.mysite.com/?topicid=1`. Input parameters passed inside the URL are available in the `$_GET` associative array. In the above example URL, the input has one key-value pair: `topicid=1`. The program fragment in Figure 1 retrieves and displays messages for the given topic.

This program is vulnerable to an SQL injection attack. An attacker can read all messages in the database (including ones intended to be private) by crafting a malicious URL like:

```
http://www.mysite.com/?topicid=1' OR '1'='1
```

Upon being invoked with that URL, the program reads the string

```
1' OR '1'='1
```

as the value of the `$my_topicid` variable, constructs an SQL query by concatenating it to a constant string, and submits the following query to the database in line 4:

```
SELECT msg FROM messages WHERE topicid='1' OR '1'='1'
```

The `WHERE` condition is always true because it contains the tautology `'1'='1'`. Thus, the query retrieves all messages, possibly leaking private information.

A programmer or an automated tool might ask, "Can an attacker exploit the `topicid` parameter and introduce a `OR '1'='1'` tautology into a syntactically-correct SQL query at line 4 in the code of Figure 1?" The HAMPI solver answers such questions and creates strings that can be used as exploits.

The HAMPI constraints in Figure 2 formalize the question in our example. Automated vulnerability-scanning tools [17, 25] can create HAMPI constraints via either static or dynamic program analysis (we demonstrate both static and dynamic techniques in our evaluation in Sections 4.1 and 4.2, respectively). Specifically, a tool could create the HAMPI input shown in Figure 2 by analyzing the code of Figure 1.

We now discuss various features of the HAMPI input language that Figure 2 illustrates. (Section 3.1 describes the input language in more detail.)

- Keyword `var` (line 2) introduces a *string variable* `v`. The variable has a size in the range of 6 to 12 characters. The goal of the HAMPI solver is to find a string that, when assigned to the string variable, satisfies all the constraints. In this example, HAMPI will search for solutions of sizes between 6 and 12.
- Keyword `cfg` (lines 5–10) introduces a *context-free grammar*, for a fragment of the SQL grammar of `SELECT` statements.
- Keyword `val` (line 13) introduces a temporary variable `q`, declared as a *concatenation* of constant strings and the string variable `v`. This variable represents an SQL query corresponding to the PHP `$sqlstmt` variable from line 3 in Figure 1.
- Keyword `assert` defines a constraint. The top-level HAMPI constraint is a conjunction of `assert` statements. Line 16 specifies that the query string `q` must be a member of the context-free language `SqlSmall` (syntactically-correct SQL). Line 17 specifies that the variable `v` must contain a specific substring (e.g., the `OR '1'='1'` tautology that can lead to an SQL injection attack).

HAMPI can solve the constraints specified in Figure 2 and find a value for `v` such as

```
1' OR '1'='1
```

which is a value for `$_GET['topicid']` that can lead to an SQL injection attack.

3 The Hampi String Constraint Solver

HAMPI finds a string that satisfies constraints specified in the input, or decides that no satisfying string exists. HAMPI works in four steps, as illustrated in Figure 3:

1. Normalize the input constraints to a *core form* (§3.2).
2. Encode core form constraints in bit-vector logic (§3.3).
3. Invoke the STP solver [8] on the bit-vector constraints (§3.3).
4. Decode the results obtained from STP (§3.3).

Users can invoke HAMPI with a text-based command-line front-end (using the input grammar in Figure 4) or with a Java API to directly construct the HAMPI constraints.

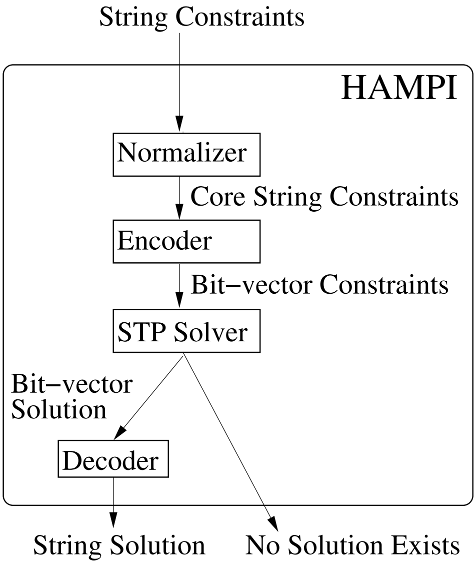


Fig. 3. Schematic view of the HAMPI string constraint solver. Input enters at the top, and output exits at the bottom. Section 3 describes the HAMPI solver.

3.1 HAMPI Input Language for String Constraints

We now discuss the salient features of HAMPI's input language (Figure 4) and illustrate them on examples. The language is expressive enough to encode string constraints generated by typical program analysis, testing, and security applications. HAMPI's language supports declaration of bounded string variables and constants, concatenation and extraction operation over string terms, equality over string terms, regular-language operations, membership predicate, and declaration of context-free and regular languages, temporaries and constraints.

Declaration of String Variable (`var` keyword). A HAMPI input must declare a *single* string variable and specify its size range as lower and upper bounds on the number of characters. If the input constraints are satisfiable, then HAMPI finds a value for the variable that satisfies all constraints. For example, the following line declares a string variable named v with a size between 5 and 20 characters:

```
var v:5..20;
```

Extraction Operation. HAMPI supports extraction of substrings from string terms (as shown in Figure 4). An example of extraction operation is as follows:

```
var longv:20;
val v1 := longv[0:9];
```

where 0 is the offset (or starting character of the extraction operation), and 9 is the length of the resultant string, in terms of the number of characters of *longv*.

Input	::=	*Var Stmt*∗	Hampi input (with a single string variable)
Var	::=	**var** *Id* : *Int..Int*	string variable (length lower..upper bound)
Stmt	::=	*Cfg* | *Reg* | *Val* | *Assert*	statement
Cfg	::=	**cfg** *Id* := *CfgProdRHS*	context-free language
CfgProdRHS	::=	*CFG declaration in EBNF*	Extended Backus-Naur Form (EBNF)
Reg	::=	**reg** *Id* := *RegElem*	regular-language
RegElem	::=	*StrConst*	string constant
	|	*Id*	variable reference
	|	**fixsize**(*Id* , *Int*)	CFG fixed-sizing
	|	**or**(*RegElem* ∗)	union
	|	**concat**(*RegElem* ∗)	concatenation
	|	**star**(*RegElem*)	Kleene star
Val	::=	**val** *Id* :=*ValElem*	temporary variable
ValElem	::=	*Id*	
	|	*StrConst*	
	|	**concat**(*ValElem* ∗)	concatenation
	|	*ValElem*[*offset* : *length*]	extraction(ValElem, offset, length)
Assert	::=	**assert** *Id* [**not**]? **in** *Reg*	regular-language membership
	|	**assert** *Id* [**not**]? **in** *Cfg*	context-free language membership
	|	**assert** *Id* [**not**]? **contains** *StrConst*	substring
	|	**assert** *Id* [**not**]? = *Id*	word equation (equality/dis-equality)
Id	::=	*String identifier*	
StrConst	::=	"*String literal constant*"	
Int	::=	*Non-negative integer*	

Fig. 4. Summary of Hampi's input language. **Terminals** are bold-faced, *nonterminals* are italicized. A Hampi input (*Input*) is a variable declaration, followed by a list of these statements: context-free-grammar declarations, regular-language declarations, temporary variables, and assertions.

Declaration of Multiple Variables. The user can simulate having multiple variables by declaring a single long string variable and using the extract operation: Disjoint extractions of the single long variable can act as multiple variables. For example, to declare two string variables of length 10 named `v1` and `v2`, use:

```
var longv:20;
val v1 := longv[0:9];
val v2 := longv[10:9];
```

The `val` keyword declares a temporary (derived) variable and will be described later in this section.

Declarations of Context-free Languages (`cfg` keyword). Hampi input can declare context-free languages using grammars in the standard notation: Extended Backus-Naur Form (EBNF). Terminals are enclosed in double quotes (e.g., `"SELECT"`), and productions are separated by the vertical bar symbol (`|`). Grammars may contain special symbols for repetition (+ and *) and character ranges (e.g., `[a-z]`). For example, lines 5–10 in Figure 2 show the declaration of a context-free grammar for a subset of SQL.

Hampi's format for context-free grammars is as expressive as that of widely-used tools such as Yacc/Lex; in fact, we have written a simple syntax-driven script that transforms a Yacc specification to Hampi format (available on the Hampi website). Hampi can only solve constraints over bounded context-free grammars. However, the user does not have to manually specify bounds, since Hampi automatically derives a bound by analyzing the bound on the input string variable and the longest possible string that can be constructed out of concatenation and extraction operations.

Declarations of Regular Languages (`reg` keyword). Hampi input can declare regular languages using the following regular expressions: (i) a singleton set with a string constant, (ii) a concatenation/union of regular languages, (iii) a repetition (Kleene star) of a regular language, (iv) bounding of a context-free language, which Hampi does automatically. Every regular language can be expressed using the first three of those operations [22].

For example, `(b*ab*ab*)*` is a regular expression that describes the language of strings over the alphabet {`a,b`}, with an even number of `a` symbols. In Hampi syntax this is:

```
reg Bstar := star("b");           // 'helper' expression
reg EvenA := star(concat(Bstar, "a", Bstar, "a", Bstar));
```

The Hampi website contains a script to convert Perl Compatible Regular Expressions (PCRE) into Hampi syntax. Also note that context-free grammars in Hampi are implicitly bounded, and hence are regular expressions.

Temporary Declarations (`val` keyword). Temporary variables are shortcuts for expressing constraints on expressions that are concatenations of the string variable and constants or extractions. For example, line 13 in Figure 2 declares a temporary variable named `q` by concatenating two constant strings to the variable `v`:

```
val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");
```

Constraints (`assert` keyword). Hampi constraints specify membership of variables in regular and context-free languages, substrings, and word equations. Hampi solves for the conjunction of all constraints listed in the input.

- Membership Predicate (`in`): Assert that a variable is in a context-free or regular language. For example, line 16 in Figure 2 declares that the string value of the temporary variable `q` is in the context-free language `SqlSmall`:

  ```
  assert q in SqlSmall;
  ```

- Substring Relation (`contains`): Assert that a variable contains the given string constant. For example, line 17 in Figure 2 declares that the string value of the temporary variable `q` contains an SQL tautology:

  ```
  assert q contains "OR '1'='1'";
  ```

- String Equalities (=): Asserts that two string terms are equal (also known as word equations). In Hampi, both sides of the equality must ultimately originate from the same single string variable. For example, the `extract` operator can assert that two portions of a string must be equal:

S	::=	*Constraint*	
	|	*S* $\wedge$ *Constraint*	conjunction
Constraint	::=	*StrExp* $\in$ *RegExp*	membership
	|	*StrExp* $\notin$ *RegExp*	non-membership
Constraint	::=	*StrExp* = *StrExp*	equality
	|	*StrExp* $\neq$ *StrExp*	dis-equality
StrExp	::=	*Var*	input variable
	|	*StrConst*	string constant
	|	*StrExp StrExp*	concatenation
	|	*StrExp*[*offset* : *length*]	extraction
RegExp	::=	*StrConst*	constant
	|	*RegExp* + *RegExp*	union
	|	*RegExp RegExp*	concatenation
	|	*RegExp*$\star$	star

Fig. 5. The grammar of core form string constraints. *Var*, *StrConst*, and *Int* are defined in Figure 4.

```
var v:20;
val v1 := v[0:9];
val v2 := v[10:9];
assert v1 = 2v;
```

All of these constraints may be negated by preceding them with a `not` keyword.

3.2 Core Form of String Constraints

After parsing and checking the input, Hampi normalizes the string constraints to a core form. The core form (grammar shown in Figure 5) is an internal intermediate representation that is easier than raw Hampi input to encode in bit-vector logic. A core form string constraint specifies membership (or its negation) in a regular language: *StrExp* $\in$ *RegExp* or *StrExp* $\notin$ *RegExp*, where *StrExp* is an expression composed of concatenations of string constants, extractions, and occurrences of the (sole) string variable, and *RegExp* is a regular expression.

Hampi normalizes its input into core form in 3 steps:

1. Expand all temporary variables, i.e., replace each reference to a temporary variable with the variable's definition (Hampi forbids recursive definitions of temporaries).
2. Calculate maximum size and bound all context-free grammar expressions into regular expressions (see below for the algorithm).
3. Expand all regular-language declarations, i.e., replace each reference to a regular-language variable with the variable's definition.

Bounding of Context-free Grammars: Hampi uses the following algorithm to create regular expressions that specify the set of strings of a fixed length that are derivable from a context-free grammar:

1. Expand all special symbols in the grammar (e.g., repetition, option, character range).
2. Remove ϵ productions [22].

3. Construct the regular expression that encodes all bounded strings of the grammar as follows: First, pre-compute the length of the shortest and longest (if exists) string that can be generated from each nonterminal (i.e., lower and upper bounds). Second, given a size n and a nonterminal N, examine all productions for N. For each production $N ::= S_1 \ldots S_k$, where each S_i may be a terminal or a nonterminal, enumerate all possible partitions of n characters to k grammar symbols (Hampi takes the pre-computed lower and upper bounds to make the enumeration more efficient). Then, create the sub-expressions recursively and combine the subexpressions with a concatenation operator. Memoization of intermediate results makes this (worst-case exponential in k) process scalable.

Here is an example of grammar fixed-sizing: Consider the following grammar of well-balanced parentheses and the problem of finding the regular language that consists of all strings of length 6 that can be generated from the nonterminal `E`.

```
cfg E := "()" | E E | "(" E ")" ;
```

The grammar does not contain special symbols or ϵ productions, so first two steps of the algorithm do nothing. Then, Hampi determines that the shortest string `E` can generate is of length 2. There are three productions for the nonterminal `E`, so the final regular expression is a union of three parts. The first production, `E := "()"`, generates no strings of size 6 (and only one string of size 2). The second production, `E := E E`, generates strings of size 6 in two ways: either the first occurrence of `E` generates 2 characters and the second occurrence generates 4 characters, or the first occurrence generates 4 characters and the second occurrence generates 2 characters. From the pre-processing step, Hampi knows that the only other possible partition of 6 characters is 3–3, which Hampi tries and fails (because `E` cannot generate 3-character strings). The third production, `E := "(" E ")"`, generates strings of size 6 in only one way: the nonterminal `E` must generate 4 characters. In each case, Hampi creates the sub-expressions recursively. The resulting regular expression for this example is (plus signs denote union and square brackets group sub-expressions):

```
()[()() + (())]  +  [()() + (())]()  +  ([()() + (())])
```

3.3 Bit-Vector Encoding and Solving

Hampi encodes the core form string constraints as formulas in the logic of fixed-size bit-vectors. A bit-vector is a fixed-size, ordered list of bits. The fragment of bit-vector logic that Hampi uses contains standard Boolean operations, extracting sub-vectors, and comparing bit-vectors (We refer the reader to [8] for a detailed description of the bit-vector logic used by Hampi) Hampi asks the STP bit-vector solver [8] for a satisfying assignment to the resulting bit-vector formula. If STP finds an assignment, Hampi decodes it, and produces a string solution for the input constraints. If STP cannot find a solution, Hampi terminates and declares the input constraints unsatisfiable.

Every core form string constraint is encoded separately, as a conjunct in a bit-vector logic formula. Hampi encodes the core form string constraint *StrExp* $\in$ *RegExp* recursively, by case analysis of the regular expression *RegExp*, as follows:

– Hampi encodes constants by enforcing constant values in the relevant elements of the bit-vector variable (Hampi encodes characters using 8-bit ASCII codes).

- Hampi encodes the union operator (+) as a disjunction in the bit-vector logic.
- Hampi encodes the concatenation operator by enumerating all possible distributions of the characters to the sub-expressions, encoding the sub-expressions recursively, and combining the sub-formulas in a conjunction.
- Hampi encodes the $\star$ similarly to concatenation — a star is a concatenation with variable number of occurrences. To encode the star, Hampi finds the upper bound on the number of occurrences (the number of characters in the string is always a sound upper bound).

After STP finds a solution to the bit-vector formula (if one exists), Hampi decodes the solution by reading 8-bit sub-vectors as consecutive ASCII characters.

3.4 Example of Hampi Constraint Solving

We now illustrate the entire constraint solving process end-to-end on a simple example. Given the following input:

```
var v:2..2; // fixed-size string of length 2
cfg E := "()" | E E | "(" E ")";
reg Efixed := fixsize(E, 6);
val q := concat( "((" , v , "))" );
assert q in Efixed;    // turns into constraint c1
assert q contains "())"; // turns into constraint c2
```

Hampi tries to find a satisfying assignment for variable `v` by following the four-step algorithm[2] in Figure 3:

Step 1. Normalize constraints to core form, using the algorithm in Section 3.2:

$$\textbf{c1}\ [\texttt{assert q in Efixed}]: \quad \texttt{((}\,v\,\texttt{))} \in \texttt{()}\big[\texttt{()()}\ +\ \texttt{(())}\big] + \big[\texttt{()()}\ +\ \texttt{(())}\big]\texttt{()}\ + \texttt{(}\big[\texttt{()()}\ +\ \texttt{(())}\big]\texttt{)}$$

$$\textbf{c2}\ [\texttt{assert q contains "())"}]: \texttt{((}\,v\,\texttt{))} \in \big[\texttt{(}\ +\ \texttt{)}\big]\star\ \texttt{())}\ \big[\texttt{(}\ +\ \texttt{)}\big]\star$$

Step 2. Encode the core-form constraints in bit-vector logic. We show how Hampi encodes constraint **c1**; the process for **c2** is similar. Hampi creates a bit-vector variable bv of length 6*8=48 bits, to represent the left-hand side of **c1** (since `Efixed` is 6 bytes). Characters are encoded using ASCII codes: '`(`' is 40 in ASCII, and '`)`' is 41. Hampi encodes the left-hand-side expression of **c1**, `((` v `))`, as formula L_1, by specifying the constant values:

$$L_1 : (bv[0] = 40) \wedge (bv[1] = 40) \wedge (bv[4] = 41) \wedge (bv[5] = 41)$$

Bytes $bv[2]$ and $bv[3]$ are reserved for v, a 2-byte variable. The top-level regular expression in the right-hand side of **c1** is a 3-way union, so the result of the encoding is a 3-way disjunction. For the first disjunct $\texttt{()}\big[\texttt{()()}\ +\ \texttt{(())}\big]$, Hampi creates the following formula D_{1a}:

[2] The alphabet of the regular expression or context-free grammar in a Hampi input is implicitly restricted to the terminals specified.

$$\begin{gathered} bv[0] = 40 \wedge bv[1] = 41 \wedge \\ \big((bv[2] = 40 \wedge bv[3] = 41 \wedge bv[4] = 40 \wedge bv[5] = 41\big) \vee \\ \big(bv[2] = 40 \wedge bv[3] = 40 \wedge bv[4] = 41 \wedge bv[5] = 41)\big) \end{gathered}$$

Formulas D_{1b} and D_{1c} for the remaining conjuncts are similar. The bit-vector formula that encodes **c1** is

$$C_1 = L_1 \wedge \big(D_{1a} \vee D_{1b} \vee D_{1c}\big)$$

Similarly, a formula C_2 (not shown here) encodes **c2**. The formula that Hampi sends to the STP solver is

$$(C_1 \wedge C_2)$$

Step 3. STP finds a solution that satisfies the formula:

$$bv[0] = 40, bv[1] = 40, bv[2] = 41, bv[3] = 40, bv[4] = 41, bv[5] = 41$$

In decoded ASCII, the solution is "`(()())`" (quote marks not part of solution string).

Step 4. Hampi reads the assignment for variable v off of the STP solution, by decoding the elements of bv that correspond to v, i.e., elements 2 and 3. Hampi reports the solution for v as "`)(`". String "`()`" is another legal solution for v, but STP only finds one solution.

4 Evaluation

We experimentally tested Hampi's applicability to practical problems involving string constraints and compared Hampi's performance and scalability to another string-constraint solver. We ran the following four experiments:

1. We used Hampi in a static-analysis tool [23] that identifies possible SQL injection vulnerabilities (Section 4.1).
2. We used Hampi in Ardilla [17], a dynamic-analysis tool that creates SQL injection attacks (Section 4.2).
3. We used Hampi in Klee, a systematic testing tool for C programs (Section 4.3).

Unless otherwise noted, we ran all experiments on a 2.2GHz Pentium 4 PC with 1 GB of RAM running Debian Linux, executing Hampi on Sun Java Client VM 1.6.0-b105 with 700MB of heap space. We ran Hampi with all optimizations on, but flushed the whole internal state after solving each input to ensure fairness in timing measurements, i.e., preventing artificially low runtimes when solving a series of structurally-similar inputs. The results of our experiments demonstrate that Hampi is expressive in encoding real constraint problems that arise in security analysis and automated testing, that it can be integrated into existing testing tools, and that it can efficiently solve large constraints obtained from real programs. Hampi's source code and documentation, experimental data, and additional results are available at `http://people.csail.mit.edu/akiezun/hampi`.

4.1 Identifying SQL Injection Vulnerabilities Using Static Analysis

We evaluated Hampi's applicability to finding SQL injection vulnerabilities in the context of a static analysis. We used the tool from Wassermann and Su [23] that, given source code of a PHP Web application, identifies potential SQL injection vulnerabilities. The tool computes a context-free grammar G that conservatively approximates all string values that can flow into each program variable. Then, for each variable that represents a database query, the tool checks whether $L(G) \cap L(R)$ is empty, where $L(R)$ is a regular language that describes undesirable strings or attack vectors (strings that can exploit a security vulnerability). If the intersection is empty, then Wassermann and Su's tool reports the program to be safe. Otherwise, the program may be vulnerable to SQL injection attacks.

An example $L(R)$ that Wassermann and Su use — the language of strings that contain an odd number of unescaped single quotes — is given by the regular expression (we used this R in our experiments):

```
R = (([^']|\')*[^\])?'
    ((([^']|\')*[^\])?'
     (([^']|\')*[^\])?'([^']|\')*
```

Using Hampi in such an analysis offers two important advantages. First, it eliminates a time-consuming and error-prone reimplementation of a critical component: the string-constraint solver. To compute the language intersection, Wassermann and Su implemented a custom solver based on the algorithm by Minamide [19]. Second, Hampi creates concrete example strings from the language intersection, which is important for generating attack vectors; Wassermann and Su's custom solver only checks for emptiness of the intersection, and does not create example strings.

Using a fixed-size string-constraint solver, such as Hampi, has its limitations. An advantage of using an unbounded-length string-constraint solver is that if the solver determines that the input constraints have no solution, then there is indeed no solution. In the case of Hampi, however, we can only conclude that there is no solution of the given size.

Experiment: We performed the experiment on 6 PHP applications. Of these, 5 were applications used by Wassermann and Su to evaluate their tool [23]. We added 1 large application (`claroline`, a builder for online education courses, with 169 kLOC) from another paper by the same authors [24]. Each of the applications has known SQL injection vulnerabilities. The total size of the applications was 339,750 lines of code.

Wassermann and Su's tool found 1,367 opportunities to compute language intersection, each time with a different grammar G (built from the static analysis) but with the same regular expression R describing undesirable strings. For each input (i.e., pair of G and R), we used both Hampi and Wassermann and Su's custom solver to compute whether the intersection $L(G) \cap L(R)$ was empty.

When the intersection is *not* empty, Wassermann and Su's tool cannot produce an example string for those inputs, but Hampi can. To do so, we varied the size N of the string variable between 1 and 15, and for each N, we measured the total Hampi solving time, and whether the result was UNSAT or a satisfying assignment.

Results: We found empirically that when a solution exists, it can be very short. In 306 of the 1,367 inputs, the intersection was *not* empty (both solvers produced identical results). Out of the 306 inputs with non-empty intersections, we measured the percentage for which Hampi found a solution (for increasing values of N): 2% for $N = 1$, 70% for $N = 2$, 88% for $N = 3$, and 100% for $N = 4$. That is, in this large dataset, all non-empty intersections contain strings with no longer than 4 characters. Due to false positives inherent in Wassermann and Su's static analysis, the strings generated from the intersection do not necessarily constitute real attack vectors. However, this is a limitation of the static analysis, not of Hampi.

We measured how Hampi's solving time depends on the size of the grammar. We measured the size of the grammar as the sum of lengths of all productions (we counted ϵ-productions as of length 1). Among the 1,367 grammars in the dataset, the mean size was 5490.5, standard deviation 4313.3, minimum 44, maximum 37955. We ran Hampi for $N = 4$, i.e., the length at which all satisfying assignments were found. Hampi solves most of these queries quickly (99.7% in less than 1 second, and only 1 query took 10 seconds).

4.2 Creating SQL Injection Attacks Using Dynamic Analysis

We evaluated Hampi's ability to automatically find SQL injection attack strings using constraints produced by running a dynamic-analysis tool on PHP Web applications. For this experiment, we used Ardilla [17], a tool that constructs SQL injection and Cross-site Scripting (XSS) attacks by combining automated input generation, dynamic tainting, and generation and evaluation of candidate attack strings.

One component of Ardilla, the *attack generator*, creates candidate attack strings from a pre-defined list of attack patterns. Though its pattern list is extensible, Ardilla's attack generator is neither targeted nor exhaustive: The generator does not attempt to create valid SQL statements but rather simply assigns pre-defined values from the attack patterns list one-by-one to variables identified as vulnerable by the dynamic tainting component; it does so until an attack is found or until there are no more patterns to try.

For this experiment, we replaced the attack generator with the Hampi string solver. This reduces the problem of finding SQL injection attacks to one of string constraint generation followed by string constraint solving. This replacement makes attack creation targeted and exhaustive — Hampi constraints encode the SQL grammar and, if there is an attack of a given length, Hampi is sure to find it.

To use Hampi with Ardilla, we also replaced Ardilla's dynamic tainting component with a concolic execution [10] component. This required code changes were quite extensive but fairly standard. Concolic execution creates and maintains symbolic expressions for each concrete runtime value derived from the input. For example, if a value is derived as a concatenation of user-provided parameter `p` and a constant string `"abc"`, then its symbolic expression is `concat(p, "abc")`. This component is required to generate the constraints for input to Hampi.

The Hampi input includes a partial SQL grammar (similar to that in Figure 2). We wrote a grammar that covers a subset of SQL queries commonly observed in Web applications, which includes `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, all with `WHERE` clauses. The grammar has size is 74, according to the metric of Section 4.1. Each terminal is represented by a single unique character.

We ran our modified Ardilla on 5 PHP applications (the same set as the original Ardilla study [17], totaling 14,941 lines of PHP code). The original study identified 23 SQL injection vulnerabilities in these applications. Ardilla generated 216 Hampi inputs, each of which is a string constraint built from the execution of a particular path through an application. For each constraint, we used Hampi to find an attack string of size $N \leq 6$ — a solution corresponds to the value of a vulnerable PHP input parameter. Following previous work [7, 13], the generated constraint defined an attack as a syntactically valid (according to the grammar) SQL statement with a tautology in the `WHERE` clause, e.g., `OR 1=1`. We used 4 tautology patterns, distilled from several security lists[3]. We separately measured solving time for each tautology and each choice of N. A security-testing tool like Ardilla might search for the shortest attack string for *any* of the specified tautologies.

4.3 Systematic Testing of C Programs

We combined Hampi with a state-of-the-art systematic testing tool, Klee [3], to improve Klee's ability to create valid test cases for programs that accept highly structured string inputs. Automatic test-case generation tools that use combined concrete and symbolic execution, also known as *concolic execution* [4, 11, 15] have trouble creating test cases that achieve high coverage for programs that expect structured inputs, such as those that require input strings from a context-free grammar [18, 9]. The parser components of programs that accept structured inputs (especially those auto-generated by tools such as Yacc) often contain complex control-flow with many error paths; the vast majority of paths that automatic testers explore terminate in parse errors, thus creating inputs that do not lead the program past the initial parsing stage.

Testing tools based on concolic execution mark the target program's input string as totally unconstrained (i.e., *symbolic*) and then build up constraints on the input based on the conditions of branches taken during execution. If there were a way to constrain the symbolic input string so that it conforms to a target program's specification (e.g., a context-free grammar), then the testing tool would only explore non-error paths in the program's parsing stage, thus resulting in generated inputs that reach the program's core functionality.

To demonstrate the feasibility of this technique, we used Hampi to create grammar-based input constraints and then fed those into Klee [3] to generate test cases for C programs. We compared the coverage achieved and numbers of legal (and rejected) inputs generated by running Klee with and without the Hampi constraints.

Similar experiments have been performed by others [18,9], and we do not claim novelty for the experimental design. However, previous studies used custom-made string solvers, while we applied Hampi as an "off-the-shelf" solver without modifying Klee. Klee provides an API for target programs to mark inputs as symbolic and to place constraints on them. The code snippet below uses `klee_assert` to impose the constraint that all elements of `buf` must be numeric before the target program runs:

```
char buf[10]; // program input
klee_make_symbolic(buf, 10); // make all 10 bytes symbolic

// constrain buf to contain only decimal digits
for (int i = 0; i < 10; i++)
  klee_assert(('0' <= buf[i]) && (buf[i] <= '9'));

run_target_program(buf); // run target program with buf as input
```

[3] http://www.justinshattuck.com/2007/01/18/mysql-injection-cheat-sheets
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku
http://pentestmonkey.net/blog/mysql-sql-injection-cheat-sheet

Table 1. The result of using Hampi grammars to improve coverage of test cases generated by the Klee systematic testing tool. ELOC lists *Executable* Lines of Code, as counted by `gcov` over all `.c` files in program (whole-project line counts are several times larger, but much of that code does not directly execute). Each trial was run for 1 hour. To create minimal test suites, Klee only generates a new input when it covers new lines that previous inputs have not yet covered; the total number of explored paths is usually 2 orders of magnitude greater than the number of generated inputs. Column symbolic shows results for runs of Klee without a Hampi grammar. Column symbolic + grammar shows results for runs of Klee with a Hampi grammar. Column combined shows accumulated results for both kinds of runs. Section 4.3 describes the experiment.

`cueconvert` (939 ELOC, 28-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	32.2%	51.4%	56.2%
% parser file line coverage (48 lines):	20.8%	77.1%	79.2%
# legal inputs / # generated inputs (%):	0 / 14 (0%)	146 / 146 (100%)	146 / 160 (91%)

`logictree` (1,492 ELOC, 7-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	31.2%	63.3%	66.8%
% parser file line coverage (17 lines):	11.8%	64.7%	64.7%
# legal inputs / # generated inputs (%):	70 / 110 (64%)	98 / 98 (100%)	188 / 208 (81%)

`bc` (1,669 ELOC, 6-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	27.1%	43.0%	47.0%
% parser file line coverage (332 lines):	11.8%	39.5%	43.1%
# legal inputs / # generated inputs (%):	2 / 27 (5%)	198 / 198 (100%)	200 / 225 (89%)

Hampi simplifies writing input-format constraints. Simple constraints, such as those above, can be written by hand, but it is infeasible to manually write more complex constraints for specifying, for example, that `buf` must belong to a particular context-free language. We use Hampi to automatically compile such constraints from a grammar down to C code, which can then be fed into Klee.

We chose 3 open-source programs that specify expected inputs using context-free grammars in Yacc format (a subset of those used by Majumdar and Xu [18]). `cueconvert` converts music playlists from `.cue` format to `.toc` format. `logictree` is a solver for propositional logic formulas. `bc` is a command-line calculator and simple programming language. All programs take input from `stdin`; Klee allows the user to create a fixed-size symbolic buffer to simulate `stdin`, so we did not need to modify these programs. For each target program, we ran the following experiment on a 3.2 GHz Pentium 4 PC with 1 GB of RAM running Fedora Linux:

1. Automatically convert its Yacc specification into Hampi's input format (described in Section 3.1), using a script we wrote. To simplify lexical analysis, we used either a single letter or numeric digit to represent certain tokens, depending on its Lex specification (this should not reduce coverage in the parser).
2. Add a fixed-size restriction to limit the input to N bytes. Klee (similarly to, for example, SAGE [11]) actually requires a fixed-size input, which matches well with Hampi's fixed-size input language. We empirically picked N as the largest input size for which Klee does not run out of memory. We augmented the Hampi input to allow for strings with arbitrary numbers of trailing spaces, so that we can generate program inputs *up to* size N.

3. Run HAMPI to compile the input grammar file into STP bit-vector constraints (described in Section 3.3).
4. Automatically convert the STP constraints into C code that expresses the equivalent constraints using C variables and calls to `klee_assert()`, with a script we wrote (the script performs only simple syntactic transformations since STP operators map directly to C operators).
5. Run Klee on the target program using an *N*-byte input buffer, first marking that buffer as symbolic, then executing the C code that imposes the input constraints, and finally executing the program itself.
6. After a 1-hour time-limit expires, collect all generated inputs and run them through the original program (compiled using `gcov`) to measure coverage and legality of each input.
7. As a control, run Klee for 1 hour using an *N*-byte symbolic input buffer (with no initial constraints), collect test cases, and run them through the original program to measure coverage and legality of each input.

Table 1 summarizes our experimental setup and results. We made 3 sets of measurements: total line coverage, line coverage in the Yacc parser file that specifies the grammar rules alongside C code snippets denoting parsing actions, and numbers of inputs (test cases) generated, as well as how many of those inputs were *legal* (i.e., not rejected by the program as a parse error).

The run times for converting each Yacc grammar into HAMPI format, fixed-sizing to *N* bytes, running HAMPI on the fixed-size grammar, and converting the resulting STP constraints into C code are negligible; together, they took less than 1 second for each of the 3 programs. Using HAMPI in Klee improved coverage. Constraining the inputs using a HAMPI grammar resulted in up to 2× improvement in total line coverage and up to 5× improvement in line coverage within the Yacc parser file. Also, as expected, it eliminated all illegal inputs.

Using *both* sets of inputs (combined column) improved upon the coverage achieved using the grammar by up to 9%. Upon manual inspection of the extra lines covered, we found that it was due to the fact that the runs with and without the grammar covered non-overlapping sets of lines: The inputs generated by runs without the grammar (symbolic column) covered lines dealing with processing parse errors, whereas the inputs generated with the grammar (symbolic + grammar column) never had parse errors and covered core program logic. Thus, combining test suites is useful for testing both error and regular execution paths.

With HAMPI's help, Klee uncovered more errors. Using the grammar, Klee generated 3 distinct inputs for `logictree` that uncovered (previously unknown) errors where the program entered an infinite loop. We do not know how many distinct errors these inputs identify. Without the grammar, Klee was not able to generate those same inputs within the 1-hour time limit; given the structured nature of those inputs (e.g., one is "`@x $y z`"), it is unlikely that Klee would be able to generate them within any reasonable time bound without a grammar.

We manually inspected lines of code that were not covered by any strategy. We discovered two main hindrances to achieving higher coverage: First, the input sizes were still too small to generate longer productions that exercised more code, especially problematic for the playlist files for `cueconvert`; this is a limitation of Klee running out of memory and not of Hampi. Second, while grammars eliminated all parse errors, many generated inputs still contained *semantic* errors, such as malformed bc expressions and function definitions (again, unrelated to Hampi).

5 Related Work

Decision procedures have received widespread attention within the context of program analysis, testing, and verification. Decision procedures exist for theories such as Boolean satisfiability [20] and bit-vectors [8]. In contrast, until recently there has been relatively little work on practical and expressive solvers that reason about strings or sets of strings directly. Since this is a tutorial paper we do not discuss related work in detail. Instead we point the reader to our ISSTA 2009 paper [16] for a detailed overview of previous work on decision procedures for theories of strings and practical string solvers.

References

1. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
2. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
3. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Symposium on Operating Systems Design and Implementation. USENIX Association, San Diego (2008)
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Conference on Computer and Communications Security. ACM Press, Alexandria (2006)
5. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: International Symposium on Software Testing and Analysis. ACM Press, London (2007)
7. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting SQL injection vulnerabilities. In: International Computer Software and Applications Conference. IEEE, Beijing (2007)
8. 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)
9. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Programming Language Design and Implementation. ACM Press, Tuscon (2008)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Programming Language Design and Implementation, Chicago, Illinois. ACM Press, New York (2005)
11. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium, San Diego, California. The Internet Society (2008)
12. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Programming Language Design and Implementation, Tuscon, Arizona. ACM Press, New York (2008)
13. Halfond, W., Orso, A., Manolios, P.: WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. Transactions on Software Engineering 34(1), 65–81 (2008)

14. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: International Symposium on Software Testing and Analysis, Portland, Oregon. ACM Press, New York (2000)
15. Jayaraman, K., Harvison, D., Ganesh, V., Kiezun, A.: jFuzz: A concolic whitebox fuzzer for Java. In: NASA Formal Methods Symposium. NASA, Moffett Field (2009)
16. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: International Symposium on Software Testing and Analysis, pp. 105–116. ACM Press, New York (2009)
17. Kiezun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: International Conference on Software Engineering. IEEE, Vancouver (2009)
18. Majumdar, R., Xu, R.-G.: Directed test generation using symbolic grammars. In: Automated Software Engineering. ACM/IEEE (2007)
19. Minamide, Y.: Static approximation of dynamically generated Web pages. In: International World Wide Web Conference, Chiba, Japan. ACM Press, New York (2005)
20. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference, Las Vegas, Nevada. ACM Press, New York (2001)
21. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Testing: Academic and Industrial Conference Practice and Research Techniques, Windsor, UK. IEEE Computer Society Press, Los Alamitos (2007)
22. Sipser, M.: Introduction to the Theory of Computation. In: Course Technology, Florence, KY (2005)
23. Wassermann, G., Su, Z.: Sound and precise analysis of Web applications for injection vulnerabilities. In: Programming Language Design and Implementation. ACM, San Diego (2007)
24. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: International Conference on Software Engineering. IEEE, Leipzig (2008)
25. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for Web applications. In: International Symposium on Software Testing and Analysis. ACM, Seattle (2008)

Using Types for Software Verification

Ranjit Jhala

University of California at San Diego

Traditional software verification algorithms work by using a combination of Floyd-Hoare Logics, Model Checking and Abstract Interpretation, to infer (and check) suitable program invariants. However, these techniques are problematic in the presence of complex (but ubiquitous) constructs like generic data structures, first-class functions.

We demonstrate that modern type systems are capable of the kind of analysis needed to analyze the above constructs, and we use this observation to develop Liquid Types, a new static verification technique which combines the complementary strengths of Floyd-Hoare logics, Model Checking, and Types.

We start in a high-level functional setting (Ocaml), and show how liquid types can be used to statically verify properties ranging from memory safety to data structure "correctness". We will then show how, by carefully reasoning about pointer arithmetic and aliasing, we can profitably use Liquid Types to verify low-level imperative (C) programs.

This presentation is based on joint work with Patrick Rondon and Ming Kawaguchi.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, p. 20, 2011.

SMT-Based Modular Analysis of Sequential Systems Code

Shuvendu K. Lahiri

Microsoft Research

Abstract. In this paper, we describe a few challenges that accompany SMT-based precise verification of systems code (device drivers, file systems) written in low-level languages such as C/C++. First, the presence of pointer arithmetic and untrusted casts make type checking difficult; we show how to formalize C type safety checking and exploit the types for disambiguation of addresses in the heap. Second, the prevalence of explicit manipulation of pointers in data structures using dereference and address arithmetic precludes abstract reasoning about data structures. We provide an expressive and efficient theory for reasoning about linked lists, which comprise most data structures in systems code. We discuss extensions to standard SMT solvers to tackle these issues in the context of the HAVOC verifier.

1 Introduction

A majority of systems software (device drivers, file systems etc.) continue to be written in low-level languages such as C and C++. These languages offer developers the potential to obtain raw performance by low-level control over object layout and object management. However, the gains come at the expense of lack of type and memory safety, lack of modularity and large bloated monolithic components with several hundred thousands of lines. These factors impose additional challenges for the analysis of systems code, in addition to those posed by higher level languages such as Java and C#.

In this work, we discuss our experience with applying *satisfiability modulo theories* (SMT) solvers [7] for *predictable* analysis of systems software, namely in the context of the HAVOC verifier [4]. Predictable analysis constitutes *precise* and *efficient* checking of assertions across loop-free and call-free program fragments.

- By precision, we denote an assertion logic (for writing pre/post conditions, loop invariants) expressive enough to be closed under *weakest liberal preconditions* [3] across a bounded code fragment.
- By efficient, we imply the complexity of the decision problem for the assertion logic. Since many efficiently solvable SMT logics (Boolean satisfiability (SAT), integer linear arithmetic, theory of arrays) have NP-complete decision problems, we consider logics with NP-complete decision problems to be efficiently decided in practice.

The use of such predictable verifiers can be extended to whole programs by combining them with user-supplied or automatically inferred procedure contracts, and loop invariants. We do not focus on the issue of inferring such annotations in this work.

We focus on two main aspects of analysis of systems software in this paper:

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 21–27, 2011.

1. *Lack of type-safety*: We discuss the challenges in checking type-safety of these low-level programs and the implications for modular property checking. We show how to formalize the type-safety of C programs as state assertions, and augmenting SMT solvers with a theory of low-level C types. Details of this work can be found in an earlier paper [2].
2. *Low-level lists*: Linked lists form a majority of linked data structures in systems code; we show the difficulty of employing abstractions on top of such lists given explicit manipulation of addresses and links. We present an SMT theory of lists that allows stating many interesting invariants for code manipulating such lists. Details of this work can be found in the following works [4,6].

In the next few sections, we briefly summarize the issues and the solutions in a semi-formal fashion to enable quick reading. Interested readers are encouraged to refer to the detailed works for more elaborate treatment on each topic.

2 Basic Memory Model

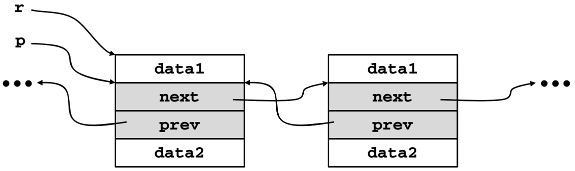
For the sake of illustration in this paper, we will assume a simplified subset of C programs where the only primitive type consists of *integers* int. Addresses and integer values are treated as integers. We ignore the issue of sub-word access, where an integer may be split up into 4 characters, or 2 shorts. The state of the heap is modeled using an mutable array Mem : int → int that maps an address to a value or another address.

Variables whose addresses are taken (using &) and structures are allocated on the heap. Read from a pointer $*e$ is modeled as $\mathsf{Mem}[||e||]$, a lookup into the array Mem at the location corresponding to the value of the C expression e (denoted by $||.||$). Similarly a write $*e = x$ is modeled as $\mathsf{Mem}[||e||] := ||x||$, an update to Mem. Field accesses $e \rightarrow f$ are compiled as pointer accesses with a field offset, $*(e + \mathrm{Offset}(f))$, where $\mathrm{Offset}(f)$ is the (static) offset of the field f in the structure pointed to by e. The different operations (arithmetic, relational) are translated as appropriate operations on integers.

3 Types

Consider the method init_record in Figure 1. The parameter p is a pointer to a list structure. In most systems program, it is common to use a structure similar to list to define a generic doubly-linked list. This structure can be embedded in any structure (such as record) as a field to create a list of such structures. The programming paradigm is uniformly used as the list manipulation routines (insertion, deletion, test for emptiness) are defined once on the list structure. A structure can have multiple fields of type list if the same structure can be part of multiple lists.

However, this also poses several challenges for type-safety as can be seen from the example. First, the type of the enclosing structure is not evident from the signature of the parameter of init_record. Second, programs need to use a macro like CONTAINING_RECORD that obtains the pointer to the enclosing structure from the address of an internal field. This involves non-trivial pointer arithmetic and type casts, the safety of which is not easy to justify.



```
struct list   { list *next; list *prev; }
struct record { int data1; list node; int data2; }

#define CONTAINING_RECORD(x, T, f) ((T *)((int)(x) - (int)(&((T *)0)->f)))

void init_record(list *p) {
    record *r = CONTAINING_RECORD(p, record, node);
    r->data2 = 42;
}

void init_all_records(list *p) {
  while (p != NULL) {
    init_record(p);
    p = p->next;
  }
}
```

Fig. 1. Example C code. The diagram shows two `record` structures in a linked list, with the embedded `list` shown in gray.

To create a sound analysis, one can completely disregard the types and field names in the program. However, this poses two main issues:

- The presence of types and checking for well-typed programs may guarantee the absence of some class of runtime memory safety errors (accesses to invalid regions in memory).
- Types also provide for disambiguation between different parts of the heap, where a read/write to pointers of one type cannot affect the values in other types/fields. For instance, any reasonable program analysis will need to establish that the value in `data1` field in any structure is not affected by `init_record`.

3.1 Formalizing Types

We address these problems by formalizing types as predicates over the program state along with an explicit type-safety invariant [2]. We introduce a map $\mathsf{Type} : \mathsf{int} \rightarrow \mathsf{type}$ that maps each allocated heap location to a type, and two predicates Match and $\mathsf{HasType}$. The Match predicate lifts Type to types that span multiple addresses. Formally, for address a and type t, $\mathsf{Match}(a, t)$ holds if and only if the Type map starting at address a matches the type t. The $\mathsf{HasType}$ predicate gives the meaning of a type. For a word-sized value v and a word-sized type t, $\mathsf{HasType}(v, t)$ holds if and only if the value v has type t.

The definitions of Match and $\mathsf{HasType}$ are given in Figure 2. For Match, the definitions are straightforward: if a given type is a word-sized type (int or $\mathsf{Ptr}(t)$ where Ptr is a pointer type constructor), we check Type at the appropriate address, and for

Definitions for Int

$$\mathsf{Match}(a, \mathsf{Int}) \triangleq \mathsf{Type}[a] = \mathsf{Int} \qquad (A)$$

$$\mathsf{HasType}(v, \mathsf{Int}) \triangleq \mathsf{true} \qquad (B)$$

Definitions for $\mathsf{Ptr}(t)$

$$\mathsf{Match}(a, \mathsf{Ptr}(t)) \triangleq \mathsf{Type}[a] = \mathsf{Ptr}(t) \qquad (C)$$

$$\mathsf{HasType}(v, \mathsf{Ptr}(t)) \triangleq v = 0 \vee (v > 0 \wedge \mathsf{Match}(v, t)) \quad (D)$$

Definitions for $\texttt{type t} = \{f_1 : \sigma_1; \ldots; f_n : \sigma_n\}$

$$\mathsf{Match}(a, \mathsf{T}) \triangleq \bigwedge_i \mathsf{Match}(a + \mathrm{Offset}(f_i), T(\sigma_i)) \qquad (E)$$

Fig. 2. Definition of HasType and Match for a, v of sort int and t of sort type

structure types, we apply Match inductively to each field. For HasType, we only need definitions for word-sized types. For integers, we allow all values to be of integer type, and for pointers, we allow either zero (the null pointer) or a positive address such that the allocation state (as given by Match) matches the pointer's base type. HasType is the core of our technique, since it explicitly defines the correspondence between values and types.

Now that we have defined HasType, we can state our type safety invariant for the heap:

$$\forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$$

In other words, for all addresses a in the heap, the value at $\mathsf{Mem}[a]$ must correspond to the type at $\mathsf{Type}[a]$ according to the HasType axioms. Our translation enforces this invariant at all program points, including preconditions and postconditions of each procedure. We have thus reduced the problem of type safety checking to checking assertions in a program.

The presence of the Type also allows us to distinguish between pointers of different types. In fact, we provide a refinement of the scheme described here to allow names of word-sized fields in the range of Type. This allows to establish that writes to the `data2` field in `init_record` does not affect the `data1` field of any other objects.

3.2 SMT Theory for Types

By using standard verification condition generation [1], the checking of the type safety assertions in a program reduces to checking a ground formula. The formula involves the application of Mem, Type, Match and HasType predicates, in addition to arithmetic symbols. The main challenge is to find an assignment that respects the definition of Match and HasType from Figure 2 and satisfies the type safety assertion; all of these can be expressed as quantified background axioms. We show that it suffices to instantiate these quantifiers at a small number of terms (with at most quadratic blowup) to produce an equisatisfiable ground formula, where the predicates Match and HasType are completely uninterpreted. This ensures that the type safety can be checked for low-level C programs in logics with NP-complete decision problem.

4 Low-Level Data Structures

Consider the procedure `init_record` in Figure 1. To ensure type safety, we need the following precondition for this procedure:

$$\mathsf{HasType}(p - 1, \mathsf{Ptr}(\mathsf{Record}))$$

to indicate that $p - 1$ is a pointer to a `record` structure. However, to prove this precondition at the call site in `init_all_records`, we need a loop invariant that asserts that for any pointer $x \in \{p, *(p + 0), *(*(p + 0) + 0), \ldots\}$, $x - 1 \neq$ `null` and $\mathsf{HasType}(x - 1, \mathsf{Ptr}(\mathsf{Record}))$ holds. This set represents the set of pointers reachable from p using the `next` field. Similarly, the set of pointers obtained by following the `prev` field is $\{p, *(p + 1), *(*(p + 1) + 1), \ldots\}$, because the `prev` field is at an offset of 1 inside the `list` structure.

Unlike most high level languages, where the internal details of a list are hidden from the clients, most systems program explicitly use the fields `next` and `prev` to iterate over lists. The lists are not well encapsulated and clients can have multiple pointers inside a list. In addition, the presence of pointer arithmetic (as described above) to obtain enclosing objects make it difficult to reason about such lists abstractly.

4.1 Reachability Predicate

We define a logic that can describe properties of a set of pointers in a list [4]. The main idea is to introduce a set constructor $\mathit{Btwn} : (\mathsf{int} \to \mathsf{int}) \times \mathsf{int} \times \mathsf{int} \to 2^{\mathsf{int}}$ that takes a map, and two integer addresses such that $\mathit{Btwn}(f, x, y)$ returns the pointers $\{x, f[x], f[f[x]], \ldots, y\}$ (between x and y) when x reaches y by dereferencing f, or $\{\}$ otherwise. The salient points of the assertion logic are:

1. The logic can model singly and doubly linked lists, including cyclic lists.
2. Formulas in this logic are closed under the weakest liberal precondition transformer with respect to the statements in a language with updates to the maps (corresponding to updating fields such as `next` or `prev`). In other words, the formula $\mathit{Btwn}(f[a := b], x, y)(z)$ can be expressed in terms of $\mathit{Btwn}(f, w_1, w_2)(w_3)$, where $w_i \in \{a, b, x, y\}$. This allows for precise reasoning for a loop-free and call-free fragment of code annotated with assertions in this logic.
3. The logic provides for expressing quantified facts about all elements in a list such as
$$\forall x : \mathsf{int} \in \mathit{Btwn}(f, a, b) \;::\; \phi(x)$$
as long as $\phi(x)$ maintains a *sort restriction* [4]. Although the logic can express a variety of useful invariants for lists (initialization, sortedness, uniqueness), it restricts the use of terms such as $f[x]$ inside $\phi(x)$. Intuitively, this may result in generating an unbounded set of terms of a list $x, f[x], f[f[x]], \ldots$ when instantiating the quantified assertions.
4. The decision problem for the resulting logic (when combined with other SMT logics such as arrays, arithmetic, equality) still remains NP-complete. We encode the decision procedure as a set of rewrite rules using *triggers* supported for quantifier reasoning in SMT solvers.

4.2 Modeling Low-Level Lists

As described earlier, elements in most lists in systems programs are obtained by first incrementing pointers by a constant offset before dereferencing the heap. For example, the list of pointers reachable through the `prev` fields are

$$\{p, \mathsf{Mem}[p+1], \mathsf{Mem}[\mathsf{Mem}[p+1]+1], \ldots, \mathtt{null}\}$$

In HAVOC, we model this list as $Btwn(shift_1(\mathsf{Mem}), p, \mathtt{null})$, where $shift_c : (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$ satisfies the following properties:

$$\begin{array}{l} \forall x : \mathsf{int} :: \; shift_c(f)[x] = f[x+c] \\ \forall x : \mathsf{int}, v : \mathsf{int} :: \; shift_c(f[x+c := v]) = shift_c(f)[x := v] \end{array}$$

The first axiom states that the content of $shift_c(f)$ is shifted by c with respect to the contents of f. The second axiom eliminates updates $f[x+c := v]$ nested inside $shift$. These axioms are carefully guarded by triggers such that they fire only for updates to the appropriate linking field such as `next` or `prev`. These extensions prevent us from guaranteeing a decision procedure for assertions about lists in C programs. However, we have observed very few unpredictable behavior in practice, even when reasoning about lists in programs several thousand lines large [6].

5 Other Challenges

In this paper, we have highlighted two issues that make precise verification difficult for systems programs. There are several other issues that pose interesting challenges. A few of them are:

- Since systems program rely on the user to perform object management, the problem of *double free* can lead to variety of undesired behaviors. When freeing objects over an unbounded data structures, we need to capture that pointers have unique references to them. These facts can often be captured as part of a module invariant, but may be broken temporarily inside the module. We describe *intra-module inference* to address similar issues in the context of large code bases [6].
- Since all the lists in a program share the `next` and `prev` fields, a modification to the field in one list may potential affect a completely disjoint list. It becomes more challenging when the procedure modifying the `next` field of a list does not have the other lists in scope. We discuss this imprecision in the context of *call invariants* [5] that offers a mechanism to leverage intraprocedural analysis to deal with interprocedural reasoning precisely.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis For Software Tools and Engineering (PASTE 2005), pp. 82–87 (2005)
2. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (POPL 2009), pp. 302–314 (2009)

3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18, 453–457 (1975)
4. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: Principles of Programming Languages (POPL 2008), pp. 171–182 (2008)
5. Lahiri, S.K., Qadeer, S.: Call invariants. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 237–251. Springer, Heidelberg (2011)
6. Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 493–508. Springer, Heidelberg (2009)
7. Satisfiability Modulo Theories Library (SMT-LIB), `http://goedel.cs.uiowa.edu/smtlib/`

Logic and Compositional Verification of Hybrid Systems*
(Invited Tutorial)

André Platzer

Carnegie Mellon University, Computer Science Department,
Pittsburgh, PA, USA
aplatzer@cs.cmu.edu

Abstract. Hybrid systems are models for complex physical systems and have become a widely used concept for understanding their behavior. Many applications are safety-critical, including car, railway, and air traffic control, robotics, physical-chemical process control, and biomedical devices. Hybrid systems analysis studies how we can build computerised controllers for physical systems which are guaranteed to meet their design goals. The continuous dynamics of hybrid systems can be modeled by differential equations, the discrete dynamics by a combination of discrete state-transitions and conditional execution. The discrete and continuous dynamics interact to form hybrid systems, which makes them quite challenging for verification.

In this tutorial, we survey state-of-the-art verification techniques for hybrid systems. In particular, we focus on a coherent logical approach for systematic hybrid systems analysis. We survey theory, practice, and applications, and show how hybrid systems can be verified in the hybrid systems verification tool KeYmaera. KeYmaera has been used successfully to verify safety, reactivity, controllability, and liveness properties, including collision freedom in air traffic, car, and railway control systems. It has also been used to verify properties of electrical circuits.

1 Introduction

Hybrid systems are a common model for systems where both discrete and continuous behavior are important [2, 8, 10, 20]. Hybrid systems arise, for instance, when a computer controls a physical process. Then the computer will cause discrete transitions and digital switching at various discrete points in time, while the physical process keeps evolving continuously.

As a common mathematical model for such complex physical systems, *hybrid systems* are dynamical systems [28] where the system state evolves over time

* This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, NSF EXPEDITION CNS-0926181, and under Grant No. CNS-0931985, and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 28–43, 2011.

according to interacting laws of discrete and continuous dynamics [1, 9, 10, 16, 20, 43].

One canonical example is a train driving on a railway track. The train moves continuously on the track while its behavior is controlled by several computer control systems supporting the train conductor even up to full automation. One of the most crucial safety-critical correctness properties of a train is that we want to ensure that the train controller prevents all train collisions. A study of the correctness of the train cannot be split into an isolated study of the software and an isolated study of the mechanical parts. They work together and need to be verified together. We cannot determine whether a software controller for a part of a train is correct unless we understand enough of the physics of the train that it controls. We cannot fully understand how a train moves physically without understanding how its digital controllers, control programs, sensors, and actuators affect its behavior. We need to look at both, i.e., the hybrid system dynamics, to find out.

Hybrid systems are equally important in the automotive, aviation, railway, and robotics industry for instance. They occur in factory automation problems and biological, chemical, and physical process control. Most of these applications are safety-critical, because badly controlled processes can have a huge impact on the system environment, especially when the processes operate close to humans. Hybrid systems verification is a very challenging but important problem for which a range of techniques have been developed [10, 12–14, 16, 17, 20–22, 24–26, 41, 42].

In this tutorial, we survey a number of state-of-the-art verification techniques for hybrid systems, especially a logical approach for hybrid systems analysis [30–32]. This approach forms the basis for the differential invariants as fixed points procedure [37] that computes the invariants and differential invariants required for verification in a fixed point loop. This logic-based verification approach has been implemented in the verification tool KeYmaera[1] for hybrid systems [39]. KeYmaera has been used successfully to verify several safety-critical properties, including collision freedom, of the cooperation protocol of the European Train Control System [40] and of aircraft roundabout maneuvers [31] and the flyable aircraft roundabout maneuver [38]. More details about the hybrid systems verification techniques surveyed in this tutorial can be found in the book *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*[2] [32].

The approach presented in this tutorial and the verification tool KeYmaera is also very instructive for teaching hybrid systems verification and the use of logic and formal methods for complex physical systems. The sophisticated graphical user interface of KeYmaera makes it easier to work with the system and learn how hybrid systems verification works. It also makes it easier to understand proofs that KeYmaera found automatically. KeYmaera's interaction capabilities, which are based on those of KeY [4], also help solving very complex verification questions and system design questions interactively that are still beyond the capabilities of today's automation techniques. The author has taught two graduate

[1] `http://symbolaris.com/info/KeYmaera.html`

[2] `http://symbolaris.com/lahs/`

courses on hybrid systems verification using the approach presented here. Course material is available at the web page[2] of the book [32].

Even though we do not focus on these extensions in this tutorial, the approach taken in this tutorial can be extended to logic and verification techniques for distributed hybrid systems [33, 34], i.e., systems that are both distributed systems and hybrid systems. These distributed hybrid systems include multi-agent hybrid systems, reconfigurable hybrid systems, and hybrid systems with an evolving and unbounded number of agents. The approach also extends to logic and verification techniques for stochastic hybrid systems [35].

2 Hybrid Systems

There is a range of models for hybrid systems [2, 5–7, 10, 15, 20, 27, 30, 31, 43]. We focus on hybrid programs [30, 32], and the related model of hybrid automata [2, 20].

Hybrid system models allow the user to specify the continuous dynamics by differential equations. Continuous dynamics results, e.g., from the continuous movement of a train along the track (train position z evolves with velocity v along the differential equation $z' = v$ where z' is the time-derivative of z) or from the continuous variation of its velocity over time ($v' = a$ with acceleration a). Other behavior can be modelled more naturally by discrete dynamics, for example, the instantaneous change of control variables like the acceleration (e.g., the changing of a by setting $a := -b$ with braking force $b > 0$) or change of status information in discrete controllers. Both kinds of dynamics interact, e.g., when measurements of the continuous state affect decisions of discrete controllers (the train switches to braking mode when velocity v is too high). Likewise, they interact when the resulting control choices take effect by changing the control variables of the continuous dynamics (e.g., changing the acceleration control variable a in $z'' = a$). The combination of continuous dynamics with analog or discrete control causes complex system behavior, which can neither be verified by purely continuous reasoning (because of the discontinuities caused by discrete transitions) nor by considering discrete change in isolation (because safety depends on continuous states).

2.1 Undecidability of Numerical Image Computation

Verification of hybrid systems is a very challenging problem. The verification problem is the problem to decide whether a given hybrid system satisfies a given correctness property (e.g., safety, liveness, and so on). Unfortunately, this problem is undecidable even for very simple hybrid systems [11, 20].

Even for absurdly limited models of hybrid systems, the verification problem is neither semidecidable nor co-semidecidable numerically, even for a bounded number of transitions and when tolerating arbitrarily large error bounds in the decision [36]. The numerical image computation problem plays a role that is almost as central as that of the halting problem for Turing machines. We refer to the literature [36] for a formal statement and proof. The basic intuition behind the undecidability result for the numerical image computation problem is shown in Fig. 1. Suppose an algorithm could decide safety of a system numerically by evaluating the value of the system flow φ at points. If the algorithm is a decision algorithm, it would have to terminate in finite time, hence, after evaluating a finite number of points, say x_1, x_2, x_3 in Fig. 1. But from the information that the algorithm has gathered at a finite number of points, it cannot distinguish the good behavior φ (solid flow safely outside B) from the bad behavior g (dashed flow reaching bad region B). The same undecidability result still holds even when restricting the flow φ to very special classes of functions and when assuming that its derivatives could be evaluated and even when tolerating arbitrarily large error bounds in the decision [36]. There is a series of extra assumptions and bounds that make the problem (approximately) decidable again by imposing extra constraints on the system; see [36]. Yet, by the general undecidability result, these extra bounds (and several other bounds that have been proposed in related work) cannot be computed numerically. Because of this strong numerical undecidability result, it is surprisingly difficult but not impossible to get hybrid systems verification techniques sound [17, 41].

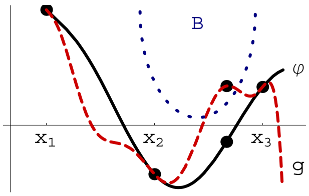


Fig. 1. Indistinguishable

Consequently, sound verification of hybrid systems needs some symbolic part. In the remainder of this tutorial, we focus on a purely symbolic and logical approach that is formally sound, i.e., the verification result is always correct.

2.2 Hybrid Programs

Hybrid programs are program models for hybrid systems and are formed using the statements and operations in Table 1.

Discrete jump sets. Discrete transitions are represented as instantaneous assignments of values to state variables. They can express resets like $a := -b$ or adjustments of control variables like $a := A$. To handle simultaneous changes of multiple variables, discrete jumps can be combined to sets of jumps with simultaneous effect. For instance, the discrete jump set $a := a + 5, A := 2a^2$ expresses that a is increased by 5 and, simultaneously, variable A is set to $2a^2$, which is evaluated *before* a receives its new value $a + 5$.

Table 1. Statements and effects of hybrid programs (HPs)

HP Notation	Operation	Effect
$x_1 := \theta_1, \dots, x_n := \theta_n$	discrete jump	simultaneously assign θ_i to variables x_i
$x_1' = \theta_1, x_2' = \theta_2, \dots$ $\dots, x_n' = \theta_n \,\&\, H$	continuous evo.	differential equations for x_i within evolution domain H (first-order formula)
$?H$	state test	test first-order formula H at current state
$\alpha; \beta$	seq. composition	HP β starts after HP α finishes
$\alpha \cup \beta$	nondet. choice	choice between alternatives HP α or HP β
α^*	nondet. repetition	repeats HP α n-times for any $n \in \mathbb{N}$

Differential equation systems. Continuous evolution in the system dynamics is represented using differential equation systems as evolution constraints. For example the (second-order) differential equation $z'' = -b$ describes deceleration with braking force b and $z' = v, v' = -b \,\&\, v \geq 0$ expresses that the evolution only applies as long as the speed is $v \geq 0$. This is an evolution along the differential equation system $z' = v, v' = -b$ that is restricted (written &) to remain within the evolution domain region $v \geq 0$, i.e., to stop braking before $v < 0$. Such an evolution can stop at any time within $v \geq 0$, it could even continue with transient grazing along the border $v = 0$, but it is never allowed to enter $v < 0$. The second-order differential equation $z'' = -b$ itself is equivalent to the first-order differential equation system $z' = v, v' = -b$, in which the velocity v is explicit.

Control structure. Discrete and continuous transitions—represented as jump sets or differential equations, respectively—can be combined to form a hybrid program with interacting hybrid dynamics using regular expression operators $(\cup, {}^*, ;)$ of regular programs [19] as control structure. For example, the hybrid program $q := accel \cup z'' = -b$ describes a train controller that can choose to either switch to acceleration mode ($q := accel$) or brake by the differential equation $z'' = -b$, by a nondeterministic choice ($\cup$). The nondeterministic choice $q := accel \cup z'' = -b$ expresses that either $q := accel$ or $z'' = -b$ happens, nondeterministically. The system can choose one of the two options. The sequential composition $a := -b\,;\, z'' = a$, instead, expresses that first, the acceleration a is updated by $a := -b$, and then the system follows the differential equation $z'' = a$ with the updated acceleration (hence brakes). In conjunction with other regular combinations, control constraints can be expressed using tests like $?z \geq SB$ as guards for the system state. This test will succeed if, indeed, the current state of the system satisfies $z \geq SB$; otherwise the test will fail and execution cannot proceed. In that respect, a test is like an assert statement in conventional programs and cuts the system run if the test is not successful.

Other control structures can easily be defined from the basic operations in Table 1. See Table 2 for a list of common additional statements that can be defined [32] from those in Table 1. For instance,

$$\text{if } H \text{ then } \alpha \text{ else } \beta \equiv (?H; \alpha) \cup (?\neg H; \beta)$$

Example 1 (Natural hybrid program for simple train). As a much simplified example of a train controller, consider the following hybrid program:

$$\big(((?z < SB;\; a := A) \,\cup\, (a := -b));\;\; z' = v, v' = a \,\&\, v \geq 0\big)^* \tag{1}$$

First, the discrete controller executes and then, after the sequential composition (;), the train follows the differential equation system $z' = v, v' = a$ that is restricted to (written &) the evolution domain $v \geq 0$. The discrete controller consists of a nondeterministic choice ($\cup$) between two options. The left option

Table 2. Additional statements and control structures definable as abbreviations

HP Notation	Operation	Effect
$x := *$	nondet. assign.	assigns any real value to x
if H then α else β	if-then-else	executes HP α if H holds, otherwise HP β
if H then α	if-then	executes HP α if H holds, otherwise no effect
while H do α	while loop	repeats α if H holds, stops if $\neg H$ holds at end
repeat α until H	repeat until	repeats α (at least once) until H holds at end
skip	do nothing	no effect and does not change the state space
abort	aborts run	blocks current run and allows no transition

performs a test $(?z < SB)$ to check whether the current position is left of the start braking point SB and then, after the left-most sequential composition (;), assigns the positive acceleration A to a by $a := A$. The right option does not perform a test, but just assigns the braking force $-b$ to the acceleration a by $a := -b$. In particular, the first control option (acceleration) is only available when the train has not yet passed the start braking point SB, while the second control option (braking) is always available. The train would choose between both options nondeterministically when both are possible. Otherwise, it can only choose the options that successfully pass their respective tests (the right option in (1) is always available because it has not tests). Finally, the repetition operator $(^*)$ at the end of hybrid program (1) expresses that the controller-plant-loop can repeat indefinitely. This pattern $(ctrl; plant)^*$ is a very common use case for hybrid programs, but by far not the only useful form of a system model.

The effect of the discrete jump set $x_1 := \theta_1, \ldots, x_n := \theta_n$ is to simultaneously change the interpretations of the x_i to the respective θ_i by a discrete jump in the state space. The new values θ_i are evaluated before changing the value of any variable x_j. The effect of $x_1' = \theta_1, \ldots, x_n' = \theta_n \,\&\, H$ is an ongoing continuous evolution respecting the differential equation system $x_1' = \theta_1, \ldots, x_n' = \theta_n$ that is restricted to remain within the evolution domain region H. The evolution is allowed to stop at any point in H. It is, however, required to stop before it leaves H. For unconstrained evolutions, we write $x' = \theta$ in place of $x' = \theta \,\&\, true$.

The test action or state check $?H$ is used to define conditions. Its semantics is that of a no-op if the formula H is true in the current state; otherwise, like *abort*, it allows no transitions. That is, if the test succeeds because formula H holds in the current state, then the state does not change, and the system execution continues normally. If the test fails because formula H does not hold in the current state, then the system execution cannot even continue. Thus, the effect of a test action is similar to an *assert* statement in Java.

The nondeterministic choice $\alpha \cup \beta$, sequential composition $\alpha;\beta$, and nondeterministic repetition α^* of programs are as in regular expressions but generalised to a semantics in hybrid systems. Choices $\alpha \cup \beta$ are used to express behavioral alternatives between the transitions of α and β. That is, the hybrid program $\alpha \cup \beta$ can choose nondeterministically to follow the transitions of the hybrid program α, or, instead, to follow the transitions of the hybrid program β. The sequential composition $\alpha;\beta$ says that the hybrid program β starts executing after α has finished (β never starts if α does not terminate). In $\alpha;\beta$, the transitions of α take effect first, until α terminates (if it does), and then β continues. Repetition α^* is used to express that the hybrid process α repeats any number of times, including zero times. When following α^*, the transitions of hybrid program α can be repeated over and over again, any nondeterministic number of times ($\geq$0). Hybrid programs form a regular-expression-style Kleene algebra with tests [23].

The formal transition semantics of hybrid programs is defined in [30, 32].

2.3 Hybrid Automata

Hybrid automata are an automaton representation of hybrid systems [2, 20]. The basic idea is to have one differential equation per mode of continuous evolution of the system with an automaton structure on top that defines how and under which condition the system switches between the various modes.

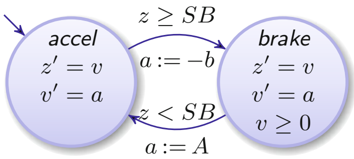
A *hybrid automaton* is a finite directed graph with a set of nodes V and a set of edges E, where

- $x_1, \ldots, x_n$ are the continuous state variables and n is the (fixed) dimension of the continuous state space.
- Each node $v \in V$ is labeled with a differential equation $x_1' = \theta_1, .., x_n' = \theta_n$ and an evolution domain constraint H, which is a quantifier-free formula of real arithmetic. The differential equation specifies how the variables $x_1, \ldots, x_n$ evolve while the system is in node v and the evolution domain constraint H has to be true all the time while in mode v.
- Each edge $e \in E$ is labeled with a guard H, which is a quantifier-free formula of real arithmetic, and a discrete jump set $x_1 := \theta_1, \ldots, x_n := \theta_n$. The guard H determines when edge e can be taken. The discrete jump set (called *reset*) $x_1 := \theta_1, \ldots, x_n := \theta_n$ determines how the variables are reassigned when the system follows edge e.

Each of the transitions of a hybrid automaton is either a discrete or a continuous transition. A continuous transition within one node is a continuous evolution along the differential equation of that node without leaving the evolution domain constraint. A discrete transition along an edge is possible if the guard H is satisfied in the current state and then the state will be reset according to the discrete jump set $x_1 := \theta_1, \ldots, x_n := \theta_n$ when following the edge. The hybrid automaton itself repeats discrete and continuous transitions indefinitely. See, e.g., [20, 32], for a formal definition of the transitions of a hybrid automaton.

We examine the relationship between hybrid programs and hybrid automata in the following example where we consider hybrid automaton and hybrid program side by side.

Example 2 (Hybrid automata versus hybrid programs). With the operations in Table 1, hybrid systems can be represented naturally as hybrid programs. For example, the right of Fig. 2 depicts a hybrid program of an (overly) simplified train control. The hybrid automaton on the left of Fig. 2 shows a corresponding hybrid automaton. Line 1 represents that, in the beginning, the current node q of the



$$
\begin{aligned}
& q := \mathit{accel}; \qquad /\!*\ \mathit{initial\ mode\ is\ node\ accel}\ */\\
& \big(\ \ (?q = \mathit{accel};\ \ z' = v, v' = a)\\
& \cup (?q = \mathit{accel} \land z \geq SB;\ a := -b;\ q := \mathit{brake};\ ?v \geq 0)\\
& \cup (?q = \mathit{brake};\ \ z' = v, v' = a \,\&\, v \geq 0)\\
& \cup (?q = \mathit{brake} \land z < SB;\ \ a := A;\ \ q := \mathit{accel})\big)^*
\end{aligned}
$$

Fig. 2. Hybrid automaton and hybrid program for a much simplified train control

system is the initial node *accel*. We represent each discrete and continuous transition of the automaton as a sequence of statements with a nondeterministic choice ($\cup$) between these transitions. Line 4 represents a continuous transition of the automaton. It tests if the current node q is *brake*, and then (i.e., if the test was successful) follows the differential equation system $z' = v, v' = a$ restricted to the evolution domain $v \geq 0$. Line 3 characterises a discrete transition of the automaton. It tests the guard $z \geq SB$ when in node *accel*, and, if successful, resets $a := -b$ and then switches q to node *brake*. By the semantics of hybrid automata [1, 20], an automaton in node *accel* is only allowed to make a transition to node *brake* if the evolution domain restriction of *brake* is true when entering the node, which is expressed by the additional test $?v \geq 0$ at the end of line 3. Observe that this test of the evolution domain region generally needs to be checked as the last operation after the guard and reset, because a reset like $v := v - 1$ could affect the outcome of the evolution domain region test. In order to obtain a fully compositional model, hybrid programs make all these implicit side conditions explicit. Line 2 represents the continuous transition when staying in node *accel* and following the differential equation system $z' = v, v' = a$. Line 5 represents the discrete transition from node *brake* of the automaton to node *accel*.

Lines 2–5 cannot be executed unless their tests succeed. In particular, at any state, the nondeterministic choice ($\cup$) among lines 2–5 reduces de facto to a nondeterministic choice between either lines 2–3 or between lines 4–5. At any state, q can have value either *accel* or *brake* (assuming these are different constants), not both. Consequently, when $q = \mathit{brake}$, a nondeterministic choice of lines 2–3 would immediately fail the tests in the beginning and not execute any further. The only remaining choices that have a chance to succeed are lines 4–5 then. In fact, only the single successful choice of line 4 would remain if the second conjunct $z < SB$ of the test in line 5 does not hold for the current state. Note that, still, all four choices in lines 2–5 are available, but at least two of these nondeterministic choices will always be unsuccessful. Finally, the repetition operator (*) at the end of Fig. 2 expresses that the transitions of a hybrid automaton, as represented by lines 2–5, can repeat indefinitely, possibly taking different nondeterministic choices between lines 2–5 at every repetition.

The hybrid program on the right of Fig. 2 directly corresponds to the hybrid automaton on the left of Fig. 2. This translation is simple and systematic. The same translation principle works for all hybrid automata and can represent them faithfully as hybrid programs [32], just like finite automata can be implemented in a conventional while-programming language. This direct translation, however, blows up the representation. A much more natural hybrid program can usually be found when directly representing the hybrid system as a hybrid program right away. More natural representations also have computational advantages for verification. This is the preferred way for designing systems.

Example 3 (Natural hybrid program corresponding to Fig. 2). The natural hybrid program corresponding to the system in Fig. 2 is the following hybrid program:

$$\big((\texttt{if}(z \geq SB)\, a := -b\, \texttt{else}\, a := A);\quad z' = v, v' = a\,\&\, v \geq 0\big)^* \tag{2}$$

This hybrid program is almost identical to that in (1), except that it has an extra test specifying that the braking option can only be chosen if the position z is after the start braking point SB. Contrast the natural hybrid program in (2) with the hybrid program on the right of Fig. 2 that has been constructed from a hybrid automaton. The natural hybrid program has the same behavior as the hybrid automaton and its corresponding hybrid program, but the natural hybrid program in (2) is significantly easier to understand and also simplifies verification. Finally, the natural hybrid program in (2) is the same as the following hybrid program when resolving abbreviations according to Table 2.

$$\big(((?z \geq SB;\ a := -b) \cup (?z < SB;\ a := A));\quad z' = v, v' = a\,\&\, v \geq 0\big)^*$$

This representational flexibility gives hybrid programs an edge over hybrid automata. The same system can be represented in many ways and a representation that is most natural to a problem often makes the verification easier. It should be noted that there is more than one hybrid automaton describing the same hybrid system, too. Nevertheless, the representation of hybrid systems as hybrid programs is more flexible, because discrete, continuous, and switching dynamics are not restricted to a specific pattern, but can be combined freely using regular expression style operators.

3 Logic for Hybrid Systems

Hybrid programs are a flexible behavioral model for hybrid systems. As a specification and verification language for hybrid systems, we have introduced the *differential dynamic logic* $\mathsf{d}\mathcal{L}$ [29, 30, 32]. In $\mathsf{d}\mathcal{L}$, operational models of hybrid systems are internalized as first-class citizens, so that correctness statements about the transition behavior of hybrid systems can be expressed as formulas. That is, correctness statements about systems can be combined into bigger formulas with arbitrary propositional operators or quantifiers, and even into nestings of formulas. As a basis, $\mathsf{d}\mathcal{L}$ includes (nonlinear) real arithmetic for describing concepts like safe regions of the state space. Further, $\mathsf{d}\mathcal{L}$ supports real-valued quantifiers for quantifying over the possible values of system parameters or durations of continuous evolutions. For talking about the transition behavior of hybrid systems, $\mathsf{d}\mathcal{L}$ provides modal operators such as $[\alpha]$ or $\langle\alpha\rangle$ that refer to the states reachable by following the transitions of hybrid program α. The logical operators of $\mathsf{d}\mathcal{L}$ are summarized in Table 3.

Table 3. Operators of differential dynamic logic for hybrid systems (d$\mathcal{L}$)

Notation	Operator	Meaning
$\theta_1 = \theta_2$	equality	value of θ_1 is equal to that of θ_2
$\theta_1 \geq \theta_2$	comparison	value of θ_1 is greater or equal that of θ_2
$\theta_1 > \theta_2$	comparison	value of θ_1 is greater than that of θ_2
$\theta_1 \leq \theta_2$	comparison	value of θ_1 is less or equal that of θ_2
$\theta_1 < \theta_2$	comparison	value of θ_1 is less than that of θ_2
$\neg\phi$	negation/not	true if ϕ is false
$\phi \land \psi$	conjunction/and	true if both ϕ and ψ are true
$\phi \lor \psi$	disjunction/or	true if ϕ is true or if ψ is true
$\phi \rightarrow \psi$	implication	true if ϕ is false or ψ is true
$\phi \leftrightarrow \psi$	equivalence	true if ϕ and ψ are both true or both false
$\forall x\, \phi$	for all quantifier	true if ϕ is true for all values of variable x
$\exists x\, \phi$	exists quantifier	true if ϕ is true for some values of variable x
$[\alpha]\phi$	$[\cdot]$ modality	true if ϕ true after all runs of HP α
$\langle\alpha\rangle\phi$	$\langle\cdot\rangle$ modality	true if ϕ true after at least one run of HP α

Within a single specification and verification language, d$\mathcal{L}$ combines operational system models with means to talk about the states that are reachable by system transitions. The logic d$\mathcal{L}$ provides parametrized modal operators $[\alpha]$ and $\langle\alpha\rangle$ that refer to the states reachable by hybrid program α and can be placed in front of any formula. The formula $[\alpha]\phi$ expresses that all states reachable by hybrid program α satisfy formula ϕ. Likewise, $\langle\alpha\rangle\phi$ expresses that there is at least one state reachable by α for which ϕ holds. These modalities can be used to express necessary or possible properties of the transition behavior of α in a natural way. They can be nested or combined propositionally. The d$\mathcal{L}$ logic supports quantifiers like $\exists p\, [\alpha]\langle\beta\rangle\phi$ which says that there is a choice of parameter p (expressed by $\exists p$) such that for all possible behaviors of hybrid program α (expressed by $[\alpha]$) there is a reaction of hybrid program β (i.e., $\langle\beta\rangle$) that ensures ϕ. Likewise, $\exists p\, ([\alpha]\phi \land [\beta]\psi)$ says that there is a choice of parameter p that makes both $[\alpha]\phi$ and $[\beta]\psi$ true, simultaneously, i.e., that makes the conjunction $[\alpha]\phi \land [\beta]\psi$ true, saying that formula ϕ holds for all states reachable by α executions and, independently, ψ holds after all β executions. This gives a flexible logic for specifying and verifying even sophisticated properties of hybrid systems, including the ability to refer to multiple hybrid systems at once.

The semantics of differential dynamic logic and more details about it can be found in [30, 32].

Example 4 (Safety in train control). Let *train* denote the hybrid program for the simple train control dynamics in (2). Consider the following d$\mathcal{L}$ formula

$$v \geq 0 \land z < m \;\rightarrow\; [\mathit{train}]z < m \tag{3}$$

It expresses that, when the system starts in an initial state where $v \geq 0 \land z < m$ is true, i.e., with nonnegative velocity and with a train position z within the movement authority limits m, then, when following the dynamics of the hybrid program *train*, then the system will always be a in a state where $z < m$ is true.

It turns out that formula (3) is a bit naive and needs additional assumptions on the parameters to be valid. For instance, the train will not remain safe, even if it starts safely within $z < m$ if its initial velocity is so high that it cannot brake in time before leaving $z < m$. Similarly, the start braking point parameter SB in (2) needs to be chosen carefully to ensure that (3) is valid. But under corresponding additional constraints, the following d$\mathcal{L}$ formula can be proven to be valid, i.e., true under all interpretations for all the variables and parameters:

$$\begin{aligned}&v^2 < 2b(m-z) \land b > 0 \land A \geq 0 \;\rightarrow\\ &\quad\big[\big(SB := m - \frac{v^2}{2b} - (\frac{A}{b}+1)(\frac{A}{2}\varepsilon^2 + \varepsilon v);\;\; \mathtt{if}(z \geq SB)\, a := -b\, \mathtt{else}\, a := A;\\ &\qquad\qquad t := 0;\; z' = v, v' = a, t' = 1 \,\&\, v \geq 0 \land t \leq \varepsilon\big)^*\big]\,(z < m) \quad (4)\end{aligned}$$

Variable t is a clock that evolves by $t' = 1$. The bound $t \leq \varepsilon$ gives an upper bound on the time of the continuous evolution until the discrete controllers have a chance to react to situation changes again. See [32, 40] for details.

4 Compositional Deductive Verification

The verification problem for hybrid systems is a very challenging problem. It is not even semidecidable numerically [36]. In the fully symbolic domain of differential dynamic logic, however, we can do better. There is a sound compositional proof system that works fully symbolically [30, 32]. It can be used to prove interesting properties of hybrid programs including safety, reactivity, controllability, and liveness. The fact that this proof system is compositional is also important for scalability purposes. Because it proves properties of complex hybrid systems by reducing them to properties about simpler systems, this compositional verification approach can scale to complex systems.

Furthermore, the proof system is a *complete axiomatization of hybrid systems relative to differential equations* [30]. That is, every true statement about a hybrid system can be proven from elementary properties of differential equations.

Theorem 1 (Relative completeness [30]). *Hybrid systems can be axiomatized completely relative to differential equations.*

The proof of this theorem is a tricky 15page proof, but the theorem has important consequences. It proves that all true properties of hybrid systems can be decomposed successfully into properties of their parts. This theorem also explains the practical verification successes with this approach in air traffic [38] and railway control [40] and shows that other systems can be verified with the approach. The reason is that the decomposition of the entire verification problem into elementary properties of the dynamical aspects makes the verification problem tractable.

Theorem 1 has another important consequence. It gives a formal reason why the handling of differential equations is at the heart of hybrid systems verification. Moreover, the proof calculus in [30, 32] completely lifts every verification technique for differential equations to a verification technique for hybrid systems. In general, it may not always be clear how verification techniques for continuous systems generalize to hybrid systems. But Theorem 1 gives a formal proof showing *that* and *how* to generalize any verification technique for differential equations to full hybrid systems completely.

A prime example of such advanced and powerful verification techniques are *differential invariants* for differential equations [31]. Differential invariants have been instrumental in enabling the verification of complex hybrid systems, including air traffic control [38], train control with disturbances in the dynamics [40], and electrical circuits [32]. Differential invariants turn the following intuition into a formally sound proof procedure. If the vector field of the differential equation always points into a direction where the differential invariant F, which is a logical formula, is becoming "more true" (see Fig. 3), then the system will always stay save if it initially starts save. This principle can be understood in a simple but formally sound way using the logic d$\mathcal{L}$ [31, 32]. Differential invariants have been introduced in [31] and later refined to a procedure that computes differential invariants in a fixed-point loop [37].

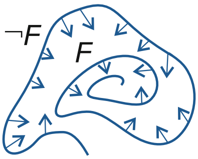


Fig. 3. Differential invariant F

5 Verification Tool KeYmaera

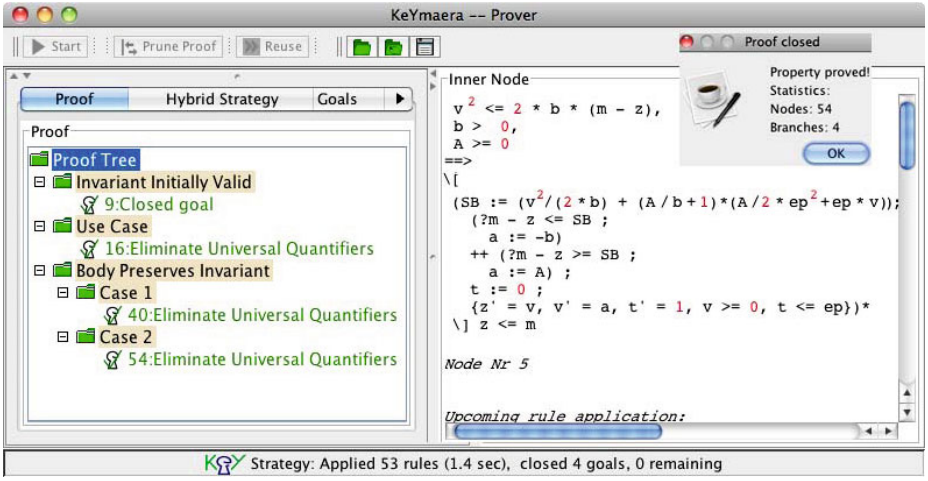
The approach surveyed in this tutorial is implemented in KeYmaera[3], which is a hybrid verification tool for hybrid systems. KeYmaera has a very powerful graphical user interface for conducting proofs and for looking at the proofs that KeYmaera found automatically; see Fig. 4.

This user interface is based on that of the prover KeY [4], from which KeYmaera also inherits its name[4]. KeYmaera has powerful automatic proof procedures that have been used to prove a number of interesting collision avoidance properties in systems including air traffic control and railway control fully automatically [37]. These automation procedures and fixedpoint loops for generating invariants and differential invariants are described in detail in [32, 37].

Nevertheless, the possibility of interacting with KeYmaera can be extremely powerful for verifying complex systems that cannot be handled automatically by any verification tool yet. A good practice for complex physical systems is to combine automatic proof search in KeYmaera with selective user guidance after inspecting the intermediate stage of a partial proof that KeYmaera found in its graphical user interface. KeYmaera also supports annotations such as `@invariant(F)` and `@candidate(F,G)` to annotate problems with possible proof hints about invariant and/or differential invariant formulas `F,G` that could help KeYmaera in finding computationally difficult proofs.

[3] `http://symbolaris.com/info/KeYmaera.html`

[4] KeYmaera is pronounced similar to the hybrid Chimaera from Greek mythology.



Proof Tree
- Invariant Initially Valid
  - 9:Closed goal
- Use Case
  - 16:Eliminate Universal Quantifiers
- Body Preserves Invariant
  - Case 1
    - 40:Eliminate Universal Quantifiers
  - Case 2
    - 54:Eliminate Universal Quantifiers

```
 v^2 <= 2 * b * (m - z),
 b >  0,
 A >= 0
==>
\[
 (SB := (v^2/(2 * b) + (A / b + 1)*(A / 2 * ep^2 + ep * v));
   (?m - z <= SB ;
     a := -b)
   ++ (?m - z >= SB ;
     a := A) ;
   t := 0 ;
   {z' = v, v' = a, t' = 1, v >= 0, t <= ep})*
 \] z <= m

Node Nr 5


Upcoming rule application:
```

Property proved!
Statistics:
Nodes: 54
Branches: 4

Strategy: Applied 53 rules (1.4 sec), closed 4 goals, 0 remaining

Fig. 4. KeYmaera verification tool for hybrid systems

```
\problem {
   \[ R ep,b,A, SB, a, v, z, t, m; \] (
      v^2 < 2*b*(m-z) & b > 0 & A>=0
  ->
     \[(
      SB := m - (v^2)/(2*b) - ((A/b) + 1) * ((A/2)*ep^2 + ep*v);
      if (z >= SB) then
            a := -b
      else
            a := A
      fi;
      t:=0; {z'=v, v'=a, t'=1, (v >= 0 & t <= ep)}
      )*@invariant(2*b*(m-z)-v^2>0)  // loop @annotation optional
     \] (z < m)
   )
}
```

Fig. 5. KeYmaera problem description for simple train control

The KeYmaera notation for the d$\mathcal{L}$ formula (4) is shown in Fig. 5. The second line declares the variables ep,b,A, SB, a, v, z, t, m of type real. The annotation @invariant(2∗b∗(m−z)−vˆ2>0) gives a proof hint that KeYmaera should use $2b(m-z) - v^2 > 0$ as a loop invariant. This proof hint is unnecessary, because KeYmaera will automatically discover an invariant that proves the formula in Fig. 5 anyhow.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138(1), 3–34 (1995)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, et al. (eds.) [18], pp. 209–229
3. Alur, R., Sontag, E.D., Henzinger, T.A. (eds.): HS 1995. LNCS, vol. 1066. Springer, Heidelberg (1996)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
5. van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schiffelers, R.R.H.: Syntax and consistent equation semantics of hybrid Chi. J. Log. Algebr. Program. 68(1-2), 129–210 (2006)
6. van Beek, D.A., Reniers, M.A., Schiffelers, R.R.H., Rooda, J.E.: Concrete syntax and semantics of the compositional interchange format for hybrid systems. In: 17th IFAC World Congress (2008)
7. Bergstra, J.A., Middelburg, C.A.: Process algebra for hybrid systems. Theor. Comput. Sci. 335(2-3), 215–280 (2005)
8. Branicky, M.S.: General hybrid dynamical systems: Modeling, analysis, and control. In: Alur, et al. (eds.) [3], pp. 186–200
9. Branicky, M.S.: Studies in Hybrid Systems: Modeling, Analysis, and Control. Ph.D. thesis, Dept. Elec. Eng. and Computer Sci. Massachusetts Inst. Technol. Cambridge, MA (1995)
10. Branicky, M.S., Borkar, V.S., Mitter, S.K.: A unified framework for hybrid control: Model and optimal control theory. IEEE T. Automat. Contr. 43(1), 31–45 (1998)
11. Cassez, F., Larsen, K.G.: The impressive power of stopwatches. In: CONCUR, pp. 138–152 (2000)
12. Chaochen, Z., Ji, W., Ravn, A.P.: A formal description of hybrid systems. In: Alur, et al. (eds.) [3], pp. 511–530
13. Chutinan, A., Krogh, B.H.: Computational techniques for hybrid system verification. IEEE T. Automat. Contr. 48(1), 64–75 (2003)
14. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B.H., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. 14(4), 583–604 (2003)
15. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. J. Log. Algebr. Program. 62(2), 191–245 (2005)
16. Davoren, J.M., Nerode, A.: Logics for hybrid systems, vol. 88(7), pp. 985–1010. IEEE, Los Alamitos (2000)
17. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT 10(3), 263–279 (2008)
18. Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.): Hybrid Systems, LNCS, vol. 736. Springer (1993)
19. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. MIT Press, Cambridge (2000)
20. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292. IEEE Computer Society, Los Alamitos (1996)

21. Jifeng, H.: From CSP to hybrid systems. In: Roscoe, A.W. (ed.) A classical mind: essays in honour of C. A. R. Hoare, pp. 171–189. Prentice Hall, Hertfordshire (1994)
22. Kesten, Y., Manna, Z., Pnueli, A.: Verification of clocked and hybrid systems. Acta Inf. 36(11), 837–912 (2000)
23. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
24. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 305–318. Springer, Heidelberg (1998)
25. Mitchell, I., Bayen, A.M., Tomlin, C.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE T. Automat. Contr. 50(7), 947–957 (2005)
26. Mysore, V., Piazza, C., Mishra, B.: Algorithmic algebraic model checking II: Decidability of semi-algebraic model checking and its applications to systems biology. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 217–233. Springer, Heidelberg (2005)
27. Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: An approach to the description and analysis of hybrid systems. In: Grossman, et al. (eds.) [18], pp. 149–178
28. Perko, L.: Differential equations and dynamical systems. Springer, New York (1991)
29. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 216–232. Springer, Heidelberg (2007)
30. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. 41(2), 143–189 (2008)
31. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
32. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
33. Platzer, A.: Quantified differential dynamic logic for distributed hybrid systems. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 469–483. Springer, Heidelberg (2010)
34. Platzer, A.: Quantified differential invariants. In: Frazzoli, E., Grosu, R. (eds.) HSCC, pp. 63–72. ACM Press, New York (2011)
35. Platzer, A.: Stochastic differential dynamic logic for stochastic hybrid programs. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE. LNCS. Springer, Heidelberg (2011)
36. Platzer, A., Clarke, E.M.: The image computation problem in hybrid systems model checking. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 473–486. Springer, Heidelberg (2007)
37. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
38. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
39. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)

40. Platzer, A., Quesel, J.-D.: European Train Control System: A Case Study in Formal Verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)
41. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. Trans. on Embedded Computing Sys. 6(1), 8 (2007)
42. Rönkkö, M., Ravn, A.P., Sere, K.: Hybrid action systems. Theor. Comput. Sci. 290(1), 937–973 (2003)
43. Tavernini, L.: Differential automata and their discrete simulators. Non-Linear Anal. 11(6), 665–683 (1987)

Using Coverage to Deploy Formal Verification in a Simulation World

Vigyan Singhal and Prashant Aggarwal

Oski Technology, Inc.

Abstract. Formal verification technology has today advanced to the stage that it can complement or replace simulation effort for selected hardware designs. Yet the completion of a formal verification effort is rarely a requirement for hardware tapeout. Simulation remains the primary verification methodology, and means of deciding when verification is complete. In this paper we discuss how formal verification can be deployed using simulation-based coverage in a simulation-based verification schedule.

1 Introduction

Application-Specific Integrated Circuits (ASICs) enable faster, better, lower-power products, both for consumer as well as enterprise markets. Competition is fierce. Product lifetimes are shrinking. Meeting project schedule and delivering first-pass silicon, is more important than ever before. Design verification consumes the largest slice of overall design effort in terms of human resources, as much as 60-70 percent [1]. Simulation remains the primary method of pre-silicon verification.

Since its introduction about 20 years ago, formal verification has become increasingly more relevant to a diverse array of applications, widening from custom processor designs to general-purpose ASICs. As with any other new technology, the true test of adoption happens with integration, in this case when formal verification becomes a signoff for chip tapeouts. We are not quite there yet, but that level of adoption is beginning to happen. For formal to become a signoff requirement for tapeout, we should be able to (a) plan for formal verification [2]; and (b) quantitatively measure the results of formal verification, and integrate those with simulation-based measurements. While multiple formal verification methodologies are in practice in the ASIC industry today (e.g., model checking, theorem proving, C-vs-RTL sequential checking), model checking accounts for practically most of the usage, with greater numbers of commercial tools, verification users, as well as licenses in use. Furthermore, major EDA vendors (Cadence, Mentor and Synopsys), as well as a few startups (Jasper, OneSpin and Real Intent) offer competitive solutions. So in this paper, we will focus on the usage of model checking as a formal verification methodology.

The question of when simulation is complete, is often answered using coverage metrics [3,4]. The adoption of formal verification in an ASIC design schedule

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 44–49, 2011.

requires formal verification coverage results, using the same metrics as simulation. Several creditable efforts have defined notions of coverage for formal [5,6]. Simulation-based metrics have also been adopted by formal [7]. The key difference in the methodology described in this paper, is that it is based on the practical impossibility of fully and formally verifying all blocks, within typical schedule and resource constraints. Yet, it is imperative that formal verification efforts yield metrics that are integrated with simulation metrics. We will describe a coverage-based methodology that achieves this in practice.

2 ASIC Verification Process

Verification is an essential part of the ASIC design cycle. Undetected bugs in any design cause loss of revenue due to delay in time-to-market, and increased non-recurring engineering cost due to additional re-spins. Well-crafted planning is key for faster verification closure. The selection of the correct verification methods and coverage metrics helps in achieving verification closure faster, and better. The design specification is an input to the RTL implementation (in an HDL), as well as a verification plan (Fig. 1). The verification plan includes choices for the right verification approaches (e.g., formal and simulation), and is based on the scope, required schedule and available resources for each approach. Verification engineers implement their verification setup based on this plan. The verification setup is executed and data is collected for simulation and formal verification runs. The data helps quantify whether planned goals are met or not.

The ASIC management team uses this quantitative data, often in the form of a dynamic verification dashboard (Fig. 2), to track the progress of verification,

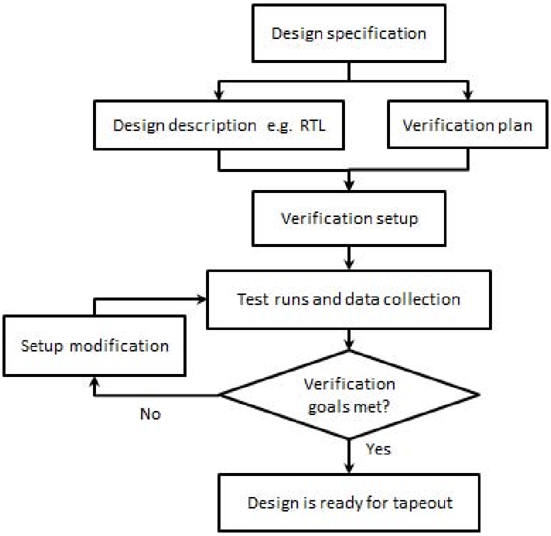


Fig. 1. ASIC verification flow

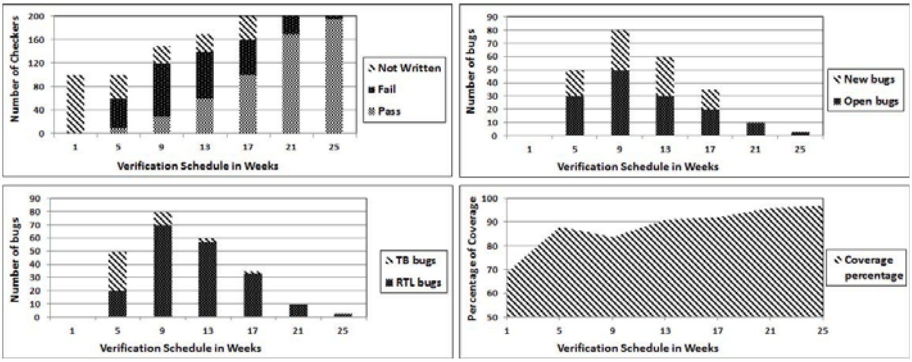


Fig. 2. A verification dashboard

and take timely corrective action. Coverage metrics form an integral part of this progress management.

3 Coverage Metrics in Simulation

A simulation testbench setup consists of the following components [8]:

- Bus Functional Models (BFMs), that contain methods used to drive inputs
- Checkers, that monitor inputs, outputs and internals of a design, and flag errors (these checkers could be in-line assertions in the RTL code, external monitors, scoreboard, or even C reference models)
- Tests, including biases, that control the inputs driven by the BFMs

Coverage is the primary tool used to determine when simulation is complete. The most common coverage metric is code coverage, and line coverage is the most widely used code coverage metric. Line coverage (similar to statement coverage for software testing [9]) computes what percentage of RTL statements were exercised by a given set of tests. For example, consider:

```
1:  always @(posedge clk) begin
2:    if ((a && b) || c)
3:      e <= d1;
4:    else
5:      e <= d2;
6:  end
```

This example results in two line coverage targets, corresponding to lines 3 and 5. If a test causes c to be 1, the line 3 will be marked as covered. If no test in a test suite covers line 5, line coverage for the suite will be reported at 50%. Other variants of code coverage, like expression, branch, toggle, and FSM coverage, are also in use, but are less popular.

Unlike software testing, 100% *judged* line coverage (given, say 99% *automated* coverage) is frequently a requirement for an ASIC tapeout – each line that is not automatically reported as covered in simulation, must be manually judged to either redundant or legacy code, or symmetric to another tested line. Tapeout would be delayed until more tests are written to cover the remaining lines. This is typically a higher bar compared to what typical software product releases are exposed to. In spite of the 100% line coverage requirement, hardware designs are taped out and shipped with bugs, some known and some unknown. Often, meeting the schedule is more important than implementing all the product features that were originally planned. And beating the competition can be even more important. One known limitation of line coverage is that it ignores the observability, or measures the quality of the checkers; we will revisit this in Section 5.

Besides code coverage, functional coverage is also beginning to gain acceptance in the ASIC world [8]. Due to space limitations, we limit this discussion to code coverage.

4 Measuring Coverage with Formal Verification

A formal verification (model checking) environment consists of the following:

- a set of constraints
- a set of checkers (or assertions)
- an optional set of manual abstractions, used to reduce the complexity of formal verification (examples of abstraction methods are data independence abstraction [10] or counter abstraction [11])

The same coverage metrics used in simulation can be applied to answer the question of whether the planned formal verification tasks are complete, or how much the formal verification tasks complement the simulation effort.

Before addressing this topic, we must counter a prevalent myth in formal literature: *whether or not bugs are missed in a model checking effort depends solely on how complete the set of checkers is.* In practice, this does not hold true – there are other important sources of missed bugs. The most useful model checking efforts end up using bounded model checking (BMC). Furthermore, for reasons of schedule or complexity, the use of (over-)constraints is also very common [2]. So bugs may also be missed because the BMC runs did not go far enough; or, because a constraint was an intentional or unintentional over-constraint. BMC is so popular in practice partly because formal verification technology is today able to solve end-to-end verification for complex blocks, especially with the use of manual abstractions.

Given this reality, the simulation-based line coverage metric can be used to mean exactly the same in formal – provided that the constraints used and the bounds reached in BMC (say, n cycles), report what percentage of line targets are reachable in n cycles. For the example in Section 3, if `((a && b) || c)`, in line 2, is reachable in n cycles, this line would be reported as covered, and otherwise, not. Thus, line coverage numbers would mean the same in simulation – whether a certain coverage target is exercised or not. And for formal, this would measure the quality of constraints, as well as the proof bounds reached, perhaps with abstractions. Commercial formal tools are beginning to compute this notion of coverage.

Since we are using the same coverage metrics, we can even merge coverage results. It is often the case that one block is verified end-to-end with formal, and a larger block containing this block is verified with simulation. Even if the line coverage with formal is not 100% for the block, as long as the unified simulation and formal line coverage is 100%, verification is considered complete from the perspective of line coverage goals. This of course relies on an important assumption – *that the set of formal checkers is as complete as the set of simulation checkers.* In the next section, we will visit the issue of quantifying the completeness of checkers, both for simulation and formal.

The methodology for integrating formal code coverage results with simulation, can be easily extended to other types of code coverage.

5 Mutation Coverage

(Strong) mutation testing applies a set of mutation operators to a design, one at a time, and determines what fraction of mutated designs (called mutants) are exposed by the verification process [12]. Mutation testing can result in a code coverage metric (e.g., line coverage), by defining a different mutant for every line of the design, resulting from changing that line. The assignment in every line can, for example, be replaced by an X-assignment, followed by an application of the standard X-propagation-based simulation methodology [13]. For the example in Section 3, the mutant for line 3 may look like this:

```
1:  always @(posedge clk) begin
2:    if ((a && b) || c)
3:      e <= 1'bX;
4:    else
5:      e <= d2;
6:  end
```

Mutation line coverage is defined as the percent of lines whose mutants will cause some verification test to fail. For example, if the mutant for line 3 is exposed by verification, but the mutant for line 5 is not, we would claim 50% mutation line coverage.

Clearly, by adding the notion of observability, mutation line coverage becomes strictly more useful than (weak) line coverage described in previous sections. It is accepted in the ASIC verification industry that the latter does not measure the completeness of the checkers, and is a known verification hole. However, mutation coverage is also more expensive to compute.

Along the same lines, model checking tools should be able to implement a feature that measures mutant code coverage, to assess the completeness of the formal checkers. For the example code shown above, the line mutant will be achieved by replacing the assignment to e in line 3, by an assignment to new primary input. Of course this will not be an easy metric to compute, given the lines of code in a design. We leave this an important problem that needs to be solved in the future.

In simulation, mutant coverage is often approximated by giving the mutant testing tool a fixed amount of computation resources, and reporting a lower bound on the coverage. The same approach can be adopted for formal. Once we have metrics for both formal and simulation, the results could be combined, and incorporated into the verification dashboard (Fig. 2).

6 Conclusion

An optimal verification flow deploys both simulation and formal verification approaches. The choice of common coverage metrics for formal and simulation enables ASIC management teams to integrate formal to complete a comprehensive, planned verification effort.

References

1. Chayut, I.: Functional verification from a manager's perspective. Synopsys Insight, 1(2) (2006), http://www.synopsys.com/news/pubs/insight/2006/art1_verinvidia_v1s2.html
2. Foster, H., Loh, L., Rabii, B., Singhal, V.: Guidelines for creating a formal verification testplan. In: Proc. DVCon (2006)
3. Kantrowitz, M., Noack, L.M.: I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In: Proc. Design Automation Conf. pp. 325–330 (1996)
4. Tasiran, S., Keutzer, K.: Coverage metrics for functional validation of hardware designs. IEEE Des. Test. 18(4), 36–45 (2001)
5. Hoskote, Y.V., Kam, T., Ho, P.-H., Zhao, X.: Coverage estimation for symbolic model checking. In: Proc. Design Automation Conf. pp. 300–304 (1996)
6. Katz, S., Grumberg, O., Geist, D.: Have I written enough properties? - A method of comparison between specification and implementation. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 280–297. Springer, Heidelberg (1999)
7. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for formal verification. J. Software Tools Technology Transfer 8(4), 373–386 (2006)
8. Bergeron, J.: Writing testbenches using SystemVerilog. Springer, Heidelberg (2006)
9. Myers, G.J.: The art of software testing, 2nd edn. Wiley, Chichester (2004)
10. Wolper, P.: Expressing interesting properties of programs in propositional logic. In: Proc. Symp. Principles Programming Lang POPL, pp. 184–192 (1986)
11. Datta, A., Singhal, V.: Formal Verification of a public-domain DDR2 controller design. In: Proc. VLSI Design, pp. 475–480 (2008)
12. Hampton, M., Petithomme, S.: Leveraging a commercial mutation analysis tool for research. In: Proc. Workshop on Mutation Analysis, pp. 203–209 (2007)
13. Bening, L.C.: Accurate simulation of high speed computer logic. In: Proc. Design Automation Conf. pp. 103–112 (1969)

Stability in Weak Memory Models

Jade Alglave[1,2] and Luc Maranget[2]

[1] Oxford University
[2] INRIA

Abstract. Concurrent programs running on weak memory models exhibit relaxed behaviours, making them hard to understand and to debug. To use standard verification techniques on such programs, we can force them to behave as if running on a Sequentially Consistent (SC) model. Thus, we examine how to constrain the behaviour of such programs via synchronisation to ensure what we call their stability, i.e. that they behave as if they were running on a stronger model than the actual one, e.g. SC. First, we define sufficient conditions ensuring stability to a program, and show that Power's locks and read-modify-write primitives meet them. Second, we minimise the amount of required synchronisation by characterising which parts of a given execution should be synchronised. Third, we characterise the programs stable from a weak architecture to SC. Finally, we present our offence tool which places either lock-based or lock-free synchronisation in a x86 or Power program to ensure its stability.

Concurrent programs running on modern multiprocessors exhibit subtle behaviours, making them hard to understand and to debug: modern architectures ($e.g.$ x86 or Power) provide *weak memory models*, allowing optimisations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [2]. Thus an execution of a program may not be an interleaving of its instructions, as it would be on a Sequentially Consistent (SC) architecture [18]. Hence standard analyses for concurrent programs might be unsound, as noted by M. Rinard in [25]. Memory model aware verification tools exist, $e.g.$ [24,11,15,30], but they often focus on one model at a time, or cannot handle the write atomicity relaxation exhibited $e.g.$ by Power: generality remains a challenge.

Fortunately, we can force a program running on a weak architecture to behave as if it were running on a stronger one ($e.g.$ SC) by using *synchronisation primitives*; this underlies the *data race free guarantee* (DRF guarantee) of S. Adve and M. Hill [3].

Hence, as observed $e.g.$ by S. Burckhart and M. Musuvathi in [12], *"we can sensibly verify the relaxed executions [. . .] by solving the following two verification problems separately: 1. Use standard verification methodology for concurrent programs to show that the [SC] executions [. . .] are correct. 2. Use specialized methodology for* memory model safety *verification"*. Here, *memory model safety* means checking that the executions of a program, although running on a weak architecture, are actually SC. To apply standard verification techniques to concurrent programs running on weak memory models, we thus first need to ensure that our programs have a SC behaviour. S. Burckhart and M. Musuvathi focus in [12] on the *Total Store Order* (TSO) [28] memory model. We generalise their idea to a wider class of models (defined in [5], and recalled in Sec. 1): we examine how to force a program running on a weak architecture A_1 to behave as if running on a stronger one A_2, a property that we call *stability from* A_1 *to* A_2.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 50–66, 2011.

To ensure stability to a program, we examine the problem of placing *lock-based* or *lock-free* synchronisation primitives in a program. We call *synchronisation mapping* an insertion of synchronisation primitives (either *barriers* (or *fences*), *read-modify-writes*, or *locks*) in a program. We study whether a given synchronisation mapping ensures stability to a program running on a weak memory model, $e.g.$ that we placed enough primitives in the code to ensure that it only has SC executions. D. Shasha and M. Snir proposed in [27] the *delay set analysis* to insert barriers in a program, but their work does not provide any semantics for weak memory models. Hence questions remain $w.r.t.$ the adequacy of their method in the context of such models.

On the contrary, locks allow the programmer to ignore the details of the memory model, but are costly from a compilation point of view. As noted by S. Adve and H.-J. Boehm in [4], *"on hardware that relaxes write atomicity [$e.g.$ Power], it is often unclear that more efficient mappings (than the use of locks) are possible; even the fully fenced implementation may not be sequentially consistent."* Hence not only do we need to examine the *soundness* of our synchronisation mappings ($i.e.$ that they ensure stability to a program), but also their cost. Thus, we present several new contributions:

1. We define in Sec. 2 sufficient conditions on synchronisation to ensure stability to a program. As an illustration, we provide in Sec. 3 semantics to the locks and read-modify-writes (rmw) of the Power architecture [1] ($i.e.$ to the `lwarx` and `stwcx.` instructions) and show in Coq that they meet these conditions.
2. We propose along the way several synchronisation mappings, which we prove in Coq to enforce a SC behaviour to an x86 or Power program.
3. We optimise these mappings by generalising in Sec. 4 the approach of [27] to weak memory models and both lock-based and lock-free synchronisation, and characterise in Coq the executions stable from a weak architecture to SC.
4. We describe in Sec. 5 our new offence tool, which places either lock-based or lock-free synchronisation in a x86 or Power assembly program to ensure its stability, following the aforementioned characterisation. We detail how we used offence to test and measure the cost of our synchronisation mappings.

We formalised our results in Coq; we omit the proofs for brevity. A long version with proofs, the Coq development, the documentation and sources of offence and the experimental details can be found at `http://offence.inria.fr`.

1 Context

We give here the background on which we build our results. This section summarises our previous generic model [5], which embraces SC [18], Sun TSO, PSO and RMO [28], Alpha [7] and a fragment of Power [1]. Fig. 1 shows a table of our relations. The **iriw** test [10] (independent reads of independent writes), in Fig. 2, is our running example.

Executions. An *event* e is a read or a write, composed of a direction R (read) or W (write), a location $\mathrm{loc}(e)$, the instruction from which it comes $\mathrm{ins}(e)$, a value $\mathrm{val}(e)$, a processor $\mathrm{proc}(e)$, and a unique identifier. We represent each instruction by the events it issues. In Fig. 2, we associate the store $(e)\ x \leftarrow 1$ on P_2 with the event $(e)\mathrm{W}x1$. We write $\mathbb{E}$ for the set of events, and $\mathbb{W}$ (resp. $\mathbb{R}$) for the subset of write (resp. read) events. We write w (resp. r) for a write (resp. read), and m or e when the direction is irrelevant.

Name	Notation	Comment
program order	$m_1 \xrightarrow{\text{po}} m_2$	per-processor total order
preserved program order	$m_1 \xrightarrow{\text{ppo}} m_2$	pairs maintained in program order; $\xrightarrow{\text{ppo}} \subseteq \xrightarrow{\text{po}}$
read-from map	$w \xrightarrow{\text{rf}} r$	links a write to a read reading its value
write serialisation	$w_1 \xrightarrow{\text{ws}} w_2$	total order on writes to the same location
from-read map	$r \xrightarrow{\text{fr}} w$	r reads from a write preceding w in $\xrightarrow{\text{ws}}$
barriers	$m_1 \xrightarrow{\text{ab}} m_2$	ordering induced by barriers

Fig. 1. Table of relations

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x` (b) `r2` $\leftarrow$ `y`	(c) `r3` $\leftarrow$ `y` (d) `r4` $\leftarrow$ `x`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`

Observed? `r1=1`; `r2=0`; `r3=2`; `r4=0`;

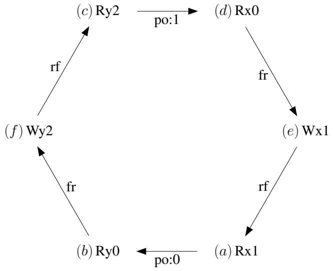


Fig. 2. The **iriw** test and a non-SC execution

We associate a program with an *event structure* $E \triangleq (\mathbb{E}, \xrightarrow{\text{po}})$, composed of its events $\mathbb{E}$ and the *program order* $\xrightarrow{\text{po}}$, a per-processor total order over $\mathbb{E}$. In Fig. 2, the read (a) from x on P_0 is in program order with the read (b) from y on P_0, $i.e.$ $(a)\text{R}x1 \xrightarrow{\text{po}} (b)\text{R}y0$. The $\xrightarrow{\text{dp}}$ relation (included in $\xrightarrow{\text{po}}$, the source being a read) models the dependencies between instructions, $e.g.$ when we compute the address of a load or store from the value of a preceding load.

Given an event structure E, we represent an execution $X \triangleq (\xrightarrow{\text{ws}}, \xrightarrow{\text{rf}})$ of the corresponding program by two relations over $\mathbb{E}$. The *write serialisation* $\xrightarrow{\text{ws}}$ is a per-location total order on writes modeling the *memory coherence* assumed by modern architectures [13], linking a write w to any write w' to the same location hitting the memory after w. The *read-from map* $\xrightarrow{\text{rf}}$ links a write w to a read r from the same location that reads from w. We derive the *from-read map* $\xrightarrow{\text{fr}}$ from $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$. A read r is in $\xrightarrow{\text{fr}}$ with a write w when the write w' from which r reads hit the memory before w did: $r \xrightarrow{\text{fr}} w \triangleq \exists w', w' \xrightarrow{\text{rf}} r \wedge w' \xrightarrow{\text{ws}} w$.

In Fig. 2, the specified outcome corresponds to the execution on the right, if each location and register initially holds 0. If `r1=1` in the end, the read (a) read its value from the write (e) on P_2, hence $(e) \xrightarrow{\text{rf}} (a)$. If `r2=0`, the read (b) read its value from the initial state, thus before the write (f) on P_3, hence $(b) \xrightarrow{\text{fr}} (f)$. Similarly, we have $(f) \xrightarrow{\text{rf}} (c)$ from `r3=2`, and $(d) \xrightarrow{\text{fr}} (e)$ from `r4=0`.

Architectures. In a shared-memory multiprocessor, a write may be committed first into a store buffer, then into a cache, and finally into memory. Hence, while a write transits in store buffers and caches, a processor may read a past value.

Code	Comment	Doc
`mfence`	WR non-cumulative barrier	[16, p. 291]
`cmp;bne;isync`	this sequence forms a RW, RR non-cumulative barrier	[1, p. 661]
`lwsync`	RW, RR, WW non-, A- and B-cumulative barrier	[1, p. 700]
`sync`	RW, RR, WW, WR non-, A- and B-cumulative barrier	[1, p. 700]

Fig. 3. Table of x86 and Power barriers

We model this by some subrelation of $\xrightarrow{\text{rf}}$ being *non-global*: they can be ignored by some processors. We write $\xrightarrow{\text{rfi}}$ (resp. $\xrightarrow{\text{rfe}}$) for the *internal* (resp. *external*) read-from map, $i.e.$ a read-from map between two events from the same (resp. distinct) processor(s). Hence we model a read r by a processor P_0 reading from a write w in P_0's store buffer by $w \xrightarrow{\text{rfi}} r$ being non-global. When r reads from a write w by a distinct processor P_1 into a cache shared by P_0 and P_1 only (a case of *write atomicity* relaxation [2]), $w \xrightarrow{\text{rfe}} r$ is non-global, and w is said to be *non-atomic*. TSO authorises $e.g.$ store buffering ($i.e.$ $\xrightarrow{\text{rfi}}$ is non-global) but considers stores to be atomic ($i.e.$ $\xrightarrow{\text{rfe}}$ is global). We write $\xrightarrow{\text{grf}}$ for the global subrelation of $\xrightarrow{\text{rf}}$. We consider $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ global, since $\xrightarrow{\text{ws}}$ is the order in which the writes to a certain location hit the memory.

Moreover, some pairs of events in the program order may be reordered. Thus only a subset of the pairs of events in $\xrightarrow{\text{po}}$, gathered in a subrelation $\xrightarrow{\text{ppo}}$ (*preserved program order*), is guaranteed to occur in this order. TSO for example authorises write-read pairs to be reordered, but nothing else: $\xrightarrow{\text{ppo}} = \xrightarrow{\text{po}} \setminus (\mathbb{W} \times \mathbb{R})$.

Finally, architectures provide barrier instructions to order certain pairs of events; Fig. 3 gives the x86 and Power ones that we use. We gather the orderings induced by barriers in the global relation $\xrightarrow{\text{ab}}$. Following [5], the relation $\xrightarrow{\text{fence}} \subseteq \xrightarrow{\text{po}}$ induced by a barrier `fence` is *non-cumulative* when it orders certain pairs of events surrounding the barrier: $\text{NC}(\xrightarrow{\text{fence}}) \triangleq (\xrightarrow{\text{fence}} \subseteq \xrightarrow{\text{ab}})$. For example, the x86 `mfence` barrier is a non-cumulative barrier ordering write-read pairs only: $(w \xrightarrow{\text{mfence}} r) \Rightarrow (w \xrightarrow{\text{ab}} r)$. If there is a *dataflow dependency*, $e.g.$ *via* a comparison `cmp`, from a read to a conditional branch ($e.g.$ `bne`), Power `isync` forms a non-cumulative barrier when placed in $\xrightarrow{\text{po}}$ after the `cmp;bne` sequence, for read-read and read-write pairs : $(r \xrightarrow{\text{cmp;bne;isync}} m) \Rightarrow (r \xrightarrow{\text{ab}} m)$.

The relation $\xrightarrow{\text{fence}}$ is *cumulative* $w.r.t.$ another relation $\xrightarrow{\text{s}} \subseteq \xrightarrow{\text{rf}}$ when it makes the writes of $\xrightarrow{\text{s}}$ atomic ($e.g.$ by flushing the store buffers and caches). Formally, we define an A-cumulative (resp. B-cumulative) barrier as $\text{AC}(\xrightarrow{\text{fence}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{s}}; \xrightarrow{\text{fence}}) \subseteq \xrightarrow{\text{ab}}$ (resp. $\text{BC}(\xrightarrow{\text{fence}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{fence}}; \xrightarrow{\text{s}}) \subseteq \xrightarrow{\text{ab}}$). For example, Power `sync` barrier is non- (resp. A- and B-) cumulative for all pairs: we have $(m_1 \xrightarrow{\text{sync}} m_2)$ (resp. $(m_1 \xrightarrow{\text{rf}} w \xrightarrow{\text{sync}} m_2)$ and $(m_1 \xrightarrow{\text{sync}} w \xrightarrow{\text{rf}} m_2)$) implies $(m_1 \xrightarrow{\text{ab}} m_2)$. Power `lwsync` is non- (resp. A- and B-) cumulative for all pairs except write-read ones; we have $(m_1 \xrightarrow{\text{lwsync}} m_2)$ (resp. $(m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{lwsync}} m_2)$ and $(m_1 \xrightarrow{\text{lwsync}} w \xrightarrow{\text{rf}} m_2)$) implies $(m_1 \xrightarrow{\text{ab}} m_2)$ if $(m_1, m_2) \notin (\mathbb{W} \times \mathbb{R})$.

An *architecture* $A \triangleq (\text{ppo}, \text{grf}, \text{ab})$ specifies the function ppo (resp. grf, ab) returning the relation $\xrightarrow{\text{ppo}}$ (resp. $\xrightarrow{\text{grf}}$, $\xrightarrow{\text{ab}}$) when given an execution.

Validity The $\mathrm{uniproc}(E, X) \triangleq \mathrm{acyclic}(\xrightarrow{\mathrm{ws}} \cup \xrightarrow{\mathrm{fr}} \cup \xrightarrow{\mathrm{rf}} \cup \xrightarrow{\text{po-loc}})$ condition (where $\xrightarrow{\text{po-loc}}$ is the program order restricted to events with the same location) forces a processor in a multiprocessor context to respect the memory *coherence* [13]. The $\mathrm{thin}(E, X) \triangleq \mathrm{acyclic}(\xrightarrow{\mathrm{rf}} \cup \xrightarrow{\mathrm{dp}})$ condition prevents executions where values seem to come *out of thin air* [21]. We define the *global happens-before* relation $A\,.\,\mathrm{ghb}(E, X)$ of an execution (E, X) on an architecture A as the union of the relations global on A:

$$A\,.\,\mathrm{ghb}(E, X) \triangleq \xrightarrow{\mathrm{ws}} \cup \xrightarrow{\mathrm{fr}} \cup \xrightarrow{\mathrm{ppo}} \cup \xrightarrow{\mathrm{grf}} \cup \xrightarrow{\mathrm{ab}}$$

An execution (E, X) is *valid* on an architecture A, written $A\,.\,\mathrm{valid}(E, X)$, when the relation $A\,.\,\mathrm{ghb}(E, X)$ is acyclic (together with the two checks above):

$$A\,.\,\mathrm{valid}(E, X) \triangleq \mathrm{uniproc}(E, X) \wedge \mathrm{thin}(E, X) \wedge \mathrm{acyclic}(A\,.\,\mathrm{ghb}(E, X))$$

Finally, we consider an architecture A_1 to be *weaker* than an architecture A_2, written $A_1 \leq A_2$, when A_1 authorises at least all the executions valid on A_2. TSO is weaker than SC, hence all the SC executions of a program are valid on TSO. In the following, we consider A_2 to be without barriers, $i.e.$ $\xrightarrow{\mathrm{ab}_2} = \emptyset$.

2 Covering Relations

We examine now how to force the executions of a program running on a weak architecture A_1 to be valid on a stronger one A_2, which we call *stability from* A_1 *to* A_2, $i.e.$ we examine when the following property holds for all (E, X):

$$\mathrm{stable}_{\mathrm{A}_1,\mathrm{A}_2}(E, X) \triangleq A_1\,.\,\mathrm{valid}(E, X) \Rightarrow A_2\,.\,\mathrm{valid}(E, X)$$

The execution of **iriw** in Fig. 2 is not stable from Power to SC, for it is valid on Power yet not on SC. We can stabilise it using *synchronisation idioms*, $e.g.$ barriers or locks. Synchronisation idioms *arbitrate conflicts* between accesses, $i.e.$ ensure that one out of two conflicting accesses occurs before the other. We formalise this with an irreflexive *conflict* relation $\xrightarrow{\mathrm{c}}$ over $\mathbb{E}$, such that $\forall xy, x \xrightarrow{\mathrm{c}} y \Rightarrow \neg(y \xrightarrow{\mathrm{po}} x)$ and a *synchronisation* relation $\xrightarrow{\mathrm{s}}$ over $\mathbb{E}$. An execution (E, X) is *covered* when $\xrightarrow{\mathrm{s}}$ *arbitrates* $\xrightarrow{\mathrm{c}}$:

$$\mathrm{covered}_{c,s}(E, X) \triangleq \forall xy, x \xrightarrow{\mathrm{c}} y \Rightarrow x \xrightarrow{\mathrm{s}} y \vee y \xrightarrow{\mathrm{s}} x$$

We consider a relation $\xrightarrow{\mathrm{s}}$ to be *covering* when ordering by $\xrightarrow{\mathrm{s}}$ the conflicting accesses of an execution (E, X) valid on A_1 guarantees its validity on A_2, $i.e.$ the synchronisation $\xrightarrow{\mathrm{s}}$ arbitrates enough conflicts to enforce a strong behaviour:

$$\mathrm{covering}(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) \triangleq \forall EX, (A_1\,.\,\mathrm{valid}(E, X) \wedge \mathrm{covered}_{c,s}(E, X)) \Rightarrow A_2\,.\,\mathrm{valid}(E, X)$$

Lock-based synchronisation. For example, the DRF guarantee [3] ensures that if the *competing accesses* (defined below) of an execution are ordered by locks, then this execution is SC, $i.e.$ locks are covering $w.r.t.$ the competing accesses. Two events are *competing* if they are from distinct processors, to the same location, and at least one of them is a write ($e.g.$ in Fig. 2, the read (a) from x on P_0 and the write (e) to x on P_2):

$$m_1 \overset{\mathrm{cmp}}{\leftrightarrow} m_2 \triangleq \mathrm{proc}(m_1) \neq \mathrm{proc}(m_2) \wedge \mathrm{loc}(m_1) = \mathrm{loc}(m_2) \wedge (m_1 \in \mathbb{W} \vee m_2 \in \mathbb{W})$$

We describe the ordering induced by locks by a relation $\xrightarrow{\text{lock}}$ (instantiated in Sec. 3.1) over $\mathbb{E}$, such that acyclic($\xrightarrow{\text{lock}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rf}}$), corresponding in Fig. 2 to placing locks to a variable ℓ_1 on the accesses (a), (d) and (e) relative to x, and locks to a different variable ℓ_2 on the accesses (b), (c) and (f) relative to y. Thus we have a cycle in $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$: $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{lock}} (f) \xrightarrow{\text{lock}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{lock}} (e) \xrightarrow{\text{lock}} (a)$. If $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ is acyclic, then the execution of Fig. 2 is forbidden. Formally, we have:

Lem. 1. acyclic($\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$) $\Rightarrow$ covering($\xleftrightarrow{\text{cmp}}, (\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})^+$)

This lemma leads to a mapping which we call L (for locks), which simply places a lock by the same lock variable on each side of a given conflict edge. By Lem. 1, it ensures stability to a program for any pair (A_1, A_2).

Lock-free synchronisation. We give here an example of a covering lock-free synchronisation relation. A program can distinguish between two architectures $A_1 \leq A_2$ for one of two reasons. First, if the program involves a pair (x, y) maintained in program order on A_2 ($i.e.$ $x \xrightarrow{\text{ppo}_2} y$) but not on A_1 ($i.e.$ $\neg(x \xrightarrow{\text{ppo}_1} y)$). In Fig. 2, we have $(a) \xrightarrow{\text{po}} (b)$. Hence on a strong architecture A_2 such as SC where $\xrightarrow{\text{ppo}_2} = \xrightarrow{\text{po}}$, we have $(a) \xrightarrow{\text{ppo}_2} (b)$. On a weak architecture A_1 such as Power, where the read-read pairs in program order are not maintained, we have $\neg((a) \xrightarrow{\text{ppo}_1} (b))$.

Second, if the program reads from a write atomic on A_2 but not on A_1. In Fig. 2, we have $(e) \xrightarrow{\text{rfe}} (a)$. On a strong architecture A_2 such as SC where the writes are atomic, $i.e.$ $\xrightarrow{\text{grf}} = \xrightarrow{\text{rf}}$, we have $(e) \xrightarrow{\text{grf}} (a)$. On a weak architecture A_1 such as Power, which relaxes write atomicity, we have $\neg((e) \xrightarrow{\text{grf}} (a))$. We call such reads *fragile reads* and define them as ($\xrightarrow{\text{r}_2\setminus 1} \triangleq \xrightarrow{\text{r}_2} \setminus \xrightarrow{\text{r}_1}$ being the set difference):

$$\text{fragile}(r) \triangleq \exists w, w \xrightarrow{\text{grf}_2\setminus 1} r$$

We consider such differences between architectures as conflicts, and formalise this notion as follows. We consider that two events form a *fragile pair* (written $\xrightarrow{\text{frag}}$) if they are maintained in the program order on A_2, and either they are not maintained in the program order on A_1, or the first event is a fragile read:

$$m_1 \xrightarrow{\text{frag}} m_2 \triangleq m_1 \xrightarrow{\text{ppo}_2} m_2 \wedge \big(\neg(m_1 \xrightarrow{\text{ppo}_1} m_2) \vee \text{fragile}(m_1)\big)$$

An execution is covered if the relation $\xrightarrow{\text{ab}_1}$ arbitrates the fragile pairs. In Fig. 2, this corresponds to placing a barrier between (c) and (d) on P_1, $i.e.$ $(c) \xrightarrow{\text{ab}_1} (d)$, and another barrier between (a) and (b) on P_0, $i.e.$ $(a) \xrightarrow{\text{ab}_1} (b)$. Hence we have a cycle in $\xrightarrow{\text{ab}_1} \cup \xrightarrow{\text{rf}}$: $(d) \xrightarrow{\text{rfe}} (a) \xrightarrow{\text{ab}_1} (b) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{ab}_1} (d)$. If $\xrightarrow{\text{ab}_1}$ is A-cumulative $w.r.t.$ $\xrightarrow{\text{grf}_2\setminus 1}$, we create a cycle in $\xrightarrow{\text{ghb}_1}$, which forbids the execution: $(d) \xrightarrow{\text{ghb}_1} (b) \xrightarrow{\text{ghb}_1} (d)$. Formally, we have:

Lem. 2. AC($\xrightarrow{\text{ab}_1}, \xrightarrow{\text{grf}_2\setminus 1}$) $\Rightarrow$ covering($\xrightarrow{\text{frag}}, \xrightarrow{\text{ab}_1}$)

This lemma leads to a mapping which we call F (for fences), given in Fig. 4. This mapping places a barrier between each fragile pair of a program. Following Lem. 2, it enforces stability to a program for any pair (A_1, A_2). Recall that we give the semantics of the barriers that we use in the mapping F in Sec. 1, § *Architectures*, on p. 4 and Fig. 3.

In x86, stores are atomic, and only the write-read pairs in program order are not preserved, $i.e.$ the fragile pairs are the pairs $w \xrightarrow{\text{po}} r$. We do not need cumulativity in x86, $i.e.$ we only need a non-cumulative write-read barrier: $w \xrightarrow{\text{mfence}} r$.

Arch.	Fragile pair	Barriers (mapping F)
Power	$r \xrightarrow{\text{po}} r$	$r \xrightarrow{\text{sync}} r$ (need A-cumulativity)
	$r \xrightarrow{\text{po}} w$	$r \xrightarrow{\text{lwsync}} w$ (A-cumulativity OK)
	$w \xrightarrow{\text{po}} w$	$w \xrightarrow{\text{lwsync}} w$ (no need for A-cumulativity)
	$w \xrightarrow{\text{po}} r$	$w \xrightarrow{\text{sync}} r$ (need for write-read non-cumulativity)
x86	$w \xrightarrow{\text{po}} r$	$w \xrightarrow{\text{mfence}} r$ (need for write-read non-cumulativity)

Fig. 4. Mapping F: barriers

Name	Code	Comment	Doc [1]
load reserve	`lwarx r1,0,r2`	loads from the address in `r2` into `r1` and reserves the address in `r2`	p. 718
store conditional	`stwcx. r1,0,r2`	checks if the address in `r2` is reserved; if so, stores from `r1` into this address and writes 1 into register `cr`; if not, writes 0 into `cr`	p. 721
branch not equal	`bne L`	checks if register `cr` holds 0, if not branches to `L`	p. 63
compare	`cmpw r4, r6`	compares values in `r4` and `r6`	p. 102

Fig. 5. Table of Power assembly instructions, excluding barriers

In Power, no pair is preserved in program order except the read-read and read-write pairs with a dependency between the accesses [5]. But since stores are not atomic, even the dependent read-read and read-write pairs are fragile. For a read-read pair $r_1 \xrightarrow{\text{po}} r_2$, since r_1 can read from a non-atomic write w, we need a cumulative barrier between r_1 and r_2. But `lwsync` does not order write to read chains, *i.e.* `lwsync` between r_1 and r_2 will not order w and r_2. Therefore we need a `sync`: $r_1 \xrightarrow{\text{sync}} r_2$. For a read-write pair $r \xrightarrow{\text{po}} w$, we need a cumulative barrier as well, but `lwsync` is sufficient here, for it will order the write from which r may read, and w. In the write-write and write-read cases, there is no need for cumulativity. In the write-write case, a `lwsync` is enough, for it orders write-write pairs; but in the write-read case, we need a `sync`.

The mapping F agrees with D. Lea's JSR-133 Cookbook for Compiler Writers [19] for write-write and write-read pairs. Our mapping is much more conservative than D. Lea's for read-read and read-write pairs: it is unclear whether D. Lea's mapping (meant to implement Java's volatiles) intends to restore SC like ours, or rather a weaker memory model. The mapping F on write-write and write-read pairs corresponds to the optimised version of P. McKenney and R. Silvera's Example Power Implementation for C/C++ Memory Model [22] for "Store Seq Cst". Their "Load Seq Cst" is implemented by `sync;ld;cmp;bc; isync`. The use of `sync` before a load access corresponds to our mapping on read-read and read-write pairs. The sequence `cmp;bc;isync` after the same load access ensures that the Load Seq Cst has, in addition to an SC semantics, a *load acquire* semantics.

3 Synchronisation Idioms

To illustrate Sec. 2, we now study the semantics of Power's locks and rmw [1]. As noted by S. Adve and H.-J. Boehm in [4] *"on hardware that relaxes write atomicity [such as Power] even the fully fenced implementation may not be sequentially consistent."* Thus it is unclear whether the synchronisation primitives provided by the architecture

```
     loop:
(a1) lwarx r1,0,r5
     [...]
(a2) stwcx. r2,0,r5
 (b) bne loop
```

(a) rmw

Initially r3 = ℓ, r4 = 0 and r5 = 1

```
     loop:
(a1) lwarx r6,0,r3
 (b) cmpw r4,r6
 (c) bne loop
(a2) stwcx. r5,0,r3
 (d) bne loop
 (e) isync
     [...]
```

(b) Lock

```
     [...]
 (f) lwsync
 (g) stw r4,0,r3
```

(c) Unlock

Fig. 6. Read-modify-write, lock and unlock in Power

actually restore SC: it could perfectly be the architect's intent ($e.g.$ `lwsync` is not strong enough to restore SC, but is faster than `sync`, as we show in Sec. 5), or a bug in the implementation [5]. Hence we need to define the semantics of the synchronisation primitives given in the documentation, and study whether they allow us to restore SC, $i.e.$ that we can use them to build covering relations, as defined in Sec. 2.

We first define *atomic pairs*, which are the stepping stone to build locks, studied in Sec. 3.1 and rmw, studied in Sec. 3.2. We show how to use these primitives to build covering relations. Second, because cumulativity might be too costly in practice, or its implementation challenging, we propose in Sec. 3.2 two lock-free mappings restoring a strong architecture from Power without using cumulativity, as an alternative to the mapping F (see Sec. 2) which uses cumulativity.

Atomicity. Fig. 6(a) gives a generic Power rmw (see Fig. 5 for the instructions we use). The `lwarx` (a_1) loads from its source address in register `r5` and *reserves* it. Any subsequent store to the reserved address from another processor and any subsequent `lwarx` from the same processor invalidates the reservation. The `stwcx.` (a_2) checks if the reservation is valid; if so, it is *successful*: it stores into the reserved address and the code exits the loop. Otherwise, `stwcx.` does not store and the code loops. Thus these instructions ensure *atomicity* to the code they surround (if this code does not contain any `lwarx` nor `stwcx.`), as no other processor can write to the reserved location between the `lwarx` and the successful `stwcx.`.

We distinguish the reads and writes issued by such instructions from the plain ones: we write $\mathbb{R}^*$ (resp. $\mathbb{W}^*$) for the subset of $\mathbb{R}$ (resp. $\mathbb{W}$) issued by a `lwarx` (resp. a successful `stwcx.`), and define two events r and w to form an atomic pair $w.r.t.$ a location ℓ if (a) w was issued by a successful `stwcx.` to ℓ, (b) r was issued by the last `lwarx` from ℓ before (in $\xrightarrow{\text{po}}$) the `stwcx.` that issued w, and (c) no other processor wrote to ℓ between r and w:

$$\begin{aligned}
\text{atom}(r, w, \ell) \triangleq r \in \mathbb{R}^* \wedge w \in \mathbb{W}^* \wedge \text{loc}(r) = \text{loc}(w) = \ell \wedge \qquad & (a)\\
r = \max_{\xrightarrow{\text{po}}}(\{m \mid m \in (\mathbb{R}^* \cup \mathbb{W}^*) \wedge m \xrightarrow{\text{po}} w\}) \wedge \qquad & (b)\\
\neg(\exists w' \in \mathbb{W}, \text{proc}(w') \neq \text{proc}(r) \wedge \text{loc}(w') = \ell \wedge r \xrightarrow{\text{fr}} w' \xrightarrow{\text{ws}} w) \quad & (c)
\end{aligned}$$

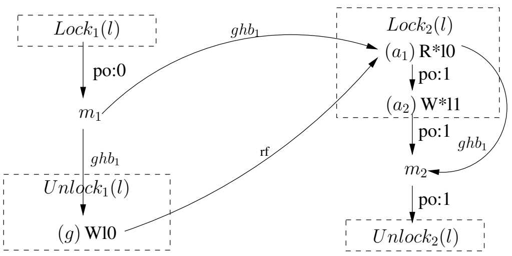


Fig. 7. Opening lock and unlock

3.1 Locks

Atomic pairs are used *e.g.* in *lock* and *unlock* primitives [1, App. B]. The idiomatic Power lock (resp. unlock) is shown in Fig. 6(b) (resp. Fig. 6(c)).

Critical sections. A lock reads the lock variable ℓ to see if it is free; an unlock writes to ℓ to free it. The instructions between a lock and an unlock form a *critical section*. Thus, a critical section consists of a lock $\mathrm{Lock}(\ell, r)$ and an unlock $\mathrm{Unlock}(\ell, r, w)$ (we define these two predicates in the next paragraph) with the same variable ℓ, and the events in $\xrightarrow{\mathrm{po}}$ between the lock's read and the unlock's write:

$$\mathrm{cs}(\mathcal{E}, \ell, r, w) \triangleq \mathrm{Lock}(\ell, r) \wedge \mathcal{E} = \{e \mid r \xrightarrow{\mathrm{po}} e \xrightarrow{\mathrm{po}} w\} \wedge \mathrm{Unlock}(\ell, r, w)$$

We write $\mathrm{loc(cs)}$ for the location of a critical section cs. Two critical sections cs_1 and cs_2 with the same location ℓ are *serialised* if cs_2 reads from cs_1, as in Fig. 7: on the left is cs_1, composed of a lock $\mathrm{Lock}_1(\ell)$, an event m_1 and an unlock $\mathrm{Unlock}_1(\ell)$, which writes into ℓ *via* the write (g). The second critical section cs_2 is on the right: the read (a_1) of its lock $\mathrm{Lock}_2(\ell)$ reads from (g). Thus, cs_1 and cs_2 are serialised if cs_2 Lock's read (written $\mathrm{R}(\mathrm{cs}_2)$) reads from cs_1 Unlock's write (written $\mathrm{W}(\mathrm{cs}_1)$):

$$\mathrm{cs}_1 \xrightarrow{\mathrm{css}_\ell} \mathrm{cs}_2 \triangleq \mathrm{loc}(\mathrm{cs}_1) = \mathrm{loc}(\mathrm{cs}_2) = \ell \wedge \mathrm{W}(\mathrm{cs}_1) \xrightarrow{\mathrm{rf}} \mathrm{R}(\mathrm{cs}_2)$$

Given a location ℓ, two events m_1 and m_2 are in $\xrightarrow{\mathrm{lock}_\ell}$ if they are in two serialised critical sections (as in Fig. 7), or m_1 is in $\xrightarrow{\mathrm{lock}_\ell}$ with an event itself in $\xrightarrow{\mathrm{lock}_\ell}$ with m_2 ($m \in \mathrm{cs}$ ensures m is between cs import and export barriers in $\xrightarrow{\mathrm{po}}$):

$$m_1 \xrightarrow{\mathrm{lock}_\ell} m_2 \triangleq (\exists \mathrm{cs}_1 \xrightarrow{\mathrm{css}_\ell} \mathrm{cs}_2, m_1 \in \mathrm{cs}_1 \wedge m_2 \in \mathrm{cs}_2) \vee (\exists m, m_1 \xrightarrow{\mathrm{lock}_\ell} m \xrightarrow{\mathrm{lock}_\ell} m_2)$$

Finally, two events m_1 and m_2 are in $\xrightarrow{\mathrm{lock}}$ if there exists ℓ such that $m_1 \xrightarrow{\mathrm{lock}_\ell} m_2$.

Lock and unlock In the Power lock of Fig. 6(b), the lines (a_1) to (a_2) form an atomic pair, as in Fig. 6(a); this sequence loops until it acquires the lock. Here, acquiring the lock means that the `lwarx` read the lock variable ℓ, and that ℓ was later written to by a successful `stwcx.`. Thus, the read r of the `lwarx` takes a lock ℓ if it forms an atomic pair with the write w from the successful `stwcx.`:

$$\mathrm{taken}(\ell, r) \triangleq \exists w, \mathrm{atom}(r, w, \ell)$$

The acquisition is followed by a sequence `bne; isync` (lines (d) and (e)), forming an *import barrier* [1, p. 721]. An import barrier prevents any event to float above a read issued by a `lwarx`: in Fig. 7, the event m_2 in cs_2 is in $\xrightarrow{\text{ghb}_1}$ with the read (a_1) from its Lock's `lwarx`. Hence the read r of a lock's `lwarx` satisfies the import predicate when no access m after r can be speculated before r:

$$\text{import}(r) \triangleq \forall rm, (r \in \mathbb{R}^* \wedge r \xrightarrow{\text{po}} m) \Rightarrow (r \xrightarrow{\text{ab}_1} m)$$

Fig. 6(c) shows Power's unlock, starting (line (f)) with an *export barrier* [1, p. 722], here a `lwsync`. The export barrier forces the accesses before the write w of the unlock to be committed to memory before the next lock primitive takes the lock: in Fig. 7, the event m_1 in cs_1 is in $\xrightarrow{\text{ghb}_1}$ with the read (a_1) of cs_2's Lock. Thus we define an export barrier as B-cumulative, but only $w.r.t.$ reads issued by the `lwarx` of an atomic pair:

$$\text{export}(w) \triangleq \forall rm, (r \in \mathbb{R}^* \wedge (m \xrightarrow{\text{po}} w \xrightarrow{\text{rf}} r)) \Rightarrow (m \xrightarrow{\text{ab}_1} r)$$

Then a store to the lock variable (line (g)), or more precisely the next write event to ℓ in program order after a lock acquisition, frees the lock:

$$\begin{aligned}\text{free}(\ell, r, w) \triangleq{} & w \in \mathbb{W} \wedge \text{loc}(w) = \ell \wedge r \xrightarrow{\text{po}} w \wedge \text{taken}(\ell, r) \wedge \\ & \neg(\exists w' \in \mathbb{W}, \text{loc}(w') = \ell \wedge r \xrightarrow{\text{po}} w' \xrightarrow{\text{po}} w)\end{aligned}$$

A lock primitive thus consists of a taken operation (see Fig. 6(b), lines (a_1) to (a_2)) followed by an import barrier. An unlock consists of an export barrier (line (f)) followed by a write freeing the lock (line (g)):

$$\text{Lock}(\ell, r) \triangleq \text{taken}(\ell, r) \wedge \text{import}(r)$$

$$\text{Unlock}(\ell, r, w) \triangleq \text{free}(\ell, r, w) \wedge \text{export}(w)$$

We show that this semantics ensures the acyclicity of $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$, $i.e.$ following Lem. 1, $(\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})^+$ is covering for the competing accesses. Hence locks on the competing accesses ensures a SC behaviour to Power programs:

Lem. 3. $\forall EX, A_1 \,.\, \text{valid}(E, X) \Rightarrow \text{acyclic}(\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})$

Our import barrier allows events to be delayed so that they are performed inside the critical section. Our export barrier allows the events after the unlock to be speculated before the lock is released. Such relaxed semantics already exist for high-level lock and unlock primitives [8,26]. In the documentation [1, p. 721], the import barrier is a sequence `bne;isync` ($i.e.$ a read-read, read-write non-cumulative barrier) or a `lwsync`, $i.e.$ cumulative [1, p.721]. Lem. 3 shows that the first one is enough, for our import barrier does not need cumulativity. The export barrier is a `sync` ($i.e.$ cumulative for all pairs) or a `lwsync` [1, p. 722]. Lem. 3 shows that we only need a B-cumulative barrier towards reads issued by a `lwarx`, $i.e.$ a `sync` is unnecessarily costly. Moreover, although a `lwsync` is not B-cumulative towards plain reads, its implementations appear experimentally to treat the reads issued by the `lwarx` of an atomic pair specially. We tested and confirmed this semantics of `lwsync` with our diy tool [5], by running our automatically generated tests up to 10^{10} times each (see the logs online).

3.2 Read-Modify-Write Primitives

By Lem. 2, we can restore SC in the **iriw** test of Fig. 2 using A-cumulative barriers between the fragile pairs (a) and (b) on P_0, and (c) and (d) on P_1. Yet, cumulativity

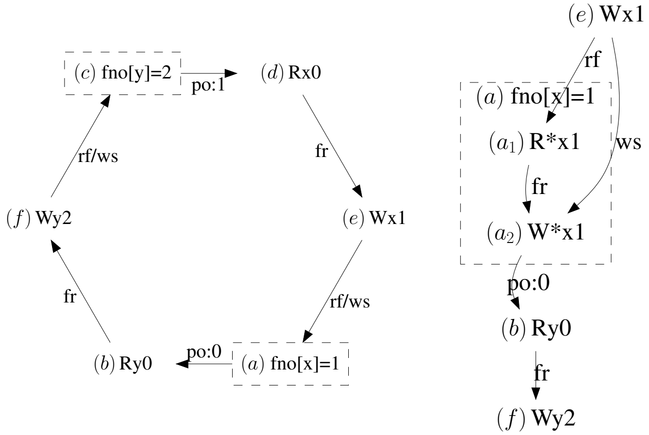


Fig. 8. (a) **iriw** after mapping P (b) Opening fno on P_0

may be challenging to implement or too costly in practice [5]. We propose a mapping of certain reads to rmw (as in Fig. 6(a)), and show that this restores a strong architecture from a weaker one without using cumulativity.

In Fig. 8(a), we replaced the fragile reads (a) and (c) of **iriw** by rmw: we say these fragile reads are *protected* (a notion defined below). In the example we use *fetch and no-op* (fno) primitives [1, p.719] to implement atomic reads. Yet, our results hold for any kind of rmw. We show that when the fragile reads are protected, we do not need cumulative barriers, but just non-cumulative ones. If a read is protected by a rmw, then the rmw compensates the need for cumulativity by enforcing enough order to the write from which the protected read reads.

Protecting the fragile reads. We consider that two events r and w form a rmw $w.r.t.$ a location ℓ if they form an atomic pair $w.r.t.$ ℓ ($i.e.$ the code in Fig. 6(a) does not loop), or there is a read r' after r in the program order forming an atomic pair $w.r.t.$ ℓ with w, such that r' is the last read issued by the loop before the `stwcx.` succeeds ($i.e.$ the code in Fig. 6(a) loops). We do not consider the case where the loop never terminates:

$$\mathrm{rmw}(r, w, \ell) \triangleq \mathrm{atom}(r, w, \ell) \vee (\exists r', r \xrightarrow{\mathrm{po}} r' \wedge \mathrm{loc}(r) = \mathrm{loc}(r') \wedge \mathrm{atom}(r', w, \ell))$$

In Fig. 8(b), we open up the fno box protecting the read (a) from x on P_0. We suppose that the fno is immediately successful, $i.e.$ the code in Fig. 6(a) does not loop. Hence we expand the fno event (a) on P_0 to the r^* (a_1) (from the `lwarx`) in program order with the w^* (a_2) (from the successful `stwcx.`).

We define a read to be *protected* when it is issued by the `lwarx` of a rmw immediately followed in program order by a non-cumulative barrier; an execution (E, X) is protected when its fragile reads are:

$$\mathrm{protected}(r) \triangleq \exists w, \mathrm{rmw}(r, w, \mathrm{loc}(r)) \wedge (\forall m, w \xrightarrow{\mathrm{po}} m \Rightarrow w \xrightarrow{\mathrm{ab_1}} m)$$

$$\mathrm{protected}(E, X) \triangleq \forall r, \mathrm{fragile}(r) \Rightarrow \mathrm{protected}(r)$$

In Fig. 8(b), the write (e) from which (a_1) reads hits the memory before (a_2), $i.e.$ $(e) \xrightarrow{\mathrm{ws}} (a_2)$. Hence there are two paths from (e) to (b): $(e) \xrightarrow{\mathrm{rf}} (a_1) \xrightarrow{\mathrm{po}} (b)$ and $(e) \xrightarrow{\mathrm{ws}}$

Arch.	Fragile pair	rmw (mapping A)	rmw (mapping P)
Power	$r \xrightarrow{\text{po}} r$	fno $\xrightarrow{\text{po}}$ fno	fno $\xrightarrow{\text{sync}} r$
	$r \xrightarrow{\text{po}} w$	fno $\xrightarrow{\text{po}}$ sta	fno $\xrightarrow{\text{lwsync}} w$
	$w \xrightarrow{\text{po}} w$	sta $\xrightarrow{\text{po}}$ sta	$w \xrightarrow{\text{lwsync}} w$
	$w \xrightarrow{\text{po}} r$	sta $\xrightarrow{\text{po}}$ fno	$w \xrightarrow{\text{sync}} r$
x86	$w \xrightarrow{\text{po}} r$	`xchg` $\xrightarrow{\text{po}} r$	na

Fig. 9. Mappings A and P: rmw

$(a_2) \xrightarrow{\text{po}} (b)$. Thus we can trade the fragile pair (a_1, b) for (a_2, b) and compensate the lack of write atomicity of (e) (*i.e.* $(e) \xrightarrow{\text{rfe}} (a)$ not global) with the write serialisation between (e) and (a_2) (thanks to the rmw) instead of cumulativity before. Formally, we prove that a sequence $w \xrightarrow{\text{grf}_2 \setminus 1} r \xrightarrow{\text{ppo}_2} m$ with r protected is globally ordered on A_1:

Lem. 4. $\forall wrm, (\text{protected}(r) \wedge w \xrightarrow{\text{grf}_2 \setminus 1} r \xrightarrow{\text{ppo}_2} m) \Rightarrow w \xrightarrow{\text{ws}}; \xrightarrow{\text{ghb}_1} m$

Thus, if we protect the fragile reads, the only remaining fragile pairs are the ones in $\xrightarrow{\text{ppo}_2 \setminus 1}$. In Fig. 8(a), we have $(e) \xrightarrow{\text{ws}} (a_2) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (f)$ and $(f) \xrightarrow{\text{ws}} (c_2) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (e)$, hence a cycle in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{po}}$. Since $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ are global, to invalidate this cycle, we need to order globally (*e.g.* by a barrier) the accesses (a_2) and (b) on P_0 and (c_2) and (d) on P_1. Indeed, if an execution is protected, non-cumulative barriers placed between the remaining fragile pairs in $\xrightarrow{\text{ppo}_2 \setminus 1}$ ensure stability:

Lem. 5. $A_1 \,.\, \text{valid}(E, X) \wedge \text{protected}(E, X) \wedge (\xrightarrow{\text{ppo}_2 \setminus 1} \subseteq \xrightarrow{\text{ab}_1}) \Rightarrow A_2 \,.\, \text{valid}(E, X)$

This lemma leads to a mapping which we call P (for protected reads), given in Fig. 9. This mapping places a fno on the first read of the fragile pairs, and a barrier between this fno and the second access of the fragile pairs. If the first access of the fragile pair is a write, it remains unchanged and we only place a barrier between the two accesses, following the mapping F. For the read-read (resp. read-write) case, since replacing a read by a fno amounts to replacing the read by a sequence of events ending with a write, we choose a barrier ordering write-read (resp. write-write) pairs, *i.e.* Power `sync` (resp. `lwsync`). Following Lem. 5, it enforces stability to a program for any pair (A_1, A_2).

H.-J. Boehm and S. Adve propose in [10] a mapping of all stores into rmw (*i.e.* `xchg`) on x86 (which has no fragile reads), to provide a SC semantics to C++ atomics. We call this mapping A-x86 (for atomics), and give it in Fig. 9. For models with fragile reads, *e.g.* Power, they question in [4] the existence of *"more efficient mappings (than the use of locks)"*. The mapping P could be more efficient, since it removes the need for cumulativity. Yet, mapping reads to rmw introduces additional stores (issued by `stwcx.`), which may impair the performance. Moreover, we have to use cumulative barriers in the mapping P, for Power does not provide non-cumulative barriers. Yet, we show in Sec. 5 that the mapping P is more efficient than locks on Power machines.

We propose another mapping, given in Fig. 9, which we call A-Power. All reads and writes are mapped into rmw (using fno for reads and fetch-and-store (sta) [1, p. 719] for writes). The documentation stipulates indeed that *"a processor has at most one* *reservation at any time"* [1, p. 663]. Hence two rmw on the same processor in program order may be preserved in this order, because the writes issued by their `stwcx.`, though to different locations, would be ordered by a dependency over the reservation. Although the documentation does not state if this dependency exists, we show in Sec. 5 that the mapping A-Power restores SC experimentally and is more efficient than locks as well.

4 Stability from a Weak Architecture to SC

We now want to minimise the synchronisation that we use, $i.e.$ we would like to synchronise only the conflicting accesses (either competing accesses or fragile pairs) that are actually necessary. For example, if in the **iriw** test of Fig. 2, we add a write (g) to a fresh variable z after (in program order) the write (e) to x on P_2, (e) and (g) may not be preserved in program order, $i.e.$ (e) and (g) may form a fragile pair. Yet, there is no need to maintain them, since they do not contribute to the cycle we want to forbid.

D. Shasha and M. Snir provide in [27] an analysis to place barriers in a program, in order to enforce a SC behaviour. They examine in [27, Thm. 3.9 p. 297] the *critical cycles* of an execution, and show that placing a barrier along each program order arrow of such a cycle (each *delay* arrow) is enough to restore SC. Yet, this work does not provide any semantics of weak memory models. We show in Coq that their technique applies to the models embraced by our framework, $e.g.$ models with store buffering, like TSO or relaxing store atomicity, like Power.

Given an architecture A and event structure E, a cycle $\xrightarrow{\sigma} \subseteq (\overset{\text{cmp}}{\leftrightarrow} \cup \xrightarrow{\text{po}})^+$ (where $\overset{\text{cmp}}{\leftrightarrow}$ is the competing relation of Sec. 2) is *critical on* A, written $\text{critical}_A(E, \xrightarrow{\sigma})$, when it is not a cycle in $(\overset{\text{cmp}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_A})^+$ and satisfies the two following properties. **(i)** Per processor, there are at most two memory accesses (x, y) on this processor and $\text{loc}(x) \neq \text{loc}(y)$. **(ii)** For a given memory location x, there are at most three accesses relative to x, and these accesses are from distinct processors ($w \overset{\text{cmp}}{\leftrightarrow} w$, $w \overset{\text{cmp}}{\leftrightarrow} r$, $r \overset{\text{cmp}}{\leftrightarrow} w$ or $r \overset{\text{cmp}}{\leftrightarrow} w \overset{\text{cmp}}{\leftrightarrow} r$). In Fig. 2, the execution of **iriw** has a critical cycle on Power.

In our framework, we show that the execution witnesses X of an event structure E are stable from A to SC if and only if E contains no critical cycle on A, $i.e.$ that an execution valid on A is SC if and only if E contains no critical cycle on A:

Thm. 1. $\forall E, (\forall X, \text{stable}_{\text{A,SC}}(E, X)) \Leftrightarrow \neg(\exists \xrightarrow{\sigma}, \text{critical}_A(E, \xrightarrow{\sigma}))$

This theorem means that we do not have to synchronise all the conflicts to ensure stability from a weak architecture to SC, but only those occurring in critical cycles. Hence to restore SC, we should arbitrate (with a covering relation) the conflicting accesses (competing accesses or fragile pairs) occurring in the critical cycles.

5 offence: A Synchronisation Tool

We implemented our study in our new offence tool, illustrating techniques that can be included in a compiler. Given a program in x86 or Power assembly, offence places either lock-based or lock-free synchronisation along the critical cycles of its input, following the mapping A, P, L or F, to enforce a SC behaviour.

5.1 Control Flow Graphs and Critical Cycles

offence builds one control flow graph (cfg) per thread of the input program, containing *static events* ($i.e.$ nodes representing memory accesses), and control flow instructions. A static memory event f has a direction, a location, originating instruction and processor, as events do, but no value component.

Given an event structure and two events $e_1 \xrightarrow{\text{po}} e_2$, mapping to static events f_1 and f_2, we compute the *static program order* $\xrightarrow{\text{pos}}$ such that $e_1 \xrightarrow{\text{po}} e_2$ entails $f_1 \xrightarrow{\text{pos}} f_2$ using a standard forward data flow analysis. If memory locations accessed by a given instruction are constant, we have $\text{loc}(e_1) = \text{loc}(f_1)$ and $\text{loc}(e_2) = \text{loc}(f_2)$. Hence static conflicts computed from the cfg, written $\overset{\text{cmps}}{\leftrightarrow}$, abstract the conflicts of the event structures. When locations are not constant, we would need alias analysis to compute an over-approximation of the locations of each static event, considering for example that all pairs of memory accesses by distinct processors conflict, if one of them is a write.

With F the set of static events, we call the triple $(F, \xrightarrow{\text{pos}}, \overset{\text{cmps}}{\leftrightarrow})$ *static event structure*. Following Sec. 4, we enumerate the cycles of F that have properties **(i)** and **(ii)**, $i.e.$ we build an over-approximation of the runtime critical cycles.

5.2 Placing Synchronisation Primitives

We then collect the fragile pairs (*i.e.* the write-read pairs in x86 and all pairs in Power) occurring in the critical cycles of F. By Thm. 1 it is necessary and sufficient to maintain these fragile pairs to reach stability, $i.e.$ to restore SC.

Barriers. Then, offence follows the mapping F on these fragile pairs. Given a pair (f_1, f_2), offence issues the barrier request (i_1, i_2, b) where $i_1 = \text{ins}(f_1)$, $i_2 = \text{ins}(f_2)$ and b is the required barrier. Every path from i_1 to i_2 in the cfg should pass through a barrier instruction b. We use the global barrier placement of [20], which maximises the number of pairs maintained by a given barrier.

Alternative to barriers. offence can also follow the mappings A and P. For A-x86, the xchg instruction has an implicit write-read barrier semantics [10]. Thus, we use the global barrier placement of [20] for xchg. For locks, offence follows the mapping L on the conflict edges of the cfg. Sec. 3.1 describes the lock and unlock idioms that we use for Power. For x86, lock uses the xchg instruction to build a compare-and-swap loop, while unlock uses a single store instruction.

5.3 Experiments

Generating tests. We generated two kinds of tests to exercise offence, using our previous diy tool [5], which computes tests in x86 or Power assembly from a cycle of relations. First, we generate tests from critical cycles, $e.g.$ **iriw** in Fig. 2. Second, using a new tool, we mix such tests: given two tests built from critical cycles, we randomly permute processors of one of the given tests, turn its memory locations and registers to fresh ones, and interleave the codes of the programs. We produced two series of tests, written X, each series consisting of 209 tests for Power and 58 tests for x86.

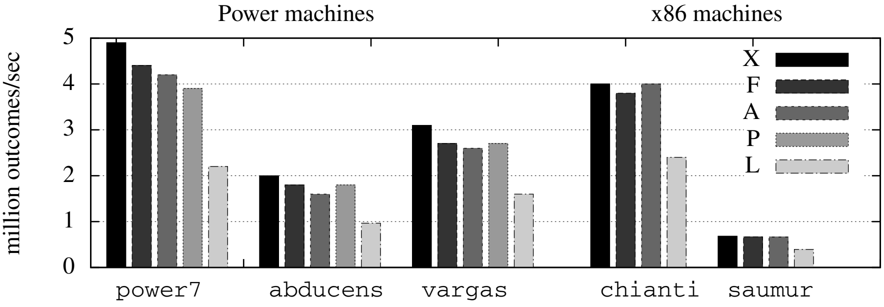


Fig. 10. Productivity observed during soundness experiments

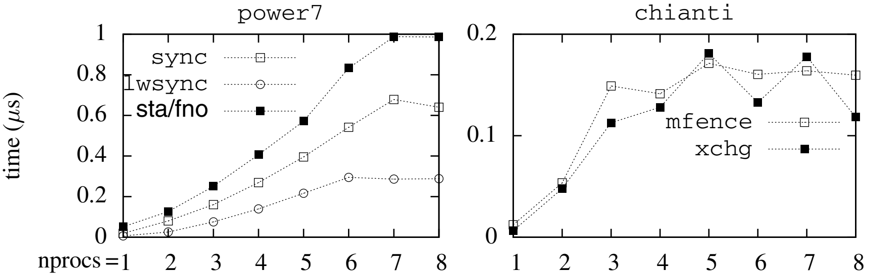


Fig. 11. Time of synchronisation constructs, in microseconds

Experimental soundness. We run these tests against hardware using our litmus tool [6]. We observed that all tests from the initial X series exhibit violations of SC and that the tests transformed by offence (following the mappings F, A, P and L) do *not* exhibit violations of SC, running each test at least 10^9 times. Thus we confirmed experimentally that our mappings enforce SC, which we established formally for the mappings F (Lem. 2), P (Lem. 5) and L (Lem. 1 and 3).

Cost measures. Fig. 10 shows the *productivity*, *i.e.* the number of outcomes per second, for the initial series of tests X, and for the tests transformed by offence following the mappings F, A, P and L. We ran our tests on three Power machines: power7 (Power7, 8 cores 4-ways SMT), abducens (Power6, 4 cores 2-ways SMT) and vargas (Power6, 32 cores 2-ways SMT); and on two AMD64 machines: chianti (Intel Xeon, 8 cores, 2-ways HT) and saumur (Intel Xeon, 4 cores, 2-ways HT). Our mappings F, P and A outperform the L one, *i.e.* provide *"more efficient mappings (than the use of locks)"*, answering the question of [4].

To compare the barriers and rmw more precisely, we consider 8 specific tests from 1 to 8 threads, where we add with offence only one synchronisation primitive per thread, and insert the code for each thread inside a tight loop. We then measure running times on our two 8 core machines, power7 and chianti, substract the time of the original test from the time of synchronised tests and divide the result by loop size. We give the results in Fig. 11. While fences and rmw are fast in isolation (10–20 ns on one thread), their cost raises to hundreds of ns when communication by shared memory occurs.

6 Related Work and Conclusion

Related work. The DRF guarantee [3,10,23], the semantics of synchronisation idioms [9,8], and the insertion of barriers [27,14,11,17] have been extensively studied, but most of these works focus on one kind of synchronisation at a time, and none of them addresses Power traits such as cumulativity or the lack of write atomicity.

S. Burckhardt and M. Musuvathi examine in [12] whether we can simulate a program running on TSO by enumerating only its SC executions. They distinguish a class of such executions, the *TSO-safe* ones. We believe these executions to be an instance of our stable ones, *i.e.* the stable executions from TSO to SC. Yet, our characterisation of stability in the general case is a novel contribution.

J. Lee and D. Padua examine in [20] how to restore SC at compiler level: we used their global fence placement algorithm. Our work improves on [20] *w.r.t.* semantical fundations: as a result, we use Power `lwsync` when possible and we do not use x86 `lfence` and `sfence` barriers, irrelevant in user-level code. Our mappings could be included in their Java compiler [29], *i.e.* using `lwsync` for Power, and `xchg` for x86.

Conclusion. Our formal study of stability in weak memory models allows us to define several mappings of Power or x86 assembly code, which, as we prove in Coq, give a SC behaviour to a program. Along the way, we give a semantics to Power's `lwarx` and `stwcx.` instructions and show how to use the lightweight Power barrier `lwsync`, which are novel contributions. In addition, we characterise the executions stable from a weak architecture to SC, hence generalise the result of [27] to weak memory models. Finally, we implement our study in our offence tool, to measure the cost of these mappings: our lock-free mappings outperform locks on our test set. Our work could for example benefit to compiler writers and semanticists interested in standardisation and implementability (*e.g.* of Java volatiles or C++ atomics on Power platforms).

Acknowledgements. We thank Susmit Sarkar, Peter Sewell, Michael Tautschnig, Jules Villard and Boris Yakobowski for comments on a draft.

References

1. Power ISA Version 2.06 (2009)
2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29, 66–76 (1995)
3. Adve, S.V., Hill, M.D.: Weak Ordering - A New Definition. In: ISCA (1990)
4. Adve, S.V., Boehm, H.-J.: Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 258–272. Springer, Heidelberg (2010)
6. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: litmus: Running tests against hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011)
7. Alpha Architecture Reference Manual, 4th edn. (2002)
8. Boehm, H.-J.: Reordering Constraints for Pthread-Style Locks. In: PPoPP (2007)

9. Boehm, H.-J.: Threads Cannot Be Implemented As a Library. In: PLDI (2005)
10. Boehm, H.-J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: PLDI (2008)
11. Burckhardt, S., Alur, R., Martin, M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
12. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
13. Cantin, J., Lipasti, M., Smith, J.: The Complexity of Verifying Memory Coherence. In: SPAA (2003)
14. Fang, X., Lee, J., Midkiff, S.: Automatic fence insertion for shared memory multiprocessing. In: ICS (2003)
15. Huynh, T.Q., Roychoudhury, A.: A memory model sensitive checker for C#. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 476–491. Springer, Heidelberg (2006)
16. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30 (March 2009)
17. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
18. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
19. Lea, D.: The JSR-133 Cookbook for Compiler Writers (September 2006), `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`
20. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Transactions on Computers 50, 824–833 (2001)
21. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL (2005)
22. McKenney, P., Silvera, R.: Example Power Implementation for C/C++ Memory Model (August 2008), `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2010.02.19a.html`
23. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
24. Park, S., Dill, D.: An executable specification, analyzer and verifier for RMO. In: SPAA 1995 (1995)
25. Rinard, M.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, p. 1. Springer, Heidelberg (2001)
26. Sevcik, J.: Program Transformations in Weak Memory Models. PhD thesis, University of Edinburgh (2008)
27. Shasha, D., Snir, M.: Efficient and Correct Execution of Parallel Programs that Share Memory. In: TOPLAS (1988)
28. Sparc Architecture Manual Version 9 (1994)
29. Sura, Z., Fang, X., Wong, C.-L., Midkiff, S.P., Lee, J., Padua, D.A.: Compiler techniques for high performance SC Java programs. In: PPoPP 2005. ACM Press, New York (2005)
30. Yang, Y., Gopalakrishnan, G.C., Lindstrom, G.: Memory-model-sensitive data race analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 30–45. Springer, Heidelberg (2004)

Verification of Certifying Computations

Eyad Alkassar[1], Sascha Böhme[2], Kurt Mehlhorn[3], and Christine Rizkallah[3]

[1] Universität des Saarlandes, Germany
[2] Institut für Informatik, Technische Universität München, Germany
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. Formal verification of complex algorithms is challenging. Verifying their implementations goes beyond the state of the art of current verification tools and proving their correctness usually involves non-trivial mathematical theorems. Certifying algorithms compute in addition to each output a witness certifying that the output is correct. A checker for such a witness is usually much simpler than the original algorithm – yet it is all the user has to trust. Verification of checkers is feasible with current tools and leads to computations that can be completely trusted. In this paper we develop a framework to seamlessly verify certifying computations. The automatic verifier VCC is used for checking code correctness, and the interactive theorem prover Isabelle/HOL targets high-level mathematical properties of algorithms. We demonstrate the effectiveness of our approach by presenting the verification of a typical example of the algorithmic library LEDA.

1 Introduction

One of the most prominent and costly problems in software engineering is correctness of software. In this paper, we are concerned with software for difficult algorithmic problems, e.g., matchings in graphs. The algorithms for such problems are complex; formal verification of the resulting programs is beyond the state of the art. We show how to obtain *formal instance correctness*, i.e., formal proofs that outputs for particular inputs are correct. We do so by combining the concept of certifying algorithms with methods for code verification and theorem proving.

A *certifying algorithm* [3,18,13] produces with each output a *certificate* or *witness* that the *particular output* is correct. By inspecting the witness, the user can convince himself that the output is correct, or reject the output as buggy. Figure 1 contrasts a standard algorithm with a certifying algorithm for computing a function f.

A user of a certifying algorithm inputs x and receives the output y and the witness w. He then checks that w proves that y is a correct output for input x. The process of checking w can be automated with a *checker*, which is an algorithm for verifying that w proves that y is a correct output for x. Having checked the witness, the user may proceed with complete confidence that output y has not been compromised. Certifying algorithms are the design principle of the algorithmic library LEDA [14]: Checkers are an integral part of the library and may (optionally) be invoked after every execution of a LEDA algorithm. Adoption of the principle greatly improved the reliability of the library.

We take the principle a step further and develop a methodology for formal proofs of instance correctness. We demonstrate it on one of the more complex algorithms in

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 67–82, 2011.

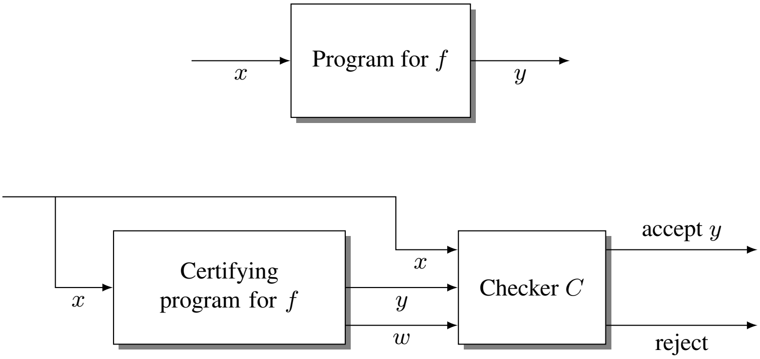


Fig. 1. The top figure shows the I/O behavior of a conventional program for computing a function f. The user feeds an input x to the program and the program returns an output y. A certifying algorithm for f computes y and a witness w. The checker C accepts the triple (x, y, w) if and only if w is a valid witness for the equality $y = f(x)$.

LEDA, maximum cardinality matching in graphs. The description of the algorithm and its implementation in [14] comprises 15 pages. In contrast, the checker is less than a page. Our formalization revealed that the checker program in LEDA is incomplete.

We outline our approach in Section 2 and give a detailed case study in Section 4. See also `http://www4.in.tum.de/~boehmes/certifying_algorithms.html` for related files. In Section 3 we survey the verfication tools VCC and Isabelle/HOL. Section 5 discusses related work and Section 6 offers conclusions.

2 Outline of Approach

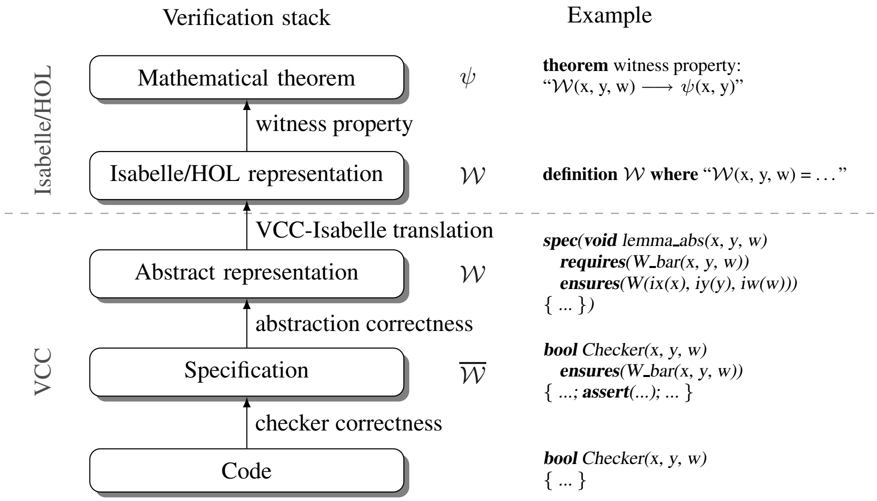
We consider algorithms taking an input from a set X and producing an output in a set Y and a witness in a set W. The input $x \in X$ is supposed to satisfy a precondition $\varphi(x)$ and the input together with the output $y \in Y$ is supposed to satisfy a postcondition $\psi(x, y)$. For simplicity, we only consider algorithms with trivial preconditions in this paper, i.e., $\varphi(x)$ for all $x \in X$. A *witness predicate* for a specification with postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$ with the following *witness property*

$$\mathcal{W}(x, y, w) \implies \psi(x, y) \tag{1}$$

In contrast to algorithms which work on abstract sets X, Y, and W, programs as their implementations operate on concrete representations of abstract objects. We use $\overline{X}$, $\overline{Y}$, and $\overline{W}$ for the set of representations of objects in X, Y, and W, respectively and assume mappings $i_X : \overline{X} \to X$, $i_Y : \overline{Y} \to Y$, and $i_W : \overline{W} \to W$. We also have a concrete version $\overline{\mathcal{W}} \subseteq \overline{X} \times \overline{Y} \times \overline{W}$ of the witness predicate and a program C that checks it. The concrete version $\overline{\psi}$ of the postcondition is defined as

$$\overline{\psi}(\overline{x}, \overline{y}) \equiv \psi(i_X(\overline{x}), i_Y(\overline{y})). \tag{2}$$

We have the following proof obligations:



```
theorem witness property:
"W(x, y, w) ⟶ ψ(x, y)"

definition W where "W(x, y, w) = ..."

spec(void lemma_abs(x, y, w)
  requires(W_bar(x, y, w))
  ensures(W(ix(x), iy(y), iw(w)))
{ ... })

bool Checker(x, y, w)
  ensures(W_bar(x, y, w))
{ ...; assert(...); ... }

bool Checker(x, y, w)
{ ... }
```

Fig. 2. Verification Framework

Checker Correctness: A formal proof that C checks $\overline{\mathcal{W}}(\overline{x},\overline{y},\overline{w})$, i.e., the checker C accepts $(\overline{x},\overline{y},\overline{w})$ if and only if $\overline{\mathcal{W}}(\overline{x},\overline{y},\overline{w})$ holds.

Abstraction Correctness: A formal proof of

$$\overline{\mathcal{W}}(\overline{x},\overline{y},\overline{w}) \implies W(i_X(\overline{x}), i_Y(\overline{y}), i_W(\overline{w})). \tag{3}$$

Witness Property: A formal proof for the implication (1).

Theorem 1. *Assume that the proof obligations are fulfilled and C accepts a triple $(\overline{x},\overline{y},\overline{w})$. Then $\overline{\psi}(\overline{x},\overline{y})$ by a formal proof.*

Proof. Since C accepts $(\overline{x},\overline{y},\overline{w})$ and we have a formal proof for the correctness of C, we have a formal proof for $\overline{\mathcal{W}}(\overline{x},\overline{y},\overline{w})$. By implication (3), we have a formal proof for $\mathcal{W}(i_X(\overline{x}), i_Y(\overline{y}), i_W(\overline{w}))$ and then by (1) a formal proof for $\psi(i_X(\overline{x}), i_Y(\overline{y}))$. The latter is equivalent to $\overline{\psi}(\overline{x},\overline{y})$ by definition (2). □

We next discuss how we fulfill the various proof obligations in a *comprehensive* and *efficient* framework, see Fig. 2. Comprehensive means that the final proof formally combines (as much as possible at syntactic level) the correctness arguments for all levels (implementation, abstraction and mathematical theory). Efficient means to use the right tool for the right target. For example, applying a general theorem prover to verify imperative code would involve a lot of language-specific overhead and lead to less automatization; similarly, a specialized code verifier is often not powerful enough to cover non-trivial mathematical properties. The aims comprehensiveness and efficiency seem to be conflicting, as different tools usually come with different languages, axiomatization sets, etc. Our solution is to use second-order logic as a common interface language.

LEDA is written in C++ [14]. Our aim is to verify code which is as near as possible to the original implementation; by this we demonstrate the feasibility of verifying already established libraries written in imperative languages such as C. Thus we verify code with VCC [6], an automatic code verifier for full C. Our choice is motivated by the maturity of the tool and the provision of a assertion language which is rich enough for our requirements. In the Verisoft XT project [20] VCC was successfully used to verify tens of thousands of non-trivial C code. VCC offers a second-order logic assertion language with ghost code and types such as maps and unbounded integers. This gives us enough expressiveness to quantify over graphs, labellings, etc. and simplifies the translation to other proof systems. For verifying the mathematical part, we resort to Isabelle/HOL, a higher-order-logic interactive theorem prover [17], due to the large set of already formalized mathematics, its descriptive proof format and its various automatic proof methods and tools. In Section 3 we overview both systems. Figure 2 shows the work-flow for verifying checkers.

Checker Verfication: Starting point is the checker code written in C. Using VCC we annotate the functions and data structures, such that the witness predicate $\overline{\mathcal{W}}$ can be established as postcondition of the checker function.

Abstraction Correctness: The witness predicate $\overline{\mathcal{W}}$ is defined over C data-structures, e.g. pointers, arrays, unions and bounded numbers. A one-to-one translation to Isabelle/HOL would have to unveil the complete type and memory axiomatization of C and VCC and would thus generate an extremely large proof context. We avoid this overhead by first abstracting all involved data structures and properties to pure mathematical objects and definitions (using VCC ghost types) by defining mappings i_X, i_Y and i_W. As a result we obtain a second-order logic formula in VCC for the witness property $\mathcal{W}$. We justify this abstraction by proving correspondence lemmas between abstract and concrete properties in VCC.

Export to Isabelle/HOL: Next—based on the abstract postcondition of the checker—we formulate the overall correctness theorem in VCC, i.e., implication (1)[1]. Establishing such a theorem may involve non-trivial mathematical reasoning. Therefore we translate it to Isabelle/HOL. Due to the level of abstraction this translation is purely syntactical and does not involve any VCC specifics.

Witness Property: We prove the final theorem using Isabelle/HOL.

We stress that the overall correctness theorem is formulated in VCC; this is important for usability. A user of a verified checker only has to look at its VCC specification; the fact that we outsource the proof of the witness property to Isabelle/HOL is of no concern to him.

3 Tool Overview: VCC and Isabelle/HOL

VCC [6,7,15] is an assertional, first-order deductive code verifier for full C code. To overcome the restrictions of first-order reasoning, ghost state and code are used, e.g.,

[1] Mathematical theorems can be formulated in VCC using pure ghost functions, i.e., functions that do not alter the state.

to maintain inductively defined information. Specifications in the form of function contracts or data invariants are added directly into the C source code. During regular build, these annotations are ignored. From the annotated program, VCC generates verification conditions for (partial) correctness, which it then tries to discharge using the Boogie verifier [2] and the automatic theorem prover Z3 [16].

Verification in VCC makes heavy use of ghost data and code (indicated by keyword ***spec()***) used for reasoning about the program but omitted from the concrete implementation. VCC provides ghost objects, ghost fields of structured data types, local ghost variables, ghost function parameters, and ghost code. Ghosts can not only use C data types but also additional mathematical data types, e.g., mathematical integers (***mathint***), records and maps. VCC ensures that information does not flow from ghost state to non-ghost state, and that all ghost code terminates; these checks guarantee that program execution when projected to non-ghost code is not affected by ghost code.

Isabelle/HOL [17,12] is an interactive theorem prover for classical higher-order logic based on Church's simply-typed lambda calculus. Internally, the system is built on top of an inference kernel which provides only a small number of rules to construct theorems; complex deductions (especially by automatic proof methods) ultimately rely on these rules only. This approach, called LCF due to its pioneering system [11], guarantees correctness as long as the inference kernel is trusted. Isabelle/HOL comes with a rich set of already formalized theories, among which are natural numbers and integers as well as sets and finite sets. New types can also be introduced. Proofs in Isabelle/HOL are written in a style close to that of mathematical textbooks. The user structures the proof and the system fills in the gaps by its automatic proof methods.

4 Case Study: Maximum Cardinality Matching in Graphs

We present a case study: maximum cardinality matchings in graphs. We obtain formal instance correctness. Our starting point is the certifying algorithm and the corresponding checker in LEDA. We give a formal proof for the correctness of the checker, for the witness property, and the connection between them[2].

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint. A matching has maximum cardinality if its cardinality is at least as large as that of any other matching. Figure 3 shows a graph, a maximum cardinality matching, and a witness of this fact. An *odd-set cover* OSC of a graph G is a labeling of the nodes of G with integers such that every edge of G is either incident to a node labeled 1 or connects two nodes labeled with the same number $i \geq 2$.

Theorem 2 (Edmonds [9]). *Let M be a matching in a graph G and let OSC be an odd-set cover of G. For any $i \geq 0$, let n_i be the number of nodes labeled i. If*

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor \tag{4}$$

then M is a maximum cardinality matching.

[2] All files related to our formalization can be obtained from the following URL: http://www4.in.tum.de/~boehmes/certifying_algorithms.html

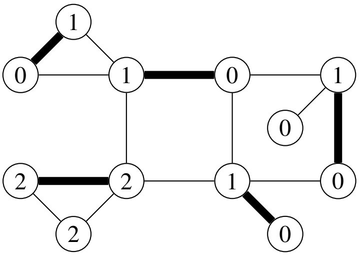


Fig. 3. The node labels certify that the indicated matching is of maximum cardinality: All edges of the graph have either both endpoints labelled as 2 or at least one endpoint labelled as 1. Therefore, any matching can use at most one edge with both endpoints labelled 2 and at most four edges that have an endpoint labelled 1. Therefore, no matching has more than five edges. The matching shown consists of five edges.

Proof. Let N be any matching in G. For $i \geq 2$, let N_i be the edges in N that connect two nodes labeled i and let N_1 be the remaining edges in N. Then, by the definition of odd-set cover, every edge in N_1 is incident to a vertex labeled 1. Since edges in a matching do not share endpoints, we have

$$|N_1| \leq n_1 \quad \text{and} \quad |N_i| \leq \lfloor n_i/2 \rfloor \quad \text{for } i \geq 2.$$

Thus $|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor = |M|$. □

It can be shown (but this is non-trivial) that for any maximum cardinality matching M there is an odd-set cover OSC satisfying equality (4). The cover uses non-negative node labels in the range 0 to $|V| - 1$ and all n_i's with $i \geq 2$ are odd. The *certifying algorithm for maximum cardinality matching* in LEDA returns a matching M and an odd-set cover OSC such that (4) holds. The relationship to Section 2 is as follows:

$$\begin{aligned} X, Y &= \text{the set of all finite undirected graphs without self-loops} \\ \psi(G, M) &= M \text{ is a maximum cardinality matching in } G \\ W &= \text{odd-set covers} \\ \mathcal{W}(G, M, \mathit{osc}) &= M \text{ is a matching in } G,\ \mathit{osc} \text{ is an odd-set cover for } G, \text{ and (4) holds.} \end{aligned}$$

Theorem 2 is the witness property. We give a formal proof for it in Section 4.2. Writing a correct program which checks whether a set of edges is a matching and a node labeling is an odd-set cover which together satisfy Eq. (4) is easy. In Section 4.1 we describe the verification of such a checker. In Section 4.3, we link both results.

4.1 Checker

First, we define the specification $\overline{\mathcal{W}}$ of the checker and consider its verification against the code. Next, we abstract the postcondition to $\mathcal{W}$ and define the witness property $\mathcal{W}(x, y, w) \implies \psi(x, y)$ which is then translated to Isabelle/HOL. Except for the witness property, which is proven in Isabelle/HOL, all presented abstractions and functions have been formally verified using VCC.

```
struct edge {unsigned s; unsigned t;};

typedef struct graph {
   unsigned m; unsigned n; // m edges and n nodes
   struct edge∗ es; // array of edges

   // data−type invariants
   invariant(∀(unsigned e; e < m  ⟹  es[e].s < n ∧ es[e].t < n ∧ es[e].s ≠ es[e].t))
   // further technical invariants are omitted here
} graph;
```

Listing 1. Data structures and invariants

Specification. First, we specify well-defined graphs as a property over the implementation data-type (see Listing 1). In the implementation, nodes are identified by unsigned integers, edges are represented as C structs with two components, the source and the target node, and graphs are encoded by structs with three components, numbers of nodes and edges and an edges array. In VCC we can specify data-type invariants, which are guaranteed to hold whenever an object of that data-type is ***wrapped***[3]. The graph invariant (see Listing 1) excludes self-loops and requires that endpoints of edges are in range. To establish memory safety, a set of additional invariants specifying ownership relations between different graph components are required. For convenience we have omitted them here.

Next, we specify the postcondition $\overline{\mathcal{W}}$ of the checker function. Its arguments are the original graph *G*, the alleged maximum matching *M* and two witnesses; an odd set cover *osc* and a graph embedding *id*, which are both mappings from ***unsigned*** to ***unsigned***. We specify the matching as a graph *M* plus an embedding *id* that maps edges of *M* to edges of *G*. Alternatively, we could have specified it as a list of edges of *G*. The postcondition states that the checker outputs ***true*** if and only if the following four properties hold, three of which can be expressed straightforwardly as first-oder logic formulae.

Matching: Let the ghost predicate *eadj* denote adjacent edges. Then *M* is a matching precisely if the following condition holds:

```
spec(ispure bool is_matching(graph∗ M)
  ensures(result  ⟺  ∀(unsigned e1, e2; e1 < M→m ∧ e2 < M→m ∧ e1 ≠ e2  ⟹
    ¬eadj(M→es[e1], M→es[e2])));)
```

where ***ensures(...)*** defines a postcondition of a function and ***spec()*** and ***ispure()*** mark a function as ghost and free of side-effects.

Subgraph: Checking that *M* is a subgraph of *G* is made efficient by an embedding function *id*, which maps edge identifiers in *M* to those in *G*. This check is missing in the LEDA checker. Let the ghost predicate *eeq* denote equality of edges. Then subgraph is specified by:

```
spec(ispure bool is_subgraph(graph∗ G, graph∗ M, unsigned∗ id)
  returns(∀(unsigned e; e < M→m  ⟹  id[e] < G→m ∧ eeq(M→es[e],
    G→es[id[e]])));)
```

where ***returns(x)*** abbreviates ***ensures(result*** ≡ *x)*.

[3] As long as an object is wrapped, its data may not be modified. Whenever the state of an object is changed to wrapped its corresponding data-type invariants are checked.

Odd-set cover: The mapping *osc* is an odd-set cover for a graph *G* if and only if:

```
spec(ispure bool is_odd_set_cover(graph∗ M, unsigned∗ osc)
  returns( ∀(unsigned k; k < G→n  ⟹  osc[k] < G→n) ∧
           ∀(unsigned e; e < G→m  ⟹
             osc[G→es[e].s] ≡ 1 ∨ osc[G→es[e].t] ≡ 1 ∨
             (osc[G→es[e].t] ≡ osc[G→es[e].s] ∧ osc[G→es[e].t] > 1)));)
```

Equation (4): It states the equality of *M→m* and a sum. Specifying sums without using recursive functions is, however, a bit intricate[4]. Given a sum $\sum_{i<N} expr(i)$, the usual trick is to define a (finite) sequence $S[i]$ of partial sums where $S[i+1] = S[i] + expr(i)$ for $i > 0$ and 0 otherwise. The last element $S[n]$ of the sequence then defines the desired sum.

In VCC we specify the sequences of partial sums by ghost maps. Our checker has to compute (i) the sums n_i, denoting the number of nodes with label i and (ii) the overall sum. The VCC map defining n_i is specified by the equation *N[k + 1][i] ≡ N[k][i] + (i ≡ osc[k] ? 1 : 0)* with the base case *N[0][i] ≡ 0*. The map for the overall sum is defined by the equation *SU[i + 1] ≡ SU[i] + N[G→n][i]/2* with the base case *SU[2] ≡ N[G→n][1]* and *SU[0] ≡ 0 ∧ SU[1] ≡ 0* for trivial graphs. The following predicate encapsulates these conditions:

```
spec(ispure bool consistent_sums(graph∗ G, unsigned∗ osc,
     mathint N[mathint][mathint], mathint SU[mathint])
  returns(
    ∀(mathint i; 0 ≤ i ∧ i < G→n  ⟹  N[0][i] ≡ 0) ∧
    ∀(mathint i, k; 0 ≤ i ∧ i < G→n ∧ 0 ≤ k ∧ k < G→n  ⟹
       N[k + 1][i] ≡ N[k][i] + (i ≡ osc[k] ? 1 : 0)) ∧
    SU[0] ≡ 0 ∧ SU[1] ≡ 0 ∧ SU[2] ≡ N[G→n][1] ∧
    ∀(mathint i; 1 < i ∧ i < G→n  ⟹  SU[i + 1] ≡ SU[i] + N[G→n][i/2]);)
```

Map types are declared analogously to array types, e.g., ***mathint** map[**mathint**]* denotes a map from unbounded integers to unbounded integers.

Based upon these definitions, Eq. (4) is formulated as *M→m ≡ SU[G→n].*

The complete postcondition of the checker is defined by the specification function *W_bar_holds* given in Listing 2. Note that the partial sums *N* and *SU* are passed as ghost parameters to the predicate[5].

Implementation and Verification. The checker function is written in plain C. Its data structures have already been introduced in Listing 1. The implementation is straightforward and consists of seven loops.

We verify the checker code by proving that its postcondition is equivalent to the witness predicate $\overline{\mathcal{W}}$[6]. As precondition we require that the graph and the matching are well-defined (by requiring that they are wrapped, i.e., that their data-type invariant

[4] We do not use recursive specifications because VCC does not yet support termination proofs.

[5] Note, that the maps SU and N could be hidden by existential quantification. We have not choosen this solution due to technical problems with existential quantifiers in VCC (presumably solved soon).

[6] For soundness, implication would suffice. However, then a trivial checker returning always false would also satisfy the postcondition.

```
spec(ispure bool W_bar_holds(bool checker_out, graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
    mathint N[mathint][mathint], mathint SU[mathint])
  ensures(
    consistent_sums(G, osc, N, SU) ∧ // sum correctly computed
    checker_out ⟺
      is_matching(M) ∧ is_subgraph(G, M, id) ∧ is_odd_set_cover(G, osc) ∧
      SU[G→n] ≡ M→m // sum equals cardinality of M
    ));)

bool max_card_match_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id
    spec(out mathint N[mathint][mathint]) spec(out mathint SU[mathint])) // ghost output
  requires(wrapped(G)) // wrapped implies that the datatype invariants hold
  requires(wrapped(M))
  requires(∀(unsigned k; k < G→n  ⟹  osc[k] < G→n))
  ensures(W_bar_holds(result, G, M, osc, id, N, SU))
```

Listing 2. Implementation correctness. The VCC keywords **requires()** and **ensures()** denote pre- and postconditions of functions. The code enclosed in **spec()** is ghost code and only taken into account during verification. With **spec(out ..)** we specify ghost output variables to functions.

holds), and that the odd-set cover is in range. The contract of the checker program is given in Listing 2. Note, that the partial sums N and SU are computed in ghost code and returned as ghost output values.

Most of the work in proving the postcondition lies in finding the right loop invariants. In Listing 3 we give an excerpt from the verification of the matching property. The presented code allocates memory for a local array *deg_in_M* (note, that we assume that allocation will never fail, i.e., that enough memory is available), which is used to count the number of edges adjacent to any node. We have a matching if the degree of every node is smaller than two. The proof of this fact is non-trivial and requires the use of ghost maps *w* and *b*. As we iterate over the edges of *M*, we build for each node *k* its adjacency list in *w[k]* and record for each edge *f* incident to *k* its position *b[k][f]* in the adjaceny list. The loop invariants guarantee that all edges adjacent to node *k* are stored in the sequence *w[0],...,w[deg_in_M[k]]* and that no two edges in this sequence are equal. Thus, in case *deg_in_M[k]* > *1* we have found two adjacent edges: *w[k][0]* and *w[k][1]*. In case *deg_in_M[k]* ≤ *1* we conclude that all edges adjacent to *k* are equal to *w[k][0]*, thus establishing that no two edges share the same node *k*. If *deg_in_M[k]* ≤ *1* holds for all nodes *k*, we easily can conclude that no two edges are adjacent, i.e., that *M* is a matching.

Abstraction. As preparation for the translation to Isabelle/HOL we define the checker predicate without referring to concrete C data structures. We do so by defining pure ghost data types (e.g., graphs) and corresponding abstraction functions i_X, i_Y and i_W. Moreover we prove that our abstraction is sound and complete.

In a naive approach one would put the coupling relation between abstraction and implementation into the data structure invariant. This, however, would make it necessary to discharge the correctness of abstraction during code verification.

Instead we chose to separate the verification of the code and the correctness of abstraction. Listing 4 presents the abstract ghost types, the abstraction functions and the lemmas establishing soundness and completeness of our abstraction. The predicate *W_holds* is derived from *W_bar_holds* by substituting arrays by maps and unsigned numbers by unbounded integers. This gives us the witness predicate $\mathcal{W}$. Using the

```
spec(unsigned w[unsigned][unsigned];) // w[k] ghost list of edges adjacent to k
spec(unsigned b[unsigned][unsigned];) // b[k][f] position in w of edge f adjacent to node k

unsigned∗ deg_in_M = malloc(M→n ∗ sizeof (unsigned));
assume(deg_in_M ≠ NULL); // enough memory available
// zeroing
for (k = 0; k < M→n; k++)
    invariant(∀(mathint l; 0 ≤ l ∧ l < k ⟹ deg_in_M[l] ≡ 0))
    deg_in_M[k] = 0;

for(e=0; e < M→m; e++)
    invariant(M→m ≠ 0 ⟹ ∀(unsigned k, i; w[k][i] < M→m))
    invariant(∀(unsigned k, i; k < M→n ∧ i < deg_in_M[k] ⟹ w[k][i] < e))
    invariant(∀(unsigned k, i, j; k < M→n ∧ j < deg_in_M[k] ∧ i < j ⟹ w[k][i] < w[k][j]))
    invariant(∀(unsigned k; k < M→n ⟹ deg_in_M[k] ≤ e))
    invariant(∀(unsigned k, n; k < M→n ∧ i < deg_in_M[k] ⟹ adj(M→es[w[k][i]], k)))
    invariant(∀(unsigned k, f; k < M→n ∧ f < e ⟹
      (adj(M→es[f], k) ⟺ w[k][b[k][f]] ≡ f ∧ b[k][f] < deg_in_M[k])))
    // further technical invariants are omitted here
{
    spec(w[M→es[e].s][deg_in_M[M→es[e].s]] = e;)
    spec(w[M→es[e].t][deg_in_M[M→es[e].t]] = e;)

    spec(b[M→es[e].t][e] = deg_in_M[M→es[e].t];)
    spec(b[M→es[e].s][e] = deg_in_M[M→es[e].s];)

    deg_in_M[M→es[e].s]++;
    deg_in_M[M→es[e].t]++;
};

// if deg_in_M[k]>1 then we found two adjacent edges
assert(∀(unsigned k; k < M→n ∧ deg_in_M[k] > 1 ⟹ w[k][0] ≠ w[k][1] ∧ eadj(M→es[w[k][0]],
  M→es[w[k][1]])));

// if deg_in_M[k]<2 then all edges adjacent to k are equal to w[k][0]
assert(∀(unsigned k; k < M→n ∧ deg_in_M[k] < 2 ⟹ ∀(unsigned f; f < M→m ∧ adj(M→es[f],k) ⟹ f ≡
  w[k][0])));
```

Listing 3. Extract from code verification. The keyword **assert()** denotes an assertion which guides the prover. Assumptions are denoted by **assume()**.

abstract types we can finally state the overall correctness theorem (where the predicate *spec_invariants* specifies well-defined graphs):

```
spec(ispure bool final_theorem(spec_graph G, spec_graph M, funType osc, funType id)
  ensures(∀(mathint N[mathint][mathint]; ∀(mathint SU[mathint];
    W_holds(true, G, M, osc, id, N, SU) ⟹
      ∀(struct spec_graph M2; ∀(funType id2; spec_invariants(M2) ∧
              is_subgraph(G,M2,id2) ∧ is_matching(M2) ⟹ M2.m ≤ M.m)))));)
```

It states that whenever the checker returns ***true***, the given matching is maximal. Since this theorem is not referencing any C types, it can easily be translated to Isabelle/HOL.

4.2 Formal Proof of Witness Property

We explain the Isabelle proof for the witness property, i.e., Theorem 2. See Figures 4, 5, and 6 for excerpts from it. The formal proof follows the scheme of the informal proof and is split into two main parts.

For $i \geq 2$, let M_i be the edges in M that connect two nodes labeled i and let M_1 be the remaining edges in M. We use the definition of odd-set cover to prove that

```
spec(
  // ghost record types instead of C structs
  struct vcc(record) spec_edge { mathint s; mathint t; };
  struct vcc(record) spec_graph { mathint m; mathint n; spec_edge es[mathint]; };

  typedef mathint funType[mathint];

  // abstraction functions (only declarations)
  ispure spec_graph abs_g(graph∗ G)
  ispure funType abs_fun(unsigned∗ id, unsigned s)

  // abstract postcondition (only declaration)
  ispure bool W_holds(bool checker_output, spec_graph G, spec_graph M, funType osc, funType id,
    mathint N[mathint][mathint], mathint SU[mathint])

  // soundness of abstraction
  ispure void lemma_sound_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
      mathint N[mathint][mathint], mathint SU[mathint])
    requires(wrapped(G) ∧ wrapped(M))
    requires(W_bar_holds(true, G, M, osc, id, N, SU))
    ensures(W_holds(true, abs_g(G), abs_g(M), abs_fun(osc, G→n), abs_fun(id, M→m), N, SU)) {};

  // completeness of abstraction
  ispure void lemma_complete_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
      mathint N[mathint][mathint], funType SU)
    requires(wrapped(G) ∧ wrapped(M))
    requires(W_holds(true, abs_g(G), abs_g(M), abs_fun(osc, G→n), abs_fun(id, M→m), N, SU))
    ensures(W_bar_holds(true, G, M, osc, id, N, SU)) {};
)
```

Listing 4. Abstraction of postcondition. A ghost VCC record is declared with the keyword **vcc(record)**.

$M \subseteq \bigcup_{i \geq 1} M_i$ and thus $|M| \leq \sum_i |M_i|$. Let V_i be the nodes labeled i and let $n_i = |V_i|$. We formally prove: $|M_1| \leq n_1$ and $|M_i| \leq \lfloor n_i/2 \rfloor$.

In order to prove $|M_1| \leq n_1$, we exhibit an injective function from M_1 to V_1. We first prove, using the definition of odd-set cover, that every edge $e \in M_1$ has at least one endpoint in V_1. This gives rise to a function $\mathit{endpoint}_{V_1} : M_1 \mapsto V_1$. We then use the fact that edges in a matching do not share endpoints, i.e., are disjoint when interpreted as sets, to conclude that $\mathit{endpoint}_{V_1}$ is injective. This establishes $|M_1| \leq |V_i|$.

For $i \geq 2$ the proof of the inequality $|M_i| \leq \lfloor n_i/2 \rfloor$ is similar, but more involved. M_i is a set of edges. If we represent edges as sets (each has cardinality equals two), then M_i is a collection of sets. We define the set of vertices V_i' to be $\bigcup M_i$ and use the definition of odd-set cover to prove $V_i' \subseteq V_i$. Then, we use the fact that the edges in a matching are pairwise disjoint to prove $|V_i'| = 2 * |M_i|$. Note also that $|V_i'|$ must be even since $|M_i|$ is a natural number. Thus we can prove that $|M_i| \leq \lfloor |V_i'| /2 \rfloor$ and hence $|M_i| \leq \lfloor |V_i'| /2 \rfloor \leq \lfloor |V_i| /2 \rfloor = \lfloor n_i/2 \rfloor$.

4.3 Linking VCC and Isabelle

We have extended VCC to export purely mathematical specifications as Isabelle theories, essentially a syntactic rewriting. More precisely, VCC ghost records are translated into Isabelle records, and pure VCC ghost functions are translated into Isabelle function definitions. The former is sound and complete because the semantics of records is the

types vertex = nat **types** label = nat **types** edge = (vertex × vertex)

definition finite-graph :: vertex set => edge set ⇒ bool **where**

finite-graph V E ≡ finite V ∧ finite E ∧ (∀ e ∈ E. fst e ∈ V ∧ snd e ∈ V ∧ fst e ≠ snd e)

definition edge-as-set :: edge ⇒ vertex set **where** edge-as-set e ≡ { fst e , snd e }

definition N :: vertex set ⇒ (vertex ⇒ label) ⇒ nat ⇒ nat **where** n_i

N V L i ≡ card {v ∈ V. L v = i}

definition weight:: label set ⇒ (vertex ⇒ nat) ⇒ nat **where** weight LV f ≡ f 1 + (∑ i∈LV. (f i) div 2)

definition OSC :: (vertex ⇒ label) ⇒ edge set ⇒ bool **where**

OSC L E ≡ ∀ e ∈ E. L (fst e) = 1 ∨ L (snd e) = 1 ∨ L (fst e) = L (snd e) ∧ L (fst e) > 1

definition disjoint-edges :: edge ⇒ edge ⇒ bool **where**

disjoint-edges e1 e2 ≡ fst e1 ≠ fst e2 ∧ fst e1 ≠ snd e2 ∧ snd e1 ≠ fst e2 ∧ snd e1 ≠ snd e2

definition matching :: vertex set ⇒ edge set ⇒ edge set ⇒ bool **where** M

matching V E M ≡ M ⊆ E ∧ finite-graph V E ∧ (∀ e1 ∈ M. ∀ e2 ∈ M. e1 ≠ e2 ⟶ disjoint-edges e1 e2)

definition matching-i :: nat ⇒ vertex set ⇒ edge set ⇒ edge set ⇒ (vertex ⇒ label) ⇒ edge set **where** M_i

matching-i i V E M L ≡ {e ∈ M. i=1 ∧ (L (fst e) = i ∨ L (snd e) = i) ∨ i>1 ∧ L (fst e) = i ∧ L (snd e) = i}

definition V-i:: nat ⇒ vertex set ⇒ edge set ⇒ edge set ⇒ (vertex ⇒ label) ⇒ vertex set **where** V_i'

V-i i V E M L ≡ ⋃ edge-as-set ‘ matching-i i V E M L

definition endpoint-inV :: vertex set ⇒ edge ⇒ vertex **where**

endpoint-inV V e ≡ if fst e ∈ V then fst e else snd e

Fig. 4. Excerpt from the Isabelle proof: Definitions

same in both systems; the latter is sound and complete as we embed VCC's second-order logic into the stronger higher-order logic of Isabelle/HOL. Thereby, VCC's specification types (***bool***, ***mathint***, and map types) are mapped to equivalent Isabelle types (bool, int, and function types). Expressions of VCC comprising logical connectives, quantifiers, integer arithmetic operations, and specification functions are mapped to equivalent Isabelle terms.

Bridging the gap between the rather low-level definitions stemming from VCC and the high-level definitions from the formalization is straightforward and in large parts automatic, except for a number of cumbersome issues: (1) The VCC specification enforces a fixed numbering scheme of vertices and edges, whereas the Isabelle formalization has no such restriction – vertices are arbitrary natural numbers and edges are modelled as sets of vertices. (2) Edges of a graph and a matching in VCC do not necessarily need to be indexed by the same number, whereas in Isabelle/HOL we model a matching as a subset of a graph (which is simply a set of edges). (3) In the VCC specification, edges of a matching are not required to have the same representation as edges in the corresponding graph, i.e., sink and target vertices may be swapped. This cannot be the case in the Isabelle formalization due to the subset relationship between matchings and graphs. (4) Moreover, we require two inductive arguments for relating the VCC ghost functions *N* and *SU* with the definition of weight in Isabelle.

4.4 Evaluation

The checker in LEDA does not verify that M is a subgraph of G. This was revealed by the formalization.

The matching algorithm for general graphs and its efficient implementation is an advanced topic in graph algorithms. It is a highly non-trivial algorithm which is not covered in the standard textbooks on algorithms. The following page numbers illustrate the complexity gap between the original algorithm and the checker: In the LEDA book,

lemma card-M1-le-NVL1:
assumes matching V E M
assumes OSC L E
shows card (matching-i 1 V E M L) $\leq$ (N V L 1) $|M_1| \leq n_1$

lemma card-Mi-twice-card-Vi:
assumes OSC L E $\wedge$ matching V E M $\wedge$ i $>$ 1
shows 2 $*$ card (matching-i i V E M L) $=$ card (V-i i V E M L) $2 * |M_i| = |V_i'|$

lemma card-Mi-le-floor-div-2-Vi:
assumes OSC L E $\wedge$ matching V E M $\wedge$ i $>$ 1
shows card (matching-i i V E M L) $\leq$ (card (V-i i V E M L)) div 2 $|M_i| \leq \lfloor |V_i'| /2 \rfloor$

lemma card-Vi-le-NVLi:
assumes i$>$1 $\wedge$ matching V E M
shows card (V-i i V E M L) $\leq$ N V L i $|V_i'| \leq n_i$

lemma card-Mi-le-floor-div-2-NVLi:
assumes OSC L E $\wedge$ matching V E M $\wedge$ i $>$ 1
shows card (matching-i i V E M L) $\leq$ (N V L i) div 2 $|M_i| \leq \lfloor n_i/2 \rfloor$

lemma card-M-le-sum-card-Mi:
assumes matching V E M **and** OSC L E
shows card M $\leq$ ($\sum$ i $\in$ L'V. card (matching-i i V E M L)) $|M| \leq \sum_{i \in LV} |M_i|$

theorem card-M-le-weight-NVLi:
assumes matching V E M **and** OSC L E
shows card M $\leq$ weight {i $\in$ L ' V. i $>$ 1} (N V L) $|M| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$

theorem maximum-cardinality-matching:
assumes matching V E M **and** OSC L E
and card M $=$ weight {i $\in$ L ' V. i $>$ 1} (N V L)
shows matching V E M$'$ $\longrightarrow$ card M$'$ $\leq$ card M Witness Property (Theorem 2)

Fig. 5. Excerpt from the Isabelle proof: Lemmas and Theorems

the description of the algorithm for computing the maximum cardinality matching and the proof of its correctness takes ca. 15 pages, compared to a one page description of the checker implementation and a few corresponding proof lines.

All described theorems and lemmas have been formally verified, using either VCC or Isabelle/HOL. The C code of the checker, without annotations, spans 102 lines, including empty lines and sparse comments. The specification and verification adds another 318 lines for code and 245 lines for abstraction correctness. This results in a ratio of ca. 2.4 for the annotation overhead due to code verification. Overall proof time is less than 1 minute on one core of a 2.66 GHz Intel Core Duo machine.

The Isabelle theories bring in additional 632 lines of declarations and proofs for 28 lemmas and theorems. More than half of the Isabelle theories are concerned with the witness theorem (Theorem 2) and the rest links this theorem with the abstract specification exported from VCC.

It took several months to develop the framework and to do the first example. Follow-up verifications will benefit from this framework.

5 Related Work

The notion of a certifying algorithm is ancient. Already al-Khawarizmi in his book on algebra described how to (partially) check the correctness of a multiplication. The extended Euclidean algorithm for greatest common divisors is also certifying; it goes back to the 17th century. Yet, formal verification of a checker for certificates has not seen many instances so far.

```
lemma injectivity:
 assumes is-osc: OSC L E
 assumes is-m: matching V E M
 assumes e1-in-M1: e1 ∈ matching-i 1 V E M L
   and e2-in-M1: e2 ∈ matching-i 1 V E M L
 assumes diff: (e1 ≠ e2)
 shows endpoint-inV {v ∈ V. L v = 1} e1 ≠ endpoint-inV {v ∈ V. L v = 1} e2
proof –
 from e1-in-M1 have e1 ∈ M by (auto simp add: matching-i-def)
 moreover
 from e2-in-M1 have e2 ∈ M by (auto simp add: matching-i-def)
 ultimately
 have disjoint-edge-sets: edge-as-set e1 ∩ edge-as-set e2 = {}
   using diff is-m matching-disjointness by fast
 then show ?thesis by (auto simp add: edge-as-set-def endpoint-inV-def)
qed
```

Fig. 6. Excerpt from the Isabelle proof: Proof of an injectivity lemma

Bulwahn et al [5] describe a verified SAT checker, i.e., a checker for certificates of unsatisfiability produced by a SAT solver. They develop and prove correct the checker within Isabelle/HOL. Similar proof checkers have been formalized in the Coq proof assistant [8,1]. CeTA [19], a tool for certified termination analysis, is also based on formally verified checkers, done in Isabelle/HOL. As opposed to our approach, all mentioned checkers are entirely developed and verified within the language of a theorem prover. This is acceptable when extending the capabilities of a theorem prover, but it is unsuitable for verifying checkers of algorithm implementations in C or similar languages.

Integrating powerful interactive theorem provers as backends to code verification systems has been exercised for VCC and Boogie with Isabelle/HOL as backend [4] as well as for Why with a Coq backend [10]. Both systems have a C verifier frontend. Usually, such approaches for connecting code verifiers and proof assistants give the latter the same information that is made available to the first-order engine, overwhelming the users of the proof assistants with a mass of detail. Instead we allow only clean chunks of mathematics to move between the verifier and the proof assistant. This hides from the proof assistant details of the underlying programming language, thus, requiring the user only to discharge interesting proof obligations.

6 Conclusion and Future Work

We described a framework for verification of certifying computations and applied it to a non-trivial combinatorial problem: maximum cardinality matchings in graphs. Our work lifts the reliability of LEDA's maximum matching algorithm to a new level. For each instance of the maximum matching problem, we can now give a formal proof of the correctness of the result. Thus, the user has neither to trust the implementation of the original algorithm or the checker, nor does he have to understand why the witness property holds. We stress that we did not prove the correctness of the program, but only verify the result of the computation.

Our approach applies to any problem for which a certifying algorithm is known; see [13] for a survey. Most algorithms in LEDA [14] are certifying and, in future work, we plan to verify all of them. The checkers and the proof of the witness properties for all other graph algorithms in LEDA are simpler than the presented one and hence will proceed analogously.

Our framework is not only applicable to verifying certifying computations. The integration of VCC and Isabelle/HOL should be useful whenever verification of a program requires non-trivial mathematical reasoning.

Acknowledgment. We thank Ernie Cohen for his advice on VCC idioms and Norbert Schirmer for his initial Isabelle support.

References

1. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)
2. Barnett, M., Chang, B.Y.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 (2006)
3. Blum, M., Kannan, S.: Designing programs that check their work. In: Proceedings of the 21th Annual ACM Symposium on Theory of Computing (STOC 1989), pp. 86–97 (1989)
4. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. J. Automated Reasoning 44(1-2), 111–144 (2010)
5. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
6. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
7. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)
8. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 260–274. Springer, Heidelberg (2010)
9. Edmonds, J.: Maximum matching and a polyhedron with 0,1 - vertices. Journal of Research of the National Bureau of Standards 69B, 125–130 (1965)
10. Filliâtre, J.-C., Marché, C.: The why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979)
12. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems, Real-Time Safety Critical Systems Series, vol. 2, ch. 3, pp. 49–70. Elsevier, Amsterdam (1994)
13. McConnell, R., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Revie (2010) (in Press, Corrected Proof)
14. Mehlhorn, K., Näher, S.: The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
15. Microsoft Corp.: VCC: A C Verifier (2009), `http://vcc.codeplex.com/`

16. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Sullivan, G., Masson, G.: Using certification trails to achieve software fault tolerance. In: Randell, B. (ed.) Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS 1990), pp. 423–433. IEEE, Los Alamitos (1990)
19. Thiemann, R., Sternagel, C.: Certification of termination proofs using ceTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)
20. Verisoft XT, `http://www.verisoft.de/index_en.html`

Parameter Identification for Markov Models of Biochemical Reactions

Aleksandr Andreychenko, Linar Mikeev, David Spieler, and Verena Wolf

Saarland University, Saarbrücken, Germany

Abstract. We propose a numerical technique for parameter inference in Markov models of biological processes. Based on time-series data of a process we estimate the kinetic rate constants by maximizing the likelihood of the data. The computation of the likelihood relies on a dynamic abstraction of the discrete state space of the Markov model which successfully mitigates the problem of state space largeness. We compare two variants of our method to state-of-the-art, recently published methods and demonstrate their usefulness and efficiency on several case studies from systems biology.

1 Introduction

A widely-used strategy in systems biology research is to refine mathematical models of biological processes based on both computer simulations and wet-lab experiments. Therefore, in this area parameter estimation methods for quantitative models play a major role. In other domains, such as probabilistic model checking, similar problems of parameter synthesis occur [8] but aim at finding parameters for which certain temporal properties are satisfied. In systems biology, time series data is analyzed to learn the structure of a biochemical reaction network and to calibrate the reaction rate parameters. Direct measurement of parameters through wet-lab experiments is often difficult or even impracticable. There are extensive research efforts to estimate the reaction rate parameters of ordinary differential equations (ODEs) that describe the evolution of the chemical concentrations over time (see, for instance, [1, 4, 5] and the references therein). The problem of finding parameters that minimize the difference between observed and predicted data is usually multimodal due to non-linear constraints and thus requires global optimization techniques.

The assumption that chemical concentrations change deterministically and continuously in time is not always appropriate for biological processes. In particular, if certain substances in the cell are present in small concentrations the resulting stochastic effects cannot be adequately described by deterministic models. In that case, discrete-state stochastic models are advantageous because they take into account the discrete random nature of chemical reactions. The theory of stochastic chemical kinetics provides a rigorously justified framework for the description of chemical reactions where the effects of molecular noise are taken into account [6]. It is based on discrete-state Markov processes that explicitly represent the reactions as state-transitions between population vectors. When the molecule numbers are large, the solution of the ODE description of a reaction network and the mean of the corresponding stochastic model agree up to

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 83–98, 2011.

a small approximation error. If, however, small populations are involved, then only a stochastic description can provide probabilities of events of interest such as probabilities of switching between different expression states in gene regulatory networks or the distribution of gene expression products. Moreover, even the mean behavior of the stochastic model can largely deviate from the behavior of the deterministic model [13]. In such cases the parameters of the stochastic model rather then the parameters of the deterministic model have to be estimated [16, 18, 20].

Here, we consider noisy time series measurements of the system state as they are available from wet-lab experiments. Recent experimental imaging techniques such as high-resolution fluorescence microscopy can measure small molecule counts with measurement errors of less than one molecule [7]. We assume that the structure of the underlying reaction network is known but the rate parameters of the network are unknown. Then we identify those parameters that maximize the likelihood of the time series data.

Our main contribution consists in devising an efficient algorithm for the numerical approximation of the likelihood and its derivatives w.r.t. the reaction rate constants. Previous techniques are based on Monte-Carlo sampling [18, 20] because the discrete state space of the underlying model is typically infinite in several dimensions and a priori a reasonable truncation of the state space is not availabe. Our method is not based on sampling but directly calculates the likelihood using a dynamic truncation of the state space. More precisely, we first show that the computation of the likelihood is equivalent to the evaluation of a product of vectors and matrices. This product includes the transition probability matrix of the associated continuous-time Markov process, i.e., the solution of the Kolmogorov differential equations (KDEs). Since solving the KDEs is infeasible, we propose two iterative approximation algorithms during which the state space is truncated in an on-the-fly fashion, that is, during a certain time interval we consider only those states that significantly contribute to the likelihood. One approach exploits equidistant observation intervals while the other approach is particularly well suited for observation intervals that are not equidistant. Both approaches take into account measurement noise during the observations.

After introducing the stochastic model in Section 2, we discuss dynamic state space truncations for the transient probability distribution and its derivatives in Section 3. We introduce the maximum likelihood method in Section 4 and present the approximation methods in Section 5. Finally, we report on experimental results for two reaction networks (Section 6) and discuss related work in Section 7.

2 Discrete-State Stochastic Model

According to Gillespie's theory of stochastic chemical kinetics, a well-stirred mixture of n molecular species in a volume with fixed size and fixed temperature can be represented as a continuous-time Markov chain $\{\mathbf{X}(t), t \geq 0\}$ [6]. The random vector $\mathbf{X}(t) = (X_1(t), \ldots, X_n(t))$ describes the chemical populations at time t, i.e., $X_i(t)$ is the number of molecules of type $i \in \{1, \ldots, n\}$ at time t. Thus, the state space of $\mathbf{X}$ is $\mathbb{Z}_+^n = \{0, 1, \ldots\}^n$. The state changes of $\mathbf{X}$ are triggered by the occurrences of chemical reactions, which are of m different types. For $j \in \{1, \ldots, m\}$ let $\mathbf{v}_j \in \mathbb{Z}^n$ be the nonzero *change vector* of the j-th reaction type, that is, $\mathbf{v}_j = \mathbf{v}_j^- + \mathbf{v}_j^+$ where $\mathbf{v}_j^-$ contains only non-positive entries, which specify how many molecules of each species are consumed (*reactants*) if an instance of the reaction occurs. The vector $\mathbf{v}_j^+$ contains only non-negative entries, which specify how many molecules of each species are produced (*products*). Thus, if $\mathbf{X}(t) = \mathbf{x}$ for some $\mathbf{x} \in \mathbb{Z}_+^n$ with $\mathbf{x} + \mathbf{v}_j^-$ being non-negative, then $\mathbf{X}(t + dt) = \mathbf{x} + \mathbf{v}_j$ is the state of the system after the occurrence of the j-th reaction within the infinitesimal time interval $[t, t + dt)$.

Each reaction type has an associated *propensity function*, denoted by $\alpha_1, \ldots, \alpha_m$, which is such that $\alpha_j(\mathbf{x}) \cdot dt$ is the probability that, given $\mathbf{X}(t) = \mathbf{x}$, one instance of the j-th reaction occurs within $[t, t + dt)$. The value $\alpha_j(\mathbf{x})$ is proportional to the number of distinct reactant combinations in state $\mathbf{x}$. More precisely, if $\mathbf{x} = (x_1, \ldots, x_n)$ is a state for which $\mathbf{x} + \mathbf{v}_j^-$ is nonnegative then, for reactions with at most two reactants,

$$\alpha_j(\mathbf{x}) = \begin{cases} c_j & \text{if } \mathbf{v}_j^- = (0, \ldots, 0), \\ c_j \cdot x_i & \text{if } \mathbf{v}_j^- = -\mathbf{e}_i, \\ c_j \cdot x_i \cdot x_\ell & \text{if } \mathbf{v}_j^- = -\mathbf{e}_i - \mathbf{e}_\ell, \\ c_j \cdot \binom{x_i}{2} = c_j \cdot \frac{x_i \cdot (x_i - 1)}{2} & \text{if } \mathbf{v}_j^- = -2 \cdot \mathbf{e}_i, \end{cases} \tag{1}$$

where $i \neq \ell$, $c_j > 0$, and $\mathbf{e}_i$ is the vector with the i-th entry 1 and all other entries 0.

Example 1. We consider the simple gene expression model described in [16] that involves three chemical species, namely DNA$_{\text{ON}}$, DNA$_{\text{OFF}}$, and mRNA, which are represented by the random variables $X_1(t)$, $X_2(t)$, and $X_3(t)$, respectively. The three possible reactions are DNA$_{\text{ON}} \rightarrow$ DNA$_{\text{OFF}}$, DNA$_{\text{OFF}} \rightarrow$ DNA$_{\text{ON}}$, and DNA$_{\text{ON}} \rightarrow$ DNA$_{\text{ON}}+$ mRNA. Thus, $\mathbf{v}_1^- = (-1, 0, 0)$, $\mathbf{v}_1^+ = (0, 1, 0)$, $\mathbf{v}_2^- = (0, -1, 0)$, $\mathbf{v}_2^+ = (1, 0, 0)$, $\mathbf{v}_3^- = (-1, 0, 0)$ and $\mathbf{v}_3^+ = (1, 0, 1)$. For a state $\mathbf{x} = (x_1, x_2, x_3)$, the propensity functions are $\alpha_1(\mathbf{x}) = c_1 \cdot x_1$, $\alpha_2(\mathbf{x}) = c_2 \cdot x_2$, and $\alpha_3(\mathbf{x}) = c_3 \cdot x_1$. Note that given the initial state $\mathbf{x} = (1, 0, 0)$, at any time, either the DNA is active or not, i.e. $x_1 = 0$ and $x_2 = 1$, or $x_1 = 1$ and $x_2 = 0$. Moreover, the state space of the model is infinite in the third dimension. For a fixed time instant $t > 0$, no upper bound on the number of mRNA is known a priori. All states $\mathbf{x}$ with $x_3 \in \mathbb{Z}_+$ have positive probability if $t > 0$ but these probabilities will tend to zero as $x_3 \rightarrow \infty$.

In general, the reaction rate constants c_j refer to the probability that a randomly selected pair of reactants collides and undergoes the j-th chemical reaction. It depends on the volume and the temperature of the system as well as on the microphysical properties of the reactant species. Since reactions of higher order (requiring more than two reactants) are usually the result of several successive lower order reactions, we do not consider the case of more than two reactants.

The Chemical Master Equation. For $\mathbf{x} \in \mathbb{Z}_+^n$ and $t \geq 0$, let $p(\mathbf{x}, t)$ denote the probability $Pr(\mathbf{X}(t) = \mathbf{x})$ and let $\mathbf{p}(t)$ be the row vector with entries $p(\mathbf{x}, t)$.

Given $\mathbf{v}_1^-, \ldots, \mathbf{v}_m^-, \mathbf{v}_1^+, \ldots, \mathbf{v}_m^+, \alpha_1, \ldots, \alpha_m$, and some initial distribution $\mathbf{p}(0)$, the Markov chain $\mathbf{X}$ is uniquely specified and its evolution is given by the chemical master equation (CME)

$$\tfrac{d}{dt}\mathbf{p}(t) = \mathbf{p}(t)Q, \tag{2}$$

where Q is the infinitesimal generator matrix of $\mathbf{X}$ with $Q(\mathbf{x}, \mathbf{y}) = \alpha_j(\mathbf{x})$ if $\mathbf{y} = \mathbf{x} + \mathbf{v}_j$ and $\mathbf{x} + \mathbf{v}_j^- \geq 0$. Note that, in order to simplify our presentation, we assume here that all vectors $\mathbf{v}_j$ are distinct. All remaining entries of Q are zero except for the diagonal entries which are equal to the negative row sum. The ordinary first-order differential equation in (2) is a direct consequence of the Kolmogorov forward equation. Since $\mathbf{X}$ is a regular Markov process, (2) has the general solution $\mathbf{p}(t) = \mathbf{p}(0) \cdot e^{Qt}$, where e^{A} is the matrix exponential of a matrix A. If the state space of X is infinite, then we can only compute approximations of $\mathbf{p}(t)$. But even if Q is finite, its size is often large because it grows exponentially with the number of state variables. Therefore standard numerical solution techniques for systems of first-order linear equations of the form of (2) are infeasible. The reason is that the number of nonzero entries in Q often exceeds the available memory capacity for systems of realistic size. If the populations of all species remain small (at most a few hundreds) then the CME can be efficiently approximated using projection methods [3, 10, 15] or fast uniformization methods [14, 17]. The idea of these methods is to avoid an exhaustive state space exploration and, depending on a certain time interval, restrict the analysis of the system to a subset of states.

Here, we are interested in the partial derivatives of $\mathbf{p}(t)$ w.r.t. the reaction rate constants $\mathbf{c} = (c_1, \ldots, c_m)$. In order to explicitly indicate the dependence of $\mathbf{p}(t)$ on the vector $\mathbf{c}$ we write $\mathbf{p}(\mathbf{c}, t)$ instead of $\mathbf{p}(t)$ and $p(\mathbf{x}, \mathbf{c}, t)$ instead of $p(\mathbf{x}, t)$ if necessary. We define the row vectors $\mathbf{s}_j(\mathbf{c}, t)$ as the derivative of $\mathbf{p}(\mathbf{c}, t)$ w.r.t. c_j, i.e.,

$$\mathbf{s}_j(\mathbf{c}, t) = \tfrac{\partial \mathbf{p}(\mathbf{c},t)}{\partial c_j} = \lim_{\Delta c \to 0} \tfrac{\mathbf{p}(\mathbf{c}+\mathbf{\Delta c}_j, t) - \mathbf{p}(\mathbf{c},t)}{\Delta c},$$

where the vector $\mathbf{\Delta c}_j$ is zero everywhere except for the j-th position that is equal to Δc. We denote the entry in $\mathbf{s}_j(\mathbf{c}, t)$ that corresponds to state $\mathbf{x}$ by $s_j(\mathbf{x}, \mathbf{c}, t)$. Using (2), we find that $\mathbf{s}_j(\mathbf{c}, t)$ is the unique solution of the system of ODEs

$$\tfrac{d}{dt}\mathbf{s}_j(\mathbf{c}, t) = \mathbf{s}_j(\mathbf{c}, t)Q + \mathbf{p}(\mathbf{c}, t)\tfrac{\partial}{\partial c_j}Q, \tag{3}$$

where $j \in \{1, \ldots, m\}$. The initial condition is $s_j(\mathbf{x}, \mathbf{c}, 0) = 0$ for all $\mathbf{x}$ and $\mathbf{c}$ since $p(\mathbf{x}, \mathbf{c}, 0)$ is independent of c_j.

3 Dynamic State Space Truncation

The parameter estimation method that we propose in Section 5.1 builds on the approximation of the transient distribution $\mathbf{p}(t)$ and the derivatives $\mathbf{s}_j(\mathbf{c}, t)$ for all j at a fixed time instant $t > 0$. Therefore we now discuss how to solve (2) and (3) simultaneously using an explicit fourth-order Runge-Kutta method and a dynamically truncated state space. This truncation is necessary because models of chemical reaction networks typically have a very large or infinite number of states $\mathbf{x}$ with nonzero values for $p(\mathbf{x}, t)$ and $s_j(\mathbf{x}, \mathbf{c}, t)$. For instance, the system in Example 1 is infinite in one dimension. In order to keep the number of states, that are considered in a certain step of the numerical integration, manageable we suggest a dynamic truncation of the state space that, for a given time interval, neglects those states being not relevant during that time, that is, we neglect states that have a probability that is smaller than a certain threshold.

First, we remark that the equation that corresponds to state $\mathbf{x}$ in (2) is given by

$$\tfrac{d}{dt}p(\mathbf{x}, t) = \textstyle\sum_{j:\mathbf{x}-\mathbf{v}_j^- \geq 0} \alpha_j(\mathbf{x} - \mathbf{v}_j)p(\mathbf{x} - \mathbf{v}_j, t) - \alpha_j(\mathbf{x})p(\mathbf{x}, t). \tag{4}$$

and it describes the change of the probability of state $\mathbf{x}$ as the difference between inflow of probability at rate $\alpha_j(\mathbf{x}-\mathbf{v}_j)$ from direct predecessors $\mathbf{x}-\mathbf{v}_j$ and outflow of probability at rate $\alpha_j(\mathbf{x})$. Assume now that an initial distribution $\mathbf{p}(0)$ is given. We choose a small positive constant δ and define the set of significant states $S = \{\mathbf{x} \mid p(\mathbf{x}, 0) > \delta\}$. We then only integrate equations in (2) and (3) that belong to states in S. If h is the time step of the numerical integration, then for the interval $[t, t+h)$ we use the following strategy to modify S according to the probability flow. We check for all successors $\mathbf{x}+\mathbf{v}_j \notin S$ of a state $\mathbf{x} \in S$ whether $p(\mathbf{x}+\mathbf{v}_j, t+h)$ becomes greater than δ at time $t+h$ as they receive "inflow" from their direct predecessors (see Eq. (4)). If the probability that $\mathbf{x}+\mathbf{v}_j$ receives is greater δ, then we add $\mathbf{x}+\mathbf{v}_j$ to S. Note that since we use a fourth-order method, states reachable within at most four transitions from a state in S can be added during one step of the integration. On the other hand, whenever $p(\mathbf{x}, t)$ becomes less or equal to δ for a state $\mathbf{x} \in S$ then we remove $\mathbf{x}$ from S. We approximate the probabilities and derivatives of all states that are not considered during $[t, t+h)$ with zero. In this way the computational costs of the numerical integration is drastically reduced, since typically the number of states with probabilities less than δ is large and the main part of the probability mass is concentrated on a small number of significant states. Due to the regular structure of $\mathbf{X}$, the probability of a state decreases exponentially with its distance to the "high probability" locations. If δ is small (e.g. 10^{-15}) and the initial distribution is such that the main part of the probability mass (e.g. 99.99%) distributes on a manageable number of states, then even for long time horizons the approximation of the transient distribution is accurate. For arbitrary Markov models, the approximation error of the derivatives could, in principle, be large. For biochemical reaction networks, however, the underlying Markov process is well-structured and the sensitivity of the transient distribution w.r.t. the rate constants is comparatively small, i.e., small changes of the rate constants result in a transient distribution that differs only slightly from the original distribution. Therefore, the derivatives of insignificant states are small and, in order to calibrate parameters, it is sufficient to consider the derivatives of probabilities of significant states. It is impossible to explore the whole state space and those parts containing most of the probability mass are most informative w.r.t. pertubations of the rate constants.

Example 2. We consider a simple enzyme reaction with three reactions that involve four different species, namely enzymes (E), substrates (S), complex molecules (C), and proteins (P). The reactions are complex formation (E+S⟶ C), dissociation of the complex (C⟶E+S), and protein production (C⟶E+P). The corresponding rate functions are $\alpha_1(\mathbf{x}) = c_1 \cdot x_1 \cdot x_2$, $\alpha_2(\mathbf{x}) = c_2 \cdot x_3$, and $\alpha_3(\mathbf{x}) = c_3 \cdot x_3$ where $\mathbf{x} = (x_1, x_2, x_3, x_4)$. The change vectors are given by $\mathbf{v}_1^- = (-1, -1, 0, 0)$, $\mathbf{v}_1^+ = (0, 0, 1, 0)$, $\mathbf{v}_2^- = (0, 0, -1, 0)$, $\mathbf{v}_2^+ = (1, 1, 0, 0)$, $\mathbf{v}_3^- = (0, 0, -1, 0)$, and $\mathbf{v}_3^+ = (1, 0, 0, 1)$. We start initially with probability one in state $\mathbf{x} = (1000, 200, 0, 0)$ and compute $\mathbf{p}(t)$ and $\mathbf{s}_j(\mathbf{c}, t)$ for $t = 10$, $\mathbf{c} = (1, 1, 0.1)$, and $j \in \{1, 2, 3\}$. In Table 1 we list the results of the approximation of $\mathbf{p}(t)$ and $\mathbf{s}_j(\mathbf{c}, t)$. We chose this model because it has a finite state space and we can compare our approximation with the values obtained for $\delta = 0$. The column "Time" lists the running times of the computation. Obviously, the smaller δ the more time consuming is the computation. The remaining columns refer to the maximum absolute error of all entries in the vectors $\mathbf{p}(t)$ and $\mathbf{s}_j(\mathbf{c}, t)$ where we use as exact values

Table 1. Approximated transient distribution and derivatives of the enzyme reaction network

δ	Time	Maximum absolute error			
		$\mathbf{p}(t)$	$\mathbf{s}_1(\mathbf{c},t)$	$\mathbf{s}_2(\mathbf{c},t)$	$\mathbf{s}_3(\mathbf{c},t)$
0	10 h	0	0	0	0
10^{-20}	47 sec	$1 \cdot 10^{-11}$	$1 \cdot 10^{-12}$	$1 \cdot 10^{-12}$	$4 \cdot 10^{-9}$
10^{-15}	25 sec	$1 \cdot 10^{-11}$	$8 \cdot 10^{-11}$	$9 \cdot 10^{-11}$	$2 \cdot 10^{-8}$
10^{-10}	10 sec	$7 \cdot 10^{-7}$	$3 \cdot 10^{-6}$	$4 \cdot 10^{-6}$	$2 \cdot 10^{-4}$

those obtained by setting $\delta = 0$. Clearly, even if $\delta = 0$ we have an approximation error due to the numerical integration of (2) and (3), which is, however, very small compared to the error that originates from the truncation of the state space.

A similar truncation effect can be obtained by sorting the entries of $\mathbf{p}(t)$ and successively removing the smallest entries until a fixed amount ε of probability mass is lost. If ε is chosen proportional to the time step, then it is possible to bound the total approximation error of the probabilities, i.e., $\varepsilon = \tilde{\varepsilon}h/t$ where $\tilde{\varepsilon}$ is the total approximation error for a time horizon of length t. If memory requirements and running time are more pressing then accuracy, then we can adjust the computational costs of the approximation by keeping only the k most probable states in each step for some integer k.

4 Parameter Inference

Following the notation in [16], we assume that observations of a biochemical network are made at time instances $t_1, \ldots, t_R \in \mathbb{R}_{\geq 0}$ where $t_1 < \ldots < t_R$. Moreover, we assume that $O_i(t_\ell)$ is the observed number of species i at time t_ℓ for $i \in \{1, \ldots, n\}$ and $\ell \in \{1, \ldots, R\}$. Let $\mathbf{O}(t_\ell) = (O_1(t_\ell), \ldots, O_n(t_\ell))$ be the corresponding vector of observations. Since these observations are typically subject to measurement errors, we assume that $O_i(t_\ell) = X_i(t_\ell) + \epsilon_i(t_\ell)$ where the error terms $\epsilon_i(t_\ell)$ are independent and identically normally distributed with mean zero and standard deviation σ. Note that $X_i(t_\ell)$ is the true population of the i-th species at time t_ℓ. Clearly, this implies that, conditional on $X_i(t_\ell)$, the random variable $O_i(t_\ell)$ is independent of all other observations as well as independent of the history of $\mathbf{X}$ before time t_ℓ.

We assume further that for the unobserved process $\mathbf{X}$ we do not know the values of the rate constants $c_1, \ldots, c_m$ and our aim is to estimate these constants. Similarly, the exact standard deviation σ of the error terms is unknown and must be estimated[1]. Let f denote the joint density of $\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R)$. Then the likelihood of the observations is [12]

$$\begin{aligned}\mathcal{L} &= f\left(\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R)\right) \\ &= \textstyle\sum_{\mathbf{x}_1} \cdots \sum_{\mathbf{x}_R} f\left(\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R) \mid \mathbf{X}(t_1) = \mathbf{x}_1, \ldots, \mathbf{X}(t_R) = \mathbf{x}_R\right) \qquad (5) \\ &\quad\; Pr\left(\mathbf{X}(t_1) = \mathbf{x}_1, \ldots, \mathbf{X}(t_R) = \mathbf{x}_R\right),\end{aligned}$$

[1] We remark that it is straightforward to extend the estimation framework that we present in the sequel such that a covariance matrix for a multivariate normal distribution of the error terms is estimated. In this way, different measurement errors of the species can be taken into account as well as dependencies between error terms.

that is, $\mathcal{L}$ is the probability to observe $\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R)$. Note that $\mathcal{L}$ depends on the chosen rate parameters $\mathbf{c}$ since the probability measure $Pr(\cdot)$ does. Furthermore, $\mathcal{L}$ depends on σ since the density f does. When necessary, we will make this dependence explicit by writing $\mathcal{L}(\mathbf{c}, \sigma)$ instead of $\mathcal{L}$. We now seek constants $\mathbf{c}^*$ and a standard deviation σ^* such that

$$\mathcal{L}(\mathbf{c}^*, \sigma^*) = \max_{\sigma, \mathbf{c}} \mathcal{L}(\mathbf{c}, \sigma) \tag{6}$$

where the maximum is taken over all $\sigma > 0$ and vectors $\mathbf{c}$ with all components strictly positive. This optimization problem is known as the maximum likelihood problem [12]. Note that $\mathbf{c}^*$ and σ^* are random variables because they depend on the (random) observations $\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R)$.

If more than one sequence of observations is made, then the corresponding likelihood is the product of the likelihoods of all individual sequences. More precisely, if $\mathbf{O}^k(t_l)$ is the k-th observation that has been observed at time instant t_l where $k \in \{1, \ldots, K\}$, then we define $\mathcal{L}_k(\mathbf{c}, \sigma)$ as the probability to observe $\mathbf{O}^k(t_1), \ldots, \mathbf{O}^k(t_R)$ and maximize

$$\textstyle\prod_{k=1}^{K} \mathcal{L}_k(\mathbf{c}, \sigma). \tag{7}$$

In the sequel, we concentrate on expressions for $\mathcal{L}_k(\mathbf{c}, \sigma)$ and $\frac{\partial}{\partial c_j} \mathcal{L}_k(\mathbf{c}, \sigma)$. We first assume $K = 1$ and drop index k. We consider the case $K > 1$ later. In (5) we sum over all state sequences $\mathbf{x}_1, \ldots, \mathbf{x}_R$ such that $Pr(\mathbf{X}(t_\ell) = \mathbf{x}_\ell, 1 \leq \ell \leq R) > 0$. Since $\mathbf{X}$ has a large or even infinite state space, it is computationally infeasible to explore all possible sequences. In Section 5 we propose an algorithm to approximate the likelihoods and their derivatives. We truncate the state space in a similar way as in Section 3 and use the fact that (5) can be written as a product of vectors and matrices. Let ϕ_σ be the density of the normal distribution with mean zero and standard deviation σ. Then

$$\begin{aligned} &f\left(\mathbf{O}(t_1), \ldots, \mathbf{O}(t_R) \mid \mathbf{X}(t_1) = \mathbf{x}_1, \ldots, \mathbf{X}(t_R) = \mathbf{x}_R\right) \\ &= \textstyle\prod_{\ell=1}^{R} \prod_{i=1}^{n} f\left(O_i(t_\ell) \mid X_i(t_\ell) = x_{i\ell}\right) \\ &= \textstyle\prod_{\ell=1}^{R} \prod_{i=1}^{n} \phi_\sigma(O_i(t_\ell) - x_{i\ell}), \end{aligned}$$

where $\mathbf{x}_\ell = (x_{1\ell}, \ldots, x_{n\ell})$. If we write $w(\mathbf{x}_\ell)$ for $\prod_{i=1}^{n} \phi_\sigma(O_i(t_\ell) - x_{i\ell})$, then the sequence $\mathbf{x}_1, \ldots, \mathbf{x}_R$ has weight $\prod_{\ell=1}^{R} w(\mathbf{x}_\ell)$ and, thus,

$$\mathcal{L} = \sum_{\mathbf{x}_1} \ldots \sum_{\mathbf{x}_R} Pr(\mathbf{X}(t_1) = \mathbf{x}_1, \ldots, \mathbf{X}(t_R) = \mathbf{x}_R) \prod_{\ell=1}^{R} w(\mathbf{x}_\ell). \tag{8}$$

Moreover, for the probability of the sequence $\mathbf{x}_1, \ldots, \mathbf{x}_R$ we have

$$Pr(\mathbf{X}(t_1) = \mathbf{x}_1, \ldots, \mathbf{X}(t_R) = \mathbf{x}_R) = p(\mathbf{x}_1, t_1) P_2(\mathbf{x}_1, \mathbf{x}_2) \cdots P_R(\mathbf{x}_{R-1}, \mathbf{x}_R)$$

where $P_\ell(\mathbf{x}, \mathbf{y}) = Pr(\mathbf{X}(t_\ell) = \mathbf{y} \mid \mathbf{X}(t_{\ell-1}) = \mathbf{x})$. Hence, (8) can be written as

$$\mathcal{L} = \sum_{\mathbf{x}_1} p(\mathbf{x}_1, t_1) w(\mathbf{x}_1) \sum_{\mathbf{x}_2} P_2(\mathbf{x}_1, \mathbf{x}_2) w(\mathbf{x}_2) \ldots \sum_{\mathbf{x}_R} P_R(\mathbf{x}_{R-1}, \mathbf{x}_R) w(\mathbf{x}_R). \tag{9}$$

Let P_ℓ be the matrix with entries $P_\ell(\mathbf{x},\mathbf{y})$ for all states $\mathbf{x},\mathbf{y}$. Note that P_ℓ is the transition probability matrix of $\mathbf{X}$ for time step $t_\ell - t_{\ell-1}$ and thus the general solution $e^{Q(t_\ell - t_{\ell-1})}$ of the Kolmogorov forward and backward differential equations

$$\tfrac{d}{dt}P_\ell = QP_\ell, \qquad\qquad \tfrac{d}{dt}P_\ell = P_\ell Q.$$

Using $\mathbf{p}(t_1) = \mathbf{p}(t_0)P_1$ with $t_0 = 0$, we can write (9) in matrix-vector form as

$$\mathcal{L} = \mathbf{p}(t_0)P_1W_1P_2W_2\cdots P_RW_R\mathbf{e}. \tag{10}$$

Here, $\mathbf{e}$ is the vector with all entries equal to one and W_ℓ is a diagonal matrix whose diagonal entries are all equal to $w(\mathbf{x}_\ell)$ with $\ell \in \{1,\ldots,R\}$, where W_ℓ is of the same size as P_ℓ. Since it is in general not possible to analytically obtain parameters that maximize $\mathcal{L}$, we use optimization techniques to find $\mathbf{c}^*$ and σ^*. Typically, such techniques iterate over values of $\mathbf{c}$ and σ and increase the likelihood $\mathcal{L}(\mathbf{c},\sigma)$ by following the gradient. Therefore, we need to calculate the derivatives $\frac{\partial}{\partial c_j}\mathcal{L}$ and $\frac{\partial}{\partial\sigma}\mathcal{L}$. For $\frac{\partial}{\partial c_j}\mathcal{L}$ we obtain

$$\begin{aligned}\tfrac{\partial}{\partial c_j}\mathcal{L} &= \tfrac{\partial}{\partial c_j}\left(\mathbf{p}(t_0)P_1W_1P_2W_2\cdots P_RW_R\mathbf{e}\right)\\ &= \mathbf{p}(t_0)\left(\sum_{\ell=1}^{R}\left(\tfrac{\partial}{\partial c_j}P_\ell\right)W_\ell\prod_{\ell'\neq\ell}P_{\ell'}W_{\ell'}\right)\mathbf{e}.\end{aligned} \tag{11}$$

The derivative of $\mathcal{L}$ w.r.t. the standard deviation σ is derived analogously. The only difference is that $P_1,\ldots,P_R$ are independent of σ but $W_1,\ldots,W_R$ depend on σ. It is also important to note that expressions for partial derivatives of second order can be derived in a similar way. These derivatives can then be used for an efficient gradient-based local optimization.

For $K > 1$ observation sequences we can maximize the log-likelihood

$$\log\prod_{k=1}^{K}\mathcal{L}_k = \sum_{k=1}^{K}\log\mathcal{L}_k, \tag{12}$$

instead of the likelihood in (7), where we abbreviate $\mathcal{L}_k(\mathbf{c},\sigma)$ by $\mathcal{L}_k$. Note that the derivatives are then given by

$$\tfrac{\partial}{\partial\lambda}\sum_{k=1}^{K}\log\mathcal{L}_k = \sum_{k=1}^{K}\frac{\frac{\partial}{\partial\lambda}\mathcal{L}_k}{\mathcal{L}_k}, \tag{13}$$

where λ is either c_j or σ. It is also important to note that only the weights $w(\mathbf{x}_\ell)$ depend on k, that is, on the observed sequence $\mathbf{O}^k(t_1),\ldots,\mathbf{O}^k(t_R)$. Thus, when we compute $\mathcal{L}_k$ based on (10) we use for all k the same transition matrices $P_1,\ldots,P_R$ and the same initial conditions $\mathbf{p}(t_0)$, but possibly different matrices $W_1,\ldots,W_R$.

5 Numerical Approximation Algorithm

In this section, we focus on the numerical approximation of the likelihood and the corresponding derivatives w.r.t. the rate constants $c_1,\ldots,c_m$. We propose two approximation algorithms for the likelihood and its derivatives, a state-based likelihood approximation (SLA) and a path-based likelihood approximation (PLA). Both are based on a dynamic truncation of the state space as suggested in Section 3. They differ in that the PLA method exploits equidistant time series, that is, it is particularly efficient if $h = t_{\ell+1} - t_\ell$ for all ℓ and if σ is not too large. The SLA algorithm works for arbitrarily spaced time series and is efficient even if σ is large.

5.1 State-Based Likelihood Approximation

The SLA algorithm calculates an approximation of the likelihood based on (10) by traversing the matrix-vector product from the left to the right. The main idea behind the algorithm is that instead of explicitly computing the matrices P_ℓ, we express the vector-matrix product $\mathbf{u}(t_{\ell-1})P_\ell$ as a system of ODEs similar to the CME (cf. Eq. (2)). Here, $\mathbf{u}(t_0), \ldots, \mathbf{u}(t_R)$ are row vectors obtained during the iteration over time points $t_0, \ldots, t_R$, that is, we define $\mathcal{L}$ recursively as $\mathcal{L} = \mathbf{u}(t_R)\mathbf{e}$ with $\mathbf{u}(t_0) = \mathbf{p}(t_0)$ and

$$\mathbf{u}(t_\ell) = \mathbf{u}(t_{\ell-1})P_\ell W_\ell \qquad \text{for all } 1 \le \ell \le R,$$

where $t_0 = 0$. Instead of computing P_ℓ explicitly, we solve R systems of ODEs

$$\tfrac{d}{dt}\tilde{\mathbf{u}}(t) = \tilde{\mathbf{u}}(t)Q \tag{14}$$

with initial condition $\tilde{\mathbf{u}}(t_{\ell-1}) = \mathbf{u}(t_{\ell-1})$ for the time interval $[t_{\ell-1}, t_\ell)$ where $\ell \in \{1, \ldots, R\}$. After solving the ℓ-th system of ODEs we set $\mathbf{u}(t_\ell) = \tilde{\mathbf{u}}(t_\ell)W_\ell$ and finally compute $\mathcal{L} = \mathbf{u}(t_R)\mathbf{e}$. Since this is the same as solving the CME for different initial conditions, we can use the dynamic truncation of the state space proposed in Section 3. Since the vectors $\tilde{\mathbf{u}}(t_\ell)$ do not sum up to one, we scale all entries by multiplication with $1/(\tilde{\mathbf{u}}(t_\ell)\mathbf{e})$. This simplifies the truncation of the state space using the significance threshold δ since after scaling it can be interpreted as a probability. In order to obtain the correct (unscaled) likelihood, we compute $\mathcal{L}$ as $\mathcal{L} = \prod_{\ell=1}^{R}(\tilde{\mathbf{u}}(t_\ell)\mathbf{e})$. We handle the derivatives of $\mathcal{L}$ in a similar way. To shorten our presentation, we only consider the derivative $\frac{\partial}{\partial c_j}\mathcal{L}$ in the sequel. An iterative scheme for $\frac{\partial}{\partial \sigma}\mathcal{L}$ is derived analogously. From (11) we obtain $\frac{\partial}{\partial c_j}\mathcal{L} = \mathbf{u}_j(t_R)\mathbf{e}$ with $\mathbf{u}_j(t_0) = \mathbf{0}$ and

$$\mathbf{u}_j(t_\ell) = (\mathbf{u}_j(t_{\ell-1})P_\ell + \mathbf{u}(t_{\ell-1})\tfrac{\partial}{\partial c_j}P_\ell)W_\ell \qquad \text{for all } 1 \le \ell \le R,$$

where $\mathbf{0}$ is the vector with all entries zero. Thus, during the solution of the ℓ-th ODE in (14) we simultaneously solve

$$\tfrac{d}{dt}\tilde{\mathbf{u}}_j(t) = \tilde{\mathbf{u}}_j(t)Q + \tilde{\mathbf{u}}(t)\tfrac{\partial}{\partial c_j}Q \tag{15}$$

with initial condition $\tilde{\mathbf{u}}_j(t_{\ell-1}) = \mathbf{u}_j(t_{\ell-1})$ for the time interval $[t_{\ell-1}, t_\ell)$. As above, we set $\mathbf{u}_j(t_\ell) = \tilde{\mathbf{u}}_j(t_\ell)W_\ell$ and obtain $\frac{\partial}{\partial c_j}\mathcal{L}$ as $\mathbf{u}_j(t_R)\mathbf{e}$.

Solving (14) and (15) simultaneously is equivalent to the computation of the partial derivatives in (3) with different initial conditions. Thus, we can use the approximation algorithm proposed in Section 3 to approximate $\mathbf{u}_j(t_\ell)$. Experimental results of the finite enzyme reaction network (see Example 2) show that the approximation errors of the likelihood and its derivatives are of the same order of magnitude as those of the transient probabilities and their derivatives (not shown). Note, however, that, if σ is small only few states contribute significantly to the likelihood. In this case, truncation strategies based on sorting of vectors are more efficient without considerable accuracy losses since the main part of the likelihood concentrates on very few entries (namely those that correspond to states that are close to the observed populations).

In the case of K observation sequences we repeat the above algorithm in order to sequentially compute $\mathcal{L}_k$ for $k \in \{1, \ldots, K\}$. We exploit (12) and (13) to compute the total log-likelihood and its derivatives as a sum of individual terms. Obviously, it is possible to parallelize the SLA algorithm by computing $\mathcal{L}_k$ in parallel for all k.

5.2 Path-Based Likelihood Approximation

If $\Delta t = t_\ell - t_{\ell-1}$ for all ℓ then the matrices $P_1, \ldots, P_R$ in (10) are equal to the Δt-step transition matrix $T(\Delta t)$ with entries $Pr(\mathbf{X}(t+\Delta t) = \mathbf{y} \mid \mathbf{X}(t) = \mathbf{x})$. Note that since we consider a time-homogeneous Markov process $\mathbf{X}$, the matrix $T(\Delta t)$ is independent of t. The main idea of the PLA method is to iteratively compute those parts of $T(\Delta t)$ that correspond to state sequences (paths) $\mathbf{x}_1, \ldots, \mathbf{x}_R$ that contribute significantly to $\mathcal{L}$. The algorithm can be summarized as follows, where we omit the argument Δt of T to improve the readability and refer to the entries of T as $T(\mathbf{x}, \mathbf{y})$:

1. We compute the transient distribution $\mathbf{p}(t_1)$ and its derivatives (w.r.t. $\mathbf{c}$ and σ) as outlined in Section 3 using a significance threshold δ.
2. For each state $\mathbf{x}_1$ with significant probability $p(\mathbf{x}_1, t_1)$ we approximate the rows of T and $\frac{\partial}{\partial c_j} T$ that correspond to $\mathbf{x}_1$ based on a transient analysis for Δt time units. More precisely, if $\mathbf{e}_{\mathbf{x}_1}$ is the vector with all entries zero except for the entry that corresponds to state $\mathbf{x}_1$ which is one, then we solve (2) with initial condition $\mathbf{e}_{\mathbf{x}_1}$ for Δt time units in order to approximate $T(\mathbf{x}_1, \mathbf{x}_2)$ and $\frac{\partial}{\partial c_j} T(\mathbf{x}_1, \mathbf{x}_2)$ for all $\mathbf{x}_2$. During this transient analysis we again apply the dynamic truncation of the state space proposed in Section 3 with threshold δ.
3. We then store for each pair $(\mathbf{x}_1, \mathbf{x}_2)$ the (partial) likelihood $a(\mathbf{x}_1, \mathbf{x}_2)$ and its derivatives:
$$\begin{aligned} a(\mathbf{x}_1, \mathbf{x}_2) &= p(\mathbf{x}_1, t_1) \cdot w(\mathbf{x}_1) \cdot T(\mathbf{x}_1, \mathbf{x}_2) \cdot w(\mathbf{x}_2) \\ \tfrac{\partial}{\partial c_j} a(\mathbf{x}_1, \mathbf{x}_2) &= \tfrac{\partial}{\partial c_j} p(\mathbf{x}_1, t_1) \cdot w(\mathbf{x}_1) \cdot T(\mathbf{x}_1, \mathbf{x}_2) \cdot w(\mathbf{x}_2) \\ &\quad + p(\mathbf{x}_1, t_1) \cdot w(\mathbf{x}_1) \cdot \tfrac{\partial}{\partial c_j} T(\mathbf{x}_1, \mathbf{x}_2) \cdot w(\mathbf{x}_2). \end{aligned}$$
4. We reduce the number of considered pairs by sorting $a(\mathbf{x}_1, \mathbf{x}_2)$ for all pairs $(\mathbf{x}_1, \mathbf{x}_2)$ calculated in the previous step and keep the most probable pairs (see also Section 3).
5. Next, we repeat steps 2-4, where in step 2 we start the analysis from all states $\mathbf{x}_2$ that are the last element of a pair kept in the previous step. In step 3 we store triples of states, say, $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ and recursively compute their likelihood and the corresponding derivatives by multiplication with $T(\mathbf{x}_2, \mathbf{x}_3)$ and $w(\mathbf{x}_3)$, i.e., for the likelihood we compute
$$\begin{aligned} a(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) &= a(\mathbf{x}_1, \mathbf{x}_2) \cdot T(\mathbf{x}_2, \mathbf{x}_3) \cdot w(\mathbf{x}_3) \\ \tfrac{\partial}{\partial c_j} a(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) &= \tfrac{\partial}{\partial c_j} a(\mathbf{x}_1, \mathbf{x}_2) \cdot T(\mathbf{x}_2, \mathbf{x}_3) \cdot w(\mathbf{x}_3) \\ &\quad + a(\mathbf{x}_1, \mathbf{x}_2) \cdot \tfrac{\partial}{\partial c_j} T(\mathbf{x}_2, \mathbf{x}_3) \cdot w(\mathbf{x}_3). \end{aligned}$$
Note that we may reuse some of the entries of T since they already have been calculated in a previous step. In step 4 we again reduce the number of triples $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ by sorting them according to their likelihood. We then keep the most probable triples, and so on. Note that in step 4 we cannot use a fixed truncation threshold δ to reduce the number of state sequences (or paths) since their probabilities may become very small as the sequences become longer.
6. We stop the prolongation of paths $\mathbf{x}_1, \ldots, \mathbf{x}_\ell$ when the time instance $t_R = \Delta t \cdot R$ is reached and compute an approximation of $\mathcal{L}$ and its derivatives by summing up the corresponding values of all paths (cf. Eq. (8)).

If we have more than one observation sequence, i.e., $K > 1$, then we repeat the procedure to compute $\mathcal{L}_k$ for all k and use (12) to calculate the total log-likelihood. Note that the contribution of each path $\mathbf{x}_1, \ldots, \mathbf{x}_R$ to $\mathcal{L}_k$ may be different for each k. It is, however, likely that the entries of T can be reused not only during the computation of each single $\mathcal{L}_k$ but also for different values of k. If many entries of T are reused during the computation, the algorithm performs fast compared to other approaches. For our experimental results in Section 6, we keep the ten most probable paths in step 4. Even though this enforces a coarse approximation, the likelihood is approximated very accurately if σ is small, since in this case only few paths contribute significantly to $\mathcal{L}_k$. On the other hand, if σ is large, then the approximation may become inaccurate depending on the chosen truncation strategy. Another disadvantage of the PLA method is that for non-equidistant time series, the performance is slow since we have to compute (parts of) different transition matrices and, during the computation of $\mathcal{L}_k$, the transition probabilities cannot be reused.

6 Experimental Results

In this section we present experimental results of the SLA and PLA method. For equidistant time series, we compare our approach to the approximate maximum likelihood (AML) and the singular value decomposition (SVDL) method described by Reinker et al. [16] (compare also Section 7). Since an implementation of the AML and SVDL method was not available to us, we chose the same examples and experimental conditions for the time series as Reinker et al. and compared our results to those listed in the results section in [16]. We also consider non-equidistant time series. To the best of our knowledge there exists no direct numerical approach for non-equidistant time series with measurement error that is based on the maximum likelihood method.

We generated time series data for two different examples from systems biology using Monte-Carlo simulation [6] and added error terms $\epsilon_i(t_\ell)$ to the population of the i-th species at time t_ℓ. Besides the simple network described in Example 1 we consider a more complex network with eight reactions and five species for the transcription regulation of a repressor protein [16]:

1:	mRNA $\rightarrow$ mRNA + M	5:	DNA + D $\rightarrow$ DNA.D
2:	M $\rightarrow \emptyset$	6:	DNA.D $\rightarrow$ DNA+D
3:	DNA.D $\rightarrow$ mRNA + DNA.D	7:	M + M $\rightarrow$ D
4:	mRNA $\rightarrow \emptyset$	8:	D $\rightarrow$ M + M

The initial molecular populations are $(2, 4, 2, 0, 0)$ for M, D, DNA, mRNA, and DNA.D. The reachable state space of the model is infinite in three dimensions since the populations of mRNA, M, and D are unbounded. The rate constants are $\mathbf{c} = (0.043, 0.0007, 0.0715, 0.00395, 0.02, 0.4791, 0.083, 0.5)$. For the network in Example 1 we chose the same parameters as Reinker et al., namely $\mathbf{c} = (0.0270, 0.1667, 0.40)$.

For the generation of time series data we fix the (true) constants $\mathbf{c}$ and the standard deviation σ of the error terms. We use the SLA and PLA method to estimate $\mathbf{c}$ and σ such that the likelihood of the time series becomes maximal under these parameters. Since in practice only few observation sequences are available, we estimate the parameters based on $K = 5$ observation sequences. As suggested by Reinker et al., we repeat the generation of batches of five observation sequences and the estimation of parameters 100 times to approximate the mean and the standard deviation of the estimators.

Our algorithms for the approximation of the likelihood are implemented in C++ and we run them on an Intel Core i7 at 2.8 Ghz with 8 GB main memory. They are linked to MATLAB's optimization toolbox which we use to minimize the negative log-likelihood. Since we use a global optimization method (MATLAB's global search), the running time of our method depends on the tightness of the intervals that we use as constraints for the unknown parameters as well as on the number of starting points of the global search procedure. We chose intervals that correspond to the order of magnitude of the parameters, i.e., if $c_j \in O(10^n)$ for some $n \in \mathbb{Z}$ then we use the interval $[10^{n-1}, 10^{n+1}]$ as constraint for c_j. E.g. if $c_j = 0.1$ then $n = -1$ and we use the interval $[10^{-2}, 10^0]$. Moreover, for global search we used 20 starting points for the gene expression example and 5 for the transcription regulation example. Note that this is the only difference of our experimental conditions compared to Reinker et al. who use a local optimization method and start the optimization with the true parameters.

In both algorithms we choose a significance threshold of $\delta = 10^{-15}$. Since the PLA method becomes slow if the number of paths that are considered is large, in step 4 of the algorithm we reduce the number of paths that we consider by keeping only the 10 most probable paths. In this way, the computational effort of the PLA method remains tractable even in the case of the transcription regulation network.

Table 2. Estimates for the simple gene expression model using equidistant time series

Δt (R)	σ	Method	Time	Average (standard deviation) of parameter estimates			
				$c_1 = 0.027$	$c_2 = 0.1667$	$c_3 = 0.4$	σ
1.0 (300)	0.1	AML	–	0.0268(0.0061)	0.1523(0.0424)	0.3741(0.0557)	0.1012(0.0031)
		SVDL	–	0.0229(0.0041)	0.1573(0.0691)	0.4594(0.1923)	–
		SLA	29.4	0.0297(0.0051)	0.1777(0.0361)	0.3974(0.0502)	0.1028(0.0612)
		PLA	2.2	0.0300(0.0124)	0.1629(0.0867)	0.3892(0.0972)	0.1010(0.0792)
	1.0	AML	–	0.0257(0.0054)	0.1409(0.0402)	0.3461(0.0630)	1.0025(0.0504)
		SVDL	–	0.0295(0.0102)	0.1321(0.0787)	0.3842(0.2140)	–
		SLA	8.3	0.0278(0.0047)	0.1868(0.0339)	0.3946(0.0419)	0.9976(0.0476)
		PLA	1.8	0.0278(0.0041)	0.1810(0.0294)	0.3938(0.0315)	0.9938(0.0465)
	3.0	AML	–	0.0250(0.0065)	0.1140(0.0337)	0.3160(0.0674)	3.0292(0.1393)
		SVDL	–	–	–	–	–
		SLA	11.1	0.0285(0.0043)	0.1755(0.0346)	0.3938(0.0508)	2.9913(0.0733)
		PLA	1.7	0.0275(0.0086)	0.1972(0.0902)	0.3894(0.0722)	3.0779(0.0887)
10.0 (30)	0.1	AML	–	–	–	–	–
		SVDL	–	–	–	–	–
		SLA	40.9	0.0273(0.0069)	0.1788(0.04786)	0.3931(0.0599)	0.1086(0.0630)
		PLA	5.2	0.0277(0.0080)	0.1782(0.0517)	0.4057(0.0678)	0.1234(0.0523)
	1.0	AML	–	–	–	–	–
		SVDL	–	–	–	–	–
		SLA	10.2	0.0283(0.0070)	0.1787(0.0523)	0.4018(0.0681)	0.9898(0.0829)
		PLA	3.5	0.0243(0.0057)	0.1665(0.0400)	0.4031(0.0638)	1.0329(0.0859)
	3.0	AML	–	–	–	–	–
		SVDL	–	–	–	–	–
		SLA	12.3	0.0300(0.0110)	0.1960(0.0788)	0.4025(0.0689)	2.9402(0.1304)
		PLA	4.2	0.0210(0.0054)	0.1511(0.0534)	0.4042(0.0616)	3.0629(0.2249)

Table 3. Estimates for the transcription regulation model using equidistant time series

Δt (R)	Method	Average (standard deviation) of parameter estimates			
		$c_1 = 0.043$	$c_2 = 0.0007$	$c_3 = 0.0715$	$c_4 = 0.00395$
1.0 (500)	SVDL	0.0477(0.0155)	0.0006(0.0004)	0.0645(0.0190)	0.0110(0.0195)
	PLA/SLA	0.0447(0.0036)	0.0007(0.0001)	0.0677(0.0115)	0.0034(0.0014)
10.0 (50)	PLA/SLA	0.0417(0.0069)	0.0005(0.0002)	0.0680(0.0075)	0.0038(0.0026)

Δt (R)	Method	Average (standard deviation) of parameter estimates			
		$c_5 = 0.02$	$c_6 = 0.4791$	$c_7 = 0.083$	$c_8 = 0.5$
1.0 (500)	SVDL	0.0159(0.0107)	0.2646(0.0761)	0.0149(0.0143)	0.0615(0.0332)
	PLA/SLA	0.0193(0.0008)	0.4592(0.0169)	0.0848(0.0024)	0.5140(0.0166)
10.0 (50)	PLA/SLA	0.0188(0.0039)	0.4359(0.0822)	0.0836(0.0016)	0.4892(0.0164)

6.1 Equidistant Time Series

In the equidistant case, the length of the observation intervals is $\Delta t = t_\ell - t_{\ell-1}$ for all $\ell \in \{1, \ldots, R\}$. In Table 2 and 3 we list the results given in [16] as well as the results of our methods. Reinker et al. do not evaluate the AML method for larger intervals than $\Delta t = 1$ because, as we will discuss in Section 7, the approximation error of the AML method becomes huge in that case. Also, the SVDL method performs poor if σ is large since it does not include measurement errors in the likelihood. Therefore, no results for $\sigma > 1.0$ are provided in [16] for SVDL. In the first three columns we list Δt, the number R of observation points and the true standard deviation σ of the error terms. In column "Time" we compare the average running time (in seconds) of one parameter estimation (out of 100) for SLA and PLA, i.e., the average running time of the maximization of the likelihood based on $K = 5$ observation sequences. It is not meaningful to compare the running times with those in [16] since different optimization methods are used and experiments were run on different machines. Finally, we list estimation results for all four methods (if available). We give the true parameters in the column headings and list the average of 100 estimations and the standard deviation of the estimates (in brackets).

For the simple gene expression (Table 2) and $\Delta t = 1.0$, we find that SLA and PLA have a similar accuracy for the estimation of σ but are consistently more accurate than AML and SVDL for estimating the rate constants. If $\sigma = 0.1$, then the total absolute error for the estimation of $\mathbf{c}$ is $0.041, 0.073, 0.016, 0.018$ for AML, SVDL, SLA, PLA, respectively. For $\sigma = 1.0$ we have total absolute errors of $0.081, 0.053, 0.026, 0.021$ for AML, SVDL, SLA, PLA. Finally, for $\sigma = 3.0$, AML has a total error of 0.139 while the error for SLA and PLA is 0.017 and 0.041. For $\Delta t = 10$, the results of the SLA and PLA method are accurate even though only 30 observation points are given. Since PLA gives a much coarser approximation, its running time is always shorter (about three to ten times shorter). If σ is large, SLA gives more accurate results than PLA.

In Table 3 we compare results of the transcription regulation for $\sigma = 0$. Note that, for this example, Reinker et al. only give results for the SVDL method with $\Delta t \leq 1.0$ and $\sigma = 0$. Here, we compare results for $\Delta t = 1.0$ since in this case the SVDL method performs best compared to smaller values of Δt. The SLA and PLA method consistently perform better than the SVDL method since they approximate the likelihood more accurately. If $\sigma = 0$, then the accuracy of SLA and PLA is the same (up to the fifth digit).

Therefore the results of SLA and PLA are combined in Table 3. The running time of SLA is, however, much slower since it does not reuse the entries of the transition probability matrix T. For $\Delta t = 1.0$, one parameter estimation based on $K = 5$ observations takes about 30 minutes for SLA and only about 2.4 minutes for PLA. For $\Delta t = 10.0$ we have running times about 5 hours(SLA) and 27 minutes (PLA). As for the gene expression example, we expect for larger values of σ the results of SLA to be more accurate than those of PLA.

6.2 Non-equidistant Time Series

Finally, we consider non-equidistant time series, which can only be handled by the SLA method. During the Monte-Carlo simulation, we generate non-equidistant time series by iteratively choosing $t_{\ell+1} = t_\ell + \mathcal{U}(0,5)$, where $\mathcal{U}(0,5)$ is a random number that is uniformly distributed on $(0,5)$ and $t_0 = 0$. Note that the intervals are not only different within an observation sequence but also for different k, i.e., the times $t_1, \dots, t_R$ depend on the number k of the corresponding sequence. We consider the transcription regulation model with $\sigma = 1.0$ and $K = 5$ as this is our most complex example. Note that, since the accuracy of the estimation decreases as σ increases, we cannot expect a similar accuracy as in Table 3. For a time horizon of $t = 500$ the average number of observation points per sequence is $R = 500/2.5 = 200$. The estimates computed by SLA are $c_1^* = 0.0384(0.0343)$, $c_2^* = 0.0010(0.0001)$, $c_3^* = 0.0642(0.0249)$, $c_4^* = 0.0044(0.0047)$, $c_5^* = 0.0273(0.0073)$, $c_6^* = 0.5498(0.1992)$, $c_7^* = 0.0890(0.0154)$, $c_8^* = 0.5586(0.0716)$, and $\sigma^* = 0.9510(0.0211)$, where we averaged over 100 repeated estimations and give the standard deviation in brackets. Recall that the true constants are $c_1 = 0.043$, $c_2 = 0.0007$, $c_3 = 0.0715$, $c_4 = 0.00395$, $c_5 = 0.02$, $c_6 = 0.4791$, $c_7 = 0.083$, and $c_8 = 0.5$. The average running time of one estimation was 19 minutes.

7 Related Work

In the context of stochastic chemical kinetics, parameter inference methods are either based on Bayesian inference [2, 19, 21] or maximum likelihood estimation [16, 18, 20]. The advantage of the latter method is that the corresponding estimators are, in a sense, the most informative estimates of unknown parameters [11] and have desirable mathematical properties such as unbiasedness, efficiency, and normality [12]. On the other hand, the computational complexity of maximum likelihood estimation is high. If an analytic solution of (6) is not possible, then, as a part of the nonlinear optimization problem, the likelihood and its derivatives have to be calculated. Monte-Carlo simulation has been used to estimate the likelihood [18, 20]. During the repeated random sampling it is difficult to explore those parts of the state space that are unlikely under the current rate parameters. Thus, especially if the rates are very different from the true parameters, many simulation runs are necessary to calculate an accurate approximation of the likelihood. To the best of our knowledge, Reinker et al. provide the first maximum likelihood estimation that is not based on Monte-Carlo simulation but calculates the likelihood numerically [16]. They propose the AML method during which the matrices P_ℓ are approximated. In order to keep the computational effort low, they allow at most two jumps of the Markov process during $[t_\ell, t_{\ell+1})$. Moreover, they ignore all states for which $|O_i(t_\ell) - x_{i\ell}|$ is greater than 3σ. This has the disadvantage that $\mathcal{L}$ is zero (and its derivative) if the values for the rate constants are far off the true values. If $\mathcal{L}$ is zero, then the derivatives provide no information about how the rate constants have to be altered in order to increase the likelihood. Thus, initially very good estimates for the rate constants must be known to apply this kind of truncation. On the other hand, the method that we propose neglects only insignificant terms of the likelihood. For this reason the likelihood and its derivatives do not become zero during the computation and it is always possible to follow the gradient in order to obtain higher likelihoods. Another disadvantage of the AML method is that, if the observation intervals are longer, the likelihood may not be approximated accurately since the assumption that only two reactions occur within an observation interval is not valid. Extending the AML approach to more than two steps would result in huge space requirements and perform slow since the state space is explored in a breath-first search manner and too many states would be considered even though their contribution to the likelihood is very small. In our approach we allow an arbitrary number of reactions during $[t_\ell, t_{\ell+1})$[2]. Therefore, our method is not restricted to reaction networks where the speed of all reactions is at most of the same time scale as the observation intervals. The second approach proposed by Reinker et al., called SVDL method, is based on the assumption that the propensities α_j stay constant during $[t_\ell, t_{\ell+1})$. Again, this assumption only applies to small observation intervals. Moreover, the SVDL method does not take into account measurement errors and is thus only appropriate if σ is very small. Further differences between the approach of Reinker et al. and our approach are that we use a global optimization technique (MATLAB's global search) while Reinker et al. use a local solver, namely the quasi-Newton method. Finally, the approach in [16] requires observations at equidistant time instances, which is not necessary for the SLA method.

8 Conclusion

Parameter inference for stochastic models of cellular processes demands huge computational resources. We proposed two numerical methods, called SLA and PLA, that approximate maximum likelihood estimators for a given set of observations. Both methods do not make any assumptions about the number of reactions that occur within an observation interval. The SLA method allows for an estimation based on arbitrarily spaced intervals while the PLA method requires equidistant intervals.

Many reaction networks involve both small populations and large populations. In this case stochastic hybrid models are most appropriate since they combine the advantages of deterministic and stochastic representations. We plan to extend our algorithms to the stochastic hybrid setting proposed in [9] to allow inference for more complex networks. Further future work also includes more rigorous truncations for the SLA method and the parallelization of the algorithm.

[2] During one step of our numerical integration, we assume that only four reactions are possible. The time step h of the numerical integration does, however, not depend on the $[t_\ell, t_{\ell+1})$ but is dynamically chosen in such a way that performing more than four steps is very unlikely.

References

[1] Banga, J.R., Balsa-Canto, E.: Parameter estimation and optimal experimental design. Essays Biochem. 45, 195–209 (2008)

[2] Boys, R., Wilkinson, D., Kirkwood, T.: Bayesian inference for a discretely observed stochastic kinetic model. Statistics and Computing 18, 125–135 (2008)

[3] Burrage, K., Hegland, M., Macnamara, F.: A Krylov-based finite state projection algorithm for solving the chemical master equation arising in the discrete modelling of biological systems. In: Proceedings of the Markov 150th Anniversary Conference, Boson Books, pp. 21–38 (2006)

[4] Chou, I., Voit, E.: Recent developments in parameter estimation and structure identification of biochemical and genomic systems. Mathematical Biosciences 219(2), 57–83 (2009)

[5] Fernández Slezak, D., Suárez, C., Cecchi, G., Marshall, G., Stolovitzky, G.: When the optimal is not the best: Parameter estimation in complex biological models. PLoS ONE 5(10), e13283 (2010)

[6] Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. 81(25), 2340–2361 (1977)

[7] Golding, I., Paulsson, J., Zawilski, S., Cox, E.: Real-time kinetics of gene activity in individual bacteria. Cell 123(6), 1025–1036 (2005)

[8] Hahn, E.M., Han, T., Zhang, L.: Synthesis for PCTL in parametric markov decision processes. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 146–161. Springer, Heidelberg (2011)

[9] Henzinger, T., Mateescu, M., Mikeev, L., Wolf, V.: Hybrid numerical solution of the chemical master equation. In: Proc. of CMSB 2010. ACM Press, New York (2010)

[10] Henzinger, T.A., Mateescu, M., Wolf, V.: Sliding window abstraction for infinite markov chains. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 337–352. Springer, Heidelberg (2009)

[11] Higgins, J.J.: Bayesian inference and the optimality of maximum likelihood estimation. Int. Stat. Rev. 45(1), 9–11 (1977)

[12] Ljung, L.: System Identification: Theory for the User, 2nd edn. Prentice Hall PTR, Upper Saddle River (1998)

[13] Loinger, A., Lipshtat, A., Balaban, N.Q., Biham, O.: Stochastic simulations of genetic switch systems. Physical Review E 75, 21904 (2007)

[14] Mateescu, M., Wolf, V., Didier, F., Henzinger, T.A.: Fast adaptive uniformisation of the chemical master equation. IET Systems Biology 4(6), 441–452 (2010)

[15] Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. J. Chem. Phys. 124, 44144 (2006)

[16] Reinker, S., Altman, R.M., Timmer, J.: Parameter estimation in stochastic biochemical reactions. IEEE Proc. Syst. Biol. 153, 168–178 (2006)

[17] Sidje, R., Burrage, K., MacNamara, S.: Inexact uniformization method for computing transient distributions of Markov chains. SIAM J. Sci. Comput. 29(6), 2562–2580 (2007)

[18] Tian, T., Xu, S., Gao, J., Burrage, K.: Simulated maximum likelihood method for estimating kinetic rates in gene expression. Bioinformatics 23, 84–91 (2007)

[19] Toni, T., Welch, D., Strelkowa, N., Ipsen, A., Stumpf, M.: Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems. J. R. Soc. Interface 6(31), 187–202 (2009)

[20] Uz, B., Arslan, E., Laurenzi, I.: Maximum likelihood estimation of the kinetics of receptor-mediated adhesion. J. Theor. Biol. 262(3), 478–487 (2010)

[21] Wilkinson, D.J.: Stochastic Modelling for Systems Biology. Chapman & Hall, Boca Raton (2006)

Getting Rid of Store-Buffers in TSO Analysis*

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], and Gennaro Parlato[2]

[1] Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se
[2] LIAFA, CNRS and University Paris Diderot, France
{abou,gennaro}@liafa.jussieu.fr

Abstract. We propose an approach for reducing the TSO reachability analysis of concurrent programs to their SC reachability analysis, under some conditions on the explored behaviors. First, we propose a *linear* code-to-code translation that takes as input a concurrent program P and produces a concurrent program P' such that, running P' under SC yields the same set of reachable (shared) states as running P under TSO with at most k context-switches for each thread, for a fixed k. Basically, we show that it is possible to use only $O(k)$ additional copies of the shared variables of P as local variables to simulate the store buffers, even if they are unbounded. Furthermore, we show that our translation can be extended so that an *unbounded* number of context-switches is possible, under the condition that each write operation sent to the store buffer stays there for at most k context-switches of the thread. Experimental results show that bugs due to TSO can be detected with small bounds, using off-the-shelf SC analysis tools.

1 Introduction

The classical memory model for concurrent programs with shared memory is the sequential consistency (SC) model, where the behaviors of the different threads are interleaved while the order between actions of each single thread is maintained. For performance reasons, modern multi-processors as well as compilers may reorder some memory access operations. This leads to the adoption of weak (or relaxed) memory models such as TSO (Total Store Ordering). In this model, store operations are not immediately visible to all threads (as in SC). Each thread is supposed to have a store buffer where store operations are kept in order to be executed later. While the order of the store operations issued by a same thread are executed (i.e., written in the main memory) in the same order (i.e., the store buffers are FIFO), load operations by this thread can overtake pending stores in the buffer if they concern different variables, and read values from the main memory. Loads from a variable for which there is a store operation in the buffer gets the value of the last of such operation. The TSO model is in some sense the kernel of many common weak memory models [15,19].

Verifying programs by taking into account the effect of the weak memory models such as TSO is a nontrivial problem, both from the theoretical and the practical point of views. Although store buffers are necessarily finite in actual machines and implementations, we should not assume any fixed bound on their size in order to reason about the

* Partially supported by the french ANR-09-SEGI-016 project VERIDYC.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 99–115, 2011.

correctness of general algorithms. For safety properties, the general question to address is whether there is a size of the buffers for which the program can reach some bad configuration, which is equivalent to check the reachability of bad configurations by considering unbounded buffers. This leads to the adoption of formal models (based on state machines with queues) for which the decidability of problems such as checking state reachability is not straightforward. It has been shown in [1] that the state reachability problem for TSO is actually decidable (for finite-data programs), but highly complex (nonprimitive recursive). This leaves open the problem of defining efficient verification techniques for TSO. Necessarily, such verification techniques should be based on upper/under-approximate analysis.

Roughly speaking, the source of complexity in TSO verification is that store buffers can encode lossy channels, and vice-versa. Then, the issue we address in this paper is how to define a verification approach for TSO that allows an efficient encoding of the store buffers, i.e., in a way that *does not depend on their size*. More precisely, we investigate an approach for reducing, with a limited overhead (i.e., a polynomial increase in the size of the program) the reachability problem under TSO (with unbounded store buffers) to the same problem under SC.

Our first idea is to consider the concept of context-bounded analysis in the case of TSO. Context-bounding has been shown (experimentally) to be a suitable notion of behavior coverage for effective bug detection in concurrent programs running under the SC model [14]. Moreover, this approach provides a decidable analysis (under SC) in the case of programs with recursive procedure calls [17]. In this paper, we extend this concept to TSO as follows. We consider that a *context* in this case is a computation segment where only one thread is active, and where all updates of the main memory use store operations taken from the store buffer of that thread. Then, we prove that for every fixed bound k, and for every concurrent program P, it is possible to construct, using a code-to-code translation, another concurrent program P' such that running P' under SC yields the same set of reachable (shared) states as running P under TSO with at most k context-switches for each thread. Our translation preserves the class of the original program in the sense that P and P' have the same features (e.g., recursive procedure calls, dynamic creation of threads, data manipulation). Basically, we show that encoding store buffers can be done using $O(k)$ additional copies of the shared variables as local variables. The obtained program has the same type of data structures and variables and the same control features (recursion, dynamic thread creation) as the original one. As a consequence, we obtain for instance that for finite-data programs, even when recursion is allowed, the context-bounded analysis of TSO programs is decidable (whereas the unrestricted reachability problem in this case is undecidable as in SC).

The translation we provide from TSO to SC, regardless of the decidability issue, does not depend fundamentally from the fact that we have a finite number of context switches for each thread. The key property we use is the fact that each store operation produced by some thread cannot stay in its store buffer for more than a bounded number of context switches of that thread. (This of course does not exclude that each thread may have an unbounded number of context switches.) Therefore, we define a notion for restricting the set of analyzed behaviors of TSO programs which consists in bounding the *age* of each store operation in the buffers. The age of a store operation, produced by a thread T, is the number of context switches made by T since its production. We show that as before, for any bound on the age of all stores, it is possible to translate the reachability problem from TSO to SC. For the case of programs with recursion this translation does not provide a decision procedure. (The targeted class of programs is concurrent programs with an unbounded number of context switches.) However, in the case of finite-data programs without recursive procedures, this translation provides a decision procedure for the TSO reachability problem under store-age bounding (since obviously SC reachability for finite-state concurrent programs is decidable).

Our code-to-code translations allow to smoothly transfer present and future decidability and complexity results from the SC to the TSO case for the same class of programs. More importantly, our translations allow to use existing analysis and verification tools designed for SC in order to perform the same kind of analysis and verification for TSO. To show its practicability, we have applied our approach in checking that standard mutual exclusion protocols for SC are incorrect under TSO, using the tools POIROT [10] and ESBMC [4]. In our experiments, bugs appear for small bounds (≤ 2).

Related work. Context-bounded analysis has been developed in a series of paper in the recent year [17,3,14,11,8,9]. So far, it has been considered only for the SC memory model. As far as we know, the only works addressing the verification problem for TSO programs with unbounded store buffers are [1,13]. In [1], the decidability and the complexity of the state reachability problem under TSO (and other memory models) is considered for a finite number of finite-state threads. The decision procedure for TSO given in that paper is based on a reduction to the reachability problem in lossy channel systems, through a nontrivial and complex encoding. In [13], an approach based on Regular Model Checking is adopted. The paper proposes techniques for computing the set of reachable configurations in TSO programs. If the algorithm terminates, it provides the precise set of reachable configurations, however termination is not guaranteed.

2 Concurrent Programs

We define in this section the class of programs we consider. Basically, we consider concurrent programs with procedure calls and dynamic thread creation. We give the syntax of these programs and describe their semantics according to both SC (Sequential Consistency) and TSO (Total Store Order) memory models.

2.1 Syntax

The syntax of concurrent programs is given by the grammar in Fig. 1. A program has a finite set of processes defining the code executed by parallel threads that can be created dynamically by the spawn statement. The program has a distinguished process `main` that is initially executed to start running the program. We assume that there is a finite number of variables `Svar` that are shared by all the threads. They are used for the communication between threads at context switch points. We also assume that there is a finite number of variables `Gvar` that are global to all procedures. During its execution, each thread has its own copy of these global variables (that are not shared with the other threads) which can be used for value passing at procedure calls and returns. We consider

$$
\begin{array}{rcl}
\langle \mathit{pgm} \rangle & ::= & \mathtt{\textbf{Svar}}\ \bar{s}\ \mathtt{\textbf{Gvar}}\ \bar{g}\ \langle \mathit{main} \rangle \langle \mathit{process} \rangle^{*} \langle \mathit{procedure} \rangle^{*} \\
\langle \mathit{main} \rangle & ::= & \mathtt{\textbf{main}}\ \mathtt{p}\ \mathtt{\textbf{begin}}\ \langle \mathit{lstmt} \rangle^{+}\ \mathtt{\textbf{end}} \\
\langle \mathit{process} \rangle & ::= & \mathtt{\textbf{process}}\ \mathtt{p}\ \mathtt{\textbf{begin}}\ \langle \mathit{lstmt} \rangle^{+}\ \mathtt{\textbf{end}} \\
\langle \mathit{procedure} \rangle & ::= & \mathtt{\textbf{procedure}}\ \mathtt{f}\ \mathtt{\textbf{begin}}\ \langle \mathit{lstmt} \rangle^{+}\ \mathtt{\textbf{end}} \\
\langle \mathit{lstmt} \rangle & ::= & \mathtt{loc} : \langle \mathit{stmt} \rangle \mathtt{;} \\
\langle \mathit{stmt} \rangle & ::= & \langle \mathit{simp\_stmt} \rangle \mid \langle \mathit{comp\_stmt} \rangle \mid \langle \mathit{sync\_stmt} \rangle \\
\langle \mathit{simp\_stmt} \rangle & ::= & \mathtt{\textbf{skip}} \mid \mathtt{\textbf{assume}}(\langle \mathit{pred} \rangle) \mid \mathtt{\textbf{assert}}(\langle \mathit{pred} \rangle) \mid \bar{\mathtt{g}} := \langle \mathit{expr} \rangle \mid \mathtt{\textbf{call}}\ \mathtt{f} \mid \mathtt{\textbf{return}} \\
\langle \mathit{comp\_stmt} \rangle & ::= & \mathtt{\textbf{if}}(\langle \mathit{pred} \rangle)\ \mathtt{\textbf{then}}\ \langle \mathit{lstmt} \rangle^{+}\ \mathbf{else}\ \langle \mathit{lstmt} \rangle^{+}\ \mathbf{fi} \mid \mathtt{\textbf{while}}\ (\langle \mathit{pred} \rangle)\ \mathtt{\textbf{do}}\ \langle \mathit{lstmt} \rangle^{+}\ \mathtt{\textbf{od}} \\
\langle \mathit{sync\_stmt} \rangle & ::= & \mathtt{\textbf{atomic\_begin}} \mid \mathtt{\textbf{atomic\_end}} \mid \mathtt{\textbf{spawn}}\ \mathtt{p} \mid \mathtt{\textbf{fence}} \mid \mathtt{g} := \mathtt{s} \mid \mathtt{s} := \mathtt{g}
\end{array}
$$

Fig. 1. The grammar for concurrent programs

that variables range over some (potentially infinite) data domain $\mathbb{D}$. We assume that we dispose of a language of expressions $\langle \mathit{expr} \rangle$ interpreted over $\mathbb{D}$, and of a language of predicates $\langle \mathit{pred} \rangle$ on global variables ranging over $\mathbb{D}$. The program has a finite number of control locations *Loc*. Its code is a nonempty sequence of labelled statements $\mathtt{loc} : \langle \mathit{stmt} \rangle$ where `loc` is a control location, and $\langle \mathit{stmt} \rangle$ belongs to a simple language of C-like statements.

2.2 SC Semantics

We describe the semantics informally and progressively. Let us first consider the case of sequential programs where statements are restricted to simple statements $\langle \mathit{simp\_stmt} \rangle$ and composed statements $\langle \mathit{comp\_stmt} \rangle$. Then, the program has a single thread that can make procedure calls and manipulate only global variables. In that case, shared variables are omitted, and a configuration can be represented by a triple $\langle \mathit{globals}, \mathtt{loc}, \mathit{stack} \rangle$ where *globals* is a valuation of the global variables, `loc` is a control location, and *stack* is a content of the call stack. The elements of this stack are the control locations of the pending procedure calls. A transition relation between these configurations can be defined as usual. At a procedure call, the current location of the caller is pushed in the stack, and the control moves to the initial location of the callee. At a procedure return, the first control location in the stack is popped, and the control moves to that location.

Now, for the general case, a concurrent program has several parallel threads $T_1, \ldots, T_n$ that have been created using the spawn statement. As mentioned above, each thread has its own copy of the global variables `Gvar` that is used throughout its procedure calls and returns, and all the threads share the variables in `Svar`. Then, a SC-configuration is a tuple of the form $\langle \mathit{shared}, \mathit{thread}_1, \ldots, \mathit{thread}_n \rangle$ for some $n \geq 1$, where *shared* is a valuation of the shared variables, and for each $i \in \{1, \ldots, n\}$, thread_i is the local configuration of thread T_i. Such local configuration is defined as for a sequential program by a triple $\langle \mathit{globals}_i, \mathtt{loc}_i, \mathit{stack}_i \rangle$, plus an additional flag $\mathit{critical}_i$ that indicates if the current thread is executing a critical section of the code that has to be executed atomically (without interference of other threads). When the thread executes an `atomic_begin` statement, this flag is set to 1, and it is set to 0 at the next `atomic_end`. The `spawn` statement creates a new thread, making the configuration of the program grow by the addition of the local configuration of the new thread (i.e., the number n of threads can get arbitrarily large, in general). Actions of different threads are interleaved in a nondeterministic way, under the restriction that if a thread T has opened a critical section, no other thread can execute an action until T closes its section. In the SC model, write operations to shared variables are immediately visible to all threads. Then, a transition relation between global configurations is defined, where at each step one single thread is active. We denote this relation $\xRightarrow{i}_{SC}$ where $i \in \{1, \ldots, n\}$ is the index of the thread T_i that has performed the corresponding step.

2.3 TSO Semantics

In the TSO memory model, the SC semantics is relaxed by allowing that read operations can overtake write operations by the same thread on different shared variables. This corresponds to the use of FIFO buffers where write operations to shared variables can be stored and executed in a delayed way, allowing read operations from the main memory (but on different variables) to overtake them. We define hereafter an operational semantics corresponding to this memory model, in the spirit of the formal model defined in [1]. A store buffer is associated with each thread. Then, a global TSO-configuration of the program is defined as in the SC case, except that a local configuration of a thread includes also the content of its store buffer, i.e., it is a tuple of the form $\langle globals_i, \mathtt{loc}_i, stack_i, critical_i, buffer_i \rangle$. Then, the semantics is defined as for SC, except for assignment operations involving shared variables, and for the synchronization actions atomic_begin and atomic_end. Let us consider each of these cases, and assume that the active thread is T_i: For an assignment of the form $\mathtt{s} := \mathtt{g}$ that writes some value $\mathtt{d}$ (the one stored in $\mathtt{g}$) to the shared variable $\mathtt{s}$, a pair $(\mathtt{s}, \mathtt{d})$ is sent to the store buffer, that is, the buffer of T_i is updated to $buffer_i' = (\mathtt{s}, \mathtt{d}) buffer_i$. For an assignment of the form $\mathtt{g} := \mathtt{s}$ that loads a value from the shared variable $\mathtt{s}$ to the variable $\mathtt{g}$, two cases can occur. First, if a pair $(\mathtt{s}, \mathtt{d})$ is still pending in $buffer_i$, then the load returns the value $\mathtt{d}$ corresponding to the last of such pair in the buffer. Otherwise, the returned value is the one stored for $\mathtt{s}$ in the main memory. As for atomic_begin and atomic_end, they have the same semantics as in the SC cases, except that it is required that their execution can only occur when $buffer_i$ is empty. Notice that these statements allow in particular to encode fences, i.e., actions that cannot be reordered w.r.t. any other actions. Indeed, a fence can be encoded as atomic_begin;atomic_end.

In addition to transitions due to the different threads, memory updates can occur at any time. A memory update consists in getting some $(\mathtt{s}, \mathtt{d})$ from some store buffer (of any thread) and updating the value of $\mathtt{s}$ in the main memory to $\mathtt{d}$, i.e., if for some $j \in \{1, \ldots, n\}$, $buffer_j = buffer_j'(\mathtt{s}, \mathtt{d})$, then $\mathtt{d}$ is stored in the main memory as a new value for $\mathtt{s}$, and the buffer of T_j is updated to $buffer_j'$. Then, we can define a transition relation between global configurations $\xRightarrow{\alpha}_{TSO}$, where α is equal to the index $j \in \{1, \ldots, n\}$ if the transition corresponds to a memory update using $buffer_j$, or otherwise, to the index j of the thread T_j that has performed the transition step.

2.4 Reachability Problems

Let $\sharp \in \{SC, TSO\}$. We define $\Longrightarrow_\sharp$ to be the union of the relations $\xRightarrow{i}_\sharp$ for all $i \in \{1, \ldots, n\}$, and we denote by $\xRightarrow{*}_\sharp$ the reflexive-transitive closure of $\Longrightarrow_\sharp$.

Then, the $\sharp$-*reachability problem* is, given a $\sharp$-configuration γ and a valuation of the shared variables *shared*, to determine if there is a $\sharp$-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) $\gamma \overset{*}{\Longrightarrow}_\sharp \gamma'$. In such a case, we say that γ' and *shared* are $\sharp$-*reachable* from γ.

Let us consider a computation $\rho = \gamma_0 \overset{\alpha_0}{\Longrightarrow}_\sharp \gamma_1 \overset{\alpha_1}{\Longrightarrow}_\sharp \gamma_2 \cdots \overset{\alpha_{m-1}}{\Longrightarrow}_\sharp \gamma_m$. A *context-switch* point in ρ is a configuration γ_j, for some $j \geq 1$, such that $\alpha_{j-1} \neq \alpha_j$. A *computation round* of a thread T_i in ρ is a computation segment (1) occurring between two consecutive points in ρ that are either context-switch or extremal points, and (2) where all transitions are labeled by the same index i, i.e., all transitions are either made by T_i, or are memory updates using the store buffer of T_i (in the case of TSO). Clearly, every computation can be seen as a sequence of computation rounds of different threads. In general, the number of rounds that a thread can have along a computation is unbounded.

Given a bound $k \in \mathbb{N}$, the *k-round* $\sharp$-*reachability problem* is, given a $\sharp$-configurations γ and a valuation of the shared variables *shared*, to determine if there is a $\sharp$-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) γ' is reachable from γ by a computation where every thread T_i has at most k computation rounds. In that case, we say that γ' and *shared* are *k-round* $\sharp$-*reachable* from γ.

3 Bounded-Round Reachability: From TSO to SC

In this section we provide a code-to-code translation that, given a concurrent program P and a fixed bound $k \in \mathbb{N}$, builds another concurrent program P' which simulates P with the property that for any shared state *shared*, *shared* is k-round TSO-reachable in P iff *shared* is k-round SC-reachable in P'. The interesting feature of our translation is that the size of the constructed program P' is *linear* in the size of P. Furthermore, P' is in the same class of programs as P in the sense that it uses the same kind of control primitives (procedure calls and thread creation) and the same kind of data-structures and variables; the encoding of the unbounded store buffers requires only adding, as global variables, $(k+1)$ copies of the shared variables and k Boolean variables per process.

In the following, we assume that the number of rounds that a thread of P can have along any computation is bounded by $(k+1)$, and these rounds are indexed from 0 to k.

3.1 Simulating Store Buffers: Case $k = 1$

Before giving the details of the translation, let us present the main ideas behind it and justify its correctness. Assume (for the moment) that $k = 1$. Then, let us focus on the behavior of one particular thread, say T, and consider its computation rounds and its interactions with its environment (i.e., the set of all the other threads) at context switch points. For that, let us project computations on what is visible to T, i.e., the configurations are projected on the shared variables and the local configuration of T, and we only consider the two computation rounds of T which are of the form:

$$\langle \mathbf{shared_0}, (\mathit{globals}_0, \mathtt{loc}_0, \mathit{stack}_0, \mathit{critical}_0, \mathit{buffer}_0) \rangle \overset{*}{\Longrightarrow}_{TSO} \langle \mathbf{shared'_0}, (\mathit{globals}_1, \mathtt{loc}_1, \mathit{stack}_1, \mathit{critical}_1, \mathbf{buffer_1}) \rangle \quad (1)$$

$$\langle \mathbf{shared_1}, (\mathit{globals}_1, \mathtt{loc}_1, \mathit{stack}_1, \mathit{critical}_1, \mathbf{buffer_1}) \rangle \overset{*}{\Longrightarrow}_{TSO} \langle \mathbf{shared'_1}, (\mathit{globals}_2, \mathtt{loc}_2, \mathit{stack}_2, \mathit{critical}_2, \mathit{buffer}_2) \rangle \quad (2)$$

Notice that the local configurations of T at the end of round 0 and at the beginning of the round 1 are the same.

Encoding the store buffers. In the following, we show that we can use a finite number of global variables to encode the (unbounded) store buffers. This can be done based on two main observations. First, in order to execute correctly a load operation of T on some shared variable x, we need to know whether a store operation on x is still pending in its store buffer, and in this case, we need the last value of such operation, or otherwise we need the value of x in the main memory. Since in each round, only T is active and only operations in its store buffer can be used to modify the main memory, the number of information needed to execute correctly loads is finite and corresponds to the last values written by T to each of the variables composed with the initial content of the main memory (at the beginning of the round). For this purpose, we introduce a vector of data named *View* which is indexed with the shared variables of P. More precisely, $View[x]$ contains the valuation for the load of the variable x.

On the other hand, the order in which store operations of T (sent to the store buffer) on different variables have been consumed (written into the main memory) is not important. In fact, only the last consumed store operation to each variable is relevant. Again, this is true because only T is active during a round, and only its own store buffer can be used to update the main memory. Therefore, given a round j we also define (1) a Boolean vector $Mask_j$ such that $Mask_j[x]$ holds if there is a store operation on x in the buffer of T that is used to update the main memory at round j, and (2) a vector of data $Queue_j$ such that, if $Mask_j[x]$ holds then $Queue_j[x]$ contains the last value that will be written in the shared memory corresponding to x at round j (otherwise it is undefined).

Let us consider the concurrent program P' running under SC built from the concurrent program P by adding to each process of P the following global variables: (1) a vector of data named *View*, (2) two Boolean vectors $Mask_0$ and $Mask_1$, and (3) two vectors of data $Queue_0$ and $Queue_1$. Then, for any local TSO configuration $\langle shared, (globals, \texttt{loc}, stack, critical, buffer)\rangle$ of P, we can associate the following local SC configuration $\langle shared, ((globals, View, Mask_0, Mask_1, Queue_0, Queue_1), \texttt{loc}, stack, critical)\rangle$ of P' such that the following conditions are satisfied: (i) the value of $View[x]$ corresponds to the value of the last store operation to x still pending in the store buffer *buffer* if such operation exists, otherwise the value $View[x]$ is the value of the variable x in the main memory, and (ii) for every $j \in \{0,1\}$, $Mask_j[\mathrm{x}]$ holds true iff at least one store operation on x pending in the store buffer *buffer* will update the shared memory in round j, and $Queue_j[\mathrm{x}]$ contains the last pending value written into x and consumed at round j.

Simulation of P by P'. In the following, we construct for any two computation rounds of a thread of P, a two computation rounds of a thread of P' such that the invariants between the configurations of P and P' are preserved along the simulation.

For the issue of updating the main memory and of passing the store buffer from a round to the next one. We assume w.l.o.g. that the store buffer of any thread of P is empty at the end of the considered computation.

Let us consider first the special case where all store operations produced (sent to the store buffer) in round j are also consumed (written to the main memory) in the same round. It is actually possible to consider that all stores are immediately written to the main memory without store buffering, i.e., as in the SC model.

Consider now the case where not all stores produced in round 0 are consumed in round 0. So for instance, at the end of the execution of round 0 given by (1), we must ensure that the main memory contains $\mathbf{shared'_0}$, and we must pass $\mathbf{buffer_1}$ to the second round. The computation in round 0 can be seen as the concatenation of two subcomputations, ρ_0^0 where all produced stores are consumed in round 0, followed by ρ_1^0 where all stores are consumed in round 1. (Notice that, since the store buffer is a FIFO queue, store operations that are consumed in round 0 are necessarily performed (i.e., sent to the buffer) by T before those that will remain for round 1.) Then, it is clear that $\mathbf{shared'_0}$ is the result of executing ρ_0^0, and $\mathbf{buffer_1}$ contains all stores produced in ρ_1^0.

During the simulation of round 0 by P', the point p separating ρ_0^0 and ρ_1^0 is nondeterministically guessed. The stores produced along the segment ρ_0^0 are written immediately to the main memory as soon as they are produced. So, when the point p is reached, the content of the main memory is precisely $\mathbf{shared'_0}$. During the simulation of ρ_1^0, two operations are performed by P': (1) maintaining the view of T in round 0 by updating *View*, and (2) keeping in $Mask_1$ and $Queue_1$ the information about the last values sent to each variable in ρ_1^0. So, at the end of round 0, the pair $Mask_1$ and $Queue_1$ represent the summary of $\mathbf{buffer_1}$.

The simulation of round 1 by P' starts from the new state of the shared memory $\mathbf{shared_1}$ (which may be different from $\mathbf{shared'_0}$ as other threads could have changed it). Then, the main memory is immediately updated by P' using the content of $Mask_1$ and $Queue_1$. Intuitively, since all stores in $\mathbf{buffer_1}$ are supposed to be consumed in round 1, and again since T is the only active thread, we execute all these store operations at the beginning of round 1. The vector *View* is now updated as follows. Starting from the view obtained at the end of the previous round, we only change the valuation of all those variables x for which no store operations are pending in the store buffer for it. For all such variables x we update its valuation with the one of x contained in the shared memory ($View[x] := x$). Now, the simulation of round 1 can proceed. Since all stores produced in this last round are supposed to be consumed by the end of this round, they are immediately written into the shared memory.

3.2 Simulating Store Buffers: General Case

The generalization to bounds k greater than 1 requires some care. The additional difficulty comes from the fact that stores produced at some round will not necessarily be consumed in the next one (as in the previous case), but may stay in the buffer for several rounds. We start by defining the set of shared and global variables of P', denoted S' and G' respectively, and describe the role they play in P':

Shared variables: the set of shared variables of P and P' are the same, that is, $S' = S$, with $dom_P(\mathtt{x}) = dom_{P'}(\mathtt{x})$ for every $\mathtt{x} \in S$.

Global variables: The set of global variables P' is defined as

$G' = G \cup \left(\bigcup_{j=0}^{k} (Queue_j \cup Mask_j) \right) \cup View \cup \{\mathtt{r\_TSO}, \mathtt{r\_SC}, \mathtt{sim}\}$, where

- for each $j \in \{0, \ldots, k\}$, $Queue_j = \{\mathtt{queue\_j\_x} \mid \mathtt{x} \in S\}$, and $dom_{P'}(\mathtt{queue\_j\_x}) = dom_P(\mathtt{x})$ for every $\mathtt{x} \in S$;
- for each $j \in \{0, \ldots, k\}$, $Mask_j = \{\mathtt{mask\_j\_x} \mid \mathtt{x} \in S\}$, and $dom_{P'}(\mathtt{queue\_j\_x}) = \{\mathtt{true}, \mathtt{false}\}$ for every $\mathtt{x} \in S$;

- $View = \{\texttt{view\_x} \mid \texttt{x} \in S\}$, with $dom_{P'}(\texttt{view\_x}) = dom_P(\texttt{x})$ for every $\texttt{x} \in S$;
- `r_TSO` and `r_SC` are two fresh variables whose domain is the set of round indices, that is, $\{0,\ldots,k\}$;
- `sim` is a new variable whose domain is $\{\texttt{true},\texttt{false}\}$.

For sake of simplicity, we denote a variable named `queue_j_x` also as $Queue_j[\texttt{x}]$, for every j and $\texttt{x} \in S$. Similarly, for the set $Mask_j$ and $View$.

Next, we associate a "meaning" to each variable of P' which represents also the invariant we maintain during the simulation of P by P'.

Invariants for variables. The shared variables of S' keep the same valuation of the ones of P at context-switch points along the simulation. The variables in $View$ is defined as in Sec. 3.1. The variables $Queue_j$ and $Mask_j$, with $j \in \{0,\ldots,k\}$, maintain the invariant that at the beginning of the simulation of round j, $Mask_j[\texttt{x}]$ holds true iff at least one write operation on `x` produced in the previous rounds will update the shared memory in round j, and $Queue_j[\texttt{x}]$ contains the last value written into `x`. Variable `r_SC` keeps track of the round under simulation, and `r_TSO` maintains the round number in which next write operation will be applied to the shared-memory. The global variable `sim` holds true iff the thread is simulating a round (which is mainly used to detect when a new round starts), and the global variables in G are used in the same way they were used in P. The program P' we are going to define maintains the invariants defined above along all its executions.

Simulation of P by P'. Here we first describe how P' simulates P for $k = 2$, and then generalize it for arbitrary values of k. For round 0, there is of course the case where all stores are consumed in the same round, or in round 0 and in round 1. Those cases are similar to what we have seen for $k = 1$. The interesting case is when there are stores that are consumed in round 2. Let us consider that the computation in round 0 is the concatenation of three sub-computations ρ_0^0, ρ_1^0, and ρ_2^0 such that ρ_i^0 represent the segment where all stores are consumed in round i.

The simulation of ρ_0^0 and ρ_1^0 is as before. (Stores produced in ρ_0^0 are written immediately to the main memory, and stores produced in ρ_1^0 are summarized using $Mask_1$ and $Queue_1$.) Then, during the simulation of ρ_2^0, the sequence of stores is summarized using a new pair of vectors $Mask_2$ and $Queue_2$. (Notice that stores produced in ρ_1^0 and ρ_2^0 are also used in updating $View$ in order to maintain a consistent view of the store buffer during round 0.)

Then, at the beginning of round 1 (i.e., after the modification of the main memory due to the context switch), the needed information about the store buffer can be obtained by composing the contents of $Mask_2$ and $Queue_2$ with $Mask_1$ and $Queue_1$ which allow us to compute the new valuation of $View$. (Indeed, the store buffer at this point contains all stores produced in round 0 that will be consumed in rounds 1 and 2.) Moreover, for the same reason we have already explained before, it is actually possible at this point to write immediately to the memory all stores that are supposed to be executed in round 1. After this update of the main memory, the simulation of round 1 can start, and since there are stores in the buffer that will be consumed in round 2, this means that all forthcoming stores are also going to be consumed in round 2. Therefore, during this simulation, the vectors $Mask_2$ and $Queue_2$ must be updated. At the end of round 1, these vectors contain the summary of all the stores that have been produced in rounds 0 and 1 and that will consumed in round 2.

After the change of the main memory due to the context switch, the memory content can be updated using $Mask_2$ and $Queue_2$ (all stores in the buffer can be flushed), and the simulation of round 2 can be done (by writing immediately stores to the memory).

The extension to any k should now be clear. In general, we maintain the invariant that at the beginning of every round j, for every $\ell \in \{j,\ldots,k\}$, the vectors $Mask_\ell$ and $Queue_\ell$ represent the summary of all stores produced in rounds $i < j$ that will be consumed at round ℓ. Moreover, we also know what is the round $r \geq j$ in which the next produced store will be consumed. The simulation starts of round j by updating the main memory using the content of $Mask_j$ and $Queue_j$, and then, when $r = j$, the simulation is done by writing stores to the memory, and when r is incremented (nondeterministically), the stores are used to update $Mask_r$ and $Queue_r$.

From what we have seen above, it is possible to simulate store buffers using additional copies of the shared variables, and therefore, it is possible to simulate the TSO behaviors of a concurrent program P under a bounded number of rounds by SC behaviors of a concurrent program P'. Notice that the latter is supposed to be executed under the SC semantics without any restriction on its behaviors. In order to capture the fact that P' will perform only execution corresponding to rounds in P, we must enforce in the code of P' that the simulation of each round of P must be done in an atomic way.

3.3 Code-to-Code Translation

In this section we provide our code-to-code translation from P to P'. The translation from P to P' that we provide is quite straightforward except for particular points in the simulation: (1) at the beginning of the simulation of each thread, (2) at the beginning of the simulation of each round j, with $j > 0$, (3) at the end of the simulation of each round, (4) during the execution of a statement `x` $:= g$, where `x` is a shared variables, and (5) the execution of a `fence` statement. Let us assume that P has $S = \{\mathtt{x\_1}, \mathtt{x\_2}, \ldots, \mathtt{x\_n}\}$ as a set of shared variables and G as a set of global variables. Next, we describe the procedures for these cases, which are used as building blocks for the general translation.

Init of each thread. Before starting the simulation of a thread, we set both `r_TSO` and `r_SC` to 0. Then, we initialize the *view-variables* to the evaluation of the shared variables, as the store-buffer is initially empty and the valuation of the view coincides with that of the shared variables. Finally, we set to `false` the variables of all *masks*. Procedure `init_thread`() is shown in Fig. 2.

```
procedure init_thread()
begin
 atomic_begin;   sim := true;   r_TSO := r_SC := 0;
     // set the view to the shared valuation
 for(i=1,i<=n,i++) do view_x_i := x_i; od
    // initialize the masks
 for(j=0,j<=k,j++) do mask_j_x_1:=mask_j_x_2:=...:=mask_j_x_n:= false; od
end
```

Fig. 2. Procedure `init_thread`()

```
procedure init_round()
begin
[*] if(r_SC == k) then atomic_end; assume(false);//last round simulated
[*] else
[*]  if(r_TSO == r_SC) then r_TSO:=r_TSO+1; fi //update r_TSO if needed
[*]  r_SC := r_SC+1; // increment of the round number
[*] fi
    for(j=0,j<=k,j++) do
      if (r_SC == j) then
        for(i=1,i<=n,i++) do
           // updating the shared memory
           if (mask_j_x_i) then x_i :=queue_j_x_i; mask_j_x_i:=false; fi
           // rebuild the view
[+]        if (!mask_j_x_i & ... & !mask_k_x_i) then view_x_i := x_i; fi
         od  fi  od
end
```

Fig. 3. Procedure `init_round()`

Starting a new round. When a new round is "detected" we accomplish the following operations. If the last round has been simulated we close the atomic section and block the execution of the thread. Otherwise, we increment `r_SC`, as well as `r_TSO` in case it becomes smaller than `r_SC` (next write operation can only modify the shared memory starting from the current round). Then we dump the part of the store-buffer that was supposed to change the shared memory during the execution of round `r_SC`. Let `r_SC` $=$ j. The way we simulate such an operation is by using $Mask_j$ and $Queue_j$: for every shared variable `x`, we assign to `x` the value $Queue_j[\mathrm{x}]$ provided $Mask_j[\mathrm{x}]$ holds true. The last step is that of updating the view for the current round. A variable $View[\mathrm{x}]$ changes its valuation if no write operation is pending for `x` in the store-buffer, and its new value is that variable `x` in the shared memory ($View[\mathrm{x}] := \mathrm{x}$). Procedure `init_round` of Fig. 3 encodes the phases described above.

Finally, procedure `is_init_round()` of Fig. 4 detects that a new round has started checking that `sim` holds *false*. In such a case, we open an atomic section and set `sim` to `true`, and then call procedure `init_round` to initialize the round.

Terminating a round. We terminate non-deterministically a round by setting the variable `sim` to `false` and then closing the atomic section. (Next time the current thread will be scheduled it detects that a new round is started by checking the valuation of `sim`.) Procedure `is_end_round()` of Fig. 4 encoperates such operations.

Write into a shared variable. Consider a statement `x` $:=$ g where `x` and g are a shared and global variable of P, respectively. In the simulation of such assignment, we first update the view for `x` to g. The next step consists in incrementing non-deterministically the value of the auxiliary variable `r_TSO` which represent the round where the current write operation will occur in the memory. Now, let `r_TSO` $= i$. In case `r_SC` is equal to i, we update `x` in the shared-memory. Instead, if `r_SC` $<$ `r_TSO`, we update $Mask_i[\mathrm{x}]$ to `true` and $Queue_i[\mathrm{x}]$ to g which captures that the write operation `x` $:=$ g will modify the shared memory exactly at round i and it is the last operation for `x`. Notice that if

```
procedure is_init_round()
begin
  if ( !sim ) then
    atomic_begin;
    sim := true;
    init_round();
  fi
end
```

```
procedure is_end_round()
begin
  if (*) then
     sim := false;
     atomic_end;
  fi
end
```

```
procedure fence()
begin
 if(r_TSO!=r_SC)
 then
    atomic_end;
    assume(F);
  fi
end
```

Fig. 4. Procedures `is_init_round`(), `is_end_round`(), and `fence`()

```
procedure memory_update_x(g)
begin
     view_x:=g; // updating the view
     // non-deterministically increase r_TSO
[*]  while (*) do  if (r_TSO < k) then r_TSO:=r_TSO + 1; fi od
     if (r_SC==r_TSO) then x:=g; // shared memory update
     else  // updating the mask and the queue
       for(i=0,i<=k,i++) do
          if (r_TSO==i) then mask_i_x:=true; queue_i_x:=g; fi od
     fi
end
```

Fig. 5. Procedure `memory_update_x`, for each shared variable `x`

another write operation will be performed for `x`, when `r_TSO` contains the value i, then the value of $Queue_i$[x] will contain only the latest operation, and the previous value will be overwritten thus reestablishing the invariant. The procedure `memory_update_x`(g) in Fig. 5 subsumes the operations described in Sec. 3.2.

Fences. The statement `fence` is simply translated into a procedure, called `fence`(), that checks whether `r_TSO` is equal to `r_SC` and in case they are different blocks the execution. However, before blocking it, it first executes the statement `atomic_end` so that other threads can continue their evolution. Fig. 4 illustrates procedure `fence`().

General translation. We are now ready to give the general translation by defining a map $[\![\cdot]\!]_{tr}$ in which $P' = [\![P]\!]_{tr}$. The definition of the translation is given in Fig. 6. The new program P' first declares the variables as described in Sec. 3.2.

- $\overline{\texttt{mask}}$ is the list of the variables `mask_i_x`, for all $i \in \{0,\ldots,k\}$ and $\mathtt{x} \in S$;
- $\overline{\texttt{queue}}$ is the list of the variables `queue_i_x` for all $i \in \{0,\ldots,k\}$ and $\mathtt{x} \in S$;
- $\overline{\texttt{view}}$ is the list of the variables `view_i_x` for all $i \in \{0,\ldots,k\}$ and $\mathtt{x} \in S$.

Each `process` procedure starts with a call to procedure `init_thread`() that initializes the auxiliary variables used for the simulation. Then, the translation consists of an in-place replacement of each statement. Each statement `stmt` of P is translated by the sequence of statements `is_init_round`(); $[\![\texttt{stmt;}]\!]_{\texttt{tr}}$ `is_end_round`();. The call to `is_init_round`() checks whether a new round has just started and hence appropriately initialize the variables for the simulation of the new round; the call `is_end_round`();

$$
\begin{aligned}
\llbracket \mathbf{Svar}\ \bar{\mathtt{s}}\ \mathbf{Gvar}\ \bar{\mathtt{g}}\ \langle main\rangle\langle process\rangle^*\langle procedure\rangle^*\ \rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{Svar}\ \bar{\mathtt{s}}\ \mathbf{Gvar}\ \bar{\mathtt{g}}, \overline{\mathtt{mask}}, \overline{\mathtt{queue}}, \overline{\mathtt{view}}, \mathtt{r\_TSO}, \mathtt{r\_SC}, \mathtt{sim} \\
&\qquad \llbracket\langle main\rangle\rrbracket_{tr}\ \llbracket\langle process\rangle\rrbracket^*_{tr}\llbracket\langle procedure\rangle\rrbracket^*_{tr} \\
\llbracket \mathbf{main}\ \mathtt{p}\ \mathbf{begin}\ \langle lstmt\rangle^+\ \mathbf{end}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{main}\ \mathtt{p}\ \mathbf{begin}\ \mathtt{init\_thread}();\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{end} \\
\llbracket \mathbf{process}\ \mathtt{p}\ \mathbf{begin}\ \langle lstmt\rangle^+\ \mathbf{end}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{process}\ \mathtt{p}\ \mathbf{begin}\ \mathtt{init\_thread}();\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{end} \\
\llbracket \mathbf{procedure}\ \mathtt{p}\ \mathbf{begin}\ \langle lstmt\rangle^+\ \mathbf{end}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{procedure}\ \mathtt{p}\ \mathbf{begin}\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{end} \\
\llbracket \mathtt{loc} : \langle stmt\rangle\,\mathbf{;}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathtt{is\_init\_round}();\ \mathtt{loc} : \llbracket\langle stmt\rangle\rrbracket_{tr}\mathbf{;}\ \mathtt{is\_end\_round}() \\
\llbracket \mathbf{skip}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{skip} \\
\llbracket \mathbf{if}\ \ (\langle pred\rangle)\ \mathbf{then}\ \langle lstmt\rangle^+\ \mathbf{else}\ \langle lstmt\rangle^+\ \mathbf{fi}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{if}\ \ (\langle pred\rangle)\ \mathbf{then}\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{else}\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{fi} \\
\llbracket \mathbf{while}\ (\langle pred\rangle)\ \mathbf{do}\ \langle lstmt\rangle^+\ \mathbf{od}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{while}\ (\langle pred\rangle)\ \mathbf{do}\ \llbracket\langle lstmt\rangle\rrbracket^+_{tr}\ \mathbf{od} \\
\llbracket \mathbf{assume}\ (\langle pred\rangle)\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{assume}\ (\langle pred\rangle) \\
\llbracket \mathbf{assert}\ (\langle pred\rangle)\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{assert}\ (\langle pred\rangle) \\
\llbracket \bar{\mathtt{g}} := \langle expr\rangle\rrbracket_{tr} &\stackrel{\text{def}}{=} \bar{\mathtt{g}} := \langle expr\rangle \\
\llbracket \mathbf{call}\ \mathtt{f}\ \rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{call}\ \mathtt{f} \\
\llbracket \mathbf{return}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{return} \\
\llbracket \mathbf{atomic\_begin}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{atomic\_begin;}\ \mathtt{fence}() \\
\llbracket \mathbf{atomic\_end}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathtt{fence}();\ \mathbf{atomic\_end} \\
\llbracket \mathbf{spawn}\ \mathtt{p}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathbf{spawn}\ \mathtt{p} \\
\llbracket \mathbf{fence}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathtt{fence}() \\
\llbracket \mathtt{g} := \mathtt{s}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathtt{g} := \mathtt{view\_s} \\
\llbracket \mathtt{s} := \mathtt{g}\rrbracket_{tr} &\stackrel{\text{def}}{=} \mathtt{memory\_update\_s}(\mathtt{g})
\end{aligned}
$$

Fig. 6. Translation map $\llbracket \cdot \rrbracket_{tr}$

allows to non-deterministically terminate a round at any point in the simulation. The remaining part of the translation concerns the translation of each single statement `stmt`:

- $\mathtt{g} := \mathtt{x}$ is translated into $\mathtt{g} := View_x$;
- $\mathtt{x} := \mathtt{g}$ is replaced with the procedure call `memory_update_s(g)`;
- `fence` is translated as the call to the procedure `fence()`;
- `atomic_begin` (resp. `atomic_end;`) is translated into the sequence `atomic_begin; fence()` (resp. `fence(); atomic_end`);
- All remaining kind of statements remain unchanged in the translation.

From the construction given above, and the reasoning followed in Sec. 3.2 we can prove the following theorem:

Theorem 1. *Let k be a fixed positive integer. A shared state shared is k-round TSO-reachable in P if and only if shared is SC-reachable in P'. Furthermore, if shared is SC-reachable in P' then shared is k'-round TSO-reachable in P for some $k' \leq k$. Moreover, the* size *of P' is linear in the size of P.*

4 Bounded Store-Age Reachability

In this section, we introduce a new notion for restricting the set of behaviors of concurrent programs to be analyzed under TSO. We impose that each store operation produced by a thread T can not stay in the store-buffer more than k consecutive rounds. (Notice that this notion does not restrict the number of rounds that the thread T may have.) We show that, under this restriction, it is still possible to define a code-to-code translation (similar to that of Sec. 3) that associates with each concurrent program P another concurrent program P' such that running P' under SC captures precisely the set of behaviors of P under TSO. More precisely, we associate to each store operation an *age*. The age is initialized at 0 when this store operation is produced by T and sent in the store-buffer. Now, this age is incremented at each context-switch of thread T.

Let $k \in \mathbb{N}$ be a fixed bound. The *k-store-age* TSO-*reachability problem* is, given a TSO-configurations γ and a valuation of the shared variables *shared*, to determine if there is a TSO-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) γ' is reachable from γ by a computation where at each step all the pending store operations have an age equal or less than k.

Let us consider a concurrent program P defined as in Sec. 3. In the following, we construct another concurrent program P' such that the k-store-age TSO-reachability problem for P can be reduced to the SC-reachability problem for P'. The provided code-to-code translation is very similar to the one given in Sec. 3. In fact, if we use the previous translation to simulate a thread T of P, we need to use an unbounded number of vectors of type *Mask* and *Queue*. The key observations (to overcome this difficulty) are that : (1) in order to simulate a round j of a thread T, we only use vectors $Mask_i$ and $Queue_i$ with $i \geq j$, and (2) at each moment of the simulation of a round j, we need only the vectors $Mask_l$ and $Queue_l$ with $l \leq j+k$ (since the age of any store operation is bounded by k). Therefore, we can define our translation using only k vectors of type *Mask* and *Queue* in a circular manner (modulo k). For instance, if the current simulated round of the thread T is 1, the variables $Mask_0$ and $Queue_0$ can be used in the simulation of the round $k+1$. Technically, we introduce only two modifications in the translation given in Sec. 3:
In Fig. 3, the piece of code marked with [*] is replaced with the following one:

```
[*]  // update r_TSO if needed
[*]  if (r_TSO == r_SC) then  r_TSO := (r_TSO+1 mod k+1); fi
[*]  // resetting the boolean vector mask_i
[*]  if (r_SC == i) then  mask_i_x_1 :=...:= mask_i_x_n := false;  fi
[*]  r_SC := (r_SC+1 mod k+1);  // increment of the round number
```

In Fig. 3 the line of code marked with [+] is replaced with the following one:

```
[+] if (!mask_1_x_i & ... & !mask_k_x_i) then view_x_i := x_i; fi
```

In Fig. 5 we replace the line of code marked with [*] with the following:

```
[*] if ( (r_SC - r_TSO) mod k+1 != 1) then r_TSO:=(r_TSO+1 mod k+1); fi
```

Finally, the relation between the given concurrent program P and the constructed program $P' = [\![P]\!]_{tr}$ is given by the following theorem:

Theorem 2. *A shared state shared is k-store-age TSO-reachable in P if and only if SC-reachable in P'. Moreover, the size of P' is linear in the size of P.*

Table 1. Experimental results for 4 mutual exclusion protocols by using POIROT and ESBMC

Mutual exclusion Protocols	**Poirot** ($L = 2$) Time (s)			**ESBMC** Time (s)	
	Version with no fences (Buggy for TSO) ($D = 1$)	Version with fences (Correct for TSO) ($D = 1$)	 ($D = 2$)	Version with no fences (Buggy for TSO) ($L = 2, C = 3$)	Version with fences (Correct for TSO) ($L = 3, C = 4$)
DEKKER	7	6	72	-	6
LAMPORT	26	110	1608	1	7
PETERSON	5	6	47	1	1
SZYMANSKI	8	6	978	1	6

5 Experiments

To show the practicability of our approach, we have experimented its application in detecting bugs due to the TSO semantics. For that, we have considered four well-known mutual exclusion protocols designed for the SC semantics: Dekker's [5], Lamport's [12], Peterson's [16], and Szymanski's [18]. All of these protocols are incorrect under TSO. Through our translations, we have analyzed the behaviors of these protocols under TSO using two SMT-based bounded model-checkers for SC concurrent programs. Our experimental results show that errors due to TSO appear within few rounds, and that off-the-shelf analysis tools designed for the SC semantics can be used for their detection.

In more details, we consider the four protocols mentioned above instantiated for two threads. We consider for each protocol two versions, one without fences (the original version of the protocol) that is buggy, and one with fences (neutralizing TSO) which is known to be correct. We have encoded each of these protocols (with and without fences) as C programs and manually translated by using the k-store-age translation with $k = 2$. We have instrumented the obtained C programs for both POIROT [10] and ESBMC [4] – two SMT-based bounded model-checkers for SC concurrent programs.

Table 1 illustrates the results of the analysis for the four mutual exclusion protocols we carried with both POIROT and ESBMC. The parameters L in the table indicate the number of loop unrolling. POIROT considers all runs by bounding L and the number of *delays* (we refer to [6] for the definition of delay). In our experiments with POIROT, we consider $L = 2$, and a bound D = 1 or D = 2 (= to the number of delays + 1). Turning to ESBMC, it analyzes all executions by bounding the number of loop unrolling and the number of context-switches. In the experiments with ESBMC, we consider a bound $L = 2$ or $L = 3$ on the number of loop unrolling, and a bound $C = 3$ or $C = 4$ on the number of context-switches. Both of tools are able to answer correctly, i.e., by finding the bugs for the buggy versions, except that ESBMC does not answer correctly for the buggy version of Dekker.)

6 Conclusion

We have presented a code-to-code translation from concurrent to concurrent programs such that the reachable shared states of the obtained program running under SC is exactly the same set of reachable shared states of the original program running under the

TSO semantics. The main characteristic of our translations is that it does not introduce any other auxiliary storage to model store buffers but only requires few copies of the shared variables that are local to threads in the resulted translated program (this is important for compositional analyses which track at each moment only one copy of the locals). Furthermore, our translations produce programs of linear size with respect the original ones, provided a constant value of k. Such characteristics allows, and this is the main interest of our approach, the use for relaxed memory models of mature tools designed for the SC semantics (such as BDD-based model-checkers [7], SMT/SAT-based model-checkers [10,4]) as well as tools for sequential analysis based on compositional sequentialization techniques for SC concurrent programs [11,8,6].

Moreover, our translations allow to transfer decidability and complexity results from the SC to the TSO case. In the following we discuss on the decidability/undecidability of the k bounded-round and k-store-age TSO-reachability for concurrent programs with variables ranging over finite domains. We consider first the case in which all processes are non-recursive. When a finite number of threads are involved in the computation, the problem is decidable by using the classical reachability algorithm for finite state concurrent programs. The same problem remains decidable if we add dynamic thread creation, by a reduction to the coverability problem for Petri nets [2]. On the other hand, if we have at least two recursive threads involved in the computation, the k-store-age TSO-reachability becomes undecidable for any k: For every concurrent program P we can construct a concurrent program P' (obtained from P by inserting a fence statement at each control location of P) such that the SC-reachability problem for P (which is an undecidable problem in general) can be reduced to the k-store-age TSO-reachability problem for P'. However, by retaining recursion and using context-bounded analysis for concurrent programs and our translation we can claim the decidability of a variety of restrictions of the k-store-age (and k bounded-round) TSO-reachability. For instance, TSO bounded context-switch reachability is decidable for finite number of threads [17], as well as for bounded round-robin reachability for the parametrized case [9]. Moreover, decidability results concerning the analysis of programs with dynamic thread creation for k context-switches per thread [2] can also be transferred to the TSO case.

Acknowledgments. We would like to thank Akash Lal and Lucas Cordeiro for their help with POIROT and ESBMC.

References

1. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL, pp. 7–18. ACM, New York (2010)
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. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
4. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE. ACM/IEEE (2011)

5. Dijkstra, E.W.: Cooperating sequential processes. Technical report, Technological University, TR EWD-123 (1965)
6. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL, pp. 411–422. ACM, New York (2011)
7. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM, New York (2009)
8. 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)
9. 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)
10. Lahiri, S., Lal, A., Qadeer, S.: Poirot. Microsoft Research, `http://research.microsoft.com/en-us/projects/poirot`
11. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
12. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
13. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
14. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)
15. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge (2009)
16. Peterson, G.L.: Myths about the mutual exclusion problem. IPL 12(3), 115–116 (1981)
17. 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)
18. Szymanski, B.K.: A simple solution to lamport's concurrent programming problem with linear wait. In: ICS, pp. 621–626 (1988)
19. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)

Malware Analysis with Tree Automata Inference[⋆],[⋆⋆]

Domagoj Babić, Daniel Reynaud, and Dawn Song

University of California, Berkeley
{babic,reynaud,dawnsong}@cs.berkeley.edu

Abstract. The underground malware-based economy is flourishing and it is evident that the classical ad-hoc signature detection methods are becoming insufficient. Malware authors seem to share some source code and malware samples often feature similar behaviors, but such commonalities are difficult to detect with signature-based methods because of an increasing use of numerous freely-available randomized obfuscation tools. To address this problem, the security community is actively researching behavioral detection methods that commonly attempt to understand and differentiate how malware behaves, as opposed to just detecting syntactic patterns. We continue that line of research in this paper and explore how formal methods and tools of the verification trade could be used for malware detection and analysis. We propose a new approach to learning and generalizing from observed malware behaviors based on tree automata inference. In particular, we develop an algorithm for inferring *k*-testable tree automata from system call dataflow dependency graphs and discuss the use of inferred automata in malware recognition and classification.

1 Introduction

Over the last several decades, the IT industry advanced almost every aspect of our lives (including health care, banking, traveling,...) and industrial manufacturing. The tools and techniques developed in the computer-aided verification community played an important role in that advance, changing the way we design systems and improving the reliability of industrial hardware, software, and protocols.

In parallel, another community made a lot of progress exploiting software flaws for various nefarious purposes, especially for illegal financial gain. Their inventions are often ingenious botnets, worms, and viruses, commonly known as *malware*. Malware source code is rarely available and malware is regularly designed so as to thwart static analysis through the use of obfuscation, packing, and encryption [34].

[⋆] This material is based upon work partially supported by the National Science Foundation under Grants No. 0832943, 0842694, 0842695, 0831501, 0424422, by the Air Force Research Laboratory under Grant No. P010071555, by the Office of Naval Research under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1-0352 and FA9550-09-1-0539. The work of the first author is also supported by the Natural Sciences and Engineering Research Council of Canada PDF fellowship.

[⋆⋆] The full version of this paper, the inference engine source code, all dependency graphs used in this paper, and a brief explanation of the file format, are available at `http://www.domagoj.info/`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 116–131, 2011.

For the above mentioned reasons, detection, analysis, and classification of malware are difficult to formalize, explaining why the verification community has mostly avoided, with some notable exceptions (e.g., [8,18]), the problem. However, the area is in a dire need of new approaches based on strong formal underpinnings, as less principled techniques, like signature-based detection, are becoming insufficient. Recently, we have been experiencing a flood of malware [31], while the recent example of Stuxnet (e.g., [27]) shows that industrial systems are as vulnerable as our every-day computers.

In this paper, we show how formal methods, more precisely tree automata inference, can be used for capturing the essence of malicious behaviors, and how such automata can be used to detect behaviors similar to those observed during the training phase. First, we execute malware in a controlled environment to extract dataflow dependencies among executed system calls (*syscalls*) using dynamic taint analysis [5,29]. The main way for programs to interact with their environment is through syscalls, which are broadly used in the security community as a high-level abstraction of software behavior [13,23,32]. The dataflow dependencies among syscalls can be represented by an acyclic graph, in which nodes represent executed syscalls, and there is an edge between two nodes, say s_1 and s_2, when the result computed by s_1 (or a value derived from it) is used as a parameter of s_2. Second, we use tree automata inference to learn an automaton recognizing a set of graphs. The entire process is completely automated.

The inferred automaton captures the essence of different malicious behaviors. We show that we can adjust the level of generalization with a single tunable factor and how the inferred automaton can be used to detect likely malicious behaviors, as well as for malware classification. We summarize the contributions of our paper as follows:

- Expansion of dependency graphs into trees causes exponential blowup in the size of the graph, similarly as eager inlining of functions during static analysis. We found that a class of tree languages, namely k-testable tree languages [35] can be inferred directly from dependency graphs, avoiding the expansion to trees.
- We improve upon the prior work on inference of k-testable tree languages by providing an $\mathcal{O}(kN)$ algorithm, where k is the size of the pattern and N is the size of the graph used for inference.
- We show how inferred automata can be used for detecting likely malicious behaviors and for malware classification. To our knowledge, this is the first work applying the theory of tree automata inference to malware analysis. We provide experimental evidence that our approach is both feasible and useful in practice.
- While previous work (e.g., [7,13]) often approximated dependencies by syntactic matching of syscall parameters, we implemented a tool for tracking dependencies via taint analysis [5,29] and we made the generated dependency graphs, as well as the tree automata inference engine, publicly available to encourage further research.

2 Related Work

Tree Automata Inference. Gold [17] showed that no super-finite (contains all finite languages and at least one infinite) is identifiable in the limit from positive examples[1]

[1] Positive examples are examples belonging to the language to be inferred, while negative examples are those not in the language.

only. For instance, regular and regular tree languages [9] are super-finite languages. We have two options to circumvent this negative result; either use both positive and negative examples, or focus on less expressive languages that are identifiable in the limit from positive examples only. Inference of minimal finite state automata from both positive and negative examples is known to be NP-complete [17]. The security community is discovering millions of new malware samples each year and inferring a single minimal classifier for all the samples might be infeasible. Inferring a non-minimal classifier is feasible, but the classifier could be too large to be useful in practice. Thus, we focus on a set of languages identifiable in the limit from positive examples in this paper.

A subclass of regular tree languages — k-testable tree languages [35] — is identifiable in the limit from positive examples only. These languages are defined in terms of a finite set of k-level-deep tree patterns. The k factor effectively determines the level of abstraction, which can be used as a knob to regulate the ratio of false positives (goodware detected as malware) and false negatives (undetected malware). The patterns partition dependency graphs into a finite number of equivalence classes, inducing a state-minimal automaton. The automata inferred from positive (malware) examples could be further refined using negative (goodware) examples. Such a refinement is conceptually simple, and does not increase the inference complexity, because of the properties of k-testable tree languages. We leave such a refinement for future work.

A number of papers focused on k-testable tree automata inference. Garcia and Vidal [15] proposed an $\mathcal{O}(kPN)$ inference algorithm, where k is the size of the pattern, P the total number of possible patterns, and N the size of the input used for inference. Many patterns might not be present among the training samples, so rather than enumerating all patterns, [14] and [22] propose very similar algorithms that use only the patterns present in the training set. Their algorithms are somewhat complex to implement as they require computation of three different sets (called roots, forks, and leaves). Their algorithms are $\mathcal{O}\left(M^k N \log(N)\right)$, where M is the maximal arity of any alphabet symbol in the tree language. We derive a simpler algorithm, so that computing forks and leaves becomes unnecessary. The complexity of our algorithm is $\mathcal{O}(kN)$, thanks to an indexing trick that after performing k iterations over the training sample builds an index for finding patterns in the training set. Patterns in the test set can be located in the index table in amortized time linear in the size of the pattern. In our application — malware analysis — the k factor tends to be small (≤ 5), so our algorithm can be considered linear-time.

Malware Analysis. From the security perspective, several types of malware analysis are interesting: malware detection (i.e., distinguishing malware from goodware), classification (i.e., determining the family of malware to which a particular sample belongs), and phylogeny (i.e., forensic analysis of evolution of malware and common/distinctive features among samples). All three types of analyses are needed in practice: detection for preventing further infections and damage to the infected computers, and the other two analyses are crucial in development of new forms of protection, forensics, and attribution. In this paper, we focus on detection and classification.

The origins of the idea to use syscalls to analyze software can be traced to Forrest et al. [12], who used fixed-length sequences of syscalls for intrusion detection. Christodorescu et al. [7] note that malware authors could easily reorder data-flow-independent syscalls, circumventing sequence-detection schemes, but if we analyze data-flow dependencies among syscalls and use such dependency graphs for detection, circumvention becomes harder. Data-flow-dependent syscalls cannot be (easily) reordered without changing the semantics of the program. They compute a difference between malware and goodware dependency graphs, and show how resulting graphs can be used to detect malicious behaviors. Such graph matching can detect only the exact behavioral patterns already seen in some sample, but does not automatically generalize from training samples, i.e., does not attempt to overapproximate the training set in order to detect similar, but not exactly the same behaviors.

Fredrikson et al. [13] propose an approach that focuses on distinguishing features, rather than similarities among dependency graphs. First, they compute dependency graphs at runtime, declaring two syscalls, say s_1 and s_2, dependent, if the type and value of the value returned by s_1 are equal to the type and value of some parameter of s_2 and s_2 was executed after s_1. They extract significant behaviors from such graphs using structural leap mining, and then choose behaviors that can be combined together using concept analysis. In spite of a very coarse unsound approximation of the dependency graph and lack of automatic generalization, they report 86% detection rate on around 500 malware samples used in their experiments. We see their approach as complementary to ours: the tree-automata we infer from real dependency graphs obtained through taint analysis could be combined with leap mining and concept analysis, to improve their classification power.

Bonfante et al. [3] propose to unroll control-flow graphs obtained through dynamic analysis of binaries into trees. The obtained trees are more fine-grained than the syscall dependency graphs. The finer level of granularity could, in practice, be less susceptible to mimicry attacks (e.g., [33]), but is also easier to defeat through control-flow graph manipulations. The computed trees are then declared to be tree automata and the recognizer is built by a union of such trees. Unlike inference, the union does not generalize from the training samples. The reported experiments include a large set of malware samples (over 10,000), but the entire set was used for training, and authors report only false positives on a set of goodware (2653 samples). Thus, it is difficult to estimate how well their approach would work for malware detection and classification.

Taint Analysis. Dynamic taint analysis (DTA) [29] is a technique used to follow data flows in programs or whole systems at runtime. DTA can be seen as a single-path symbolic execution [21] over a very simple domain (set of taints). Its premises are simple: *taint* is a variable annotation introduced through *taint sources*, it is propagated through program execution according to some *propagation rules* until it reaches a *taint sink*. In our case, for instance, taint sources are the syscalls' output parameters, and taint sinks are the input parameters.

As will be discussed in detail later, our implementation is based on the binary rewriting framework Pin [25] and uses the taint propagation rules from Newsome and Song [29]. Since DTA must operate at the instruction-level granularity, it poses a significant runtime overhead. Our DTA implementation executes applications several thousand times slower than the native execution. Our position is that the speed of the taint analysis is less important than the speed of inference and recognition. The taint analysis can be run independently for each sample in parallel, the dependency graph extraction is linear with the length of each execution trace, and hardware-based information flow tracking has been proposed (e.g., [30,11]) as a potential solution for improving performance. In contrast, inference techniques have to process all the samples in order to construct a single (or a small number of) recognizer(s). An average anti-virus vendor receives millions of new samples annually and the number of captured samples has been steadily growing over the recent years. Thus, we believe that scalability of inference is a more critical issue than the performance of the taint analysis.

3 Notation and Terminology

In this section, we introduce the notation and terminology used throughout the paper. First, we build up the basic formal machinery that allows us to define tree automata. Second, we introduce some notions that will help us define *k-roots* that can be intuitively seen as the top k levels of a tree. Later, we will show how k-roots induce an equivalence relation used in our inference algorithm. Towards the end of this section, we introduce *k-testable* languages, less expressive than regular tree languages, but suitable for designing fast inference algorithms.

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^*$ the free monoid generated by $\mathbb{N}$ with concatenation $(\cdot)$ as the operation and the empty string ε as the identity. The prefix order $\leq$ is defined as: $u \leq v$ for $u,v \in \mathbb{N}^*$ iff there exists $w \in \mathbb{N}^*$ such that $v = u \cdot w$. For $u \in \mathbb{N}^*, n \in \mathbb{N}$, the *length* $|u|$ is defined inductively: $|\varepsilon| = 0, |u \cdot n| = |u| + 1$. We say that a set S is *prefix-closed* if $u \leq v \wedge v \in S \Rightarrow u \in S$. A *tree domain* is a finite non-empty prefix-closed set $D \subset \mathbb{N}^*$ satisfying the following property: if $u \cdot n \in D$ then $\forall 1 \leq j \leq n \,.\, u \cdot j \in D$.

A *ranked alphabet* is a finite set $\mathscr{F}$ associated with a finite *ranking relation arity* $\subseteq \mathscr{F} \times \mathbb{N}$. Define $\mathscr{F}_n$ as a set $\{f \in \mathscr{F} | \, (f,n) \in arity\}$. The set $T(\mathscr{F})$ of *terms* over the ranked alphabet $\mathscr{F}$ is the smallest set defined by:

1. $\mathscr{F}_0 \subseteq T(\mathscr{F})$
2. if $n \geq 1$, $f \in \mathscr{F}_n$, $t_1, \ldots, t_n \in T(\mathscr{F})$ then $f(t_1, \ldots, t_n) \in T(\mathscr{F})$

Each term can be represented as a finite ordered *tree* $t : D \longrightarrow \mathscr{F}$, which is a mapping from a tree domain into the ranked alphabet such that $\forall u \in D$:

1. if $t(u) \in \mathscr{F}_n$, $n \geq 1$ then $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$
2. if $t(u) \in \mathscr{F}_0$ then $\{j \mid p \cdot j \in D\} = \emptyset$

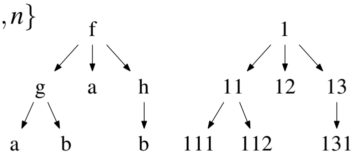


Fig. 1. An Example of a Tree t and its Tree Domain. $dom(t) = \{1, 11, 111, 112, 12, 13, 131\}$, $\mathscr{F} = \{f, g, h, a, b\}$, $\| \, t \, \| = 3$, $t(1) = f$, $t/131 = b$.

As usual in the tree automata literature (e.g., [9]), we use the letter t (possibly with various indices) both to represent a tree as a mathematical object and to name a relation that maps an element of a tree domain to the corresponding alphabet symbol. An example of a tree with its tree domain is given in Figure 1.

The set of all positions in a particular tree t, i.e., its domain, will be denoted $dom(t)$. A *subtree* of t rooted at position u, denoted t/u is defined as $(t/u)(v) = t(u \cdot v)$ and $dom(t/u)$

$= \{v \mid u \cdot v \in dom(t)\}$. We generalize the *dom* operator to sets as usual: $dom(S) = \{dom(u) \mid u \in S\}$. The *height* of a tree t, denoted $\| t \|$, is defined as:

$$\| t \| = \max(\{|u| \text{ such that } u \in dom(t)\})$$

Let $\Xi = \left\{\xi_f \mid f \in \bigcup_{i>0} \mathscr{F}_i\right\}$ be a set of new nullary symbols such that $\Xi \cap \mathscr{F} = \emptyset$. The Ξ set will be used as a set of *placeholders*, such that ξ_f can be substituted only with a tree t whose position one (i.e., the *head*) is labelled with f, i.e., $t(1) = f$. Let $T(\Xi \cup \mathscr{F})$ denote the set of trees over the ranked alphabet and placeholders. For $t, t' \in T(\Xi \cup \mathscr{F})$, we define the *link* operation $t \sharp t'$ by:

$$(t \sharp t')(n) = \begin{cases} t(n) \ \text{if}\ t(n) \notin \Xi \vee (t(n) = \xi_f \wedge f \neq t'(1)) \\ t'(z) \ \text{if}\ n = y \cdot z,\ t(y) = \xi_{t'(1)},\ y \in dom(t),\ z \in dom(t') \end{cases}$$

For any two trees, $t, t' \in T(\mathscr{F})$, the *tree quotient* $t^{-1}t'$ is defined by:

$$t^{-1}t' = \left\{t'' \in T(\Xi \cup \mathscr{F}) \mid t' = t'' \sharp t\right\}$$

The tree quotient operation can be extended to sets, as usual: $t^{-1}S = \left\{t^{-1}t' \mid t' \in S\right\}$. For any $k \geq 0$, define k-root of a tree t as:

$$root_k(t) = \begin{cases} t & \text{if}\ t(1) \in \mathscr{F}_0 \\ \xi_f & \text{if}\ f = t(1), f \in \bigcup_{i>0} \mathscr{F}_i,\ k = 0 \\ f(root_{k-1}(t_1), \ldots, root_{k-1}(t_n)) & \text{if}\ t = f(t_1, \ldots, t_n),\ \| t \| > k > 0 \end{cases}$$

A *finite deterministic bottom-up tree automaton* (FDTA) is defined as a tuple $(Q, \mathscr{F}, \delta, F)$, where Q is a finite set of states, $\mathscr{F}$ is a ranked alphabet, $F \subseteq Q$ is the set of final states, and $\delta = \bigcup_i \delta_i$ is a set of *transition relations* defined as follows: $\delta_0 : \mathscr{F}_0 \to Q$ and for $n > 0$, $\delta_n : (\mathscr{F}_n \times Q^n) \to Q$.

The *k-testable in the strict sense* (k-TSS) languages [22] are intuitively defined by a set of tree patterns allowed to appear as the elements of the language. The following theorem is due to López et al. [24]:

Theorem 1. *Let $\mathscr{L} \subseteq T(\mathscr{F})$. $\mathscr{L}$ is a k-TSS iff for any trees $t_1, t_2 \in T(\mathscr{F})$ such that $root_k(t_1) = root_k(t_2)$, when $t_1^{-1}\mathscr{L} \neq \emptyset \wedge t_2^{-1}\mathscr{L} \neq \emptyset$ then it follows that $t_1^{-1}\mathscr{L} = t_2^{-1}\mathscr{L}$.*

We choose López et al.'s theorem as a definition of k-TSS languages. Other definitions in the literature [14,22] define k-TSS languages in terms of three sets; leaves, roots, and forks. Forks are roots that have at least one placeholder as a leaf. Theorem 1 shows that such more complex definitions are unnecessary. Intuitively, the theorem says that within the language, any two subtrees that agree on the top k levels are interchangeable, meaning that a bottom-up tree automaton has to remember only a finite amount of history. In the next section, we show that we can define an equivalence relation inducing an automaton accepting a k-TSS language using only our definition of the k-root, as expected from Theorem 1.

4 *k*-Testable Tree Automata Inference

4.1 Congruence Relation

We begin with our definition of the equivalence relation that is used to induce a state-minimal automaton from a set of trees. The equivalence relation, intuitively, compares trees up to k levels deep, i.e., compares k-roots.

Definition 1 (Root Equivalence Relation $\sim_k$). *For some $k \geq 0$, two trees $t_1, t_2 \in T(\mathscr{F})$ are root-equivalent with degree k, denoted $t_1 \sim_k t_2$, if $root_k(t_1) = root_k(t_2)$.*

Lemma 1. *The $\sim_k$ relation is a congruence (monotonic equivalence) relation of finite index.*

Proof (Sketch). It is obvious that $\sim_k$ is an equivalence relation (reflexive, symmetric, and transitive). Monotonicity can be proven by a simple induction on the height of the two trees being compared and the $root_k$ definition.

The size of a k-root is bounded by M^k, where $M = \max(\{n \mid \mathscr{F}_n \in F, \mathscr{F}_n \neq \emptyset\})$. Each position u in the k-root's domain can be labelled with at most $|\mathscr{F}_{arity(t(u))}|$ symbols. Thus, $root_k$ generates a finite number of equivalence classes, i.e., is of finite index.

As a consequence of Lemma 1, inference algorithms based on the root equivalence relation need not propagate congruences using union-find [10] algorithms, as the root equivalence relation is a congruence itself.

Definition 2 ($\sim_k$-induced Automaton). *Let $T' \subseteq T(\mathscr{F})$ be a finite set of finite trees. The $A^{\sim_k}(T') = (Q, \mathscr{F}, \delta, F)$ automaton induced by the root equivalence relation $\sim_k$ is defined as:*

$$\begin{aligned} Q &= \{root_k(t') \mid \exists t \in T' \ . \ \exists u \in dom(T') \ . \ t' = t/u\} \\ F &= \{root_k(t) \mid t \in T'\} \\ \delta_0(f) &= f \quad \text{for } f \in \mathscr{F}_0 \\ \delta_n(f, root_k(t_1), \ldots, root_k(t_n)) &= root_k(f(t_1, \ldots, t_n)) \quad \text{for } n \geq 1, f \in \mathscr{F}_n \end{aligned}$$

Corollary 1 (Containment). *From the definition it follows that $\forall k \geq 0 \ . \ T' \subseteq \mathscr{L}(A^{\sim_k}(T'))$. In other words, the $\sim_k$-induced automaton abstracts the set of trees T'.*

Theorem 2. *$\mathscr{L}(A^{\sim_k})$ is a k-TSS language.*

Proof. We need to prove that $\forall t_1, t_2 \in T(\mathscr{F}),\ k \geq 0 \ . \ root_k(t_1) = root_k(t_2) \wedge t_1^{-1}\mathscr{L}(A^{\sim_k}) \neq \emptyset \wedge t_2^{-1}\mathscr{L}(A^{\sim_k}) \neq \emptyset \Rightarrow t_1^{-1}\mathscr{L}(A^{\sim_k}) = t_2^{-1}\mathscr{L}(A^{\sim_k})$. Suppose the antecedent is true, but the consequent is false, i.e., $t_1^{-1}\mathscr{L}(A^{\sim_k}) \neq t_2^{-1}\mathscr{L}(A^{\sim_k})$. Then there must exist t such that $t\sharp t_1 \in \mathscr{L}(A^{\sim_k})$ and $t\sharp t_2 \notin \mathscr{L}(A^{\sim_k})$. Let u be the position of $\xi_{t_2(1)}$, i.e., $(t\sharp t_2)/u = t_2$. Without loss of generality, let t be the tree with minimal $|u|$. Necessarily, $|u| > 1$, as otherwise $t_1^{-1}\mathscr{L}(A^{\sim_k}) = \emptyset$. Let $u = w \cdot i,\ i \in \mathbb{N}$. We prove that $t\sharp t_2$ must be in $\mathscr{L}(A^{\sim_k})$, contradicting the initial assumption, by induction on the length of w.

Base case ($|w| = 1$): Let $(t(w))(1) = f,\ f \in \mathscr{F}_n$. There are two subcases: $n = 1$ and $n > 1$. For $n = 1$, the contradiction immediately follows, as $\delta(f, root_k(t_1))$

$= \delta(f, root_k(t_2))$. For the $n > 1$ case, observe that for all positions $w \cdot j$ such that $1 \leq j \leq n$ and $j \neq i$, $(t \sharp t_1)/w \cdot j = (t \sharp t_2)/w \cdot j = t/w \cdot j$. From that observation and $root_k(t_1) = root_k(t_j)$, it follows that

$$\begin{aligned} &\delta\big((t\sharp t_1/w)(1), root_k(t\sharp t_1/w \cdot 1), \ldots, root_k(t\sharp t_1/w \cdot n)\big) \\ =\ &\delta\big((t\sharp t_2/w)(1), root_k(t\sharp t_2/w \cdot 1), \ldots, root_k(t\sharp t_2/w \cdot n)\big) \end{aligned}$$

Induction step ($|w| > 1$): Let $w = w' \cdot m,\ m \in \mathbb{N}$. From the induction hypothesis, we know that for all m, $root_k(t\sharp t_1/w) = root_k(t\sharp t_2/w)$, thus it follows:

$$\begin{aligned} &\delta\big((t\sharp t_1/w')(1), root_k(t\sharp t_1/w' \cdot 1), \ldots, root_k(t\sharp t_1/w' \cdot n)\big) \\ =\ &\delta\big((t\sharp t_2/w')(1), root_k(t\sharp t_2/w' \cdot 1), \ldots, root_k(t\sharp t_2/w' \cdot n)\big) \end{aligned}$$

Theorem 3 (Minimality). *$A^{\sim_k}$ is state-minimal.*

Proof. Follows from Myhill-Nerode Theorem [20, pg. 72] and Lemma 1.

Minimality is not absolutely crucial for malware analysis in a laboratory setting, but it is important in practice, where antivirus tools can't impose a significant system overhead and have to react promptly to infections.

Theorem 4 (Garcia [14]). $\mathscr{L}(A^{\sim_{k+1}}) \subseteq \mathscr{L}(A^{\sim_k})$

An important consequence of Garcia's theorem is that the k factor can be used as an abstraction knob — the smaller the k factor, the more abstract the inferred automaton. This tunability is particularly important in malware detection. One can't hope to design a classifier capable of perfect malware and goodware distinction. Thus, tunability of the false positive (goodware detected as malware) and false negative (undetected malware) ratios is crucial. More abstract automata will result in more false positives and fewer false negatives.

4.2 Inference Algorithm

In this section, we present our inference algorithm, but before proceeding with the algorithm, we discuss some practical aspects of inference from data-flow dependency graphs. As discussed in Section 2, we use taint analysis to compute data-flow dependencies among executed syscalls at runtime. The result of that computation is not a tree, but an acyclic directed graph, i.e., a partial order of syscalls ordered by the data-flow dependency relation, and expansion of such a graph into a tree could cause exponential blowup. Thus, it would be more convenient to have an inference algorithm that operates directly on graphs, without expanding them into trees.

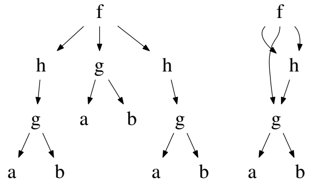


Fig. 2. Folding a Tree into a Maximally-Shared Graph

Fortunately, such an algorithm is only slightly more complicated than the one that operates on trees. In the first step, our implementation performs common subexpression elimination [1] on the dependency graph to eliminate syntactic redundancies. The result is a maximally-shared graph [2], i.e., an acyclic directed graph with shared common subgraphs. Figure 2 illustrates how a tree can be folded into a maximally-shared graph. In the second step, we compute a hash for each k-root in the training set. The hash is later used as a hash table key. Collisions are handled via chaining [10], as usual, but chaining is not described in the provided algorithms. The last step of the inference algorithm traverses the graph and folds it into a tree automaton, using the key computed in the second phase to identify equivalent k-roots, which are mapped to the same state.

To simplify the exposition, we shall use the formal machinery developed in Section 3 and present indexing and inference algorithms that work on trees. The extension to maximally-shared graphs is trivial and explained briefly later.

```
input   : Tree $t$, factor $k$
result  : Key computed for every subtree of $t$

$tmp \leftarrow hash(t(1))$
foreach $1 \leq i \leq arity(t(1))$ do
    $t_s \leftarrow t/i$
    $tmp \leftarrow tmp \oplus hash(t_s.key)$
    ComputeKey($t_s, k$)
$t.key \leftarrow tmp$
```

Algorithm 1. ComputeKey — Computing k-Root Keys (Hashes). The $\oplus$ operator can be any operator used to combine hashes, like bitwise exclusive OR. The $hash : \mathscr{F} \to \mathbb{N}$ function can be implemented as a string hash, returning an integral hash of the alphabet symbols.

Algorithm 1 traverses tree t in postorder (children before the parent). Every subtree has a field *key* associated with its head, and the field is assumed to be initially zero. If the algorithm is called once, for tree t, the key of the head of each subtree t_s will consist only of the hash of the alphabet symbol labeling t_s, i.e., $hash(t_s(1))$. If the algorithm is called twice (on the same tree), the key of the head of each subtree will include the hash of its own label and the labels of its children, and so on. Thus, after k calls to ComputeKey, the key of each node will be equal to its k-root key. Note that the temporary key, stored in the *tmp* variable, has to be combined with the children's $(k-1)$-root key. The algorithm can be easily extended to operate on maximally-shared graphs, but has to track visited nodes and visit each node only once in postorder. The complexity of the algorithm is $\mathscr{O}(k \cdot N)$, where N is the size of the tree (or maximally-shared graph). For multi-rooted graphs (or when processing multiple trees), all roots can be connected by creating a synthetic super-root of all roots, and the algorithm is then called k times with the super-root as the first operand.

Algorithm 2 constructs the $A^{\sim_k}$ automaton. The tree (alternatively maximally-shared graph) used for training is traversed in postorder, and k-root of each subtree is used to retrieve the representative for each $\sim_k$-induced equivalence class. Multi-rooted graphs can be handled by introducing super-roots (as described before). Amortized complexity is $\mathscr{O}(kN)$, where N is the size of the tree (or maximally-shared graph).

```
input  : Tree $t$, factor $k$, alphabet $\mathscr{F}$
output: $A^{\sim k} = (Q, \mathscr{F}, \delta, F)$
foreach subtree $t_s$ in $\{t/u \mid u \in dom(t)\}$ traversed in postorder do
    if $rep[t_s.key] = \emptyset$ then
        $q \leftarrow root_k(t_s)$
        $rep[t_s.key] = q$
        $Q \leftarrow Q \cup q$
    $n \leftarrow arity(t_s(1))$
    $\delta \leftarrow \Big(\big(t_s(1), rep[(t_s/1).key], \ldots, rep[(t_s/n).key]\big), rep[t_s.key]\Big)$
$F = F \cup rep[t.key]$
return $(Q, \mathscr{F}, \delta, F)$
```

Algorithm 2. k-Testable Tree Automaton Inference. The $rep : hash(root_k(T(\mathscr{F}))) \rightarrow root_k(T(\mathscr{F}))$ hash map contains representatives of equivalence classes induced by $\sim_k$. Collisions are handled via chaining (not shown).

5 Experimental Results

5.1 Benchmarks

For the experiments, we use two sets of benchmarks: the malware and the goodware set. The malware set comprises 2631 samples pre-classified into 48 families. Each family contains 5–317 samples. We rely upon the classification of Christodorescu et al. [6] and Fredrikson et al. [13].[2] The classification was based on the reports from antivirus tools. For a small subset of samples, we confirmed the quality of classification using virustotal.com, a free malware classification service. However, without knowing the internals of those antivirus tools and their classification heuristics, we cannot evaluate the quality of the classification provided to us. Our classification experiments indicate that the classification antivirus tools do might be somewhat ad-hoc. Table 1 shows the statistics for every family.

The goodware set comprises 33 commonly used applications: AdobeReader, Apple SW Update, Autoruns, Battle for Wesnoth, Chrome, Chrome Setup, Firefox, Freecell, Freeciv, Freeciv server, GIMP, Google Earth, Internet Explorer, iTunes, Minesweeper, MSN Messenger, Netcat port listen and scan, NetHack, Notepad, OpenOffice Writer, Outlook Express, Ping, 7-zip archive, Skype, Solitaire, Sys info, Task manager, Tux Racer, uTorrent, VLC, Win. Media Player, and WordPad. We deemed these applications to be representative of software commonly found on the average user's computer, from a number of different vendors and with a diverse set of behaviors. Also, we used two micro benchmarks: a HelloWorld program written in C and a file copy program. Microbenchmarks produce few small dependency graphs and therefore might be potentially more susceptible to be misidentified for malware.

[2] The full set of malware contains 3136 samples, but we eliminated samples that were not executable, executable but not analyzable with Pin (i.e., MS-DOS executables), broken executables, and those that were incompatible with the version of Windows (XP) that we used for experiments.

Table 1. Malware Statistics per Family. All dependency graphs were obtained by running each sample for 120sec in a controlled environment. The identifier that will be used in later graphs is given in the first column. The third column shows the number of samples per family. The *Avg.* column shows the average height of the dependency graphs across all the samples in the family. The *Nodes* column shows the total number of nodes in the dependency graph (after CSE). The *Trees* column shows the total number of different trees (i.e., roots of the dependency graph) across all the samples. The *Max* column gives the maximal height of any tree in the family.

ID	Family Name	Samples	Avg.	Nodes	Trees	Max.
1	ABU.Banload	16	7.71	544	303	21
2	Agent	42	8.86	965	593	27
3	Agent.Small	15	8.88	950	588	27
4	Allaple.RAHack	201	8.78	1225	761	44
5	Ardamax	25	6.21	144	69	16
6	Bactera.VB	28	7.09	333	177	28
7	Banbra.Banker	52	13.97	1218	686	37
8	Bancos.Banker	46	14.05	742	417	45
9	Banker	317	17.70	2952	1705	43
10	Banker.Delf	20	14.78	939	521	50
11	Banload.Banker	138	19.38	2370	1332	152
12	BDH.Small	5	5.82	348	199	21
13	BGM.Delf	17	7.04	339	199	25
14	Bifrose.CEP	35	11.17	1190	698	50
15	Bobax.Bobic	15	8.98	859	526	30
16	DKI.PoisonIvy	15	9.22	413	227	40
17	DNSChanger	22	12.62	874	483	36
18	Downloader.Agent	13	12.89	1104	613	49
19	Downloader.Delf	22	10.76	1486	906	32
20	Downloader.VB	17	10.80	516	266	29
21	Gaobot.Agobot	20	17.54	1812	1052	45
22	Gobot.Gbot	58	7.01	249	134	22
23	Horst.CMQ	48	16.86	1030	541	42
24	Hupigon.ARR	33	23.58	2388	1244	55
25	Hupigon.AWQ	219	24.63	7225	3758	62
26	IRCBot.Sdbot	66	16.51	3358	1852	47
27	LdPinch	16	16.88	1765	1012	66
28	Lmir.LegMir	23	9.00	1112	667	28
29	Mydoom	15	5.78	484	305	20
30	Nilage.Lineage	24	9.64	1288	657	83
31	Games.Delf	11	8.44	971	632	22
32	Games.LegMir	76	17.18	11892	8184	59
33	Games.Mmorpg	19	7.00	654	478	25
34	OnLineGames	23	7.30	718	687	16
35	Parite.Pate	71	14.31	1420	816	36
36	Plemood.Pupil	32	6.29	330	189	24
37	PolyCrypt.Swizzor	43	10.32	415	213	30
38	Prorat.AVW	40	23.47	1031	572	58
39	Rbot.Sdbot	302	14.23	4484	2442	47
40	SdBot	75	14.13	2361	1319	40
41	Small.Downloader	29	11.93	2192	1216	34
42	Stration.Warezov	19	9.76	1682	1058	34
43	Swizzor.Obfuscated	27	21.75	1405	770	49
44	Viking.HLLP	32	7.84	512	315	24
45	Virut	115	11.76	3149	1953	40
46	VS.INService	17	11.42	307	178	37
47	Zhelatin.ASH	53	12.14	1919	1146	39
48	Zlob.Puper	64	15.16	2788	1647	90

In behavioral malware detection, there is always a contention between the amount of time the behavior is observed and the precision of the analysis. For malware samples, which are regularly small pieces of software, we set the timeout to 120sec of running in our environment. For goodware, we wanted to study the impact of the runtime on the height and complexity of generated dependency graphs, and the impact of these differences on the false positive rates. Thus, we ran goodware samples for both 120 and 800sec. To give some intuition of how that corresponds to the actual native runtime, it takes approximately 800s in our DTA analysis environment for Acrobat Reader to open a document and display a window.

We noticed a general tendency that detection and classification tend to correlate positively with the average height of trees in samples used for training and testing. We provide the average heights in Table 1.

5.2 Malware and Goodware Recognition

For our malware recognition experiments, we chose at random 50% of the entire malware set for training, and used the rest and the entire goodware set as test sets. Training with $k = 4$ took around 10sec for the entire set of 1315 training samples, and the time required for analyzing each test sample was less than the timing jitter (sub-second range). All the experiments were performed in Ubuntu 10.04, running in a VMware

7.1.3 workstation, running on Win XP Pro and dual-core 2.5GHz Intel machine with 4GB of RAM. In Figure 3a (resp. 3b), we show the results, using the goodware dependency graphs produced with an 800sec (resp. 120sec) timeout.

The detection works as follows. We run all the trees (i.e., roots of the dependency graph) in each test sample against the inferred automaton. First, we sort the trees by height, and then compute how many trees for each height are accepted by the automaton. Second, we score the sample according to the following function:

$$score = \frac{\sum_i \frac{accepted_i}{total_i} * i}{\sum_i i} \tag{1}$$

where i ranges from 1 to the maximal height of any tree in the test sample (the last column of Table 1), $accepted_i$ is the number of trees with height i accepted by the automaton, and $total_i$ is the total number of trees with height i. The test samples that produce no syscall dependency graphs are assumed to have score zero.

The score can range from 0 to 1. Higher score signifies a higher likelihood the sample is malicious. The ratio in the nominator of Eq. 1 is multiplied by the depth of the tree to filter out the noise from shallow trees, often generated by standard library functions, that have very low classification power.

The results turned out to be slightly better with an 800sec timeout than with the 120sec timeout, as the average height of dependency graphs was slightly larger. As expected, we found that with the rising k factor (and therefore decreasing level of abstraction), the capability of inferred tree automaton to detect malware decreases, which obviously indicates the value of generalization achieved through tree automata inference. On the other hand, with the rising k factor, the detection becomes more precise and therefore the false positive rate drops down. Thus, it is important to find the right level of abstraction. In our experiments, we determined that $k = 4$ was the optimal abstraction level. The desired ratio between false positives and negatives can be adjusted by selecting the score threshold. All samples scoring above (resp. below) the threshold are declared malware (resp. goodware). For example, for $k = 4$, timeout of 800sec, and score 0.6, our approach reports two false positives (5%) — Chrome setup and NetHack, and 270 false negatives (20%), which corresponds to an 80% detection rate. For $k = 4$, timeout of 800sec, and score 0.6, our approach reports one additional false positive (System info), and the same number of false negatives, although a few malware samples are somewhat closer to the threshold. Obviously, the longer the behavior is observed, the better the classification.

It is interesting to notice that increasing the value of k above 4 does not make a significant difference in (mis)detection rates. We ran the experiments with k up to 10, but do not show the results as they are essentially the same as for $k = 4$. From our preliminary analysis, it seems that generalization is effective when a sequence of dependent syscalls are executed within a loop. If two samples execute the same loop body a different number of times, our approach will be able to detect that. Changing k effectively changes the window with which such loop bodies are detected. During the inference, it seems like one size (of k) does not fit all cases. We believe that by analyzing the repetitiveness of patterns in dependency graphs, we could detect the sizes of loop bodies much more accurately, and adjust the k factor according to the size of the body, which

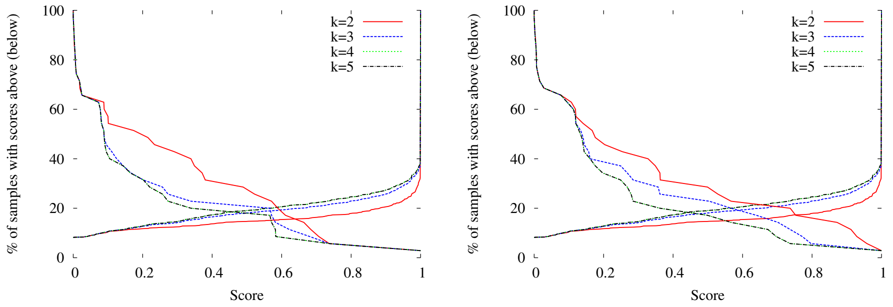


Fig. 3. Malware and Goodware Recognition. Timeouts for generating the dependency graphs were 120sec for malware test and training sets and 800sec (resp. 120sec) for the goodware test set in the figure on the left (resp. right). The training set consists of 50% of the entire malware set, chosen at random. The test set consists of the remaining malware samples (curves rising from left to right), and the goodware set (curves falling from left to right). The rising curves represent the percentage of malware samples for which the computed score was *less* than the corresponding value on the x axis. The falling curves represent the percentage of goodware samples for which the score was *greater* than the corresponding value on the x axis. The figure shows curves for four different values of k, there is essentially no difference between the cases when $k = 4$ and $k = 5$. For the rising curves, the lowest curve is for $k = 2$, the next higher one for $k = 3$, and the two highest ones for the remaining cases. For the falling curves, the ordering is reversed. The optimal score for distinguishing malware from goodware is the lowest intersection of the rising and falling curves for the same k.

should in turn improve the generalization capabilities of the inference algorithm. Many other improvements of our work are possible, as discussed later.

5.3 Malware Classification

We were wondering what is the classification power of inferred automata, so we did the following experiment. We divided at random each family into training and test sets of equal size. For each training set, we inferred a family-specific tree automaton. For each test set, we read the dependency graphs for all the samples in the set, and compute a single dependency graph, which is then analyzed with the inferred tree automaton. The scores are computed according to Equation 1, with $k = 3$. The only difference from the experiment done in the previous section is that the score is computed for the entire test set, not individual samples in the set. Results are shown in Figure 4.

The pronounced diagonal in Figure 4 shows that our inferred automata clearly have a significant classification power and could be used to classify malware into families. There is some noise as well. The noise could be attributed to many factors: over-generalization, over- and under-tainting of our DTA [5,19], insufficiently large dependency graphs, frequently used dynamic libraries that are shared by many applications and malware, and a somewhat ad-hoc pre-classification by the antivirus tools.

6 Limitations

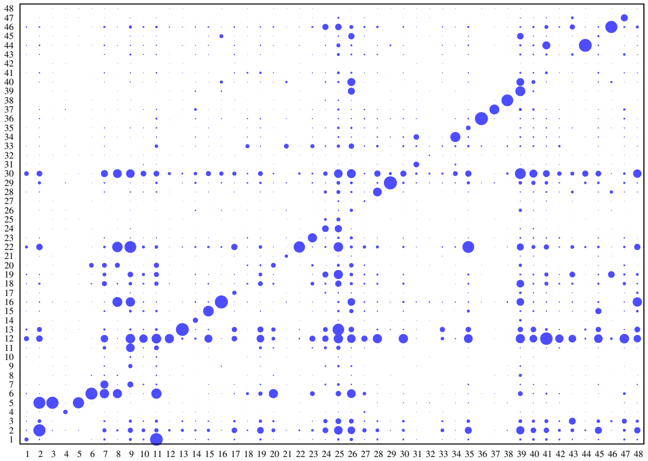


Fig. 4. Malware Classification Results. The x (y) axis represents the training (test) sets. The size of the shaded circle corresponds to the score computed by Eq. 1.

There are several inherent limitations of our approach. An attacker could try to mask syscall dependencies so as to be similar (or the same) as those of benign applications. This class of attacks are known as *mimicry attacks* [33]. All intrusion and behavioral malware detection approaches are susceptible to mimicry attacks. One way to make this harder for the attacker, is to make the analysis more precise, as will be discussed in the following section.

Triggering interesting malware behavior is another challenge. Some behaviors could be triggered only under certain conditions (date, web site visited, choice of the default language, users' actions,...). Moser et al. [28,4] proposed DART [16] as a plausible approach for detecting rarely exhibited behaviors.

As discussed earlier, our DTA environment slows the execution several thousand times, which is obviously too expensive for real-time detection. A lot of work on malware analysis is done in the lab setting, where this is not a significant constraint, but efficiency obviously has to be improved if taint-analysis based approaches are ever to be broadly used for malware detection. Hardware taint-analysis accelerators are a viable option [30,11], but we also expect we could probably achieve an order of magnitude speedup of our DTA environment with a very careful optimization.

7 Conclusions and Future Work

In this paper, we presented a novel approach to detecting likely malicious behaviors and malware classification based on tree automata inference. We showed that inference, unlike simple matching of dependency graphs, does generalize from the learned patterns and therefore improves detection of yet unseen polymorphic malware samples. We proposed an improved k-testable tree automata inference algorithm and showed how the k factor can be used as a knob to tune the abstraction level. In our experiments, our approach detects 80% of the previously unseen polymorphic malware samples, with a 5% false positive rate, measured on a diverse set of benign applications.

There are many directions for further improvements. The classification power of our approach could be improved by a more precise analysis of syscall parameters (e.g., using their actual values in the analysis), by dynamically detecting the best value of the k factor in order to match the size of loop bodies that produce patterns in the dependency graphs, by using goodware dependency graphs as negative examples during training, and by combining our approach with the leap mining approach [13].

Another interesting direction is inference of more expressive tree languages. Inference of more expressive languages might handle repeated patterns more precisely, generalizing only as much as needed to fold a repeatable pattern into a loop in the tree automaton. Further development of similar methods could have a broad impact in security, forensics, detection of code theft, and perhaps even testing and verification, as the inferred automata can be seen as high-level abstractions of program's behavior.

Acknowledgments

We are grateful to Matt Fredrikson and Somesh Jha for sharing their library of classified malware with us. We would especially like to thank Lorenzo Martignoni, who wrote the libwst library [26] for extracting and parsing arguments of Windows's system calls. We also thank reviewers for their insightful and constructive comments.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc. Redwood City (1986)
2. Babić, D.: Exploiting Structure for Scalable Software Verification. Ph.D. thesis, University of British Columbia, Vancouver, Canada (2008)
3. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Architecture of a morphological malware detector. Journal in Computer Virology 5, 263–270 (2009)
4. Brumley, D., Hartwig, C., Zhenkai Liang, J.N., Song, D., Yin, H.: Automatically Identifying Trigger-based Behavior in Malware. In: Botnet Detection Countering the Largest Security Threat, Advances in Information Security, vol. 36, pp. 65–88. Springer, Heidelberg (2008)
5. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proc. of 13th USENIX Security Symp. (2004)
6. Christodorescu, M., Jha, S.: Testing malware detectors. In: ISSTA 2004: Proc. of the 2004 ACM SIGSOFT Int. Symp. on Software Testing and Analysis, pp. 34–44. ACM Press, New York (2004)
7. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proc. of the the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering, pp. 5–14. ACM Press, New York (2007)
8. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: SP 2005: Proc. of the 2005 IEEE Symp. on Security and Privacy, pp. 32–46. IEEE Computer Society Press, Los Alamitos (2005)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
11. Crandall, J., Chong, F.: Minos: Control data attack prevention orthogonal to memory model. In: the Proc. of the 37th Int. Symp. on Microarchitecture, pp. 221–232. IEEE Computer Society Press, Los Alamitos (2005)

12. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: Proc. of the 1996 IEEE Symp. on Security and Privacy, pp. 120–129. IEEE Computer Society Press, Los Alamitos (1996)
13. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: Proc. of the 2010 IEEE Symp. on Security and Privacy, pp. 45–60. IEEE Computer Society Press, Los Alamitos (2010)
14. García, P.: Learning *k*-testable tree sets from positive data. Tech. rep. In: Dept. Syst. Inform. Comput. Univ. Politecnica Valencia, Spain (1993)
15. García, P., Vidal, E.: Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. IEEE Trans. Pattern Anal. Mach. Intell. 12, 920–925 (1990)
16. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. pp. 213–223. ACM Press, New York (2005)
17. Gold, E.M.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
18. Holzer, A., Kinder, J., Veith, H.: Using verification technology to specify and detect malware. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 497–504. Springer, Heidelberg (2007)
19. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp.San Diego, CA (2011)
20. Khoussainov, B., Nerode, A.: Automata Theory and Its Applications. Birkhäuser, Basel (2001)
21. King, J.C.: Symbolic execution and program testing. Comm. of the ACM 19(7), 385–394 (1976)
22. Knuutila, T.: Inference of *k*-testable tree languages. In: Bunke, H. (ed.) Advances in Structural and Syntactic Pattern Recognition: Proc. of the Int. Workshop, pp. 109–120. World Scientific, Singapore (1993)
23. Kolbitsch, C., Milani, P., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: The 18th USENIX Security Symp. (2009)
24. López, D., Sempere, J.M., García, P.: Inference of reversible tree languages. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(4), 1658–1665 (2004)
25. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of the 2005 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., pp. 190–200. ACM Press, New York (2005)
26. Martignoni, L., Paleari, R.: The libwst library (a part of WUSSTrace) (2010), `http://code.google.com/p/wusstrace/`
27. Matrosov, A., Rodionov, E., Harley, D., Malcho, J.: Stuxnet under the microscope. Tech. rep. Eset (2010)
28. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: SP 2007: Proc. of the IEEE Symp. on Security and Privacy, pp. 231–245. IEEE Computer Society Press, Los Alamitos (2007)
29. Newsome, J., Song, D.: Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In: Proc. of the Network and Distributed Systems Security Symp. (2005)
30. Suh, G., Lee, J., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. ACM SIGOPS Operating Systems Review 38(5), 85–96 (2004)
31. Symantec: Symantec global internet security threat report: Trends for 2009. Tech. rep.Symantec, vol. XV (2010)
32. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proc. of the IEEE Symp. on Security and Privacy, p. 156. IEEE Computer Society Press, Los Alamitos (2001)
33. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proc. of the 9th ACM Conf. on Comp. and Comm. Security, pp. 255–264. ACM Press, New York (2002)
34. You, I., Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In: Int. Conf. on Broadband, Wireless Computing, Communication and Applications, pp. 297–300 (2010)
35. Zalcstein, Y.: Locally testable languages. J. Comput. Syst. Sci. 6(2), 151–167 (1972)

State/Event-Based LTL Model Checking under Parametric Generalized Fairness

Kyungmin Bae and José Meseguer

Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana IL 61801
{kbae4,meseguer}@cs.uiuc.edu

Abstract. In modeling a concurrent system, fairness constraints are usually considered at a specific granularity level of the system, leading to many different variants of fairness: transition fairness, object/process fairness, actor fairness, etc. These different notions of fairness can be unified by making explicit their *parametrization* over the relevant entities in the system as *universal quantification*. We propose a *state/event-based* framework as well as an on-the-fly model checking algorithm to verify LTL properties under universally quantified parametric fairness assumptions, specified by *generalized* strong/weak fairness formulas. It enables verification of temporal properties under fairness conditions associated to dynamic entities such as new process creations. We have implemented our algorithm within the Maude system.

Keywords: Model checking, Parameterized Fairness, State/Event LTL.

1 Introduction

Fairness assumptions are often necessary to verify a liveness property of a concurrent system. Without fairness, unrealistic counterexamples can be produced, such as a process that is never executed even though the process is continuously enabled. A usual way to model check an LTL property under fairness assumptions is to re-formulate the property such that fairness requirements become a conjunction of premises implying the original property [6]. Since this method is impractical for model checking properties under complex fairness assumptions, several specialized algorithms have been proposed, e.g., [11,14,16,19].

In practice, however, descriptions of fairness are dependent on specific models or languages. There are many different variants of the fairness concepts, such as transition fairness, object/process fairness, actor fairness, etc [1,13]. In general, such variants do not coincide, even though their temporal behaviors like strong/weak fairness are all similar. It becomes difficult to represent fairness notions which are not directly supported by a specific modeling language.

Such different variants of fairness can be unified by making explicit the *parametrization* of fairness formulas over the relevant spatial entities in a system [20]. Fairness is then expressed by a *universally quantified* temporal formula, where variables in the formula range over the relevant system entities. We use a

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 132–148, 2011.

state/event-based LTL (SE-LTL) because fairness notions usually involve both states and events. For example, weak process fairness can be expressed by the universally quantified formula: $\forall x \in \mathit{Process}.\ \Diamond\Box \mathit{enabled}(x) \rightarrow \Box\Diamond \mathit{execute}(x)$, where $\mathit{enabled}(x)$ is a state predicate and $\mathit{execute}(x)$ is an event predicate.

We present a framework to verify SE-LTL properties under parameterized fairness constraints, given by generalized strong (resp., weak) fairness formulas of the form $\forall \overline{x}\ \Box\Diamond\Phi \rightarrow \Box\Diamond\Psi$ (resp., $\forall \overline{x}\ \Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$) in SE-LTL. For parameterized fairness, the number of entities in the system over which the parametrization ranges can be unbounded[1] and may change during execution. Instead, previous approaches require that the number of fairness conditions is already determined. For example, in process fairness with dynamic process creation, fairness is parametric over a number of processes unknown a priori. Our framework is based on the notion of *parameter abstraction* to make explicit the fact that, even though the domain of entities or parameters is infinite, only a *finite* number of parameters are meaningful in a single state for fairness purposes. In process algebra, meaningful parameters are the processes in the state, and strong/weak fairness is vacuously satisfied for the processes not existing in a system.[2]

We have developed an on-the-fly SE-LTL model checking algorithm (available at [2]) using a parameter abstraction that can directly handle universally quantified fairness formulas; its complexity is linear in the number of fairness instances (see Sec. 4). We have implemented our algorithm within the Maude system [7], which is a verification framework for concurrent systems where many concurrent systems, including object-based systems and process algebras, can be naturally described. This model checking algorithm can verify liveness properties of complex examples with dynamic entities having an unpredictable number of fairness assumptions (see Sec. 5). To the best of our knowledge, such dynamic parametric fairness assumptions cannot be easily handled by other existing model checkers.

Related Work. Parameterization has long been considered as a way to describe fairness of concurrent systems. The theorem proving of liveness properties commonly involves parameterized fairness properties. Fairness notions supported by usual modeling languages are parameterized, e.g., process fairness [13] is parameterized by processes. However, such fairness notions are parameterized *only* by specific entities, depending on the system modeling language. Localized fairness [20] was introduced as a unified notion to express different variants of fairness, depending on the chosen system granularity level. Localized fairness can be parameterized by *any* entity in a system, but generalized versions of strong/weak fairness were not discussed in [20]. Our work extends localized fairness to incorporate generalized fairness, and answers the question of how to model check LTL properties under such generalized fairness conditions.

To verify a property φ under parameterized fairness assumptions, the usual method is to construct the conjunction of corresponding instances of fairness,

[1] For finite-state systems the number is finite, but it may be impossible to determine such a number from the initial state without exploring the entire state space.

[2] E.g., $\mathit{enabled}(p)$ is false for all states if a process p does not exist in the system, and therefore, $\Diamond\Box \mathit{enabled}(p) \rightarrow \Box\Diamond \mathit{execute}(p)$ is vacuously satisfied.

and to apply either: (i) a standard LTL model checking algorithm for the reformulated property $\mathit{fair} \rightarrow \varphi$, or (ii) a specialized model checking algorithm which handles fairness, based on either explicit graph search [11,14,19], or a symbolic algorithm [16]. Approach (i) is inadequate for fairness, since the time complexity is exponential in the number of strong fairness conditions, while the latter is linear. Furthermore, compiling such a formula, expressing a conjunction of fairness conditions, into Büchi automata is usually not feasible in reasonable time [24]. There are several tools to support the specialized algorithms such as PAT [22] and Maria [19]. Our algorithm is related to the second approach to deal with fairness, but it does not require pre-translation of parameterized fairness, and can handle *dynamic* generalized fairness instances. There are also other methods to support parameterized fairness not based on standard model checking methods, such as regular model checking [4] and compositional reasoning [8].

This paper is organized as follows. Section 2 presents the necessary background about fairness and SE-LTL. Section 3 introduces a logical framework for parameterized fairness, and Section 4 presents the automata-based model checking algorithm under parameterized fairness. Section 5 illustrates case studies, Section 6 shows experimental results, and Section 7 presents some conclusions.

2 Fairness Expressed in a State/Event-Based Logic

Fairness generally means that, if a certain kind of choice is sufficiently often provided, then it is sufficiently often taken. For example, strong fairness means that, if a given choice is available infinitely often, then it is taken infinitely often. Similarly, weak fairness means that, if the choice is continuously available beyond a certain point, then it is taken infinitely often.

In order to express fairness using *only* logic formulas, we need a logic to specify properties involving both states and events. Neither state-based logics such as LTL nor event-based logics are usually sufficient to express fairness as logic formulas on the original system, although system transformations can be used to “encode” events in the state, typically at the price of a bigger state space. Many modeling languages using state-based logics incorporate specific kinds of fairness properties to avoid such problems, but the expressiveness of fairness is then limited to the given kind of fairness thus supported.

State/event linear temporal logic (SE-LTL) [5] is a simple state/event extension of linear temporal logic. The only syntactic difference between LTL and SE-LTL is that the latter can have both state propositions and event propositions. Given a set of state propositions *AP* and a set of event propositions *ACT*, the syntax of SE-LTL formulas over *AP* and *ACT* is defined by $\varphi ::= p \mid \delta \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \bigcirc\varphi \mid \varphi \mathbf{U} \varphi'$, where $p \in \mathit{AP}$ and $\delta \in \mathit{ACT}$. Other operators can be defined by equivalences, e.g., $\Diamond\varphi \equiv \mathit{true}\mathbf{U}\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

The semantics of SE-LTL is defined on a *labeled Kripke structure* (LKS), which is a natural extension of a Kripke structure with transition labels. The model checking problem of SE-LTL formulas can be characterized by automata-theoretic techniques on LKS similar to the LTL case [3,5], which use the Büchi automaton $\mathscr{B}_{\neg\varphi}$ with size $O(2^{|\varphi|})$ associated to the negated formula $\neg\varphi$, where the alphabet of $\mathscr{B}_{\neg\varphi}$ is a set of subsets of the disjoint union $AP \uplus ACT$.

Definition 1. *A* labeled Kripke structure *is a 6-tuple* $(S, S_0, AP, \mathcal{L}, ACT, T)$ *with* S *a set of* states*,* $S_0 \subseteq S$ *a set of* initial states*,* AP *a set of* atomic state propositions*,* $\mathcal{L} : S \to \mathcal{P}(AP)$ *a* state-labeling function*,* ACT *a set of* atomic events*, and* $T \subseteq S \times \mathcal{P}(ACT) \times S$ *a* labeled transition relation*.*

Note that each transition of an LKS is labeled by a *set* A of atomic events, which enables us to describe a pattern of an event. A labeled transition $(s, A, s') \in T$ is often denoted by $s \xrightarrow{A} s'$. A *path* (π, α) of an LKS is an infinite sequence $\langle \pi(0), \alpha(0), \pi(1), \alpha(1), \ldots \rangle$ such that $\pi(i) \in S$, $\alpha(i) \subseteq ACT$, and $\pi(i) \xrightarrow{\alpha(i)} \pi(i+1)$ for each $i \geq 0$. $(\pi, \alpha)^i$ denotes the suffix of (π, α) beginning at position $i \in \mathbb{N}$. A SE-LTL formula φ is satisfied on an LKS $\mathcal{K}$, denoted by $\mathcal{K} \models \varphi$, if and only if for each path (π, α) of $\mathcal{K}$ starting from an initial state, the path satisfaction relation $\mathcal{K}, (\pi, \alpha) \models \varphi$ holds, which is defined inductively as follows:

- $\mathcal{K}, (\pi, \alpha) \models p$ iff $p \in \mathcal{L}(\pi(0))$
- $\mathcal{K}, (\pi, \alpha) \models \delta$ iff $\delta \in \alpha(0)$
- $\mathcal{K}, (\pi, \alpha) \models \neg\varphi$ iff $\mathcal{K}, (\pi, \alpha) \not\models \varphi$
- $\mathcal{K}, (\pi, \alpha) \models \varphi \wedge \varphi'$ iff $\mathcal{K}, (\pi, \alpha) \models \varphi$ and $\mathcal{K}, (\pi, \alpha) \models \varphi'$
- $\mathcal{K}, (\pi, \alpha) \models \bigcirc\varphi$ iff $\mathcal{K}, (\pi, \alpha)^1 \models \varphi$
- $\mathcal{K}, (\pi, \alpha) \models \varphi\, \mathbf{U} \varphi'$ iff $\exists k \geq 0.\ \mathcal{K}, (\pi, \alpha)^k \models \varphi'$, $\forall 0 \leq i < k.\ \mathcal{K}, (\pi, \alpha)^i \models \varphi$

We can define fairness properties of an LKS as SE-LTL formulas. A strong fairness (resp. weak fairness) condition with respect to an event proposition α is expressed by the SE-LTL formula $\Box\Diamond \mathit{enabled}.\alpha \to \Box\Diamond\alpha$ (resp., $\Diamond\Box \mathit{enabled}.\alpha \to \Box\Diamond\alpha$). A special state proposition *enabled*.α is defined for each state s of $\mathcal{K}$ such that $\mathit{enabled}.\alpha \in \mathcal{L}(s)$ iff there exists a transition $s \xrightarrow{A} s' \in T$ with $\alpha \in A$. *Generalized* strong (resp., weak) fairness conditions are defined by SE-LTL formulas of the form $\Box\Diamond\Phi \to \Box\Diamond\Psi$ (resp, $\Diamond\Box\Phi \to \Box\Diamond\Psi$), where Φ and Ψ are Boolean formulas that do not contain any temporal operators. Many fairness notions, that arise in real examples, can be expressed by generalized strong/weak fairness formulas [18]. For example, the minimal progress of a set of events $\{\alpha_1, \ldots, \alpha_k\}$ can be expressed by $\Diamond\Box(\mathit{enabled}.\alpha_1 \vee \cdots \vee \mathit{enabled}.\alpha_k) \to \Box\Diamond(\alpha_1 \vee \cdots \vee \alpha_k)$. However, imposing such fairness conditions for each relevant entity, e.g., for each process, may require a large or even infinite set of such formulas.

3 Parameterized Fairness as Quantified SE-LTL

Besides a temporal perspective regarding frequency of a choice, fairness also has a spatial perspective depending on the relation between the choice and the system structure. The variants of fairness from such system structures can be unified by making explicit their *parametrization* over the chosen spatial entities [20]. To specify parameterized fairness conditions, we use first-order SE-LTL over parameterized propositions. Fairness is then expressed by a universally quantified SE-LTL formula, where variables range over the relevant entities in the system.

3.1 Quantification of Parametric SE-LTL Formulas

In order to define parametric SE-LTL formulas, we should make the state and event propositions parametric on the relevant entities. Such entities need not be states: they could be process names, messages, or other data structures. Therefore, we allow *parametric* state propositions $p \in \Pi$ (resp., event propositions $\delta \in \Omega$) of the form $p(x_1, \dots, x_n)$ (resp., $\delta(x_1, \dots, x_m)$).

Definition 2. *A* parameterized labeled Kripke structure *over a set of parameters $\mathcal{C}$ is a tuple $\mathcal{K} = (S, S_0, \Pi, \mathcal{L}, \Omega, T)$ such that $\mathcal{K}_\mathcal{C} = (S, S_0, AP_\mathcal{C}, \mathcal{L}, ACT_\mathcal{C}, T)$ is an ordinary LKS with state propositions $AP_\mathcal{C} = \{p(\overline{a}) \mid \overline{a} \in \mathcal{C}^n, p \in \Pi, n \in \mathbb{N}\}$ and event propositions $ACT_\mathcal{C} = \{\delta(\overline{b}) \mid \overline{b} \in \mathcal{C}^m, \delta \in \Omega, m \in \mathbb{N}\}$.*

We can now define the set of *universally quantified SE-LTL formulas* with respect to Π, Ω, and $\mathcal{C}$ as the set of formulas of the form $\forall \overline{x}\ \varphi$, where φ is a propositional SE-LTL formula over $AP_{\mathcal{C}\cup\mathcal{V}} = \{p(\overline{a}) \mid \overline{a} \in (\mathcal{C} \cup \mathcal{V})^n, p \in \Pi, n \in \mathbb{N}\}$ and $ACT_{\mathcal{C}\cup\mathcal{V}} = \{\delta(\overline{b}) \mid \overline{b} \in (\mathcal{C} \cup \mathcal{V})^m, \delta \in \Omega, m \in \mathbb{N}\}$, with $\mathcal{V}$ an infinite set of variables disjoint from $\mathcal{C}$, and $\overline{x} = \mathit{vars}(\varphi)$ the set of variables occurring in φ. The *satisfaction* of such formulas in a path (π, α) of a parameterized LKS $\mathcal{K} = (S, S_0, \Pi, \mathcal{L}, \Omega, T)$ is now defined in the obvious way:

$$\mathcal{K}, (\pi, \alpha) \models \forall \overline{x}\ \varphi \quad \Leftrightarrow \quad \forall (\theta : \overline{x} \to \mathcal{C}).\ \mathcal{K}_\mathcal{C}, (\pi, \alpha) \models \theta\varphi$$

where $\theta\varphi$ is the propositional SE-LTL formula obtained by applying the simultaneous substitution θ to the variables $\overline{x}$ in φ. Note that $\mathcal{K} \models \forall \overline{x}\ \varphi$ iff $\mathcal{K}, (\pi, \alpha) \models \forall \overline{x}\ \varphi$ for each path (π, α) starting from an initial state.

A parameterized strong (resp., weak) fairness formula from Φ to Ψ is a universally quantified SE-LTL formula $\forall \overline{x}\ \Box\Diamond\Phi \to \Box\Diamond\Psi$ (resp, $\forall \overline{x}\ \Diamond\Box\Phi \to \Box\Diamond\Psi$), where Φ and Ψ are Boolean formulas. Consider sets of strong (resp., weak) parameterized fairness formulas $\mathscr{F} = \{\forall \overline{x}_i\ \Box\Diamond\Phi_i \to \Box\Diamond\Psi_i \mid i \in I\}$ (resp., $\mathscr{J} = \{\forall \overline{x}_j\ \Diamond\Box\Phi_j \to \Box\Diamond\Psi_j \mid j \in J\}$), where I and J are index sets. A path (π, α) of $\mathcal{K}$ is *fair under* parameterized fairness $\mathscr{F} \cup \mathscr{J}$ iff $\mathcal{K}, (\pi, \alpha) \models \forall \overline{x}_i\ \Box\Diamond\Phi_i \to \Box\Diamond\Psi_i$ and $\mathcal{K}, (\pi, \alpha) \models \forall \overline{x}_j\ \Diamond\Box\Phi_j \to \Box\Diamond\Psi_j$ for each $i \in I$, $j \in J$. A SE-LTL formula φ is *fairly* satisfied on $\mathcal{K}$ under $\mathscr{F} \cup \mathscr{J}$, denoted by $\mathcal{K} \models_{\mathscr{F} \cup \mathscr{J}} \varphi$, iff $\mathcal{K}, (\pi, \alpha) \models \varphi$ for each fair path (π, α) under $\mathscr{F} \cup \mathscr{J}$ starting from any initial state of $\mathcal{K}$.

3.2 Finite Instantiation Property and Parameter Abstraction

Although the parameter set $\mathcal{C}$ is not a subset of the set S of states, there is however an implicit relation between $\mathcal{C}$ and S derived from an underlying LKS $\mathcal{K}$, in terms of a definable set. If $[\overline{x} \to \mathcal{C}]$ denotes the set of all substitutions $\theta : \overline{x} \to \mathcal{C}$, given a transition $s \xrightarrow{A} s'$ of $\mathcal{K}$, the *proposition-definable* sets $\mathcal{D}_{s,A}(p(\overline{x}))$ and $\mathcal{D}_{s,A}(\delta(\overline{x}))$ are the sets of substitutions that make the propositions satisfied:

$$\mathcal{D}_{s,A}(p(\overline{x})) = \{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta p(\overline{x}) \in \mathcal{L}(s)\} \quad \mathcal{D}_{s,A}(\delta(\overline{x})) = \{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta\delta(\overline{x}) \in A\}$$

In practice, the number of parameters $c \in \mathcal{C}$ that *occur* in a state is finite. For that reason, assuming that $\mathcal{L}(s)$, $\mathcal{L}(s')$, and A are finite for each transition $s \xrightarrow{A} s'$, the proposition-definable sets for all state and event propositions are finite. This is captured by the following finite instantiation property.

Definition 3. *A parameterized LKS $\mathcal{K} = (S, S_0, \Pi, \mathcal{L}, \Omega, T)$ over parameters $\mathcal{C}$ satisfies* finite instantiation property (FIP) *with respect to $\Pi' \subseteq \Pi$ and $\Omega' \subseteq \Omega$ if for each transition $s \xrightarrow{A} s'$ of $\mathcal{K}$, the sets $\mathcal{D}_{s,A}(p(\overline{x}))$ for each $p \in \Pi'$ and $\mathcal{D}_{s,A}(\delta(\overline{x}))$ for each $\delta \in \Omega'$ are finite.*

For an LKS $\mathcal{K}$ satisfying FIP with respect to Π' and Ω' over a parameter set $\mathcal{C}$ we can define abstraction of substitutions $\theta : \overline{x} \to \mathcal{C}$ with respect to proposition-definable sets of Π' and Ω'. The key idea is to collapse the cofinite[3] complement of each proposition-definable set into the abstracted substitution $\bot_{\overline{x}} : \overline{x} \to \{\bot\}$ with a fresh new constant $\bot$, which denotes a parameter that does *never* appear in the finite definable set. For example, for a state proposition $p(\overline{x})$, each substitution $\theta \notin \mathcal{D}_{s,A}(p(\overline{x}))$ is abstracted to $\bot_{\overline{x}} : \overline{x} \to \{\bot\}$. The extended parameter set $\mathcal{C}_\bot = \mathcal{C} \cup \{\bot\}$ involves the LKS $\mathcal{K}_{\mathcal{C}_\bot} = (S, S_0, AP_{\mathcal{C}_\bot}, \mathcal{L}, ACT_{\mathcal{C}_\bot}, T)$ naturally extending $\mathcal{K}_{\mathcal{C}} = (S, S_0, AP_{\mathcal{C}}, \mathcal{L}, ACT_{\mathcal{C}}, T)$ to $\mathcal{C}_\bot$. In this case, the negated satisfaction relation $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \not\models \bot_{\overline{x}} p(\overline{x})$ holds, as $\bot_{\overline{x}} p(\overline{x}) \in \mathcal{L}(\pi(0))$ is impossible.

This abstraction relation can be extended to any SE-LTL formula using a natural ordering in the abstracted domain $[\overline{x} \to \mathcal{C}_\bot]$. A partial order relation $\preceq$ between substitutions $\theta_1, \theta_2 \in [\overline{x} \to \mathcal{C}_\bot]$ is defined by:

$$\theta_1 \preceq \theta_2 \iff \text{for each } x \in \overline{x},\ \theta_1(x) = \bot \text{ or } \theta_1(x) = \theta_2(x)$$

Given a pair θ_1, θ_2 of substitutions that have a common upper bound, i.e., $\theta_1 \preceq \theta$ and $\theta_2 \preceq \theta$ for some θ, there is the least upper bound defined by:

$$\theta_1 \vee \theta_2 = (\theta_1 \vee \theta_2)(x) = \theta_1(x) \vee \theta_2(x) \text{ for each } x \in \overline{x}$$

where $c \vee \bot = \bot \vee c = c \vee c = c$ for each $c \in \mathcal{C}$. For substitutions θ_1, θ_2 with possibly different domains, we can define the combined substitution $\theta_1 \oplus \theta_2$:

$$\theta_1 \oplus \theta_2(x) = \begin{cases} \theta_1(x) & \text{if } x \in \mathit{dom}(\theta_1) \setminus \mathit{dom}(\theta_2) \\ \theta_2(x) & \text{if } x \in \mathit{dom}(\theta_2) \setminus \mathit{dom}(\theta_1) \\ \theta_1(x) \vee \theta_2(x) & \text{otherwise} \end{cases}$$

The *abstraction function* $\varrho_{(\pi,\alpha),\varphi} : [\mathit{vars}(\varphi) \to \mathcal{C}_\bot] \to [\mathit{vars}(\varphi) \to \mathcal{C}_\bot]$ is then inductively defined for a SE-LTL formula φ and a path (π, α) as follows:

- $\varrho_{(\pi,\alpha),p(\overline{x})}(\theta) = \textbf{if } \theta \in \mathcal{D}_{\pi(0),\alpha(0)}(p(\overline{x})) \textbf{ then } \theta \textbf{ else } \bot_{\overline{x}} \textbf{ fi}$
- $\varrho_{(\pi,\alpha),\delta(\overline{x})}(\theta) = \textbf{if } \theta \in \mathcal{D}_{\pi(0),\alpha(0)}(\delta(\overline{x})) \textbf{ then } \theta \textbf{ else } \bot_{\overline{x}} \textbf{ fi}$
- $\varrho_{(\pi,\alpha),\neg\varphi}(\theta) = \varrho_{(\pi,\alpha),\varphi}(\theta)$
- $\varrho_{(\pi,\alpha),\varphi_1 \wedge \varphi_2}(\theta) = \varrho_{(\pi,\alpha),\varphi_1}(\theta\restriction_{\mathit{vars}(\varphi_1)}) \oplus \varrho_{(\pi,\alpha),\varphi_2}(\theta\restriction_{\mathit{vars}(\varphi_2)})$
- $\varrho_{(\pi,\alpha),\bigcirc\varphi}(\theta) = \varrho_{(\pi,\alpha)^1,\varphi}(\theta)$
- $\varrho_{(\pi,\alpha),\varphi_1 \mathbf{U} \varphi_2}(\theta) = \bigvee_{i \geq 0} \varrho_{(\pi,\alpha)^i,\varphi_1}(\theta\restriction_{\mathit{vars}(\varphi_1)}) \ \oplus\ \bigvee_{j \geq 0} \varrho_{(\pi,\alpha)^j,\varphi_2}(\theta\restriction_{\mathit{vars}(\varphi_2)})$

Note that $\varrho_{(\pi,\alpha),\varphi_1 \wedge \varphi_2}(\theta)$ and $\varrho_{(\pi,\alpha),\varphi_1 \mathbf{U} \varphi_2}(\theta)$ are well-defined since $\varrho_{(\pi,\alpha),\varphi}(\theta) \preceq \theta$ is satisfied by induction. The satisfaction relation of φ on (π, α) for an abstracted substitution $\vartheta = \varrho_{(\pi,\alpha),\varphi}(\theta)$ is naturally defined by $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\varphi$. The following lemma asserts that the satisfaction of a formula on an LKS satisfying FIP is preserved by the abstraction function $\varrho_{(\pi,\alpha),\varphi}$ of substitutions.

[3] A set is cofinite iff the complement of the set is finite.

Lemma 1. *Given an LKS $\mathcal{K}$ satisfying FIP over a parameter set $\mathcal{C}$, a quantified SE-LTL formula $\forall \overline{x}\ \varphi$, and a substitution $\theta \in [\overline{x} \to \mathcal{C}]$, for each path (π, α), $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha) \models \theta\varphi$ iff $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha) \models \varrho_{(\pi,\alpha),\varphi}(\theta)\varphi$.*

Proof. We show the following generalized version of the lemma by structural induction on φ: $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha) \models \theta\varphi$ iff $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha) \models \vartheta\varphi$ for any substitution $\vartheta \in [\overline{x} \to \mathcal{C}_\perp]$ such that $\varrho_{(\pi,\alpha),\varphi}(\theta) \preceq \vartheta \preceq \theta$. For a state proposition $p(\overline{x})$, $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha) \not\models \theta p(\overline{x})$ iff $\theta \notin \mathcal{D}_{\pi(0),\alpha(0)}(p(\overline{x}))$ iff $\varrho_{(\pi,\alpha),p(\overline{x})}(\theta) = \perp_{\overline{x}}$, and for each substitution $\perp_{\overline{x}} \preceq \vartheta \prec \theta$, $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha) \not\models \vartheta\varphi$. To prove the $\varphi_1 \mathbf{U} \varphi_2$ case, we need the following ordering properties of $\varrho_{(\pi,\alpha),\varphi}$, which are easy consequences from the definition: (i) $\varrho_{(\pi,\alpha)^i,\varphi_1}(\theta) \preceq \varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta)\upharpoonright_{dom(\theta_1)}$, and (ii) $\varrho_{(\pi,\alpha)^i,\varphi_2}(\theta) \preceq \varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta)\upharpoonright_{dom(\theta_2)}$, for each $i \geq 0$. Thus, for each $\varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta) \preceq \vartheta \preceq \theta$, if $\overline{y} = vars(\varphi_1)$ and $\overline{z} = vars(\varphi_2)$, we have $\varrho_{(\pi,\alpha)^i,\varphi_1}(\theta \upharpoonright_{\overline{y}}) \preceq \vartheta \upharpoonright_{\overline{y}} \preceq \theta \upharpoonright_{\overline{y}}$ and $\varrho_{(\pi,\alpha)^i,\varphi_2}(\theta \upharpoonright_{\overline{z}}) \preceq \vartheta \upharpoonright_{\overline{z}} \preceq \theta \upharpoonright_{\overline{z}}$, $i \geq 0$. Hence, by induction hypothesis, we have $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha)^i \models \theta\varphi_1$ iff $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha)^i \models \vartheta\varphi_1$, and $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha)^j \models \theta\varphi_2$ iff $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha)^j \models \vartheta\varphi_2$ for each $i, j \geq 0$. Therefore, $\mathcal{K}_{\mathcal{C}}, (\pi, \alpha) \models \theta(\varphi_1 \mathbf{U} \varphi_2)$ iff $\mathcal{K}_{\mathcal{C}_\perp}, (\pi, \alpha) \models \vartheta(\varphi_1 \mathbf{U} \varphi_2)$. The other cases are similar. □

On the other hand, as a dual of $\varrho_{(\pi,\alpha),\varphi}$, the *concretization* function $I_{(\pi,\alpha),\varphi} : [vars(\varphi) \to \mathcal{C}_\perp] \to \mathcal{P}([vars(\varphi) \to \mathcal{C}])$ can be defined for a SE-LTL formula φ as follows, where $[\overline{x} \to \mathcal{C}]_{\succeq\vartheta}$ denotes the set $\{\theta \in [\overline{x} \to \mathcal{C}] \mid \vartheta \preceq \theta\}$ and the "glueing" $I_1 \odot I_2$ of two sets I_1 and I_2 of concrete substitutions is defined by $I_1 \odot I_2 = \{\theta \mid \theta\upharpoonright_{dom(I_1)} \in I_1,\ \theta\upharpoonright_{dom(I_2)} \in I_2\}$:

- $I_{(\pi,\alpha),p(\overline{x})}(\vartheta) = \textbf{if } \vartheta \in D \textbf{ then } \vartheta \textbf{ else } [\overline{x} {\to} \mathcal{C}]_{\succeq\vartheta} \setminus D \textbf{ fi},\ D = \mathcal{D}_{\pi(0),\alpha(0)}(p(\overline{x}))$
- $I_{(\pi,\alpha),\delta(\overline{x})}(\vartheta) = \textbf{if } \vartheta \in D \textbf{ then } \vartheta \textbf{ else } [\overline{x} {\to} \mathcal{C}]_{\succeq\vartheta} \setminus D \textbf{ fi},\ D = \mathcal{D}_{\pi(0),\alpha(0)}(\delta(\overline{x}))$
- $I_{(\pi,\alpha),\neg\varphi}(\vartheta) = I_{(\pi,\alpha),\varphi}(\vartheta)$
- $I_{(\pi,\alpha),\varphi_1\wedge\varphi_2}(\vartheta) = I_{(\pi,\alpha),\varphi_1}(\vartheta\upharpoonright_{vars(\varphi_1)}) \odot I_{(\pi,\alpha),\varphi_2}(\vartheta\upharpoonright_{vars(\varphi_2)})$
- $I_{(\pi,\alpha),\bigcirc\varphi}(\vartheta) = I_{(\pi,\alpha)^1,\varphi}(\vartheta)$
- $I_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\vartheta) = \bigcap_{i\geq 0} I_{(\pi,\alpha)^i,\varphi_1}(\vartheta) \ \odot\ \bigcap_{j\geq 0} I_{(\pi,\alpha)^j,\varphi_2}(\vartheta)$

It is easy to check that for each $\theta \in I_{(\pi,\alpha),\varphi}(\vartheta)$, $\vartheta \preceq \theta$ and $\varrho_{(\pi,\alpha),\varphi}(\vartheta) = \varrho_{(\pi,\alpha),\varphi}(\theta)$. The abstraction of a concrete substitution does always exist, but there may be *no* concretization for some abstracted substitution. For instance, given a path (π, α) such that $\mathcal{D}_{\pi(i),\alpha(i)}(p(x)) = \{i\}$ for each $i \geq 0$, if $\mathcal{C} = \mathbb{N}$, then $\varrho_{(\pi,\alpha),\Diamond p(x)}(\theta) = \theta$ for any $\theta \in [\{x\} {\to} \mathbb{N}]$, and $I_{(\pi,\alpha),\Diamond p(x)}(\perp_x) = \varnothing$. However, for a *finite* LKS that has only a finite set of states and a finite set of transitions, each abstracted substitution has a corresponding concrete substitution.

Lemma 2. *Given a* finite *LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, a quantified SE-LTL formula $\forall \overline{x}\ \varphi$, and $\vartheta \in [\overline{x} {\to} \mathcal{C}_\perp]$, for each path (π, α), $I_{(\pi,\alpha),\varphi}(\vartheta) \neq \varnothing$.*

Proof. It suffices to show, by structural induction on φ, that for each $x \in vars(\theta)$, $I_{(\pi,\alpha),\phi}(\vartheta)\upharpoonright_{\{x\}}$ is *cofinite* if $\vartheta(x) = \perp$, and the singleton $\{\vartheta(x)\}$ otherwise. The $\varphi_1 \wedge \varphi_2$ case comes from the fact that the intersection of two cofinite sets is cofinite. For $\varphi_1 \mathbf{U} \varphi_2$, it is enough to mention that: (i) the set of suffixes $\{(\pi, \alpha)^i \mid i \geq 0\}$ is finite when $\mathcal{K}$ is finite, and (ii) a finite intersection of cofinite sets is cofinite. The other cases are clear by definition and the induction hypothesis. □

3.3 Path-Realized Parameters for an LKS Satisfying FIP

For a *finite* LKS $\mathcal{K}$ satisfying FIP, we can determine the satisfaction of $\forall\overline{x}\ \varphi$ by considering a (possibly small) finite set of substitutions. Consider a set $\mathscr{R} \subseteq [\overline{x} \to \mathcal{C}_\bot]$ of substitutions with $\varrho_{(\pi,\alpha),\varphi}([\overline{x} \to \mathcal{C}]) \subseteq \mathscr{R}$. By definition, $\mathcal{K}, (\pi, \alpha) \models \forall\overline{x}\ \varphi$ iff $\mathcal{K}_\mathcal{C}, (\pi, \alpha) \models \theta\varphi$ for each $\theta \in [\overline{x} \to \mathcal{C}]$, and by Lemma 1, iff $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\varphi$ for each $\vartheta \in \varrho_{(\pi,\alpha),\varphi}([\overline{x} \to \mathcal{C}])$. If $\vartheta \in [\overline{x} \to \mathcal{C}_\bot] \smallsetminus \varrho_{(\pi,\alpha),\varphi}([\overline{x} \to \mathcal{C}])$, by Lemma 2, there is a concrete substitution $\theta \in [\overline{x} \to \mathcal{C}]$ such that $\varrho_{(\pi,\alpha),\varphi}(\vartheta) = \varrho_{(\pi,\alpha),\varphi}(\theta)$, which implies $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\varphi$ iff $\mathcal{K}_\mathcal{C}, (\pi, \alpha) \models \theta\varphi$. Consequently, we have:

Theorem 1. *Given a* finite *LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, a quantified SE-LTL formula $\forall\overline{x}\ \varphi$, and a path (π, α), if $\varrho_{(\pi,\alpha),\varphi}([\overline{x} \to \mathcal{C}]) \subseteq \mathscr{R} \subseteq [\overline{x} \to \mathcal{C}_\bot]$, then $\mathcal{K}, (\pi, \alpha) \models \forall\overline{x}\ \varphi$ iff for each $\vartheta \in \mathscr{R}$, $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\varphi$.*

If the operator $\oplus$ is extended to sets of substitutions by $I_1 \oplus I_2 = \{\theta_1 \oplus \theta_2 \mid \theta_1 \in I_1, \theta_2 \in I_2\}$, given a SE-LTL formula φ and a path (π, α), the *path-realized* set $\mathscr{R}_{(\pi,\alpha),\varphi} \subseteq [\mathit{vars}(\varphi) \to \mathcal{C}_\bot]$ of substitutions is defined as follows:

- $\mathscr{R}_{(\pi,\alpha),p(\overline{x})} = \mathcal{D}_{\pi(0),\alpha(0)}(p(\overline{x})) \cup \{\bot_{\overline{x}}\}$
- $\mathscr{R}_{(\pi,\alpha),\delta(\overline{x})} = \mathcal{D}_{\pi(0),\alpha(0)}(\delta(\overline{x})) \cup \{\bot_{\overline{x}}\}$
- $\mathscr{R}_{(\pi,\alpha),\neg\varphi} = \mathscr{R}_{(\pi,\alpha),\varphi}$
- $\mathscr{R}_{(\pi,\alpha),\varphi_1\wedge\varphi_2} = \mathscr{R}_{(\pi,\alpha),\varphi_1} \oplus \mathscr{R}_{(\pi,\alpha),\varphi_2}$
- $\mathscr{R}_{(\pi,\alpha),\bigcirc\varphi} = \mathscr{R}_{(\pi,\alpha)^1,\varphi}$
- $\mathscr{R}_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2} = \bigcup_{i\geq 0} \mathscr{R}_{(\pi,\alpha)^i,\varphi_1} \ \oplus\ \bigcup_{j\geq 0} \mathscr{R}_{(\pi,\alpha)^j,\varphi_2}$

Since $\mathscr{R}_{(\pi,\alpha),\varphi}$ is the aggregation of all possible values of $\varrho_{(\pi,\alpha),\varphi}$, from Theorem 1, we have the following *localization lemma*:

Lemma 3. *Given a finite LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, a quantified SE-LTL formula $\forall\overline{x}\ \varphi$, and a path (π, α), for each substitution $\theta \in [\overline{x} \to \mathcal{C}]$, there exists $\vartheta \in \mathscr{R}_{(\pi,\alpha),\varphi}$ such that $\mathcal{K}_\mathcal{C}, (\pi, \alpha) \models \theta\varphi$ iff $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\varphi$.*

If we consider a parameterized *fairness* formula $\forall\overline{x}\ \psi$,we can further reduce the set of substitutions necessary to determine the satisfaction of the formula. Since the satisfaction of a parameterized fairness formula ψ does not vary if we skip finitely many steps of a path, from the above lemma, we can consider only the set $\mathscr{R}^{\mathit{inf}}_{(\pi,\alpha),\psi}$ of *infinitely often* path-realized substitutions, whose elements belong to $\mathscr{R}_{(\pi,\alpha)^i,\psi}$ for infinitely many $i \in \mathbb{N}$. Note that $\mathscr{R}^{\mathit{inf}}_{(\pi,\alpha),\psi}$ is actually equal to $\mathscr{R}_{(\pi,\alpha)^N,\psi}$ for a sufficiently large $N \geq 0$ in which all substitutions with finite occurrences are skipped. Accordingly, by Theorem 1, we then have:

Theorem 2. *Given a* finite *LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, a parameterized fairness formula $\forall\overline{x}\ \psi$, and a path (π, α), $\mathcal{K}, (\pi, \alpha) \models \forall\overline{x}\ \psi$ iff for each $\vartheta \in \mathscr{R}^{\mathit{inf}}_{(\pi,\alpha),\psi}$, $\mathcal{K}_{\mathcal{C}_\bot}, (\pi, \alpha) \models \vartheta\psi$.*

Note that if $\mathscr{R}^{\mathit{inf}}_{\psi} \subseteq [\overline{x} \to \mathcal{C}_\bot]$ is the union of $\mathscr{R}^{\mathit{inf}}_{(\pi,\alpha),\psi}$ for each (π, α) from a initial state of $\mathcal{K}$, by the above theorem, $\mathcal{K} \models \forall\overline{x}\ \psi$ iff for each $\vartheta \in \mathscr{R}^{\mathit{inf}}_{\psi}$, $\mathcal{K}_{\mathcal{C}_\bot} \models \vartheta\psi$.

4 Automata-Based Model Checking Algorithm

Given a finite LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, the satisfiability of a quantified SE-LTL formula $\forall\overline{x}\ \varphi$ is now reduced to the satisfiability of $\vartheta\varphi$ on $\mathcal{K}_{\mathcal{C}_\perp}$ for each path-realized substitution ϑ. This reduction gives a simple algorithm to verify $\forall\overline{x}\ \varphi$ using the existing SE-LTL model checking algorithm as follows:

1. Traverse the state space of $\mathcal{K}$ to compute a path-realized set $\mathscr{R}_{(\pi,\alpha),\varphi}$ for each infinite path (π,α), witnessed by a cycle in the search graph.
2. For each substitution ϑ evaluated at Step 1, model check $\mathcal{K}_{\mathcal{C}_\perp} \models \vartheta\varphi$ using the existing algorithm. If all satisfied, then $\mathcal{K} \models \forall\overline{x}\ \varphi$. Otherwise, $\mathcal{K} \not\models \forall\overline{x}\ \varphi$.

This algorithm is not on-the-fly, since we have to traverse the entire state space first. However, for a parameterized fairness formulas, thanks to the fact that only infinitely often path-realized substitutions are necessary, we can give an on-the-fly model checking algorithm below based on a *strongly connected component* (SCC) analysis, seeing that each cycle is identified by a SCC.

4.1 Automata-Based Characterization

Given a set of parameterized strong/weak fairness formulas $\mathscr{F} \cup \mathscr{J}$ for a finite LKS $\mathcal{K}$ satisfying FIP over parameters $\mathcal{C}$, by Theorem 2, we can construct an equivalent set $\mathcal{G} = \hat{\mathscr{F}} \cup \hat{\mathscr{J}}$ of *propositional* generalized strong/weak fairness formulas by instantiating each parameterized formula $\forall\overline{x}\ \psi$ with the substitutions in $\mathscr{R}^{inf}_{\psi}$. Since generalized weak fairness formula $\Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$ can be expressed by $\Box\Diamond\top \rightarrow \Box\Diamond(\neg\Phi\vee\Psi)$, we can regard $\mathcal{G}$ as a set of strong fairness formulas. Such strong fairness conditions can be incorporated into the acceptance conditions of a transition-based *Streett* automaton.

Definition 4. *A* Streett automaton $(Q, Q_0, P, \Delta, \mathcal{F})$ *is a 5-tuple with* Q *a set of states,* $Q_0 \subseteq Q$ *a set of initial states,* P *an alphabet of transition labels,* $\Delta \subseteq Q\times P\times Q$ *a transition relation, and* $\mathcal{F} \subseteq \mathcal{P}(\Delta\times\Delta)$ *an acceptance condition.*

A *run* of a Streett automaton $\mathcal{S}$ is an infinite sequence $q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \cdots$ of transitions starting from $q_0 \in Q_0$. A run σ is *accepted* by $\mathcal{S}$ iff for each pair $(G,H) \in \mathcal{F}$, whenever σ has transitions in G infinitely often, σ has transitions in H infinitely often. Given two Streett automata $\mathcal{S}_1$ and $\mathcal{S}_2$, their synchronous product $\mathcal{S}_1 \times \mathcal{S}_2$ is defined such that $|\mathcal{S}_1 \times \mathcal{S}_2| = O(|\mathcal{S}_1|\cdot|\mathcal{S}_2|)$ and $L(\mathcal{S}_1 \times \mathcal{S}_2) = L(\mathcal{S}_1) \cap L(\mathcal{S}_2)$ [11]. Note that a Büchi automaton $\mathcal{B} = (Q, Q_0, P, \Delta, F)$ can be translated into an equivalent Streett automaton $\mathcal{S}(\mathcal{B}) = (Q, Q_0, P, \Delta, \{(\Delta, F)\})$.

Given an LKS $\mathcal{K} = (S, S_0, AP, \mathcal{L}, ACT, T)$ and a set of generalized strong fairness formulas $\mathcal{G} = \{\Box\Diamond\Phi_i \rightarrow \Box\Diamond\Psi_i \mid i \in I\}$, we can define a fair Streett automaton $\mathcal{S}^{\mathcal{G}}(\mathcal{K}) = (S, S_0, \mathcal{P}(AP \uplus ACT), \Delta, \mathcal{F}^{\mathcal{G}})$ such that:[4]

$$\begin{aligned}\Delta &= \{s \xrightarrow{\mathcal{L}(s)\uplus A} s' \mid s \xrightarrow{A} s' \in T\}\\ \mathcal{F}^{\mathcal{G}} &= \{(\Delta^{\Phi_i}, \Delta^{\Psi_i}) \mid i \in I\}, \qquad\qquad \text{where } \Delta^{\Phi} = \{s \xrightarrow{B} s' \in \Delta \mid B \models \Phi\}\end{aligned}$$

[4] $B \models \Phi$ is defined inductively as follows: $B \models p$ iff $p \in B$, $B \models \delta$ iff $\delta \in B$, $B \models \neg\Phi$ iff $B \not\models \Phi$, and $B \models \Phi_1 \wedge \Phi_2$ iff $B \models \Phi_1$ and $B \models \Phi_2$, where $p \in AP$ and $\delta \in ACT$.

Each path (π, α) of an LKS $\mathcal{K}$ is in one-to-one correspondence with a run $\pi(0) \xrightarrow{\mathcal{L}(\pi(0)) \uplus \alpha(0)} \pi(1) \xrightarrow{\mathcal{L}(\pi(1)) \uplus \alpha(1)} \cdots$ of the Streett automaton $\mathcal{S}^{\mathcal{G}}(\mathcal{K})$. Furthermore, (π, α) satisfies all fairness conditions of $\mathcal{G}$ iff the corresponding run of (π, α) is accepted by $\mathcal{S}^{\mathcal{G}}(\mathcal{K})$. Therefore, we can use a Streett automaton $\mathcal{S}^{\mathcal{G}}_{\neg\varphi}(\mathcal{K}) = \mathcal{S}^{\mathcal{G}}(\mathcal{K}) \times \mathcal{S}(\mathcal{B}_{\neg\varphi})$ to model check a SE-LTL formula in $\mathcal{K}$ under generalized strong/weak fairness conditions as follows:

Theorem 3. *Given an LKS $\mathcal{K}$, a SE-LTL formula φ, and a set of propositional generalized strong fairness formulas $\mathcal{G}$, there is a Streett automaton $\mathcal{S}^{\mathcal{G}}_{\neg\varphi}(\mathcal{K})$ with size $O(|\mathcal{K}| \cdot 2^{|\varphi|})$ such that $L(\mathcal{S}^{\mathcal{G}}_{\neg\varphi}(\mathcal{K})) = \varnothing$ iff $\mathcal{K} \models_{\mathcal{G}} \varphi$.*

Consequently, the model checking problem of SE-LTL formulas on a finite LKS $\mathcal{K}$ satisfying FIP under parameterized strong/weak fairness conditions $\mathscr{F} \cup \mathscr{J}$ is reduced to the emptiness checking problem of the Streett automaton whose acceptance condition is defined by the generalized strong/weak fairness conditions $\hat{\mathscr{F}} \cup \hat{\mathscr{J}}$ obtained by instantiating $\mathscr{F}$ and $\mathscr{J}$ for each $\vartheta \in \bigcup_{(\forall \overline{x}^k\ \psi) \in \mathscr{F} \cup \mathscr{J}} \mathscr{R}^{inf}_{\psi}$.

It is worth noting that a naive selection of such substitutions without parameter abstraction does not guarantee the equivalence between the parameterized fairness formula and the set of instantiated formulas. For example, in process algebra, the fairness formula $\forall x\ \Box\Diamond\neg \mathit{enabled}(x) \to \Box\Diamond \mathit{execute}(x)$ is always false, since for a process k not existing in a system, $\neg \mathit{enabled}(k)$ is true but $\mathit{execute}(k)$ is false. But all instantiated formulas only using the *existing* processes can be true if such processes are always enabled.

4.2 On-The-Fly Model Checking Algorithm

We present an *on-the-fly* automata-based algorithm for parameterized fairness, based on the emptiness checking algorithm for Streett automaton associated to the strong fairness conditions [11,19]. To check emptiness of a *finite* Streett automaton $(Q, Q_0, P, \Delta, \mathcal{F})$, the basic idea is to find a reachable SCC that satisfies all Streett acceptance conditions in $\mathcal{F}$ [12]. An acceptance condition $(g_i, h_i) \in \mathcal{F}$ is satisfied in a SCC $\mathfrak{S}$ iff whenever $\mathfrak{S}$ contains a transition $s_1 \xrightarrow{B} s_2$ such that $B \models g_i$, there exists some transition $s'_1 \xrightarrow{B'} s'_2 \in \mathfrak{S}$ such that $B' \models h_i$. If some (g_i, h_i) is not satisfied in $\mathfrak{S}$, then the *bad* transitions of $\mathfrak{S}$ are identified, satisfy $g_i \wedge \neg h_i$ and therefore prevent the satisfaction of (g_i, h_i).

The emptiness checking algorithm specified in Fig. 1 is to find a SCC with no bad transitions. The computeNextSCC(Q, q, Δ) function in Line 3 identifies each SCC $\mathfrak{S}$ in the graph $(Q,\ \Delta)$ containing q, which can be implemented by any *on-the-fly* algorithm to find a SCC; typically Tarjan's algorithm, or Couvreur's algorithm [9] for early finding of SCC. If $\mathfrak{S}$ satisfies all acceptance conditions (Line 4), then we can generate a counterexample given by a fair cycle from $\mathfrak{S}$ using breadth-first search [19]. Otherwise, if $\mathfrak{S}$ is a maximal strongly connected component (MSCC)[5] and contains bad transitions, then the whole $\mathfrak{S}$ is traversed again *except for* the bad transitions (Line 11), which leads to dividing $\mathfrak{S}$ into multiple smaller subcomponent with no such bad transitions.

[5] A MSCC is a SCC such that there is no other SCCs containing it.

```
findFairSCC($Q$, $Q_0$, $\varDelta$)
2  while there is a reachable state $q \in Q$ from $Q_0$ that has not been visited do
3    $\mathfrak{S}$ := computeNextSCC($Q$, $q$, $\varDelta$);
4    if fairnessSatisfied($\mathfrak{S}$) then
5      return $\mathfrak{S}$
6    else if $\mathfrak{S}$ is maximal and contains bad transitions then
7      $Q^{\mathfrak{S}}$ := the set of states in $\mathfrak{S}$;
8      $\varLambda^{\mathfrak{S}}$ := the set of bad transitions for unsatisfied acceptance conditions in $\mathfrak{S}$;
9      $Q_0^{\mathfrak{S}}$ := the set of states that occur in $\varLambda^{\mathfrak{S}}$;
10     mark each state in $Q^{\mathfrak{S}}$ as unvisited;
11     return findFairSCC($Q^{\mathfrak{S}}$, $\{q\} \cup Q_0^{\mathfrak{S}}$, $\varDelta \setminus \varLambda^{\mathfrak{S}}$) unless $\bot$
12   end if
13 end while;
14 return $\bot$;
```

Fig. 1. Streett Emptiness Checking Algorithm for $\mathcal{S} = (Q, Q_0, P, \varDelta, \mathcal{F})$

Given a Streett Automaton $\mathcal{S} = (Q, Q_0, P, \varDelta, \mathcal{F})$, if there exists a nonempty SCC $\mathfrak{T} \in (Q, \varDelta)$ satisfying $\mathcal{F}$ reachable from Q_0, then $\mathfrak{T}$ is a subcomponent of a MSCC $\mathfrak{S} = (Q^{\mathfrak{S}}, \varDelta^{\mathfrak{S}})$ given by computeNextSCC(Q, q, $\varDelta$) with q reachable from Q_0. Further, since $\mathfrak{T}$ does not contain any bad transitions $\varLambda^{\mathfrak{S}}$ of $\mathfrak{S}$, we have $\mathfrak{T} \subseteq (Q^{\mathfrak{S}}, \varDelta^{\mathfrak{S}} \setminus \varLambda^{\mathfrak{S}})$. Hence, whenever we meet findFairSCC($Q^{\mathfrak{S}}$, $\{q\} \cup Q_0^{\mathfrak{S}}$, $\varDelta \setminus \varLambda^{\mathfrak{S}}$) in Line 11, $\mathfrak{T}$ is contained in $(Q^{\mathfrak{S}}, \varDelta^{\mathfrak{S}} \setminus \varLambda^{\mathfrak{S}})$ and is reachable from $\{q\} \cup Q_0^{\mathfrak{S}}$. This yields the correctness of this algorithm as follows:

Theorem 4. *Assuming the correctness of an underlying SCC finding algorithm, given a Streett Automaton $\mathcal{S} = (Q, Q_0, P, \varDelta, \mathcal{F})$, the findFairSCC($Q$, Q_0, $\varDelta$) finds a nonempty SCC satisfying $\mathcal{F}$ reachable from Q_0 if it exists.*

To make this algorithm on-the-fly under parameterized fairness conditions, we have to check fairnessSatisfied($\mathfrak{S}$) in Line 4 using only states and transitions in $\mathfrak{S}$. Given parameterized fairness formulas $\forall \overline{x}_i\ \psi_i$, $1 \leq i \leq n$, with ψ_i either $\Box\Diamond\varPhi_i \rightarrow \Box\Diamond\varPsi_i$, or $\Diamond\Box\varPhi_i \rightarrow \Box\Diamond\varPsi_i$, since $\varPhi_i$ and $\varPsi_i$ have no temporal operators, when $\zeta(\overline{y})$ ranges over both state and event propositions, we have $\mathscr{R}_{(\pi,\alpha)^k,\varPhi_i} = \bigoplus_{\zeta(\overline{y})\in\varPhi_i}(\mathcal{D}_{\pi(k),\alpha(k)}(\zeta(\overline{y})) \cup \{\bot_{\overline{y}}\})$, $k \geq 0$, and $\mathscr{R}_{(\pi,\alpha)^k,\varPsi_i}$ similar. Thus, for any infinite path (π, α) whose infinite suffixes are included in $\mathfrak{S}$, the infinitely often path-realized set $\mathscr{R}^{inf}_{(\pi,\alpha),\psi_i}$ is a subset of the following set $\mathscr{R}_{\mathfrak{S},\psi_i}$:

$$\mathscr{R}_{\mathfrak{S},\psi_i} = \bigcup_{s \xrightarrow{A} s' \in \mathfrak{S}} \Big(\bigoplus_{\zeta(\overline{y})\in\varPhi_i} (\mathcal{D}_{s,A}(\zeta(\overline{y})) \cup \{\bot_{\overline{y}}\}) \oplus \bigoplus_{\zeta(\overline{z})\in\varPsi_i} (\mathcal{D}_{s,A}(\zeta(\overline{z})) \cup \{\bot_{\overline{z}}\}) \Big)$$

Thanks to the localization lemma, we only need to check fairness instances of ψ_i from $\mathscr{R}_{\mathfrak{S},\psi_i}$ to determine the satisfaction of $\forall \overline{x}_i\ \psi_i$ on such paths. That is, to decide whether acceptance conditions $\{(\varDelta^{\theta\varPhi_i}, \varDelta^{\theta\varPsi_i}) \mid \theta \in [\overline{x} \rightarrow \mathcal{C}]\}$ from $\forall \overline{x}_i\ \psi_i$ are all satisfied on $\mathfrak{S}$, it is enough to consider acceptance conditions $\{(\varDelta^{\vartheta\varPhi_i}, \varDelta^{\vartheta\varPsi_i}) \mid \vartheta \in \mathscr{R}_{\mathfrak{S},\psi_i}\}$, so that all parameterized acceptance conditions are *localized* to $\mathfrak{S}$ and fairnessSatisfied($\mathfrak{S}$) can be computed on-the-fly.

A Streett automaton emptiness check can be determined in time $O(|\mathcal{F}|\cdot(|Q|+|\Delta|))$ [11]. Thus, the time complexity of model checking a SE-LTL formula φ on $\mathcal{K}$ with parameterized fairness conditions is $O(f\cdot r\cdot|\mathcal{K}|\cdot 2^{|\varphi|})$, where f and r are, respectively, the numbers of parameterized fairness conditions and of infinitely often path-realized parameters in $\mathcal{K}$. That is, $f = |\mathscr{F} \cup \mathscr{J}|$, and $r = |R|$, where $R = \bigcup_{(\forall \overline{x}^k\ \psi)\in\mathscr{F}\cup\mathscr{J}} \mathscr{R}_{\psi}^{inf}$. Note that the space complexity is also exponential on $|\varphi|$, since in the worst case the whole state space can be a single SCC maintained by the underlying Streett emptiness checking algorithm.

5 Parameterized Fairness Case Studies

This section illustrates how our framework for parameterized fairness can be applied to a wide range of modeling applications, especially including nontrivial parameterized fairness assumptions such as (i) object fairness with dynamic object creation, and (ii) the fairness for the sliding window protocol, parametric not only on processes, but also on data in channels.

5.1 Evolving Dining Philosophers Problem

We illustrate *dynamic* parameterized fairness by means of the Evolving Dining Philosophers problem [17]. This problem is similar to the famous Dining Philosophers problem: there are N philosophers sitting at a circular table who are either *thinking*, *waiting*, or *eating*. A chopstick is placed in between each pair of adjacent philosophers. The thinking philosophers *wake* up to eat something. The waiting philosophers can *grab* a chopstick on their left or right, and eat when they have both. After eating, a philosopher places the chopsticks back on the table and *think*s. However, in the evolving version, a philosopher can join or leave the table, so that the number of philosophers can be dynamically changed. In this problem we cannot decide the total fairness conditions at the outset, since they apply to each philosopher.

Although there is no limit to the number of philosophers in the original problem, we give an unpredictable bound using the Collatz problem [10]. There is a global counter that symbolizes a philosophical problem, and philosophers keep thinking the problem by changing the number n to: (i) $3n + 1$ for n odd, or (ii) $n/2$ for n even. New philosophers can join the group if the global number is a multiple of the current number of philosophers. Only the last philosopher can leave the group. To keep consistency, whenever a philosopher joins or leaves the table, the related chopsticks should not be held by another philosopher.

Each philosopher is identified by a natural number $k \in \mathbb{N}$. The states of philosopher k are expressed by the parameterized state propositions $\mathit{thinking}(k)$, $\mathit{waiting}(k)$, and $\mathit{eating}(k)$. Similarly, the actions of the philosopher k are represented by the parameterized event propositions $\mathit{wake}(k)$, $\mathit{grab}(k)$, and $\mathit{think}(k)$. Initially, there are two philosophers 1 and 2 thinking. The corresponding parameterized LKS over parameters $\mathbb{N}$ can be generated from the initial state, for example, $\{\mathit{thinking}(1), \mathit{thinking}(2)\} \xrightarrow{\mathit{wake}(1)} \{\mathit{waiting}(1), \mathit{thinking}(2)\}$.

```
send_P: { $a_P \leq i < s_P + l_P$ }
  begin enqueue $I_P[i]$ to $F_Q$ end

loss_P: { $[i, w] \in F_P$ }
  begin remove $[i, w]$ from $F_P$ end

recv_P: { $[i, w] := \mathrm{dequeue}(F_P)$ }
  begin
    if $O_P[i] = \bot$ then $O_P[i] := \mathrm{w}$;
       $s_P := \min\{j \mid O_P[j] = \bot\}$;
       $a_P := \max(a_P,\ i - l_Q + 1)$ fi end
```

Fig. 2. The Balanced Sliding-Window Protocol (For P)

In order to prove, e.g., the liveness property $\Diamond eating(1)$, we need the weak fairness condition of $wake(k)$ and the strong fairness condition of $grab(k)$ for each philosopher k, given by the following parameterized fairness formulas:

$$\forall x\ \Diamond\Box enabled.wake(x) \rightarrow \Box\Diamond wake(x) \qquad \forall x\ \Box\Diamond enabled.grab(x) \rightarrow \Box\Diamond grab(x)$$

The parameterized LKS generated over a set of parameters $\mathbb{N}$ is finite due to the Collatz bound, and satisfies FIP since the propositions in the fairness formulas can be true only for the existing philosophers. Hence, we can directly model check $\Diamond eating(1)$ under the above parameterized fairness conditions in our framework.

5.2 Balanced Sliding Window Protocol

In this example we show how a liveness property of a nontrivial system with an *unbounded* number of fairness assumptions can be verified under parameterized fairness. The balanced sliding window protocol is a symmetric protocol that allows information to be sent reliably in both directions. The verification task for this protocol is not simple, since the specification involves unbounded queue and dynamic fairness conditions.

The balanced sliding window protocol description is as follows [23]: there are two entirely symmetric processes P and Q connected to each other through a lossy channel. Packets exchanged by the processes are pairs $[i, w]$ with i an index number and w a data word. The acknowledgement is implicitly provided by sending and receiving messages. Process P contains an array I_P of packets to be sent, another array O_P of items to be received, and a FIFO queue F_P of packets in transit to be received. Process P also has three variables to describe a state of the process as follows: s_P the lowest index of packet not yet received from the other process, a_P the lowest index of packet sent but not yet acknowledged, and l_P a fixed bound allowing sending packets before being acknowledged.

Process P can *send* any packet $[i, w]$ in I_P to Q if no acknowledgement for it has been yet received but within bound, i.e., $a_P \leq i < s_P + l_P$. When *receiving* a packet, an already received packet is ignored. Otherwise, the packet is added to O_P, s_P is set to the smallest index that has not been received, and a_P is set to $max(a_P, i - l_Q + 1)$ to ignore messages in the future whose index is less then or equal to $i - l_Q$. Finally, the *loss* of a packet can happen at any time. The behavior of this protocol is summarized as the pseudo code in Fig. 2.

The liveness property we are interest in is that all messages are eventually delivered, given by the LTL formula $\Diamond success$ such that the state proposition *success* holds if $I_P = O_Q$ and $I_Q = O_P$. Let the actions of a process p with packet $[i, w]$ be expressed by the parameterized event propositions $\mathit{send}(p, i, w)$ and $\mathit{recv}(p, i, w)$. The verification of $\Diamond \mathit{success}$ requires the weak fairness condition of $\mathit{send}(p, i, w)$ and the strong fairness condition of $\mathit{recv}(p, i, w)$ for each process p and packet $[i, w]$, specified by the following parameterized fairness formulas:

$$\begin{aligned} &\forall(p, i, w)\ \ \Diamond\Box \mathit{enabled.send}(p, i, w) \rightarrow \Box\Diamond \mathit{send}(p, i, w) \\ &\forall(p, i, w)\ \ \Box\Diamond \mathit{enabled.recv}(p, i, w) \ \rightarrow \Box\Diamond \mathit{recv}(p, i, w) \end{aligned}$$

Since the state space of the original system is infinite due to the unbounded queue, we apply equational-abstraction [21] to collapse the set of states to a finite number by identifying repeated packets. At the level of the abstracted system all these fairness requirements are captured by the following *generalized* parameterized fairness conditions that only use state propositions, where the parameterized state proposition $\mathit{inQueue}(p, i, w)$ (resp., $\mathit{inOutput}(p, i, w)$) holds when the packet $[i, w]$ is in the queue F_p (resp., the output array O_p):

$$\begin{aligned} \forall(i, w)\ \ &\Diamond\Box \mathit{enabled.send}(P, i, w) \rightarrow \Box\Diamond \mathit{inQueue}(Q, i, w) \\ \forall(i, w)\ \ &\Diamond\Box \mathit{enabled.send}(Q, i, w) \rightarrow \Box\Diamond \mathit{inQueue}(P, i, w) \\ \forall(p, i, w)\ \ &\Box\Diamond \mathit{inQueue}(p, i, w) \qquad\quad \rightarrow \Box\Diamond \mathit{inOutput}(p, i, w) \end{aligned}$$

Again, in the finite abstracted system, we can apply our framework to model check $\Diamond \mathit{success}$ under the above parameterized fairness conditions, since the system satisfies FIP, owing to the fact that each proposition in the fairness formula can be true only for the existing entities in the system.

6 Experimental Results

We have implemented our algorithm in the Maude system by extending the existing SE-LTL model checker [3]. Our tool accepts models with both unparameterized and parameterized fairness conditions. We have compared it with other explicit-state model checkers, such as Pat [22] and Spin [15], and then tested our algorithm on complex examples involving dynamic fairness conditions. Since Spin and Pat only support unparameterized fairness, the comparison with those tools uses a model with unparameterized fairness assumptions. The experiments in this section were conducted on an Intel Core 2 Duo 2.66 GhZ with 8GB RAM running Mac OS X 10.6. We set a timeout of 30 minutes for the experiments.

To evaluate our algorithm comparing it with other tools, we use the classical Dining Philosophers problem which requires both strong and weak fairness conditions to verify the liveness property $\Box\neg \mathit{deadlock} \rightarrow \Diamond \mathit{eating}(1)$, where *deadlock* is considered as an event proposition.[6] Table 1 shows the verification results for each tool, where "N" is the number of philosophers, and "Time" is the runtime in seconds. We can observe that in the weak-fairness case, our algorithm is comparable to Spin, and for the strong/weak fairness case, it shows similar performance with Pat. For Spin, we had to encode strong fairness conditions into the LTL formula since Spin only supports weak fairness.

[6] For the cases of Pat and Spin, we use a modified *deadlock-free* version.

Table 1. Dining Philosophers for the property $\Box\neg \mathit{deadlock} \rightarrow \Diamond \mathit{eating}(1)$

Fairness	N	MAUDE		PAT		SPIN	
		States	Time	States	Time	States	Time
Weak Only (Counter Example)	6	913	< 0.1	1596	1.0	672	< 0.1
	7	2418	0.1	5718	5.1	2765	0.2
	8	11092	0.9	21148	33.5	9404	0.8
Strong/Weak (Valid)	6	5777	1.8	18101	3.9	> 30 minutes	
	7	24475	11.5	69426	16.1		
	8	103681	77.6	260998	79.0		

Table 2. Results for models with dynamic parameterized fairness

(a) Evolving Dining Philosophers

C. Nr.	States	Time	#Fairness
6	10532	3.6	10
18	86563	44.5	12
30	86387	47.5	12
42	13258	47.3	10
48	61751	31.1	12
54	697835	385.9	12

(b) Bounded Sliding Window Protocol

Size	Bound	States	Time	#Fairness
3	1	420	0.2	12
3	2	1596	1.7	
3	3	4095	5.7	
5	1	6900	5.5	20
5	2	32256	42.6	
5	3	123888	223.8	

Most interesting cases respecting parameterized fairness are models with dynamic fairness which cannot be easily predicted from the initial state, e.g., the examples in Sec. 5. Table 2a presents the model checking results for the evolving Dining Philosophers problem from the several initial Collatz numbers, where"#Fairness" is the total number of fairness instances generated during model checking. The results for the bounded sliding window protocol are provided in Table 2b, with different input array sizes and window bounds. In both cases, considerably large numbers of fairness constraints are automatically constructed, and verified within reasonable times.

7 Conclusions

We have presented a logical framework for parameterized fairness, which makes much easier expressing a wide range of fairness constraints that can be specified by universally quantified SE-LTL fairness formulas. We have also presented an on-the-fly algorithm for model checking SE-LTL properties under parameterized fairness, and have shown that it has reasonable performance when compared to other existing model checkers that support fairness. Furthermore, it answers the question of how to verify strong/weak fairness conditions for dynamic systems, in which the number of relevant parameter entities cannot be predicted. We have shown two case studies that require a dynamic, and unpredictable number of fairness conditions, which would be hard to handle by other tools.

Acknowledgments. This work has been partially supported by NSF Grants CNS 07-16638, CNS 08-34709, and CCF 09-05584. The authors would like to thank the referees for comments which helped to improve the paper.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence, 11th edn. MIT Press, Cambridge (1986)
2. Bae, K., Meseguer, J.: The Maude LTLR model checker under parameterized fairness, manuscript (2011), `http://www.cs.uiuc.edu/homes/kbae4/fairness`
3. Bae, K., Meseguer, J.: The Linear Temporal Logic of Rewriting Maude Model Checker. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 208–225. Springer, Heidelberg (2010)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
5. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2001)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Bevilacqua, V., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, pp. 31–37. Springer, Heidelberg (2007)
8. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: A dash of fairness for compositional reasoning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 543–557. Springer, Heidelberg (2010)
9. Couvreur, J., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. Model Checking Software, 169–184 (2005)
10. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems 19, 253–291 (1997)
11. Duret-Lutz, A., Poitrenaud, D., Couvreur, J.-M.: On-the-fly emptiness check of transition-based streett automata. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 213–227. Springer, Heidelberg (2009)
12. Emerson, E.A., Lei, C.: Modalities for model checking: Branching time logic strikes back. Science of Computer Programming 8(3), 275–306 (1987)
13. Francez, N.: Fairness. Springer, Heidelberg (1986)
14. Henzinger, M., Telle, J.: Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, Springer, Heidelberg (1996)
15. Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison Wesley Publishing Company, Reading (2004)
16. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods in System Design 28(1), 57–84 (2006)
17. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering 16(11), 1293–1306 (2002)
18. Lamport, L.: Fairness and hyperfairness. Distributed Computing 13(4) (2000)
19. Latvala, T.: Model checking LTL properties of high-level petri nets with fairness constraints. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 242–262. Springer, Heidelberg (2001)

20. Meseguer, J.: Localized fairness: A rewriting semantics. In: RTA 2005. LNCS, vol. 3467, pp. 250–263. Springer, Heidelberg (2005)
21. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403(2-3), 239–264 (2008)
22. Sun, J., Liu, Y., Dong, J., Pang, J.: PAT: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
23. Tel, G.: Introduction to distributed algorithms. Cambridge University Press, Cambridge (2000)
24. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 137–150. Springer, Heidelberg (2007)

Resolution Proofs and Skolem Functions in QBF Evaluation and Applications

Valeriy Balabanov and Jie-Hong R. Jiang

Department of Electrical Engineering, Graduate Institute of Electronics Engineering,
National Taiwan University, Taipei 10617, Taiwan
balabasik@gmail.com
jhjiang@cc.ee.ntu.edu.tw

Abstract. Quantified Boolean formulae (QBF) allow compact encoding of many decision problems. Their importance motivated the development of fast QBF solvers. Certifying the results of a QBF solver not only ensures correctness, but also enables certain synthesis and verification tasks particularly when the certificate is given as a set of Skolem functions. To date the certificate of a true formula can be in the form of either a (cube) resolution proof or a Skolem-function model whereas that of a false formula is in the form of a (clause) resolution proof. The resolution proof and Skolem-function model are somewhat unrelated. This paper strengthens their connection by showing that, given a true QBF, its Skolem-function model is derivable from its cube-resolution proof of satisfiability as well as from its clause-resolution proof of unsatisfiability under formula negation. Consequently Skolem-function derivation can be decoupled from Skolemization-based solvers and computed from standard search-based ones. Fundamentally different from prior methods, our derivation in essence constructs Skolem functions following the variable quantification order. It permits constructing a subset of Skolem functions of interests rather than the whole, and is particularly desirable in many applications. Experimental results show the robust scalability and strong benefits of the new method.

1 Introduction

Quantified Boolean formulae (QBF) allow compact encoding of many decision problems, for example, hardware model checking [6], design rectification [17], program synthesis [18], two-player game solving [13], planning [15], and so on. QBF evaluation has been an important subject in both theoretical and practical computer sciences. Its broad applications have driven intensive efforts pursuing effective QBF solvers, despite the intractable PSPACE-complete complexity. Approaches to QBF evaluation may vary in formula representations, solving mechanisms, data structures, preprocessing techniques, etc. As a matter of fact, the advances of DPLL-style satisfiability (SAT) solving make search-based QBF evaluation [5] on prenex conjunctive normal form (PCNF) formulae the most popular approach.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 149–164, 2011.

As QBF evaluation procedures are much more complicated than their SAT solving counterparts, validating the results of a QBF solver is more critical than that of a SAT solver. The commonly accepted certificate formats to date are mainly resolution proofs and Skolem-function models. More precisely, for a true QBF, a certificate can be in the syntactic form of a cube-resolution proof (e.g., available in solvers QuBE-cert [12] and yQuaffle [20]) or in the semantic form of a model consisting of a set of Skolem functions (e.g., available in sKizzo [1,2], squolem [9], and Ebddres [9]); for a false QBF, it can be in the syntactic form of a clause-resolution proof (e.g., available in all the above solvers except for sKizzo). Despite some attempts towards a unified QBF proof checker [9], resolution proofs and Skolem-function models remain weakly related. Moreover, the asymmetry between the available certificate formats in the true and false QBF may seem puzzling.

From the application viewpoint, Skolem functions are more directly useful than resolution proofs. The Skolem-function model in solving a true QBF may correspond to, for example, a correct replacement in design rectification, a code fragment in program synthesis, a winning strategy in two-player game solving, a feasible plan in robotic planning, etc. Unfortunately, Skolem-function models are currently only derivable with Skolemization-based solvers, such as sKizzo, squolem, and Ebddres. Moreover, the derivation can be expensive as evidenced by empirical experience that Skolemization-based solvers usually take much longer time on solving true instances than false ones. In contrast, search-based solvers, such as QuBE-cert, can be more efficient and perform more symmetrically in terms of runtime on true and false instances.

This paper takes one step closer to a unified approach to QBF validation by showing that, for a true QBF, its Skolem-function model can be derived from its cube-resolution proof of satisfiability and also from its clause-resolution proof of unsatisfiability under formula negation, both in time linear with respect to proof sizes. Consequently, the aforementioned issues are addressed. Firstly, the connection between resolution proofs and Skolem functions is strongly established. Secondly, it practically conceives Skolem-function *countermodels* for false QBF, and thus yielding a symmetric view between satisfiability and unsatisfiability certifications. Finally, Skolem-function derivation can be decoupled from Skolemization-based solvers and achieved from the more popular search-based solvers, provided that resolution proofs are maintained. A key characteristic of the new derivation is that Skolem functions are generated for variables quantified from outside in, in contrast to the inside-out computation of Skolemization-based solvers. This feature gives the flexibility of computing some Skolem functions of interests, rather than all as in Skolemization-based solvers.

Experimental results show that search-based QBF solver QuBE-cert certifies more QBFEVAL instances[1] than Skolemization-based solvers sKizzo and squolem. Almost all of the Skolem-function models (respectively countermodels)

[1] Since negating the QBFEVAL formulae using Tseitin's conversion [19] may suffer from variable blow up, the Skolem functions are only derived with respect to the original formulae.

are computable, under resource limits, from the cube-resolution proofs of the true cases (respectively clause-resolution proofs of the false cases). On the other hand, for the relation determinization instances (all satisfiable), whose negations are concise by Tseitin's conversion from the circuit structures, their Skolem functions are obtained both from the cube-resolution proof of the original formulae and also from the clause-resolution proof of the negated formulae to compare. The latter tends to be much more robust and shows the unique value of the proposed method.

2 Preliminaries

A *literal* in a Boolean formula is either a variable (i.e., positive-phase literal) or the negation of the variable (i.e., negative-phase literal). In the sequel, the corresponding variable of a literal l is denoted $var(l)$. A *clause* is a Boolean formula consisting of a disjunction of a set of literals; a *cube* is a Boolean formula consisting of a conjunction of a set of literals. In the sequel, we may alternatively specify a clause or cube by a set of literals. A formula in *conjunctive normal form* (CNF) is a conjunction of a set of clauses whereas a *disjunctive normal form* (DNF) formula is a disjunction of a set of cubes. A (quantifier-free) formula ϕ over variables X subject to some truth assignment $\alpha : X' \to \{0, 1\}$ on variables $X' \subseteq X$ is denoted as $\phi|_\alpha$.

2.1 Quantified Boolean Formulae

A *quantified Boolean formula* (QBF) Φ over variables $X = \{x_1, \ldots, x_k\}$ in the *prenex conjunctive normal form* (PCNF) is of the form

$$Q_1 x_1, \ldots, Q_k x_k.\phi, \tag{1}$$

where $Q_1 x_1, \ldots, Q_k x_k$, with $Q_i \in \{\exists, \forall\}$ and variables $x_i \neq x_j$ for $i \neq j$, is called the *prefix*, denoted Φ_{prefix}, and ϕ, a quantifier-free CNF formula in terms of variables X, is called the *matrix*, denoted Φ_{matrix}. We shall assume that a QBF is in PCNF and is totally quantified, i.e., with no free variables. So the set X of variables of Φ can be partitioned into *existential variables* $X_\exists = \{x_i \in X \mid Q_i = \exists\}$ and *universal variables* $X_\forall = \{x_i \in X \mid Q_i = \forall\}$. A literal l is called an *existential literal* and a *universal literal* if $var(l)$ is in $X_\exists$ and $X_\forall$, respectively.

Given a QBF, the *quantification level* $\ell : X \to \mathbb{N}$ of variable $x_i \in X$ is defined to be the number of quantifier alternations between $\exists$ and $\forall$ from left (i.e., outer) to right (i.e., inner) plus 1. For example, the formula $\exists x_1, \exists x_2, \forall x_3, \exists x_4.\phi$ has $\ell(x_1) = \ell(x_2) = 1$, $\ell(x_3) = 2$, and $\ell(x_4) = 3$. For convenience, we extend the definition of ℓ to literals, with $\ell(l)$ for some literal l meaning $\ell(var(l))$.

A clause C with literals $\{l_1, \ldots, l_j\}$ in a QBF Φ over variables X is called *minimal* if

$$\max_{l_i \in C, var(l_i) \in X_\forall} \{\ell(l_i)\} < \max_{l_i \in C} \{\ell(l_i)\}.$$

Otherwise, it is *non-minimal.* A non-minimal clause C can be minimized to a minimal clause C' by removing the literals

$$\{l \in C \mid var(l) \in X_\forall \text{ and } \ell(l) = \max_{l_i \in C}\{\ell(l_i)\}\}$$

from C. This process is called *∀-reduction.* For a clause C of a QBF, we denote its ∀-reduced minimal clause as $\mathrm{MIN}(C)$. Replacing C with $\mathrm{MIN}(C)$ in a QBF does not change the formula satisfiability.

2.2 Q-Resolution

A clause is *tautological* if it contains both literals x and $\neg x$ of some variable x. Two non-tautological clauses C_1 and C_2 are of *distance k* if there are k variables $\{x_1, \ldots, x_k\}$ appearing in both clauses but with opposite phases. The ordinary *resolution* is defined on two clauses C_1 and C_2 of distance 1. If $C_1 = C_1' \vee x$ and $C_2 = C_2' \vee \neg x$, then resolving C_1 and C_2 on the *pivot variable* x yields the *resolvent* $C_1' \vee C_2'$.

Q-resolution [11] extends the ordinary resolution on CNF to PCNF formulae with two rules: First, only existential variables can be the pivot variables for resolution. Second, ∀-reduction is applied whenever possible. Unless otherwise said, "Q-resolution" is shortened to "resolution" in the sequel. In fact (Q-)resolution is a sound and complete approach to QBF evaluation.

Theorem 1 ([11]). *A QBF is false (unsatisfiable) if and only if there exists a clause resolution sequence leading to an empty clause.*

By duality, cube resolution can be similarly defined, and is also sound and complete for QBF evaluation.

Theorem 2 ([8]). *A QBF is true (satisfiable) if and only if there exists a cube resolution sequence leading to an empty cube.*

Modern search-based QBF solvers are equipped with conflict-driven learning, which performs resolution in essence. A tautological clause containing both positive and negative literals of a (universal) variable may result from resolution [21]. Since the clause is resolved from two clauses with distance greater than 1, it is referred to as *long-distance resolution.* Unlike the case in propositional satisfiability, such a clause is not totally redundant as it facilitates implication in QBF evaluation. Nevertheless, long-distance resolution is not essential, and can always be replaced by distance-1 resolution [8].

2.3 Skolemization and Skolem Functions

Any QBF Φ can be converted into the well-known *Skolem normal form* in mathematical logic, which consists of only two quantification levels, first existential and second universal. In the conversion, every existential variable x_i of Φ is replaced in Φ_{matrix} by its respective fresh function symbol, denoted $F[x_i]$, which refers only to the universal variables x_j of Φ with $\ell(x_j) < \ell(x_i)$. These function symbols, corresponding to the so-called *Skolem functions* [16], are then existentially quantified in the first quantification level before the second level of universal quantification over the original universal variables. This conversion, called *Skolemization*, is satisfiability preserving. Essentially a QBF is true if and only if the Skolem functions of its Skolem normal form exist. (Skolemization was exploited in [1] for QBF evaluation.)

In the sequel, we shall extend the notion of Skolem functions in their dual form, also known as the *Herbrand functions.* That is, the Skolem normal form (in the dual) contains two quantification levels, first universal and second existential. For a QBF Φ, in the new notion the Skolem function $F[x_i]$ of a universal variable x_i of Φ refers only to the existential variables x_j of Φ with $\ell(x_j) < \ell(x_i)$. Essentially a QBF is false if and only if the Skolem functions of its Skolem normal form (in the dual) exist.

2.4 QBF Certificates

To validate the results of a QBF solver, resolution proofs and Skolem functions are commonly accepted certificates [12]. For a true QBF, either a cube-resolution proof or a collection of Skolem functions can certify the satisfiability. For a false QBF, a clause-resolution proof can certify the unsatisfiability. In theory, a false QBF can be negated to a true QBF, whose Skolem functions can then be used as a countermodel to the original false QBF. In practice, however, such a countermodel is hardly derivable because negation may result in substantial increase in the formula size or variable count [9]. In contrast, we show that a countermodel can be obtained without formula negation, and thus practical for certifying a false QBF.

3 Model/Countermodel Construction from Resolution Proofs

This section shows a sound and complete approach to construct Skolem functions for existential (respectively universal) variables as the model (respectively countermodel) of a true (respectively false) QBF in time linear with respect to a cube (respectively clause) resolution proof. Since cube and clause resolutions both obey similar deduction rules, we keep attention on the latter only and omit the former.

We consider (Q-)resolution proofs of QBF unsatisfiability that involve no long-distance resolution. As long-distance resolution can always be avoided and replaced by distance-1 resolution [8], our discussion is applicable in general.

Before delving into the main construction, we first define the following formula structure.

Definition 1. *A* Right-First-And-Or (RFAO) formula φ *is recursively defined by*

$$\varphi ::= \mathit{clause} \mid \mathit{cube} \mid \mathit{clause} \wedge \varphi \mid \mathit{cube} \vee \varphi, \qquad (2)$$

where the symbol "::=" is read as "can be" and symbol "|" as "or".

Note that the formula is constructed in order from left to right. Due to the particular building rule of an RFAO formula with priority going to the right, we save on parentheses to enhance readability. For example, formula

$$\begin{aligned}\varphi &= clause_1 \wedge clause_2 \wedge cube_3 \vee clause_4 \wedge cube_5 \vee cube_6\\ &= (clause_1 \wedge (clause_2 \wedge (cube_3 \vee (clause_4 \wedge (cube_5 \vee cube_6))))).\end{aligned}$$

We sometimes omit expressing the conjunction symbol "$\wedge$" and interchangeably use "+" for "$\vee$" in a formula.

In our discussion we shall call a clause/cube in an RFAO formula a *node* of the formula, and omit a node's subsequent operator, which can be uniquely determined. Note that the ambiguity between a single-literal clause and a single-literal cube does not occur in an RFAO formula as the clause-cube attributes are well specified in our construction.

The RFAO formula has two important properties (which will be crucial in proving Theorem 3):

1. If $node_i$ under some (partial) assignment of variables becomes a validated clause (denoted 1-clause) or falsified cube (denoted 0-cube), then we can effectively remove $node_i$ (if it is not the last) from the formula without further valuating it.
2. If $node_i$ becomes a falsified clause (denoted 0-clause) or validated cube (denoted 1-cube), then we need not further valuate (namely, can remove) all other nodes with index greater than i.

Below we elaborate how to construct the countermodel expressed by the RFAO formula from a clause-resolution proof Π of a false QBF Φ. We treat the proof Π as a directed acyclic graph (DAG) $G_\Pi(V_\Pi, E_\Pi)$, where a vertex $v \in V_\Pi$ corresponds to a clause $v.clause$ obtained in the resolution steps of Π and a directed edge $(u, v) \in E_\Pi \subseteq V_\Pi \times V_\Pi$ from the *parent* u to the *child* v indicates that $v.clause$ results from $u.clause$ through either resolution or $\forall$-reduction. The clauses of Π can be partitioned into three subsets: those in Φ_{matrix}, those resulting from resolution, and those from $\forall$-reduction. Let V_M, V_S, and V_D denote their respective corresponding vertex sets. So $V_\Pi = V_M \cup V_S \cup V_D$. Note that in G_Π a vertex in V_M has no incoming edges and is a *source* vertex; a vertex in V_S has two incoming edges from its two parent vertices; a vertex in V_D has one incoming edge from its parent vertex. On the other hand, there can be one or many *sink* vertices, which have no outgoing edges. Since the final clause of Π is an empty clause, the graph G_Π must have the corresponding sink vertex.

The intuition behind our construction stems from the following observations. Firstly, if $V_D = \emptyset$, then the quantifier-free formula Φ_{matrix} is unsatisfiable by itself, and so is Φ. Since there exists an ordinary resolution proof, which involves no $\forall$-reduction, any functional interpretation on the universal variables forms a countermodel to Φ.

Secondly, if $V_S = \emptyset$, then Φ_{matrix} must contain a clause consisting of only universal variables. With only $\forall$-reduction, Φ can be falsified. Without loss of generality, assume this clause is $(l_1 \vee \cdots \vee l_k)$. Then letting the Skolem function of $var(l_i)$ be

$$F[var(l_i)] = \begin{cases} 0 & \text{if } l_i = var(l_i), \text{ and} \\ 1 & \text{if } l_i = \neg var(l_i), \end{cases}$$

for $i = 1, \ldots, k$, forms a countermodel of Φ. (The Skolem functions of the universal variables not in the clause are unconstrained.)

Finally, we discuss the general case where V_D and V_S are non-empty. Every clause $w.clause$ of Π with $w \in V_S$ is implied by the conjunction $u.clause \wedge v.clause$ with $(u, w), (v, w) \in E_\Pi$. (That is, the clause resulting from resolution is *unconditionally* implied by the conjunction of its parent clauses.) Even if the pivot variable of the corresponding resolution were universally quantified, the implication would still hold. So the implication is regardless of Φ_{prefix}. On the other hand, a clause $v.clause$ of Π with $v \in V_D$ is not directly implied by $u.clause$ with $(u, v) \in E_\Pi$. (That is, the clause resulting from ∀-reduction is *conditionally* implied by its parent clause.) Nevertheless Φ_{matrix} and $\Phi_{matrix} \wedge v.clause$ are equisatisfiable under Φ_{prefix}.

To characterize the conditions for an implication (especially between the two clauses involved in a ∀-reduction step) to hold, we give the following definition.

Definition 2. *Let $\alpha : X \to \{0, 1\}$ be a full assignment on variables X. Given two (quantifier-free) formulae ϕ_1 and ϕ_2 over variables X, if the implication $\phi_1 \to \phi_2$ holds under α, then we say that ϕ_2 is* α-implied *by ϕ_1.*

For a resolution proof of a false QBF Φ, when we say a clause is *α-implied*, we shall mean it is α-implied by its parent clause or by the conjunction of its parent clauses depending on whether the clause results from ∀-reduction or resolution. A clause resulting from resolution is surely α-implied for any α, but a clause resulting from ∀-reduction may not be α-implied for some α. We further say that a clause C is *α-inherited* if all of its ancestor clauses (except for the clauses of the source vertices, which have no parent clauses and are undefined under α-implication) and itself are α-implied. Clearly, if C is α-inherited, then $\Phi_{matrix}|_\alpha = (\Phi_{matrix} \wedge C)|_\alpha$.

For a false QBF Φ over variables $X = X_\exists \cup X_\forall$, let the assignment $\alpha : X \to \{0, 1\}$ be divided into $\alpha_\exists : X_\exists \to \{0, 1\}$ and $\alpha_\forall : X_\forall \to \{0, 1\}$. To construct the Skolem-function countermodel, our goal is to determine $\alpha_\forall$ for every $\alpha_\exists$ such that the empty clause of the resolution proof is α-inherited, or there exists an α-inherited clause C with $C|_\alpha = 0$. Therefore, for every assignment $\alpha_\exists$, Φ implies false. That is, such $\alpha_\forall$ provides a countermodel to Φ.

Figure 1 sketches the countermodel construction algorithm, where the Skolem functions for universal variables are computed in RFAO formulae, each of which is stored as an (ordered) array of nodes. Before proving the correctness of the algorithm, we take the following example to illustrate the computation.

```
Countermodel_ construct
input: a false QBF $\Phi$ and its clause-resolution DAG $G_\Pi(V_\Pi, E_\Pi)$
output: a countermodel in RFAO formulae
begin
  foreach universal variable $x$ of $\Phi$
    RFAO_node_array[$x$] := $\emptyset$;
  foreach vertex $v$ of $G_\Pi$ in topological order
    if $v.clause$ resulting from $\forall$-reduction on $u.clause$, i.e., $(u, v) \in E_\Pi$
      $v.cube := \neg(v.clause)$;
      foreach universal variable $x$ reduced from $u.clause$ to get $v.clause$
        if $x$ appears as positive literal in $u.clause$
          push $v.clause$ to RFAO_node_array[$x$];
        else if $x$ appears as negative literal in $u.clause$
          push $v.cube$ to RFAO_node_array[$x$];
    if $v.clause$ is the empty clause
      foreach universal variable $x$ of $\Phi$
        simplify RFAO_node_array[$x$];
      return RFAO_node_array's;
end
```

Fig. 1. Algorithm: Countermodel Construction

Example 1. Let Φ be a false QBF and Π be its resolution proof of unsatisfiability as below.

$$\begin{aligned}
\Phi_{prefix} &= \exists a \forall x \exists b \forall y \exists c \\
\Phi_{matrix} &= (a \vee b \vee y \vee c)(a \vee x \vee b \vee y \vee \neg c)(x \vee \neg b)(\neg y \vee c)(\neg a \vee \neg x \vee b \vee \neg c) \\
&\quad (\neg x \vee \neg b)(a \vee \neg b \vee \neg y)
\end{aligned}$$

$$\Pi = \left\{ \begin{array}{l}
1.\ clause_8 = resolve(clause_1, clause_2) \\
2.\ clause_9 = resolve(clause_3, clause_8) \\
3.\ clause_{10} = resolve(clause_4, clause_5) \\
4.\ clause_{11} = resolve(clause_{10}, clause_6) \\
5.\ clause_{empty} = resolve(clause_{11}, clause_9)
\end{array} \right\}$$

Note that the $\forall$-reduction steps are omitted in Π, which however can be easily filled in as shown in Figure 2, where the clauses are indexed by subscript numbers, and the $\forall$-reduction steps are indexed by the parenthesized numbers indicating the relative order.

By following the steps of the *Countermodel_ construct* algorithm in Figure 1, the RFAO node-array contents after each $\forall$-reduction step in the proof of Figure 2 are listed in order of appearance in Figure 3. The resultant Skolem functions for universal variables x and y are

$$\begin{aligned}
F[x] &= (a) \wedge (a) = a, \text{ and} \\
F[y] &= (\neg ab) \vee ((a \vee x \vee b) \wedge (ax\neg b)),
\end{aligned}$$

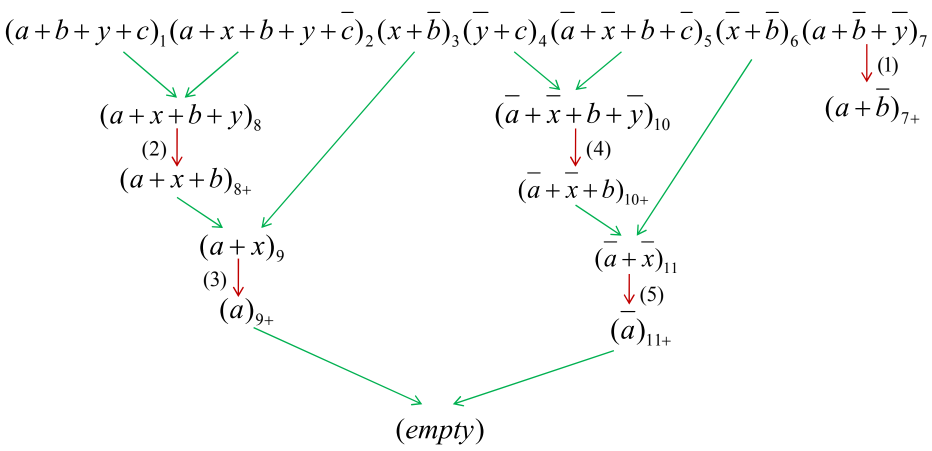


Fig. 2. DAG of resolution proof Π

respectively. Note that the computed $F[y]$ depends on variable x, which can always be eliminated by substituting $F[x]$ for x in $F[y]$. In fact, keeping such dependency may be beneficial as the countermodel can be represented in a multi-level circuit format with shared logic structures. Moreover, observe that clause 7, namely $(a \vee \neg b \vee \neg y)$, is not involved in the resolution steps leading to the empty clause. Its existence is optional in constructing the countermodel, and can be treated as don't cares for countermodel simplification. It can be verified that, for any assignment to variables a, b, and c, formula Φ_{matrix} with variables x and y substituted with $F[x]$ and $F[y]$, respectively, is false.

The correctness of the *Countermodel_construct* algorithm of Figure 1 is asserted below.

Theorem 3. *Given a false QBF Φ and a DAG G_Π corresponding to its resolution proof Π of unsatisfiability, the algorithm* Countermodel_construct*(Φ, G_Π) produces a correct countermodel for the universal variables of Φ.*

Proof. We show that, under every assignment $\alpha_\exists$ to existential variables of Φ, our constructed countermodel always induces some $\alpha_\forall$ such that $\Phi_{matrix}|_\alpha = 0$. There are two possible cases under every such α.

First, assume every clause $v.clause$ with $v \in V_D$ is α-implied. Then the empty clause must be α-inherited because other clauses resulting from resolution are always α-implied. Thus $\Phi_{matrix}|_\alpha = 0$.

Second, assume not every clause $v.clause$ with $v \in V_D$ is α-implied. Let $C_{\alpha\_violate}$ be the set of all such clauses violating α-implication. Suppose $v.clause \in C_{\alpha\_violate}$ is obtained by $\forall$-reduction from $u.clause$ with $(u, v) \in E_\Pi$ on some universal variables. Let $C_{u\backslash v}$ denote the subclause of $u.clause$ consisting of exactly the reduced literals in the $\forall$-reduction leading to $v.clause$. Then $v.clause$ must satisfy the criteria.

0.	$x : [\,]$	$y : [\,]$
1.	$x : [\,]$	$y : [\,\mathit{cube}(\neg ab)\,]$
2.	$x : [\,]$	$y : \begin{bmatrix} \mathit{cube}(\neg ab), \\ \mathit{clause}(a \vee x \vee b) \end{bmatrix}$
3.	$x : [\,\mathit{clause}(a)\,]$	$y : \begin{bmatrix} \mathit{cube}(\neg ab), \\ \mathit{clause}(a \vee x \vee b) \end{bmatrix}$
4.	$x : [\,\mathit{clause}(a)\,]$	$y : \begin{bmatrix} \mathit{cube}(\neg ab), \\ \mathit{clause}(a \vee x \vee b), \\ \mathit{cube}(ax\neg b) \end{bmatrix}$
5.	$x : \begin{bmatrix} \mathit{clause}(a), \\ \mathit{cube}(a) \end{bmatrix}$	$y : \begin{bmatrix} \mathit{cube}(\neg ab), \\ \mathit{clause}(a \vee x \vee b), \\ \mathit{cube}(ax\neg b) \end{bmatrix}$

Fig. 3. Contents of RFAO node arrays

1. $v.\mathit{clause}|_\alpha = 0$ (otherwise $v.\mathit{clause}$ would be α-implied), and
2. $C_{u \setminus v}|_{\alpha_\forall} = 1$ (otherwise $v.\mathit{clause}$ would have the same value as $u.\mathit{clause}$ and thus be α-implied).

It remains to show that, even if $C_{\alpha\_violate}$ is non-empty, there still exists some α-inherited clause C with $C|_\alpha = 0$, i.e., an induced empty clause under α.

Notice that algorithm *Countermodel_ construct* processes G_Π in a topological order, meaning that a clause in the resolution proof is processed only after all of its ancestor clauses are processed. Now we consider all clauses $v.\mathit{clause}$ with $v \in V_D$ in the topological order under the assignment α. Let $v'.\mathit{clause}$ be the first clause encountered with $v'.\mathit{clause}|_\alpha = 0$. (If there is no such $v'.\mathit{clause}$ under α, then it corresponds to the situation analyzed in the first case.) For every universal variable x being reduced from the parent clause $u'.\mathit{clause}$ of $v'.\mathit{clause}$, i.e., $(u', v') \in E_\Pi$, we examine its corresponding `RFAO_node_array[`x`]`. Suppose v' is the ith enumerated vertex that results from $\forall$-reduction involving the reduction of variable x. By the aforementioned two properties of the RFAO formula and by the way how `RFAO_node_array[`x`]` is constructed, we know that the Skolem function value of $F[x]$ under α is not determined by the first $i-1$ nodes, but by the ith node of `RFAO_node_array[`x`]`. In addition, the function value $F[x]$ makes the literal of variable x in clause $C_{u' \setminus v'}$ valuate to false. Because every literal in $C_{u' \setminus v'}$ is valuated to false, we have $u'.\mathit{clause}|_\alpha = 0$ and thus $v'.\mathit{clause}$ is α-implied. Moreover, since $v'.\mathit{clause}$ is the first clause encountered with $v'.\mathit{clause}|_\alpha = 0$, all its ancestor clauses must be α-implied. So $v'.\mathit{clause}$ is α-inherited, and thus $\Phi_{\mathit{matrix}}|_\alpha = 0$.

Because every assignment $\alpha_\exists$ together with the corresponding induced assignment $\alpha_\forall$ makes $\Phi_{matrix}|_\alpha = 0$, the Skolem functions computed by algorithm *Countermodel_construct* form a correct countermodel to Φ. ■

Proposition 1. *Given a false QBF Φ and its resolution proof of unsatisfiability, let $F[x]$ be the Skolem function computed by algorithm* Countermodel_construct *for the universal variable x in Φ. Then $F[x]$ refers to some variable y in Φ only if $\ell(y) < \ell(x)$.*

Note that, by the above strict inequality, Proposition 1 asserts that no cyclic dependency arises among the computed Skolem functions.

In fact algorithm *Countermodel_construct* of Figure 1 can be easily modified to compute the Skolem functions for some (not all) of the universal variables of a given QBF. Let k be the maximal quantification level among the universal variables whose Skolem functions are of interests. Then, by Proposition 1, algorithm *Countermodel_construct* only needs to maintain RFAO node arrays for universal variables with quantification level no greater than k. For Skolemization-based solvers, this partial derivation is not possible because Skolem functions (for existential variables) are constructed on-the-fly during QBF solving, whereas our construction is performed after the entire proof is done.

Proposition 2. *Given a false QBF and its resolution proof of unsatisfiability, algorithm* Countermodel_construct *computes the countermodel in time linear with respect to the proof size.*

Proposition 3. *The RFAO formula size (in terms of nodes) for each universal variable computed by algorithm* Countermodel_construct *is upper bounded by the number of $\forall$-reduction steps in the resolution proof.*

The resolution proofs provided by search-based QBF solvers often contain (redundant) resolution steps unrelated to yielding the final empty clause. Algorithm *Countermodel_construct* works for resolution proofs with and without redundant steps. Since a highly redundant proof may degrade the performance of the algorithm, it may be desirable to trim away redundant parts before countermodel construction. On the other hand, as illustrated in Example 1, it may be possible to exploit the redundancy for countermodel simplification.

The above discussion, concerned about countermodel construction, can be straightforwardly extended under the duality principle to model construction of a true QBF from its cube-resolution proof of satisfiability. We omit similar exposition.

4 Applications to Boolean Relation Determinization

We relate Skolem functions to the problem of *Boolean relation determinization*, which is useful in logic and property synthesis [9,10]. A *Boolean relation* over *input variables* X and *output variables* Y is a characteristic function $R : \{0,1\}^{|X|+|Y|} \to \{0,1\}$ such that assignments $a \in \{0,1\}^{|X|}$ and $b \in \{0,1\}^{|Y|}$ make $R(a,b) = 1$ if and only if (a,b) is in the relation. Relations can be exploited to specify the permissible (non-deterministic) behavior of a system, by restricting its allowed input X and output Y combinations. To be implemented with circuits, a relation has to be *determinized* in the sense that each output variable $y_i \in Y$ can be expressed by some function $f_i : \{0,1\}^{|X|} \to \{0,1\}$. Formally it can be written as a QBF $\forall X, \exists Y.R(X,Y)$, and the determinization problem corresponds to finding the Skolem functions of variables Y.

Often the formula $R(X,Y)$ is not in CNF, but rather in some circuit structure. By Tseitin's transformation, it can be rewritten in CNF $\phi_R(X,Y,Z)$ with the cost of introducing some new intermediate variables Z. Therefore the QBF is rewritten as $\forall X, \exists Y, \exists Z.\phi_R(X,Y,Z)$. By our model construction, the Skolem functions can be computed from its cube-resolution proof of satisfiability. Alternatively, we may compute the Skolem functions by finding the countermodel of $\exists X, \forall Y.\neg R(X,Y)$, which can be again by Tseitin's transformation translated into PCNF $\exists X, \forall Y, \exists Z'.\phi_{\neg R}(X,Y,Z')$ with Z' being the newly introduced intermediate variables in the circuit of $\neg R(X,Y)$. Note that after the negation, the number of quantification levels increases from two to three; on the other hand, ϕ_R and $\phi_{\neg R}$ can be simplified to have the same number of clauses and $|Z| = |Z'|$. The above two approaches are to be studied in the experiments.

It is interesting to note that, since the quantification order of a QBF affects the support variables of a Skolem functions, QBF prefix reordering may be exploited to synthesize Skolem functions with some desired variable dependencies. Moreover, in addition to the relation determinization application, the duality between model and countermodel construction may be useful in other applications whose original formulae are in circuit representation.

5 Experimental Results

The proposed method, named ResQu, was implemented in the C++ language. The experiments were conducted on a Linux machine with a Xeon 2.53 GHz CPU and 48 GB RAM for two sets of test cases: the QBF evaluation benchmarks downloaded from [14] and relation determinization ones modified from [3].

We compared various Skolem-function derivation scenarios using QBF solvers with certification capability, including sKizzo [1], squolem [9], and QuBE-cert.[2] For true QBF instances, sKizzo and squolem were applied to obtain Skolem-function models whereas the cube-resolution proofs produced by QuBE-cert were converted to Skolem-function models by ResQu. For false QBF instances, sKizzo was applied on the negated formulae to obtain Skolem-function countermodels whereas the clause-resolution proofs produced by squolem and QuBE-cert were converted to Skolem-function countermodels by ResQu.

Table 1 summarizes the results of our first experiment on the QBFEVAL'05 and QBFEVAL'06 test sets, which contain 211 and 216 instances, respectively. In the experiment, all the QBF solvers, including sKizzo, squolem, and QuBE-cert, are given a 600-second time limit and a 1-GB memory limit for solving

[2] We did not experiment with Ebddres [9] and yQuaffle [20] as the former tends to generate larger certificates for false QBF compared to squolem, and the latter has characteristics similar to QuBE-cert.

Table 1. Summary for QBFEVAL Benchmarks

		overall	sKizzo		squolem+ResQu				QuBE-cert+ResQu			
		#sv	#sv	time (sv)	#sv	time (sv)	#md	time (md)	#sv/#pg	time (sv)	#md	time (md)
true	'05	84	69	1707.27	50	1490.84	—	—	19/19	414.65	19	54.73
	'06	48	29	295.24	25	199.79	—	—	44/44	859.64	44	152.22
	total	132	98	2002.51	75	1690.63	—	—	63/63	1274.29	63	206.95
false	'05	77	0	0	42	1467.45	42	12.60	46/25	2369.91	25	12.99
	'06	29	0	0	9	85.96	9	0.80	28/22	916.57	22	2.34
	total	106	0	0	51	1553.41	51	13.40	74/47	3286.48	47	15.33

#sv: number of instances solved; #pg: number of proofs involving no long-distance resolution; #md: number of (counter)models generated by ResQu; time (sv/md): CPU time in seconds for QBF evaluation/(counter)model generation; —: data not available due to inapplicability of ResQu

each instance. Under the given resource limits, all solvers, together, solved 132 true and 106 false instances. All the (counter)models produced by ResQu were verified using MiniSat [7] while the models produced by sKizzo and squolem were assumed correct without verification.

It should be mentioned that the resolution proofs produced by QuBE-cert were not simplified, that is, including resolution steps unrelated to producing the final empty clause (or empty cube). The unrelated resolution steps were first removed (with runtime omitted) before the (counter)model construction of ResQu. Moreover, approximately 20% of all the proofs involved long-distance resolution, and ResQu did not construct their (counter)models. On the other hand, the clause-resolution proofs produced by squolem were simplified already and involved no long-distance resolution. Hence ResQu had no problems constructing their countermodels.

We compared the numbers of instances whose (counter)models generated by ResQu and by other tools. When models are concerned, ResQu (via the proofs from QuBE-cert) covered 63 (19 in QBFEVAL'05 and 44 in QBFEVAL'06), whereas sKizzo and squolem in combination covered 105 (75 in QBFEVAL'05 and 30 in QBFEVAL'06). When countermodels are concerned, ResQu (via the proofs from squolem and QuBE-cert) covered 83 (60 in QBFEVAL'05 and 23 in QBFEVAL'06), whereas sKizzo covered 0.[3] Notably, ResQu circumvents the DNF-to-CNF conversion problem and is unique in generating countermodels.

While all the (counter)models can be constructed efficiently (for proofs without long-distance resolution), some of them can be hard to verify. In fact, about 84% of the 161 (counter)models constructed by ResQu were verified within 1 second using MiniSAT; there are 5 models of the true instances in QBFEVAL'06 that remained unverifiable within 1000 seconds.

Table 2 shows the results of our second experiment on 22 relation determinization benchmarks. All the original 22 instances are true (satisfiable). We compared

[3] In addition to sKizzo, in theory squolem can also compute Skolem-function countermodels of false QBF instances by formula negation. We only experimented with sKizzo, which can read in DNF formulae and thus requires no external DNF-to-CNF conversion, arising due to formula negation. Although squolem is not experimented in direct countermodel generation by formula negation, prior experience [9] suggested that it might be unlikely to cover much more cases than sKizzo.

Table 2. Results for Relation Determinization Benchmarks

		(#in, #out, #e, #C)	sKizzo		squolem+ResQu		QuBE-cert+ResQu	
			time (sv)	size	time (sv/md/vf)	size	time (sv/md/vf)	size
true	1	(7, 3, 55, 322)	0.09	377	(0.06 , —, —)	134	(0.03, 0.01, 0.01)	28
	2	(20, 10, 963, 5772)	0.86	1311	(0.79, —, —)	1378	(0.12, 0.03, 0.02)	118
	3	(21, 9, 1280, 7672)	NA		(NA, —, —)		(5.28, 1.74, 1.23)	148883
	4	(24, 12, 1886, 11300)	NA		(1.23, —, —)	179	(0.94, 0.11, 0.03)	1947
	5	(28, 14, 1833, 10992)	NA		(NA, —, —)		(0.35, 0.05, 0.02)	61
	6	(32, 16, 3377, 20250)	NA		(NA, —, —)		(1.21, 0.18, 0.04)	1193
	7	(36, 18, 5894, 35354)	NA		(NA, —, —)		(0.23, 0.15, 0.03)	91
	8	(42, 20, 6954, 41718)	NA		(NA, —, —)		(0.27, 0.12, 0.03)	3
	9	(39, 19, 9823, 58932)	NA		(NA, —, —)		(3.08, 0.50, 0.01)	307
	10	(46, 22, 10550, 63294)	NA		(NA, —, —)		(1.89, 0.25, 0.01)	58
	11	(49, 19, 11399, 68384)	NA		(NA, —, —)		(NA, NA, NA)	
	12	(32, 18, 13477, 80856)	NA		(NA, —, —)		(NA, NA, NA)	
	13	(50, 24, 14805, 88822)	NA		(NA, —, —)		(3.14, 0.77, 0.05)	3458
	14	(53, 25, 16037, 96216)	NA		(NA, —, —)		(3.41, 0.35, 0.02)	283
	15	(56, 26, 19700, 118194)	NA		(NA, —, —)		(8.10, 1.10, 0.05)	905
	16	(59, 27, 26117, 156696)	NA		(NA, —, —)		(3.85, 0.59, 0.03)	187
	17	(65, 29, 29038, 174222)	NA		(NA, —, —)		(7.16, 0.88, 0.05)	232
	18	(62, 28, 30294, 181756)	NA		(NA, —, —)		(9.29, 1.32, 0.05)	731
	19	(72, 32, 35806, 214828)	NA		(NA, —, —)		(NA, NA, NA)	
	20	(68, 30, 50513, 303070)	NA		(NA, —, —)		(2.97, 0.62, 0.05)	11
	21	(95, 35, 57717, 346294)	NA		(NA, —, —)		(NA, NA, NA)	
	22	(41, 23, 89624, 537738)	NA		(NA, —, —)		(NA, NA, NA)	
false	1	(7, 3, 55, 322)	NA		(0.03, 0.01, 0.01)	6	(0.05, NA, NA)	
	2	(20, 10, 963, 5772)	NA		(1.14, 0.02, 0.01)	53	(0.13, NA, NA)	
	3	(21, 9, 1280, 7672)	NA		(0.20, 0.02, 0.01)	4	(1.19, NA, NA)	
	4	(24, 12, 1886, 11300)	NA		(0.31, 0.02, 0.03)	0	(0.30, NA, NA)	
	5	(28, 14, 1833, 10992)	NA		(0.29, 0.02, 0.01)	3	(1.02, NA, NA)	
	6	(32, 16, 3377, 20250)	NA		(1.95, 0.04, 0.03)	3	(0.95, NA, NA)	
	7	(36, 18, 5894, 35354)	NA		(3.08, 0.06, 0.05)	3	(4.22, NA, NA)	
	8	(42, 20, 6954, 41718)	NA		(3.23, 0.07, 0.06)	3	(9.15, NA, NA)	
	9	(39, 19, 9823, 58932)	NA		(9.41, 0.11, 0.08)	5	(10.01, NA, NA)	
	10	(46, 22, 10550, 63294)	NA		(9.87, 0.15, 0.07)	3	(3.62, NA, NA)	
	11	(49, 19, 11399, 68384)	NA		(8.33, 0.20, 0.08)	3	(14.09, NA, NA)	
	12	(32, 18, 13477, 80856)	NA		(10.42, 0.23, 0.10)	3	(10.41, NA, NA)	
	13	(50, 24, 14805, 88822)	NA		(15.82, 0.25, 0.10)	4	(509.84, NA, NA)	
	14	(53, 25, 16037, 96216)	NA		(23.65, 0.27, 0.11)	5	(7.19, NA, NA)	
	15	(56, 26, 19700, 118194)	NA		(30.18, 0.35, 0.14)	3	(25.33, NA, NA)	
	16	(59, 27, 26117, 156696)	NA		(74.19, 0.43, 0.14)	3	(202.80, NA, NA)	
	17	(65, 29, 29038, 174222)	NA		(46.90, 0.42, 0.21)	0	(24.45, NA, NA)	
	18	(62, 28, 30294, 181756)	NA		(84.48, 0.46, 0.25)	4	(94.93, NA, NA)	
	19	(72, 32, 35806, 214828)	NA		(129.84, 0.41, 0.22)	3	(80.12, NA, NA)	
	20	(68, 30, 50513, 303070)	NA		(363.12, 0.70, 7.31)	3	(26.14, NA, NA)	
	21	(95, 35, 57717, 346294)	NA		(359.40, 0.96, 8.15)	2	(86.10, NA, NA)	
	22	(41, 23, 89624, 537738)	NA		(NA, NA, NA)		(142.24, NA, NA)	

#in: number of input variables in the relation; #out: number of output variables in the relation; #e: number of innermost existential variables added due to circuit-to-CNF conversion; #C: number of clauses in final CNF; size: number of AIG nodes after performing ABC command `dc2` with negligible runtime not shown; time (sv/md/vf): CPU time in seconds for QBF evaluation/(counter)model generation/verification; NA: data not available due to computation out of resource limit; —: data not available due to inapplicability of ResQu

their models obtained in two ways: by direct model construction from the satisfiability proofs of the original formulae and by indirect model construction from the unsatisfiability proofs of their negations. Unlike the QBFEVAL cases, negating these formulae by Tseitin's transformation does not result in variable- and clause-increase, as discussed in Section 4. The experiment was conducted under the resource limit same as before. For the original instances, ResQu could have generated Skolem functions only for the existential variables of interests

Table 3. Summary for Relation Determinization Benchmarks

	overall	sKizzo		squolem+ResQu				QuBE-cert+ResQu			
	#sv	#sv	time (sv)	#sv	time (sv)	#md	time (md)	#sv/#pg	time (sv)	#md	time (md)
true	17	2	0.95	3	2.08	—	—	17/17	51.32	17	8.77
false	22	0	0	21	1175.84	21	5.20	22/0	1254.28	0	0

(Legend same as in Table 1)

(namely, the output variables of a Boolean relation rather than the intermediate variables), but it generated all for the verification purpose.

As summarized in Table 3, for the true cases, sKizzo and squolem in combination can construct models for only 3 instances, whereas from the 17 proofs of QuBE-cert, ResQu can generate (and verify) all models. For the negated cases, all the proofs provided by QuBE-cert involved long-distance resolution, so ResQu did not construct their countermodels. Nevertheless, squolem solved 21 out of 22 instances, and ResQu can generate (and verify) all their countermodels (i.e., models for the original QBF). It is interesting to see that, in the relation determinization application, countermodel generation for the negated formulae can be much easier than model generation for the original formulae. It reveals the essential value of ResQu.

6 Conclusions and Future Work

A new approach has been proposed to compute Skolem functions in the context of QBF evaluation. As a result, Skolem-function derivation is decoupled from Skolemization-based solvers, and is available from standard search-based solvers, provided that proper resolution proofs are given. The approach gives a balanced and unified view on certifying both true and false QBF using models and countermodels. Moreover, its practical value has been strongly supported by experiments. As Skolem functions can be important in various areas, we hope our results may encourage and enable QBF applications. Our on-going work is to extract Skolem functions from proofs with the presence of long-distance resolution.

Acknowledgments

The authors are grateful to Roderick Bloem and Georg Hofferek for providing the relation determinization benchmarks. This work was supported in part by the National Science Council under grants NSC 99-2221-E-002-214-MY3 and NSC 99-2923-E-002-005-MY3.

References

1. Benedetti, M.: Evaluating qBFs via symbolic skolemization. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 285–300. Springer, Heidelberg (2005)
2. Benedetti, M.: Extracting Certificates from Quantified Boolean Formulas. In: Proc. Int.Joint Conf. on Artificial Intelligence (IJCAI) (2005)

3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic Hardware Synthesis from Specifications: A Case Study. In: Proc. Design Automation and Test in Europe (2007)
4. Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, `http://www.eecs.berkeley.edu/~alanmi/abc/`
5. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. Journal of Automated Reasoning 28(2), 101–142 (2002)
6. Dershowitz, N., Hanna, Z., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause-Term Resolution and Learning in Quantified Boolean Logic Satisfiability. Artificial Intelligence Research 26, 371–416 (2006)
9. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 201–214. Springer, Heidelberg (2007)
10. Jiang, J.-H.R., Lin, H.-P., Hung, W.-L.: Interpolating Functions from Large Boolean Relations. In: Proc. Int.Conf. on Computer-Aided Design (ICCAD), pp. 779–784 (2009)
11. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Information and Computation 117(1), 12–18 (1995)
12. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and Certifying QBFs: A Comparison of State-of-the-Art Tools. AI Communications (2009)
13. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
14. QBF Solver Evaluation Protal, `http://www.qbflib.org/qbfeval/`
15. Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. Journal of Artificial Intelligence Research 10, 323–352 (1999)
16. Skolem, T.: Über die Mathematische Logik. Norsk. Mat. Tidsk. In: van Heijenoor, J. (ed.) Translation in From Frege to Gödel, A Source Book in Mathematical Logic, vol. 10, pp. 125–142. Harvard Univ. Press, Cambridge (1928)
17. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 355–368. Springer, Heidelberg (2007)
18. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S., Saraswat, V.: Combinatorial Sketching for Finite Programs. In: Int.Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS), pp. 404–415 (2006)
19. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic, 466–483 (1970)
20. Yu, Y., Malik, S.: Validating the Result of a Quantified Boolean Formula (QBF) Solvers: Theory and Practice. In: Proc. Asia and South Pacific Design Automation Conference (2005)
21. Zhang, L., Malik, S.: Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In: Proc. Int'l Conf. on Computer-Aided Design (ICCAD), pp. 442–449 (2002)

The BINCOA Framework for Binary Code Analysis*

Sébastien Bardin[1], Philippe Herrmann[1], Jérôme Leroux[2], Olivier Ly[2],
Renaud Tabary[2], and Aymeric Vincent[2]

[1] CEA, LIST, Gif-sur-Yvette CEDEX, F-91191, France
first.last@cea.fr
[2] LaBRI, 351 cours de la Libération, 33405 Talence Cedex
first.last@labri.fr

Abstract. This paper presents the BINCOA framework, whose goal is to ease the development of binary code analysers by providing an open formal model for low-level programs (typically: executable files), an XML format for easy exchange of models and some basic tool support. The BINCOA framework already comes with three different analysers, including simulation, test generation and Control-Flow Graph reconstruction.

1 Introduction

Automatic analysis of programs from their executable files is a recent and promising field of research, opening the way to new applications of software verification (mobile code, off the shelf components, legacy code, malware) and more accurate and reliable analyses (taking into account the compilation step). In the last years, a few teams have been involved in this emerging research field and a few techniques and tools have been developed [1,3,5,6,7,8,9,10,13,14], mostly based on static analysis and symbolic execution.

The problem. Besides specific theoretical challenges, binary code analysis suffers from two major practical issues. First, implementing a binary code analyser requires lots of programming efforts. There exist many different instruction set architectures (ISA), and each ISA counts several dozens of instructions. Hence adding support for a new ISA is a time-consuming, tedious and error-prone activity. Moreover, it follows that each analyser supports only very few ISAs, making different technologies and tools difficult to compare. Second, each analyser comes with its own formal model of binary code. Since the exact semantics is seldom available, modelling hypotheses are often unclear and may differ from one tool to the other, making results and models difficult to reuse.

The BINCOA framework. We describe in this paper the BINary COde Analysis (BINCOA) framework, whose aim is to ease the development of binary code analysers.

1- The framework is constructed around Dynamic Bitvector Automata (DBA), a generic and concise formal model for low-level programs. The main design ideas behind DBA are the following: (a) a small set of instructions; (b) a concise and natural modelling for common architectures; (c) self-contained models which do not require a

* Work partially funded by ANR grant ANR-08-SEGI-006.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 165–170, 2011.

separate description of the memory model or of the architecture; and (d) a sufficiently low-level formalism, so that DBA can serve as a reference semantics of the executable file to analyse. Most features of low-level programs are taken into account by this formalism, including dynamic jumps, modification of the call stack, instruction overlapping and endianness. The two main limitations are the following: the formalism cannot capture self-modifying code and it is untimed.

2- We intend to gather an ecosystem of binary code analysers around DBA, all tools being able to share their front-ends and exchange their results. To this end, we have defined an XML DTD to communicate DBA and we provide open-source code for basic DBA manipulation, including XML input/output and DBA simplifications.

3- DBA are already used by three different analysers: Osmose [3,4] for test generation, TraceAnalyzer for safe Control-Flow Graph (CFG) reconstruction (based on [6]) and Insight, a platform providing front-end, simulation and some value analysis mechanism. Altogether, these three tools prove that DBA can encode a few different architectures and ISAs (including PowerPC and x86).

Why using DBA and the BINCOA framework? The BINCOA framework eases the development of binary code analysers: (semantics) all analysers built upon DBA can be fairly compared and their results can be safely reused from one tool to the other; (engineering) the BINCOA framework provides open-source basic facilities for DBA manipulations. In the future we also plan to provide an open-source platform allowing to share front-ends and ISA support.

Moreover, we think that DBA are a good trade-off between conciseness and ease of use: there are only about two dozen of operators (to be compared with any ISA) and modelling common ISAs with DBA is straightforward. DBA have already been used to encode four different ISA, including x86 and PowerPC.

Outline. The remaining part of the paper is structured as follows: the DBA model is described in Section 2, its practical usage is discussed in Section 3, tool support for DBA and different analysers based on DBA are presented in Section 4, finally Section 5 describes related work and Section 6 provides a conclusion and some future work.

2 DBA in a Nutshell

The syntax and semantics of DBA is sketched hereafter. A detailed description can be found in the technical report [2].

DBA are automata extended with a finite set of variables ranging over fixed-width bit-vectors and a finite set of (disjoint) fixed-size arrays of bytes (bit-vectors of size 8). Some of the nodes of the automaton are labelled with addresses ranging over $\mathbb{N}$. Transitions are decorated with basic instructions: assignments (**Assign** lhs := rhs), no-operation (**Skip**), guards (**Guard** cond), jumps to a non-statically known address (**Jump** expr), and an instruction to handle absent code such as API calls (**External** φ). The first three kind of instructions are standard. The **External** instruction, followed by a first-order formula over bit-vectors and arrays, allows to introduce a non-deterministic computation step defined in a pre/post-condition style. The **Jump** instruction is described hereafter. The operational semantics is given by a transition system in a standard manner. Expressions and conditions are built upon a small set of standard fixed-width bit-vector operators, including (signed / unsigned) arithmetic operators, reified (signed / unsigned) arithmetic relational operators, logical bitwise operators, size extensions, shifts, concatenation and restriction. Contrary to real processor instructions, these operators are side-effect free. Every expression evaluates to a bit-vector of statically known size (this is not a restriction considering current ISAs).

Original features. DBA provide a few original mechanisms dedicated to low-level languages. (1) Dynamic jumps (**Jump** transitions) are *dangling*, i.e. they do not have a predefined target node. When the transition is fired, the jump expression is evaluated and turned into an integer a, and the control-flow goes to the node labelled by a. Dynamic jumps are necessary for modelling indirect branching. (2) Bit manipulations can be expressed easily thanks to the restriction operator, available both in lhs and rhs operands. (3) Multiple-byte read and write operations (and incidentally endianness) can be expressed easily thanks to a dedicated array-access operator of the form: $\mathrm{array}[expr; k^{\#}]$, where $k \in \mathbb{N}$ and $\# \in \{\leftarrow, \rightarrow\}$, denoting the k consecutive indices starting at index $expr$ and accessed in big-endian ($\rightarrow$) or little-endian ($\leftarrow$). (4) Memory zone properties define specific behaviours for segments of arrays; currently available properties are *write-is-ignored*, *write-aborts*, *read-aborts* and *volatile* (with their intuitive meanings).

A few remarks. (1) The set of operators in DBAs is not minimal, however we think that it is a nice trade-off between conciseness and ease of use. (2) Realistic programs have typically only a few dynamic jumps, hence "realistic" DBAs will behave mostly as standard extended automata. This motivates the choice of an automata-based formalism for DBA. (3) DBAs do not have native support for procedure calls and returns: they are encoded as jumps, as it is the only correct semantic for call/return instructions found in ISAs. However, since this extra information may be useful if treated with care, the XML format for DBA (see Section 4) allows to annotate jumps with call/return tags.

3 Modelling Low Level Programs with DBA

Basics. Most architectures and ISAs can be modelled accurately using the following rules. Each register in the processor is modelled by a variable in the automaton, additional variables ("local" variables) may be introduced to encode ISA instructions needing intermediate results, e.g. for side-effects. An ISA instruction at address a is translated in at least one node labelled by a and one transition. Additional ("local") nodes and transitions may be needed for intermediate computations. The additional nodes are not labelled. A single array is usually sufficient for memory. Additional arrays may allow for example to distinguish an I/O bus from a memory bus. Memory zone properties can be used for ROM (*write-is-ignored*), code section of the program (*write-aborts*, allowing to detect self-modifying code) or memory controlled by an external device (*volatile*). Instructions with side effects (e.g. flag updating) are split into several DBA instructions by adding local nodes and transitions.

Figure 1 presents a few examples of ISA instruction modelling through DBA. We suppose that each ISA instruction is encoded on four bytes. The ISA instruction is on the left column, and the corresponding DBA is on the right. ISA instructions are supposed to be located at address `0x5003` in the executable file. For the second example (an

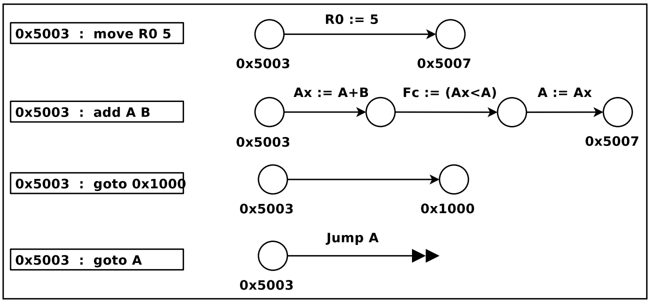


Fig. 1. DBA encoding of a few typical instructions

addition instruction), we suppose that the instruction updates a carry flag `Fc` (the carry-flag is set to 0 iff the *unsigned* addition is correct).

Open programs and interruptions. DBA provides various ways to model programs interacting with an external environment, either hardware (sensor/actuator) or software (OS). A sequential interaction may be simulated either by a stub or by a logical specification (**External**). A concurrent interaction may be simulated in the general case by a product of DBA, and in simple cases by declaring a volatile memory zone.

Limitations. The two strongest limitations of DBA are that they are untimed and that they cannot encompass self-modifying code. There are also a few "weaker" limitations, i.e. mechanisms with no native support, which can still be modelled in DBA but at a possibly high cost. These limitations are mainly dynamic memory (de-)allocation, run-time modification of endianness and asynchronous interruptions. Finally, DBA do not provide any operator for floating-point arithmetic. It is straightforward to add them to the model, but taking them into account in the analysers is much more demanding.

4 The BINCOA Framework

Tool support for DBA manipulation. We provide open-source OCaml code for basic DBA manipulation[1]. The module contains a datatype for DBA, import / export functions from / to the XML format defined in the technical report [2], as well as type checking (based on bit-vector sizes) and simplification functions for a few typical inefficient patterns observed in automatic DBA generation (typically, removing useless flag computations). These simplifications are inspired by standard code optimisation techniques (peephole, dead code elimination, etc.), and are adapted to be sound on *partial* DBAs, in case where the DBA is recovered incrementally from an executable file. We observed a reduction from 10% to 55% of the number of DBA instructions with these simplifications. The XML parser is based on xml-light [2] and the library counts about 3 kloc. The code is under GPL license.

Insight: decoding, simulation and analysis platform.[3] Insight is a platform developed mostly in C++ and offers the ability to load executable files supported by the GNU BFD

[1] `https://bincoa.labri.fr/`

[2] `http://tech.motion-twin.com/xmllight`

[3] `http://insight.labri.fr/`

library and disassemble them with a homebrew disassembler. Its internal representation is very close to DBA and import/export of DBA in XML is possible. It offers a general setting for concrete and symbolic execution of the model, as well as a generic annotation facility which allows to prove assertions using weakest precondition computation. The platform currently has three satellite tools allowing to disassemble a program, to execute it concretely or symbolically (intervals, sets of values, probability distributions), and to apply control flow graph reconstruction to polymorphic virus analysis.

Osmose: test data generation. Osmose [3,4] is a test data generation tool for binary code, based on dynamic symbolic execution and bit-vector constraint solving. The tool also offers test suite replay via a simulation engine, test suite completion, (unsafe) test suite coverage estimation, (under-approximated) CFG recovery and a graphical user interface. Front-ends are available for PowerPC, Intel 8051 and Motorola 6800. The tool can export / import DBA given in XML. The program contains 75 kloc of OCaml. A few industrial case-studies have been successfully carried out.

TraceAnalyzer: safe and precise CFG recovery. TraceAnalyzer performs safe and precise CFG reconstruction from an executable file. The core technology is a refinement-based static analysis [6]. The program is about 29 kloc of C++. A front-end for PowerPC is available, as well as import / export facilities from / to DBA.

A concrete example of cooperation between tools. TraceAnalyzer and Osmose are able to communicate in two ways. First, Osmose can receive from TraceAnalyzer an upper approximation of every set of jump targets and take advantage of it to provide a safe coverage measure, which is crucial for example in critical system certification. Second, TraceAnalyzer can receive from Osmose a set of observed jump targets, and take advantage of it to efficiently bootstrap its refinement-based static analysis.

5 Related Work

Many binary code analysers have been developed recently, for example to name a few: CodeSurfer/x86 [1], Sage [8], Bitscope [5], Osmose [3], Jakstab [9] and McVeto [14]. However most of them are based on a “private” formal model, with no available specification. We are aware of two other generic low-level models suitable for executable analysis, but none of them is open. DBA can be seen as the successor of the Generic Assembly Language (GAL) of Osmose [4], which is similar to DBA in both goals and shape. However DBA are more concise, easier to manipulate and more expressive than GAL. Actually, GAL shows a few shortcomings that have been addressed in DBA: no loops in intermediate nodes, no native support for endianness, unduly complex operators with multiple return-values. Osmose is being redesigned to work on DBAs instead of GAL. TSL [12], developed by Lim and Reps to re-implement CodeSurfer/x86 [1], is based on semantic reinterpretation: each instruction of the ISA is given a concrete semantic written in a ML-like language (with only a limited set of basic operators); adding a new analysis is mainly done by overloading every basic operator. This is rather similar to the idea behind BINCOA, the ML description serving as the reference model. TSL and DBA should have more or less the same modelling power, however comparison is difficult since TSL is not publicly available.

LLVM [11] is a generic low-level language designed for compilation rather than verification. Hence LLVM abstraction level is in between binary code and C: it provides many low-level operations, as well as higher level features based on the knowledge of the initial source code (types, native array manipulation).

6 Conclusion and Perspectives

This paper presents the BINCOA framework for binary code analysis. BINCOA aims at easing the development of binary code analysers by providing an open formal model (DBA) for low-level programs, an XML format to allow easy exchange of both models and benchmarks and some basic tool support. Future work comprises providing more open-source support (visualisation tools, x86 and ARM front-ends) as well as extending DBA with native support for memory allocation and facilities for self-modifying code.

References

1. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: CodeSurfer/x86—A Platform for Analyzing x86 Executables. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 250–254. Springer, Heidelberg (2005)
2. Bardin, S., Fleury, E., Herrmann, P., Leroux, J., Ly, O., Sighireanu, M., Tabary, R., Touili, T., Vincent, A.: Description of the BINCOA Model. Deliverable J1.1 part 2 of ANR Project BINCOA (2009), `https://bincoa.labri.fr/`
3. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
4. Bardin, S., Herrmann, P.: OSMOSE: Automatic Structural Testing of Executables. International Journal of Software Testing, Verification and Reliability (STVR) 21(1) (2011)
5. Brumley, D., Hartwig, C., Kang, M.: BitScope: Automatically Dissecting Malicious Binaries. Carnegie Mellon Uni. technical report CMU, pp. 7–133 (2007)
6. Bardin, S., Herrmann, P., Védrine, F.: Refinement-Based CFG Reconstruction from Unstructured Programs. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 54–69. Springer, Heidelberg (2011)
7. Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
8. Godefroid, P., Levin, M., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008, The Internet Society (2008)
9. 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)
10. 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)
11. Lattner, C.: The LLVM Compiler Infrastructure Project, `http://llvm.org/`
12. 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)
13. Reps, T., Lim, J., Thakur, A., Balakrishnan, G., Lal, A.: There's plenty of room at the bottom: Analyzing and verifying machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 41–56. Springer, Heidelberg (2010)
14. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)

CVC4*

Clark Barrett[1], Christopher L. Conway[1], Morgan Deters[1],
Liana Hadarean[1], Dejan Jovanović[1], Tim King[1],
Andrew Reynolds[2], and Cesare Tinelli[2]

[1] New York University
[2] University of Iowa

Abstract. CVC4 is the latest version of the Cooperating Validity Checker. A joint project of NYU and U Iowa, CVC4 aims to support the useful feature set of CVC3 and SMT-LIBv2 while optimizing the design of the core system architecture and decision procedures to take advantage of recent engineering and algorithmic advances. CVC4 represents a completely new code base; it is a from-scratch rewrite of CVC3, and many subsystems have been completely redesigned. Additional decision procedures for CVC4 are currently under development, but for what it currently achieves, it is a lighter-weight and higher-performing tool than CVC3. We describe the system architecture, subsystems of note, and discuss some applications and continuing work.

1 Introduction

The Cooperating Validity Checker series has a long history. The Stanford Validity Checker (SVC) [3] came first, incorporating theories and its own SAT solver. Its successor, the Cooperating Validity Checker (CVC) [16], had a more optimized internal design, produced proofs, used the Chaff [13] SAT solver, and featured a number of usability enhancements. Its name comes from the *cooperative* nature of decision procedures in Nelson-Oppen theory combination [14], which share amongst each other equalities between shared terms. CVC Lite [1], first made available in 2003, was a rewrite of CVC that attempted to make CVC more flexible (hence the "lite") while extending the feature set: CVC Lite supported quantifiers where its predecessors did not. CVC3 [4] was a major overhaul of portions of CVC Lite: it added better decision procedure implementations, added support for using MiniSat [11] in the core, and had generally better performance.

CVC4 is the new version, the fifth generation of this validity checker line that is now celebrating fifteen years of heritage. It represents a complete re-evaluation of the core architecture to be both performant and to serve as a cutting-edge research vehicle for the next several years. Rather than taking CVC3 and redesigning problem parts, we've taken a clean-room approach, starting from scratch.

* This work partially supported by the NSF (CCF–0644299, CCF–0914956, CNS–1049495, and 0914877), AFOSR (FA9550–09–1–0596 and FA9550–09–1–0517), SRC 2008–TJ–1850, and MIT Lincoln Laboratory.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 171–177, 2011.

Before using any designs from CVC3, we have thoroughly scrutinized, vetted, and updated them. Many parts of CVC4 bear only a superficial resemblance, if any, to their correspondent in CVC3. However, CVC4 is fundamentally similar to CVC3 and many other modern SMT solvers: it is a DPLL(T) solver [12], with a SAT solver at its core and a delegation path to different decision procedure implementations, each in charge of solving formulas in some background theory. The re-evaluation and ground-up rewrite was necessitated, we felt, by the performance characteristics of CVC3. CVC3 has many useful features, but some core aspects of the design led to high memory use, and the use of heavyweight computation (where more nimble engineering approaches could suffice) makes CVC3 a much slower prover than other tools. As these designs are central to CVC3, a new version was preferable to a selective re-engineering, which would have ballooned in short order. Some specific deficiencies of CVC3 are mentioned in this article.

2 Design of CVC4

CVC4 is organized around a central core of *engines:*

- The *SMT Engine* serves as the main outside interface point to the solver. Known in previous versions of CVC as the *ValidityChecker*, the *SMT Engine* has public functions to push and pop solving contexts, manipulate a set of currently active assumptions, and check the validity of a formula, as well as functions to request proofs and generate models. This engine is responsible for setting up and maintaining all user-related state.
- The *Prop Engine* manages the propositional solver at the core of CVC4. This, in principle, allows different SAT solvers to be plugged into CVC4. (At present, only MiniSat is supported, due to the fact that a SAT solver must be modified to dispatch properly to SMT routines.)
- The *Theory Engine* serves as an "owner" of all decision procedure implementations. As is common in the research field, these implementations are referred to as *theories* and all are derived from the base class *Theory.*

CVC3 used what was in effect a domain-specific language for proof rules, which formed the trusted code base of the system. No fact could be registered by the system without first constructing a *Theorem* object, and no *Theorem* object could be constructed except through the trusted proof rules.

CVC4's design takes a different approach. Much time and memory was spent in CVC3's *Theorem*-computation. When not producing proofs, CVC4 uses a more lightweight approach, with *Theory* objects similar to those suggested by modern DPLL(T) literature [12] and used in other solvers. CVC4's *Theory* class is responsible for checking consistency of the current set of assertions, and propagating new facts based on the current set of assertions. *Theorem* objects are not produced up front for theory-propagated facts, but rather can be computed lazily (or not at all, when the DPLL core doesn't require them).

CVC4 incorporates numerous *managers* in charge of managing subsystems:

- The *Node Manager* is one of the busiest parts of CVC4, in charge of the creation and deletion of all expressions ("nodes") in the prover. *Node* objects are immutable and subject to certain simplifying constraints.[1] Further, *Node* objects are unique; the creation of an already-extant *Node* results in a reference to the original. Node data is reference-counted (the *Node* class itself is just a reference-counted smart pointer to node data) and subject to reclamation by the *Node Manager* when no longer referenced; for performance reasons, this is done lazily (see below for performance justification).
- The *Shared Term Manager* is in charge of all shared terms in the system. Shared terms are detected by the *Theory Engine* and registered with this manager, and this manager broadcasts new equalities between shared terms.
- The *Context Memory Manager* is in charge of maintaining a coherent, backtrackable data context for the prover. At its core, it is simply a region memory manager, from which new memory regions can be requested ("pushed") and destroyed ("popped") in LIFO order. These regions contain saved state for a number of heap-allocated objects, and when a pop is requested, these heap objects are "restored" from their backups in the region. This leads to a nice, general mechanism to do backtracking without lots of *ad hoc* implementations in each theory; this is highly useful for rapid prototyping. However, as a general mechanism, it must be used sparingly; it is often beneficial to perform backtracking manually within a theory using a lighter-weight method, to timestamp to indicate when a previously-computed result is stale, or to develop approaches requiring little or no backtracking at all (*e.g.*, tableaux in Simplex).

2.1 Expressions ("nodes")

Expressions are represented by class *Node* and are considerably more efficient than CVC3's expression representation. In the latest version of CVC3, expressions maintain 14 word-sized data members (plus pointers to child expressions). In CVC4, nodes take 64 bits plus child pointers, a considerable space savings. (In part, this savings results from clever bit-packing. Part is in storing node-related data outside of *Node* objects when appropriate.)

The expression subsystem of CVC4 has been carefully designed, and we have analyzed runtime profiling data to ensure its performance is reasonable. On stress tests, it beats CVC3's expression subsystem considerably. We performed a handful of targeted experiments to demonstrate this (all results are speedups observed over a large number of iterations of the same test within the same process):

Set-up/tear-down. First, we wanted to measure raw set-up and tear-down time for the CVC4 expression subsystem with respect to CVC3. For CVC3, this

[1] For example, PLUS nodes, representing arithmetic addition, must have two or more children. This is specified by the theory of arithmetic and enforced by the *Node Manager*; this arity can then be assumed by code that manipulates arithmetic-kinded nodes. If an input language permits unary PLUS, that language's parser must convert that input expression into a valid CVC4 *Node*.

involves the creation and destruction of a *ValidityChecker* object. For CVC4, this involves the creation and destruction of an *Expression Manager*. CVC4 performs this task almost 10× faster than CVC3.

Same-exprs. CVC4 keeps a unique copy of expression information for each distinct expression. When the client requests an expression node, a lookup in an internal node table is performed to determine whether it already exists; if it does, a pointer to the existing expression data is returned (if not, a pointer to a new, freshly-constructed expression data object is returned). CVC3's behavior is similar. For this stress test, we created a simple expression, then pointed away from it, causing its reference count to drop to 0. CVC4 is 3.5× faster than CVC3 at this simple task. This is largely because CVC3 does garbage-collection eagerly; it thus does collection work when the reference count on the expression data drops to 0, and must construct the expression anew each time it is requested. CVC4's lazy garbage-collection strategy never collects the expression data (as it is dead only a short time) and therefore must never re-construct it anew.

Same-exprs-with-saving. Because the reference count on node data falls to 0 in the previous test, we performed a similar test where the reference count never drops to 0. This removes the advantage of the lazy collection strategy and measures the relative performance CVC4's lookup in its internal node table. Because the same expression is requested each time, the lookup is always successful (after the first time). CVC4's advantage in this test drops to 1.5× speedup over CVC3, less of an advantage but still considerably faster.

Separate-exprs. Finally, the performance of raw expression construction is measured by producing a stream of new expressions. These will each result in a failed lookup in the internal node table and the construction of a new expression structure. Here, CVC4 is again roughly 3.5× faster than CVC3.

As mentioned above, all of the above stress tests were run for a high number of iterations (at least ten million) to get stable performance data on which to base the comparisons. In the *separate-exprs* test, expressions were built over fresh variables, ensuring their distinctness.

We conclude that CVC4's expression subsystem has better performance in setting up and tearing down, in creating already-existing expressions, and in creating not-yet-existing expressions. We further demonstrated one case justifying the use of a lazy garbage collector implemented in CVC4 over the eager one in CVC3.

We performed similar experiments on typical linear arithmetic workloads (drawn from QF_LRA benchmarks in the SMT-LIB library). The time for the expression subsystem operations (not involving any solver machinery) was roughly 1.4× faster in CVC4 than in CVC3, and CVC4 allocated only a quarter of the memory that CVC3 did.

2.2 Theories

CVC4 incorporates newly-designed and implemented decision procedures for its theory of uninterpreted functions, its theory of arithmetic, of arrays, of bitvectors, and of inductive datatypes, based on modern approaches described in the literature. Performance generally is far better than CVC3's (see the note in the conclusion).

In a radical departure from CVC3, CVC4 implements a version of the Simplex method in its implementation of arithmetic [10], whereas CVC3 (and earlier provers in the CVC line) had used an approach based on Fourier-Motzkin variable elimination.

2.3 Proofs

CVC4's proof system is designed to support LFSC proofs [15], and is also designed to have *absolutely zero footprint* in memory and time when proofs are turned off at compile-time.

2.4 Library API

As CVC4 is meant to be used via a library API, there's a clear division between the public, outward-facing interface, and the private, inward-facing one. This is a distinction that wasn't as clear in the previous version; installations of CVC3 required the installation of *all* CVC3 header files, because public headers depended on private ones to function properly. Not so in CVC4, where only a subset of headers declaring public interfaces are installed on a user's machine.

Further, we have decided "to take our own medicine." Our own tools, *including CVC4's parser and main command-line tool,* link against the CVC4 library in the same way that any end-user application would. This helps us ensure that the library API is complete—since if it is not, the command-line CVC4 tool is missing functionality, too, an omission we catch quickly. This is a considerable difference in design from CVC3, where it has often been the case that the API for one or another target language was missing key functionality.

2.5 Theory Modularity

Theory objects are designed in CVC4 to be highly *modular:* they do not employ global state, nor do they make any other assumptions that would inhibit their functioning as a client to another decision procedure. In this way, one *Theory* can instantiate and send subqueries to a completely subservient client *Theory* without interfering with the main solver flow.

2.6 Support for Concurrency

CVC4's infrastructure has been designed to make the transition to multiprocessor and multicore hardware easy, and we currently have an experimental lemma-sharing portfolio version of CVC4. We intend CVC4 to be a good vehicle for other research ideas in this area as well. In part, the modularity of theories (above) is geared toward this—the absence of global state and the immutability of expression objects clearly makes it easier to parallelize operations. Similarly, the *Theory* API specifically includes the notion of *interruptibility*, so that an expensive operation (*e.g.*, theory propagation) can be interrupted if work in another thread makes it irrelevant. Current work being performed at NYU and U Iowa is investigating different ways to parallelize SMT; the CVC4 architecture provides a good experimental platform for this research, as it does not need to be completely re-engineered to test different concurrent solving strategies.

3 Conclusion

SMT solvers are currently an area of considerable research interest. Barcelogic [5], CVC3 [4], MathSat [6] OpenSMT [7], Yices [9], and Z3 [8] are examples of modern, currently-maintained, popular SMT solvers. OpenSMT and CVC3 are open-source, and CVC3, Yices, and Z3 are the only ones to support all of the defined SMT-LIB logics, including quantifiers.

CVC4 aims to follow in CVC3's footsteps as an open-source theorem prover supporting this wide array of background theories. CVC3 supports all of the background theories defined by the SMT-LIB initiative, and provides proofs and counterexamples upon request; CVC4 aims for full compliance with the new SMT-LIB version 2 command language and backward compatibility with the CVC presentation language.

In this way, CVC4 will be a drop-in replacement for CVC3, with a cleaner and more consistent library API, a more modular, flexible core, a far smaller memory footprint, and better performance characteristics.

The increased performance of CVC4's (over CVC3's) expression subsystem was demonstrated in section 2; CVC4's solving apparatus also performs better than CVC3's. In SMT-COMP 2010 [2], both solvers competed in the QF_LRA division. CVC4 solved more than twice the benchmarks CVC3 did, and for the benchmarks they both solved, CVC4 was almost always faster.

Our goal in CVC4 has been to provide a better-performing implementation of CVC3's feature set, while focusing on flexibility so that it can function as a research vehicle for years to come. Our first goal has been realized for the features that CVC4 currently supports, and we believe this success will continue as we complete support for CVC3's rich set of features. We have been successful in our second as well: a number of internal, complicated, non-intuitive assumptions on which CVC3 rests have been removed in the CVC4 redesign. We have been able to simplify greatly the component interactions and the data structures used in CVC4, making it far easier to document the internals, incorporate new developers, and add support for new features.

References

1. Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker Category. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
2. Barrett, C., Deters, M., Oliveras, A., Stump, A.: SMT-COMP 2010: the 2010 edition of the satisfiability modulo theories competition, `http://www.smtcomp.org/2010/`

3. Barrett, C., Dill, D., Levitt, J.: Validity checking for combinations of theories with equality, pp. 187–201. Springer, Heidelberg (1996)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008)
6. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
7. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
8. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Dutertre, B., de Moura, L.: The YICES SMT solver, `http://yices.csl.sri.com/tool-paper.pdf`
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. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 333–336. Springer, Heidelberg (2004)
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures, pp. 175–188. Springer, Heidelberg (2004)
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Annual Acm Ieee Design Automation Conference, pp. 530–535. ACM, New York (2001)
14. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
15. Oe, D., Reynolds, A., Stump, A.: Fast and flexible proof checking for SMT. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT 2009, pp. 6–13. ACM, New York (2009)
16. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A Cooperating Validity Checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 500–504. Springer, Heidelberg (2002)

SLAyer: Memory Safety for Systems-Level Code

Josh Berdine, Byron Cook, and Samin Ishtiaq

Microsoft Research

Abstract. SLAyer is a program analysis tool designed to automatically prove memory safety of industrial systems code. In this paper we describe SLAyer's implementation, and its application to Windows device drivers. This paper accompanies the first release of SLAyer.

1 Introduction

This paper describes SLAyer, a program analysis tool designed to prove the absence of memory safety errors such as dangling pointer dereferences, double frees, and memory leaks. Towards this goal, SLAyer searches for invariants that form proofs in Separation Logic [8]. The algorithms implemented in SLAyer are aimed at verifying moderately sized (*e.g.* 10K-30K LOC) systems level code bases written in C. SLAyer is fully automatic and does not require annotations or hints from the user.

2 Example

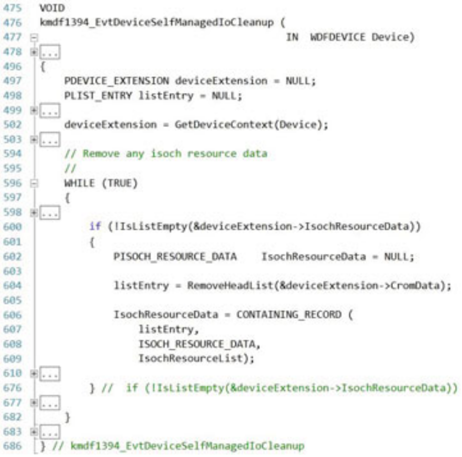
The majority of Windows faults are caused by third-party device drivers [1]. Because device drivers spend much of their time maintaining queues of requests, many of these errors are related to maintaining memory safety while operating over mutable linked data structures. Consider the code excerpt from the FireWire device driver distributed in the Windows Driver Kit (WDK) v7600 shown in Figure 1. The code is part of the cleanup routine in which allocated "isoch" resources are deleted from the IsochResourceData list. The while loop (line 596) and if test (line 600) traverse the list.

The suspicious code is on line 604 where the element is removed from the wrong list. The code then assumes, on line 606, that listEntry is pointing into the middle of an ISOCH_RESOURCE_DATA, whereas it is actually pointing into a CROM_DATA. The assignment to IsochResourceData on this line now sets it to a parent object of the wrong type. SLAyer complains that it cannot verify that the later accesses to IsochResourceData are to valid memory. This is a real bug, now fixed in the Windows 8 codebase.

3 Applying SLAyer to Device Drivers

Although the current public release of SLAyer is as a standalone tool running on vanilla C code, we have also integrated it with Static Driver Verifier (SDV) [1]. In this integration, SLAyer is called instead of the model checker SLAM. The SDV OS model used is an extension to SDV's original OS model: it is developed to be faithful both to various I/O protocols as well as to the heap.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 178–183, 2011.



```
VOID
kmdf1394_EvtDeviceSelfManagedIoCleanup (
                                        IN  WDFDEVICE Device)
...
{
    PDEVICE_EXTENSION deviceExtension = NULL;
    PLIST_ENTRY listEntry = NULL;
...
    deviceExtension = GetDeviceContext(Device);
...
    // Remove any isoch resource data
    //
    WHILE (TRUE)
    {
...
        if (!IsListEmpty(&deviceExtension->IsochResourceData))
        {
            PISOCH_RESOURCE_DATA    IsochResourceData = NULL;

            listEntry = RemoveHeadList(&deviceExtension->CromData);

            IsochResourceData = CONTAINING_RECORD (
                listEntry,
                ISOCH_RESOURCE_DATA,
                IsochResourceList);
...
        } //  if (!IsListEmpty(&deviceExtension->IsochResourceData))
...
    }
...
} // kmdf1394_EvtDeviceSelfManagedIoCleanup
```

Fig. 1. FireWire cleanup routine

The top-level user interaction is to provide the C source code of a Windows Driver Framework (WDF) device driver, and get the result at the end of the run. The result is one of: Safe (a proof the target memory safety property was found); Possibly Unsafe (a failed proof in the form of an abstract counterexample providing a path to a memory safety violation); or, Exhausted (Time/Memory constraints exceeded). Figure 2 gives the overall tool flow picture.

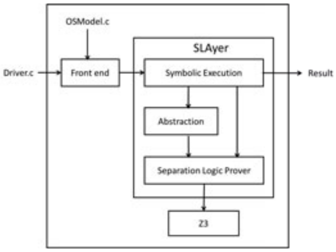


Fig. 2. SLAyer Flow

A device driver consists of a set of dispatch routines. The OS model provides a `main` function that simulates the lifetime of a driver (calls the driver's dispatch routines) under the most general assumptions; it also provides behavioral specifications of the kernel functions that the driver calls. The Microsoft Visual C compiler based frontend links the driver with the OS model to form a complete, closed, sequential system that is the input to the SLAyer analyzer.

The OS model can be viewed as the assumption under which the memory safety property is proven for the source code. Additionally, the OS model imposes the obligation that the driver maintain data structure integrity for objects passed over the Driver–OS model interface. In this sense, underlying kernel and even hardware properties can be seen to leak into the driver code.

4 SLAyer Program Analysis

SLAyer implements an analysis that attempts to prove the absence of memory safety errors. Such an error occurs whenever dereferencing a pointer to an object outside the object's dynamic lifetime. Proving this property subsumes those such as double-frees, and null or dangling pointer dereferences.

If SLAyer finds a proof, then the input program under our semantics is memory safe. There are C programs (that overrun an array or access misaligned data, for example) that are safe in our idealized semantics, but have undefined meaning according to the C standard. This situation is characteristic of automatic "sound" tools.

To elaborate, the main idealization in our semantics is the model of memory: SLAyer takes a "logical", as opposed to a "physical", view of heap memory. Memory is modeled as a collection of disjoint structured objects. The structure of each object is determined by the source-level struct definition, omitting aspects such as alignment, padding, and byte widths of scalar values. The memory model is not even close to byte-accurate, so legitimate pointer traversals that use char* as a universal type are not provable.

The focus of SLAyer is reasoning about the shape of mutable linked data structures. To this end, validity of memory is treated at a per-object granularity. In particular, arrays and structs are treated as unbounded objects where accessing any member from a valid object is valid, and any index of a valid array is valid. This is deliberately in contrast to tools aimed at catching buffer overflows such as ESP [4].

4.1 Prover

Assertion Language. The particular fragment of Separation Logic used is, like other tools such as SpaceInvader [10] and Thor [9], an extension of that introduced in Smallfoot [3].

The assertion logic does not distinguish between the various C scalar types such as pointers, integers and floats. It does distinguish offsets, which are not first-class in C and represent differences between pointers to members of structured objects, and so are expressions that can be added to pointers via field access ".", or subtracted from them via CONTAINING_RECORD. Structured values are represented using a form of records mapping offsets to scalar values similar to a first-order theory of arrays (variables, select, store), but where the domain is given by an associated C struct definition.

The pure, heap-independent, part of the logic is essentially passed through to the Z3 SMT solver [5], thereby inheriting the same generality. Atomic pure formulas in particular include (principally linear) arithmetic ($<$, $\leq$), and equality over address expressions ($p = q$, $p.\mathsf{CromData} \neq p.\mathsf{IsochResourceData}$). Pure formulas are kept in negation-normal form.

Apart from Separation Logic's emp, which describes an empty part of the heap, atomic spatial formulas are of two forms: points-to or list-segment. A points-to $l \mapsto r$ describes a single heap cell at location l that contains an object described by a record r. A list-segment $\mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n})$ describes a possibly-empty, possibly-cyclic, segment of a doubly-linked list, where the heap structure of each item of the list is given by the formula Λ. This second-order inductive predicate is used to support complex composite data structures [2].

Formulas are closed under separating conjunction $P * Q$ and disjunction $P \vee Q$, and may have a prefix of existential quantifiers $\exists \boldsymbol{x}.Q$. Unlike related tools, formulas are not restricted to disjunctive-normal form, and arbitrary nesting of

$*$ and $\vee$ is supported. While this generalization of representation does not add expressivity in principle, and is a substantial complication for the implementation, there are several motivations. One is the ability to compactly represent assertions that otherwise may blow up when expressed in DNF: for example it is common for different code branches to produce formulas whose heap structures differ in only a small region. Another is added flexibility in the design of theorem proving algorithms, where small proofs generally require (intermediate) formulas not in DNF.

Subtraction. All manipulation of formulas performed by the program analysis operations in SLAyer are, at the core, defined in terms of a judgment form called subtraction (a generalization of "frame inference" introduced in [3]), and implemented using a prover for these judgments. A subtraction judgment $M \vdash \exists \boldsymbol{x}.\, S \rightsquigarrow R$ holds if and only if the entailment $M \vdash \exists \boldsymbol{x}.\, (S * R)$ is universally valid. This is a production form, where a valid remainder R is computed as a function of the minuend M, existentials $\boldsymbol{x}$, and subtrahend S. Informally, a subtraction query $M \vdash \exists \boldsymbol{x}.\, S \rightsquigarrow$ asks the prover to re-express M, possibly weakening it, into the form $S * R$, thereby ensuring that heaps satisfying M have subheaps satisfying S, and yielding a formula R that describes the rest of each M heap.

Proof search is performed using a sequent calculus that includes deduction rules specific to the fragment's atomic formulas. A particular collection of axioms involving $\mapsto$ and ls are built into the calculus. These axioms generally require induction to prove, and by adding them to the prover it is able to prove inductive properties without searching for induction hypotheses. Additionally, these axioms, and knowledge of the semantics of the atomic forms, are used to direct aggressive use of Cut where subformulas of S are used to choose small but not atomic or quantifier-free intermediate proof goals. Searching for proofs with Cut enables localizing case analysis, resulting in much smaller proofs and search spaces, as well as more compactly represented remainders.

Reasoning about pure formulas is done using Z3, given an axiomatization of the pure fragment of the assertion logic. To enable incremental solving, during proof search SLAyer maintains a first-order approximation of the hypotheses in Z3. Leaves of proof trees are discharged by Z3, as they are implications between pure formulas. Z3 is also used to reason about equality between pointers in order to guide application of proof rules that manipulate spatial formulas. Additionally, some case splits in the sequent calculus are directed by unsatisfiable cores extracted by Z3. Overall this results in many small queries, involving the theories of arrays, data types, integer linear arithmetic, uninterpreted functions, as well as quantifier elimination for each.

The prover implemented in SLAyer is not complete, due (in part) to limited treatment of quantifiers and not attempting to find proofs using induction over the second-order list segments.

4.2 Symbolic Execution and Abstraction

Interprocedural Analysis. SLAyer uses a version of the Reps-Horowitz-Sagiv algorithm with localization [7] to perform a whole-program interprocedural analysis. The procedure specifications computed as summaries take the form

$\forall \boldsymbol{g}. \{P\} f(\boldsymbol{x}) \{Q\}$. The ghost variables $\boldsymbol{g}$ allow values on procedure entry to be compactly related to those on exit. Application of such specifications relies on subtraction's ability to reason adequately about quantified formulas. This parameterization is useful, for instance, to treat so-called heap cutpoints, as well as to express pure pre/post relations, to for instance reestablish properties of shadowed stack variables when returning from recursive procedures.

Transformers. Individual instructions, including specification statements, are symbolically executed following Smallfoot [3]. Given the generalization of subtraction to arbitrary disjunction and existential quantification, no separate "rearrangement" operation is needed in SLAyer.

Abstraction. The abstraction operation that generalizes formulas to loop invariants takes the form of a rewrite system that progressively weakens formulas, similar to SpaceInvader [6]. Instead of syntactic variable occurrence conditions, SLAyer uses rewrite rules guarded by properties involving a form of reachability through formulas modulo provable equality. Syntactic conditions proved too fragile when using a more general assertion logic. Another difference is that subtraction is used to perform the actual manipulation of the formulas, which means that the algorithms which determine how and if to rewrite do not need to know the full logical meaning of formulas.

Abstraction has also been extended to arbitrarily nested disjunction. When considering whether to apply a particular weakening of the formula, all the deeper disjuncts are considered, and distinctions where both alternatives appear are not preserved. This enables abstracting and hoisting common facts out of disjunctions, transforming formulas closer to conjunctive-normal form. So unlike the join operation of [10] which merges disjuncts of DNF formulas, the support of nested disjunction allows merging parts of formulas even when they cannot be fully merged.

5 Experimental Results and Availability

Table 1 presents some experimental results. The fw programs are extracted from the Windows FireWire driver, and are representative of device driver type code: a lot of control structures, traversal through linked lists, pointer arithmetic (the `cleanup_isochresourcedata` tests are the FireWire bug testcases). The sll

Table 1. Benchmarks

`fw/attach_buffer_insert_head_list.c`	Safe	1.8	`sll/append.c`	Safe	14.2
`fw/callback_remove_entry_list.c`	Safe	99.8	`sll/copy.c`	Safe	3.8
`fw/cleanup_isochresourcedata.c`	Safe	28.0	`sll/copy_unsafe.c`	Unsafe	0.3
`fw/cleanup_isochresourcedata_unsafe.c`	Unsafe	1.8	`sll/create.c`	Safe	0.1
`fw/cromdata_add_remove.c`	Safe	31.5	`sll/create_kernel.c`	Safe	3.8
`fw/is_on_list_flat.c`	Safe	18.2	`sll/destroy.c`	Safe	0.4
`fw/is_on_list_via_devext.c`	Safe	53.1	`sll/filter.c`	Safe	10.5
			`sll/find.c`	Safe	3.5
			`sll/reverse.c`	Safe	1.2
			`sll/traverse.c`	Safe	0.7

programs are similar but avoid using CONTAINING_RECORD to factor out pointer arithmetic. We are very conscious of the differences and prototype status of tools in this field; we present these results only as a strawman benchmark. The table gives the time, usr+sys in seconds, for SLAyer to prove each test Safe or Possibly Unsafe. The machine used was an Intel E5630 running Windows 7 64-bit.

SLAyer is available from `http://research.microsoft.com/slayer`. The download includes these benchmarks, and other C programs that stress memory safety. We plan to make a set of releases that revise the core components (analyzer performance, OS model fidelity), and to make our integration with SDV available.

References

1. 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: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, Leuven, Belgium. EuroSys 2006, pp. 73–85. ACM, New York (2006) ISBN: 1-59593-322-0, `http://doi.acm.org/10.1145/1217935.1217943`
2. 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)
3. 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)
4. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany. PLDI 2002, pp. 57–68. ACM, New York (2002) ISBN: 1-58113-463-0, `http://doi.acm.org/10.1145/512529.512538`
5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. 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)
7. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
8. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages 2001, London, United Kingdom. POPL 2001, pp. 14–26. ACM, New York (2001) ISBN:1-58113-336-7, `http://doi.acm.org/10.1145/360204.375719`
9. 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)
10. 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)

CPAchecker: A Tool for Configurable Software Verification*,**

Dirk Beyer[1,2] and M. Erkan Keremoglu[2]

[1] University of Passau, Germany
[2] Simon Fraser University, B.C., Canada

Abstract. Configurable software verification is a recent concept for expressing different program analysis and model checking approaches in one single formalism. This paper presents CPAchecker, a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, implements the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. In software verification, it takes a considerable amount of effort to convert a verification idea into actual experimental results — we aim at accelerating this process. We hope that researchers find it convenient and productive to implement new verification ideas and algorithms using this flexible and easy-to-extend platform, and that it advances the field by making it easier to perform practical experiments. The tool is implemented in Java and runs as command-line tool or as Eclipse plug-in. CPAchecker implements CPAs for several abstract domains. We evaluate the efficiency of the current version of our tool on software-verification benchmarks from the literature, and compare it with other state-of-the-art model checkers. CPAchecker is an open-source toolkit and publicly available.

1 Overview

The field of software verification is a fast growing area, and researchers contribute new ideas and approaches with enormous pace. The more new approaches are discovered, the more difficult it is to understand the essential insight or the fundamental difference that makes a new approach good and better. Experimental evaluation is often a deciding factor for whether or not a new approach is considered an advancement of the field. But it requires a considerable engineering effort to actually build the software infrastructure for evaluating verification algorithms. Adapting a suitable parser front-end and transforming the abstract syntax tree into a format that is convenient for verification algorithms is one example. The interaction with a theorem prover is yet another issue that needs to be considered. There are successful approaches in program analysis as well as in model checking, but these techniques are rarely combined; the reason is that

* This research was supported by the Canadian NSERC grant RGPIN 341819-07.

** A preliminary version of this paper appeared as Technical Report SFU-CS-2009-02 in 2009.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 184–190, 2011.

it is indeed extremely difficult to combine them. Most published approaches are not even comparable, because the choice of the parser front-end, the choice of the theorem prover, and the choice of the pointer-alias analysis algorithm in the corresponding tool implementation, considerably influence the performance and precision of the new verification algorithm. When evaluating a performance comparison of two approaches, it is often difficult to identify what the new approach contributes and what is due to the different environment. In practice, it was so far extremely difficult to perform an experimental performance evaluation of one component while keeping all other components constant.

Configurable program analysis (CPA) provides a conceptual basis for expressing different verification approaches in the same formal setting. The CPA formalism provides an interface for the definition of program analyses, which includes the abstract domain, the post operator, the merge operator, and the stop operator [4]. Consequently, the corresponding tool implementation CPAchecker provides an implementation framework that allows the seamless integration of program analyses that are expressed in the CPA framework. The comparison of different approaches in the same experimental setting becomes easy and the experimental results will be more meaningful.

Availability. The source code and all benchmark programs for CPAchecker are available online at `http://cpachecker.sosy-lab.org`. The tool is free software, released under the Apache 2.0 license.

Related Tools. In many respects, CPAchecker is similar to Blast [3]. Our predicate analysis is also based on lazy abstraction and interpolation-based refinement. The novelty of CPAchecker is that it is easy to configure. For example, the tool can run a predicate analysis using single-block encoding (SBE), like Blast [3] and Slam [1], but also using large-block encoding (LBE) [2] or even adjustable-block encoding (ABE) [5]. The advantage of the new tool over tools that implement a separated abstract-check-refine loop, like Slam [1] and SATAbs [8], is that the on-the-fly approach allows the design of more flexible, and more efficient, algorithms (like ABE [5]). We have integrated the bounded model checker CBMC [7] into CPAchecker for a bit-precise path-feasibility check.

2 Architecture and Implementation

Figure 1 shows an overview of the CPAchecker architecture. The central data structure is a set of control-flow automata (CFA), which consist of control-flow locations and control-flow edges. A location represents a program-counter value, and an edge represents a program operation, which is either an assume operation, an assignment block, a function call, or a function return (we do not consider more complex operations due to a well-known reduction called C intermediate language[1]). Before a program analysis starts, the input program is transformed into a syntax tree, and further into CFAs. The current version of CPAchecker uses the parser from the CDT, a fully functional C and C++ plug-in for the

[1] Available at `http://www.cs.berkeley.edu/~necula/cil`

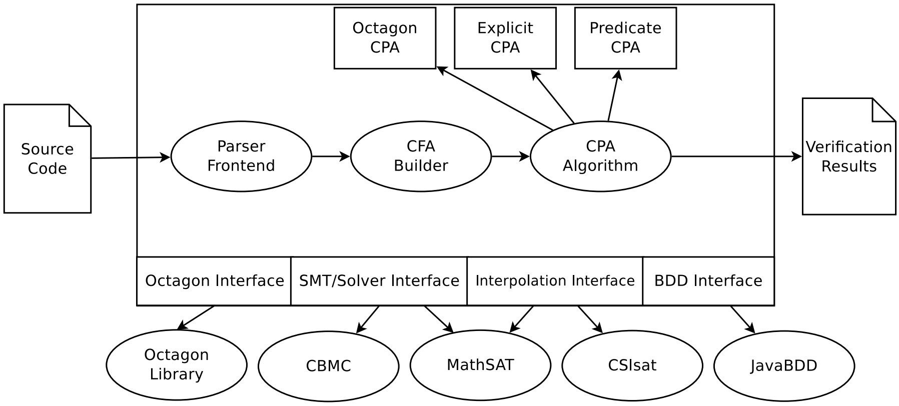


Fig. 1. CPAchecker — Architecture overview

Eclipse platform. Our framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient way. Currently we use MathSAT as an SMT solver, and CSIsat and MathSAT as interpolation procedures. We use CBMC as a bit-precise checker for the feasibility of error paths, JavaBDD as a BDD package, and provide an interface to an Octagon representation as well.[2]

The CPA algorithm is the core of CPAchecker: it performs the reachability analysis, and operates on an object of the abstract data type CPA, i.e., the algorithm applies operations from the CPA interface without knowing which concrete CPA it is analyzing [4]. For most configurations, the concrete CPA will be a composite CPA, which implements the combination of different CPAs. In order to extend CPAchecker by integrating an additional CPA for a new abstract domain, only two steps are necessary. First, an entry in the global properties file is necessary in order to announce the new CPA for composition. Second, the new CPA needs to implement the interface *CPA*, and implementations of all CPA operation interfaces need to be provided. Figure 2 shows the interaction: The CPA algorithm (shown at the top in the figure) takes as input a set of control-flow automata (CFA) representing the program, and a CPA, which is in most cases a *Composite CPA*. The interfaces correspond one-to-one to the formal framework [4]. The elements in the gray box (top right) in Fig. 2 represent the abstract interfaces of the CPA and the CPA operations. The two gray boxes at the bottom of the figure show two implementations of the interface *CPA*, one is a *CompositeCPA* that can combine several other CPAs, and the other is a *LeafCPA* (cf. the Composite design pattern). For example, suppose we want to implement a CPA for shape analysis. We would provide an implementation for *CPA*, possibly called *ShapeCPA*, and implementations for the operation interfaces that are presented in the top-right box. If we want to experiment with several different merge operators, we would provide several different implementations of *Merge*

[2] Tools available at `http://mathsat4.disi.unitn.it`, `http://www.sosy-lab.org/∼dbeyer/CSIsat`, `http://www.cprover.org/cbmc`, `http://javabdd.sourceforge.net`, `http://www.di.ens.fr/∼mine/oct`

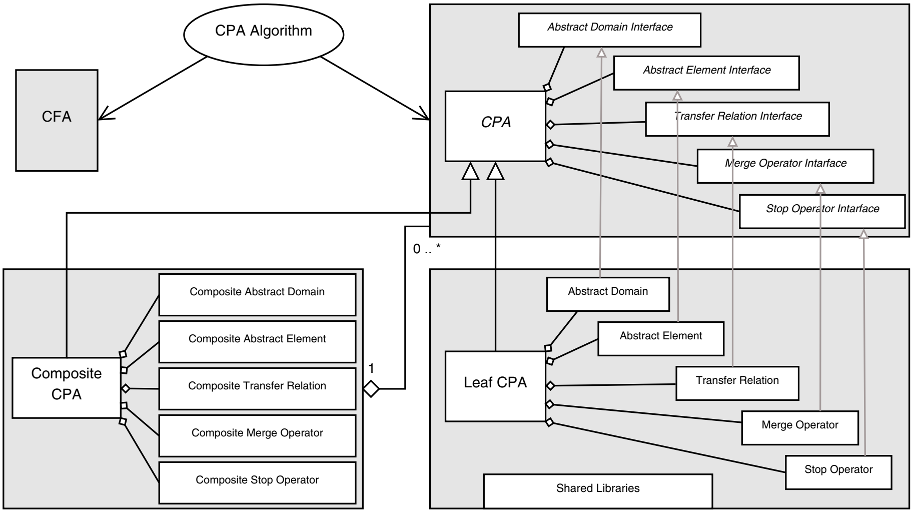


Fig. 2. CPAchecker — Design for extension

Operator Interface that can be freely configured for use in various experiments. Note that a user-defined CPA can be either a *CompositeCPA* (composing other CPAs) or a *LeafCPA* (used stand alone or as part of a *CompositeCPA*).

3 Experimental Evaluation

In this section, we compare CPAchecker with several existing tools. Our goal is to show that our CPA-based tool not only contributes a great flexibility by its configuration possibilities, but also that it can significantly improve the performance due to the possibility of constructing interesting analysis configurations.

Benchmarks. We experimented with three sets of benchmark verification problems. The first set consists of simplified, partial Windows device drivers; the second set consists of simplified versions of the state machine that handles the communication in the SSH suite. Different numbers in a program name indicate different simplifications that were applied to the source code. Both sets of benchmarks were taken from the Blast repository[3]. The third set consists of SystemC programs from the supplementary web page of SyCMC [6][4]. The string BUG in the program name indicates that the program contains a defect. All benchmarks and tools are available online at `http://www.sosy-lab.org/∼dbeyer/cpa-tool`.

Reporting. Table 1 shows the verification results for four tools. All experiments were performed on a machine with a 3.2 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.10 (64 bit), using Linux 2.6.35 as kernel and OpenJDK 1.6 as Java virtual machine. The first column reports the

[3] `http://www.sosy-lab.org/∼dbeyer/Blast`

[4] `https://es.fbk.eu/people/roveri/tests/fmcad2010`

Table 1. Performance experiments

Program	Result	CBMC		SATABS		BLAST		CPACHECKER			
	(expected)							Predicate		Explicit	
cdaudio_simpl1	SAFE	2.5	✓	–	TO	120	✓	25	✓	5.0	✓
diskperf_simpl1	SAFE	–	ER	–	TO	78	✓	21	✓	–	TO
floppy_simpl3	SAFE	.27	✓	720	✓	68	✓	12	✓	4.0	✓
floppy_simpl4	SAFE	.56	✓	–	TO	95	✓	19	✓	4.2	✓
kbfiltr_simpl1	SAFE	.09	✓	20	✓	8.3	✓	4.8	✓	2.8	✓
kbfiltr_simpl2	SAFE	.18	✓	46	✓	12	✓	7.3	✓	3.4	✓
cdaudio_simpl1_BUG	BUG	2.3	✓	–	TO	57	✓	16	✓	3.7	✓
floppy_simpl3_BUG	BUG	.28	✓	210	✓	3.2	✓	8.9	✓	3.1	✓
floppy_simpl4_BUG	BUG	.63	✓	650	✓	3.2	✓	15	✓	3.3	✓
kbfiltr_simpl2_BUG	BUG	.20	✓	100	✓	6.4	✓	5.4	✓	2.9	✓
s3_clnt_1	SAFE	–	ER	49	✓	180	✓	6.9	✓	21	✓
s3_clnt_2	SAFE	–	ER	610	✓	240	✓	9.0	✓	19	✓
s3_clnt_3	SAFE	–	ER	630	✓	–	ER	11	✓	19	✓
s3_clnt_4	SAFE	–	ER	330	✓	150	✓	11	✓	21	✓
s3_srvr_1	SAFE	–	ER	130	✓	–	ER	28	✓	–	TO
s3_srvr_2	SAFE	–	ER	170	✓	–	ER	15	✓	–	TO
s3_srvr_3	SAFE	–	ER	120	✓	–	ER	14	✓	–	TO
s3_srvr_4	SAFE	–	ER	210	✓	–	ER	13	✓	–	TO
s3_srvr_6	SAFE	–	ER	–	TO	200	✓	77	✓	–	TO
s3_srvr_7	SAFE	–	ER	–	TO	–	ER	25	✓	–	TO
s3_srvr_8	SAFE	–	ER	–	TO	85	✓	310	✓	–	TO
s3_clnt_1_BUG	BUG	7.4	✓	15	✓	9.3	✓	4.9	✓	3.2	✓
s3_clnt_2_BUG	BUG	7.7	✓	18	✓	10	✓	5.0	✓	2.6	✓
s3_clnt_3_BUG	BUG	8.9	✓	20	✓	10	✓	4.7	✓	2.7	✓
s3_clnt_4_BUG	BUG	7.9	✓	18	✓	9.1	✓	4.7	✓	2.7	✓
s3_srvr_1_BUG	BUG	12	✓	15	✓	–	ER	4.4	✓	12	✓
s3_srvr_2_BUG	BUG	10	✓	13	✓	130	✓	4.2	✓	12	✓
bist_cell	SAFE	–	ER	21	✓	430	✓	280	✓	–	ER
kundu	SAFE	–	ER	51	✓	–	TO	800	✓	–	MO
mem_slave_tlm.1	SAFE	–	TO	46	✓	–	ER	650	✓	–	ER
mem_slave_tlm.2	SAFE	–	TO	110	✓	–	ER	–	TO	–	ER
mem_slave_tlm.3	SAFE	–	TO	230	✓	–	ER	–	TO	–	ER
mem_slave_tlm.4	SAFE	–	TO	480	✓	–	ER	–	TO	–	ER
mem_slave_tlm.5	SAFE	–	TO	–	TO	–	ER	–	TO	–	ER
pc_sfifo_1	SAFE	–	ER	3.0	✓	14	✓	7.7	✓	–	TO
pc_sfifo_2	SAFE	–	ER	2.9	✓	55	✓	14	✓	–	TO
token_ring.1	SAFE	–	ER	4.2	✓	36	✓	14	✓	–	ER
token_ring.2	SAFE	–	ER	18	✓	–	ER	420	✓	–	ER
token_ring.3	SAFE	–	ER	34	✓	–	ER	–	TO	–	ER
token_ring.4	SAFE	–	TO	76	✓	–	TO	–	TO	–	ER
token_ring.5	SAFE	–	TO	200	✓	–	TO	–	TO	–	ER
token_ring.6	SAFE	–	TO	420	✓	–	TO	–	TO	–	ER
token_ring.7	SAFE	–	TO	–	TO	–	TO	–	TO	–	ER
token_ring.8	SAFE	–	TO	–	TO	–	TO	–	TO	–	ER
toy	SAFE	–	ER	10	✓	–	TO	–	TO	–	ER
kundu1_BUG	BUG	70	✓	20	✓	88	✓	11	✓	2.6	✓
kundu2_BUG	BUG	350	✓	49	✓	230	✓	57	✓	3.0	✓
toy1_BUG	BUG	380	✓	12	✓	–	TO	560	✓	3.0	✓
toy2_BUG	BUG	330	✓	8.9	✓	–	TO	270	✓	2.9	✓
transmitter_BUG.1	BUG	14	✓	3.2	✓	11	✓	3.7	✓	2.4	✓
transmitter_BUG.2	BUG	55	✓	11	✓	86	✓	8.4	✓	2.6	✓
transmitter_BUG.3	BUG	190	✓	20	✓	330	✓	40	✓	2.8	✓
transmitter_BUG.4	BUG	510	✓	62	✓	670	✓	–	TO	2.9	✓
transmitter_BUG.5	BUG	–	TO	140	✓	–	TO	–	TO	3.3	✓
transmitter_BUG.6	BUG	–	TO	340	✓	–	TO	–	TO	3.3	✓
transmitter_BUG.7	BUG	–	TO	–	TO	–	TO	–	TO	3.4	✓
transmitter_BUG.8	BUG	–	TO	–	TO	–	TO	–	TO	3.6	✓
transmitter_BUG.9	BUG	–	TO	–	TO	–	TO	–	TO	3.8	✓
transmitter_BUG.10	BUG	–	TO	–	TO	–	TO	–	TO	4.1	✓
transmitter_BUG.11	BUG	–	TO	–	TO	–	TO	–	TO	4.4	✓
transmitter_BUG.12	BUG	–	TO	–	TO	–	TO	–	TO	4.5	✓
transmitter_BUG.13	BUG	–	TO	–	TO	–	TO	–	TO	5.1	✓

program name and the second column indicates the expected verification result. The entries in the table use the following conventions: run times are given in seconds of CPU time; TO and MO indicate that the run was terminated after 900 s of run time or 12 GB of memory were consumed, respectively; ✓ indicates that the expected verification result was correctly computed; ER indicates that the checker failed to return a verification result, i.e., it gave up for some reason, or it crashed.

Tools. Column CBMC reports the results obtained using the bounded model checker CBMC 3.9. The bound was set to 10 loop iterations (`--32 --error-label "ERROR" --unwind 10`), which detects many of the bugs and proves safety for five drivers (by computing the loop bound); in general, nothing can be said about safe programs. If CBMC reports a violated loop assertion, we add `--no-unwinding-assertions` to the options and re-run the analysis, trying to identify more bugs. Column SATabs is based on SATabs 2.6 with the standard configuration (`--32 --error-label "ERROR"`); an explicit CEGAR loop is performed. Column Blast reports the results using Blast 2.5 (with MathSAT^2 as solver [2]), configured to employ lazy, interpolation-based refinement, the DFS algorithm for the state-space exploration, and the recommended predicate-search heuristic (`-craig 2 -dfs -predH 7 -nosimplemem -alias ""`). Column CPAchecker was obtained using Revision 3330 of CPAchecker, with two configuration options: on the left, we used predicate analysis where the adjustable-block encoding was configured for large blocks (`-config symbpredabsCPA-lbe.properties`) [5]; on the right, we used an explicit-value analysis that tracks explicit values for each variable, where CBMC is used to certify that an error path corresponds to a true bug by encoding the error path into a C program that is given to CBMC (`-config explicitAnalysis.properties`).

Summary. The general outcome of the evaluation is that CPAchecker's predicate analysis outperforms Blast in all but four cases. CPAchecker (predicate) outperforms SATabs on all driver and ssh programs; for the SystemC programs, SATabs is better than CPAchecker (predicate). However, for the programs with a bug, CPAchecker can be started with an explicit-value analysis and compute the result within seconds (this is especially impressive for the SystemC programs with a bug). CBMC was most successful on the driver programs. There was no false-alarm, and no tool reported a violating program as safe.

Acknowledgments. We thank G. Endler, A. Griggio, T. Henzinger, A. Holzer, S. Löwe, A. v. Rhein, M. Tautschnig, G. Théoduloz, P. Wendler, and the Blast developers for their direct and indirect contributions to the CPAchecker project.

References

1. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: POPL 2002, pp. 1–3. ACM, New York (2002)
2. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD 2009, pp. 25–32. IEEE Computer Society Press, Los Alamitos (2009)

3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
5. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD 2010, pp. 189–197 (2010)
6. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A software model checking approach. In: FMCAD 2010, pp. 51–59 (2010)
7. Clarke, E., 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)
8. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)

Existential Quantification as Incremental SAT

Jörg Brauer[1], Andy King[2,3] and Jael Kriener[3]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Portcullis Computer Security, Pinner, UK
[3] School of Computing, University of Kent, UK

Abstract. This paper presents an elegant algorithm for existential quantifier elimination using incremental SAT solving. This approach contrasts with existing techniques in that it is based solely on manipulating the SAT instance rather than requiring any reengineering of the SAT solver or needing an auxiliary data-structure such as a BDD. The algorithm combines model enumeration with the generation of shortest prime implicants so as to converge onto a quantifier-free formula presented in CNF. We apply the technique to a number of hardware circuits and transfer functions to demonstrate the effectiveness of the method.

1 Introduction

Elegant ideas and careful engineering have advanced DPLL-based SAT solvers to the point they can rapidly decide the satisfiability of structured problems that involve thousands of variables. SAT has thus been applied to the problem of existential quantifier elimination, yet existing algorithms are considered "inelegant" [9, slide 24]. This paper is concerned with computing a quantifier-free formula $\exists X : \varphi$ in CNF where X is a set of propositional variables and φ is itself presented in CNF. The quadratic nature of resolution renders it impractical when X is large compared to $\mathit{vars}(\varphi)$ and SAT-based techniques are favoured when the set of variables $Y = (\mathit{vars}(\varphi) \setminus X)$ is small compared to $\mathit{vars}(\varphi)$. These techniques apply a SAT solver to find a cube $c_1 = (\bigwedge Y_1) \wedge \neg(\bigvee Y_2)$ for which $\varphi \wedge c_1$ is satisfiable and Y_1 and Y_2 partition Y. The clause $\neg c_1$ is then added to φ and the process is repeated to enumerate all such cubes $C = \{c_1, \ldots, c_\ell\}$. During enumeration, these cubes are typically stored in a BDD which converges onto $\bigvee C$. A CNF representation of $\bigvee C$ can then be extracted from the BDD, for example, by following all paths (cubes) c which lead to 0 and then negating to obtain a clause $\neg c$. McMillan [29] critiqued this approach pointing out that:

> "CNF and SAT-based quantifier elimination can be exponentially more efficient than [..] BDDs in cases where the resulting fixed points have compact representations in CNF, but not as BDDs."

BDDs have been used to store the cubes as it is believed that they offer a space-efficient way of storing the image (quantifier-free formula). However, this does not preclude computing CNF directly, especially if the size of the CNF is smaller than that extracted from the BDD. This paper can be considered a

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 191–207, 2011.

response to the agenda set by McMillan and presents an efficient, and we believe elegant, method for computing the image as a compact CNF formula that does not require modification to a solver.

The quest for elegance is more than an exercise in aesthetics since existential quantifier elimination finds application in: unbounded model checking [29], dependency analysis [2], information flow analysis [20], transfer function synthesis [5] and the synthesis of ranking functions [14]. It also occurs in predicate abstraction [21] from which we take an example [9] that we develop in what follows. In predicate abstraction, a finite set of predicates is used to express properties of and relationships between program variables at different points in the program. State can then be described by a cube over the predicate symbols, and a set of states as a Boolean function. Quantifier elimination arises when computing successor states. Adapting an example from [9], suppose the predicates $X = \{x_1, \ldots, x_6\}$ and $Y = \{y_1, \ldots, y_6\}$ express state at two consecutive program points, and the transition relation between these states is expressed as a Boolean function:

$$\begin{aligned} \mu = \neg(x_2 \wedge y_2) \wedge \neg(y_2 \wedge y_1) \wedge ((x_4 \wedge x_6) \Rightarrow y_1) \wedge \\ (x_3 \Leftrightarrow y_4) \wedge (x_4 \Leftrightarrow y_3) \wedge \qquad (x_5 \Leftrightarrow y_6) \wedge (x_6 \Leftrightarrow y_5) \end{aligned}$$

If $\xi = (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \wedge \neg x_6) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge x_6)$ describes the state at one point, then the state at the next is given by $\exists X : (\xi \wedge \mu)$.

To summarise our work, the paper contributes an algorithm for existential quantifier elimination based solely on SAT solving. The elimination problem is reduced to that of discovering a cube of size k which entails the formula; a problem that can be completely encoded as a SAT instance. This formulation finesses the need for a complicated DPLL-like algorithm based on internal implication graphs and the application of heuristics [29, Sect. 2]. Furthermore, with a BDD-based approach, the size of resulting CNF formula is very sensitive to the variable ordering (even when dynamic reordering is applied), whereas the algorithm proposed herein actually produces a compact CNF representation, challenging the belief that BDDs are necessary for elimination.

2 Worked Example

Let φ denote a quantifier-free propositional formula and X denote a set of propositional variables. The key idea behind our approach is to converge onto the set of solutions of the formula $\exists X : \varphi$ from above by adding clauses formed from a sub-class of prime implicants of $\neg\varphi$; namely those prime implicants that contain no positive or negative occurrence of any variable of X. This approach contrasts with existing techniques in that it is based solely on manipulating the formula φ, rather than requiring any reengineering of the solver itself [29] or needing an auxiliary data-structure such as a BDD [22]. Furthermore the technique possesses the “everyone a winner” [33] enumeration property, which means that, rather than enumerating and filtering potential clauses of $\exists X : \varphi$, a new clause of $\exists X : \varphi$ is found on (virtually) each application of a SAT solver. This property is highly desirable, because it couples the computational effort required to compute the quantifier-free version of φ with its size. We build towards this technique using $\varphi = (\xi \wedge \mu)$ from the introduction and demonstrate how to eliminate quantifiers from $\exists X : \varphi$. We henceforth refer to this problem as that of projecting φ onto Y, where $Y = (\mathit{vars}(\varphi) \setminus X) = \{y_1, \ldots, y_k\}$. Intuitively this problem is that of removing all information in φ pertaining to variables other than Y.

2.1 Enumerating Implicants

The first step of our method is to enumerate the implicants of φ in the projection space. To do so, we first convert φ into CNF, for which we introduce a set of Tseitin variables T [31] which are existentially quantified. The resulting formula in CNF, which is equisatisfiable to φ, is denoted by $\exists T : \psi$. We then derive a so-called dual-rail encoding [8] by applying a transformation inspired by [28]. This amounts to introducing two disjoint sets of fresh variables $Y^+ = \{y_1^+, \ldots, y_k^+\}$ and $Y^- = \{y_1^-, \ldots, y_k^-\}$ and replacing each occurrence of the literal y_i in φ with y_i^+, and likewise each occurrence of the literal $\neg y_i$ with y_i^-. To ensure that y_i^+ and y_i^- cannot hold simultaneously, the transformed formula is augmented with the clauses $\bigwedge_{i=1}^{k}(\neg y_i^+ \vee \neg y_i^-)$. Let $\tau(\psi)$ denote this syntactic transformation, which yields a formula in CNF, defined over the variables $V = X \cup Y^+ \cup Y^- \cup T$. Passing $\tau(\psi)$ to a SAT solver yields a model $\mathbf{m}_1 : V \rightarrow \mathbb{B}$, such as for example:

$$\mathbf{m}_1 = \left\{ \begin{array}{l} x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_3 \mapsto 1,\ x_4 \mapsto 0,\ x_5 \mapsto 1,\ x_6 \mapsto 0 \\ y_1^+ \mapsto 0,\ y_2^+ \mapsto 0,\ y_3^+ \mapsto 0,\ y_4^+ \mapsto 1,\ y_5^+ \mapsto 0,\ y_6^+ \mapsto 1 \\ y_1^- \mapsto 1,\ y_2^- \mapsto 1,\ y_3^- \mapsto 1,\ y_4^- \mapsto 0,\ y_5^- \mapsto 1,\ y_6^- \mapsto 0 \end{array} \right\}$$

(Note that the Tseitin variables T have been omitted for the purpose of presentation.) The same model $\mathbf{m}_1$ can be represented as a subset of V, namely, $\{v \in V \mid \mathbf{m}_1(v) = 1\}$ and, henceforth, we shall use this representation interchangeably with $\mathbf{m}_1$. The variables in $\mathbf{m}_1 \cap (Y^+ \cup Y^-)$ then define a conjunction of literals, a cube, $\xi(\mathbf{m}_1)$ over the variables in Y, which is given as:

$$\xi(\mathbf{m}_1) = \left(\bigwedge\{y_i \mid y_i^+ \in (\mathbf{m}_1 \cap Y^+)\}\right) \wedge \left(\bigwedge\{\neg y_i \mid y_i^- \in (\mathbf{m}_1 \cap Y^-)\}\right)$$

Therefore, we have $\xi(\mathbf{m}_1) = (\neg y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$. Furthermore, the cube $\xi(\mathbf{m}_1)$ is a so-called implicant of $\exists X : \varphi$ since $\xi(\mathbf{m}_1) \models \exists X : \varphi$. It constitutes an under-approximation of $\exists X : \varphi$ since the set of all models of $\xi(\mathbf{m}_1)$ is a subset of the set of all models of $\exists X : \varphi$. To find another under-approximation, and specifically one that is not itself entailed by $\xi(\mathbf{m}_1)$, we augment $\tau(\psi)$ with the blocking clause

$$\beta(\mathbf{m}_1) = \left(\bigvee\{y_i^- \mid y_i^+ \in (\mathbf{m}_1 \cap Y^+)\}\right) \vee \left(\bigvee\{y_i^+ \mid y_i^- \in (\mathbf{m}_1 \cap Y^-)\}\right)$$

which gives $\beta(\mathbf{m}_1) = (y_1^+ \vee y_2^+ \vee y_3^+ \vee y_4^- \vee y_5^+ \vee y_6^-)$. Of course enumerating implicants in this way dovetails with the advances in incremental SAT.

Applying a solver to the augmented formula $\tau(\psi)' = \tau(\psi) \wedge \beta(\mathbf{m}_1)$ gives another model $\mathbf{m}_2$ as follows:

$$\mathbf{m}_2 = \left\{ \begin{array}{l} x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_3 \mapsto 1,\ x_4 \mapsto 0,\ x_5 \mapsto 1,\ x_6 \mapsto 0 \\ y_1^+ \mapsto 0, y_2^+ \mapsto 1, y_3^+ \mapsto 0, y_4^+ \mapsto 1, y_5^+ \mapsto 0, y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, y_2^- \mapsto 0, y_3^- \mapsto 1, y_4^- \mapsto 0, y_5^- \mapsto 1, y_6^- \mapsto 0 \end{array} \right\}$$

The model $\mathbf{m}_2$ defines another implicant $\xi(\mathbf{m}_2) = (\neg y_1 \wedge y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$ of $\exists X : \varphi$, hence $\xi(\mathbf{m}_1) \vee \xi(\mathbf{m}_2) \models \exists X : \varphi$. Repeating this strategy to derive implicants yields an unsatisfiable formula after the fourth step, and thus

$$\bigvee_{i=1}^{4} \xi(\mathbf{m}_i) = \left\{ \begin{array}{l} (\neg y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge \ \ y_4 \wedge \neg y_5 \wedge \ \ y_6) \vee \\ (\neg y_1 \wedge \ \ y_2 \wedge \neg y_3 \wedge \ \ y_4 \wedge \neg y_5 \wedge \ \ y_6) \vee \\ \ (y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge \ \ y_4 \wedge \neg y_5 \wedge \ \ y_6) \vee \\ \ (y_1 \wedge \neg y_2 \wedge \ \ y_3 \wedge \neg y_4 \wedge \ \ y_5 \wedge \neg y_6) \end{array} \right.$$

satisfies $\bigvee_{i=1}^{4} \xi(\mathbf{m}_i) = \exists X : \varphi$. However, observe that $\bigvee_{i=1}^{4} \xi(\mathbf{m}_i)$ is in DNF and also contains redundancies, e.g. $\xi(\mathbf{m}_1) \vee \xi(\mathbf{m}_2) = (\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$. In the following section, we will demonstrate the use of cardinality constraints based on sorting networks to avoid such redundancies during model enumeration.

2.2 Enumerating Shortest Implicants

Observe that the aforementioned redundancy would not have occured, had the SAT solver first found the following model:

$$\mathbf{m}_1' = \left\{ \begin{array}{l} x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_3 \mapsto 1,\ x_4 \mapsto 0,\ x_5 \mapsto 1,\ x_6 \mapsto 0 \\ y_1^+ \mapsto 0, y_2^+ \mapsto 0, y_3^+ \mapsto 0, y_4^+ \mapsto 1, y_5^+ \mapsto 0, y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, y_2^- \mapsto 0, y_3^- \mapsto 1, y_4^- \mapsto 0, y_5^- \mapsto 1, y_6^- \mapsto 0 \end{array} \right\}$$

The model $\mathbf{m}_1'$ defines an implicant $\xi(\mathbf{m}_1') = (\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$, which is entailed by both $\xi(\mathbf{m}_1)$ and $\xi(\mathbf{m}_2)$ (since, given two models $\mathbf{m}_1$ and $\mathbf{m}_2$, $\xi(\mathbf{m}_2) \models \xi(\mathbf{m}_1)$ if $\mathbf{m}_1 \cap (Y^+ \cup Y^-) \subseteq \mathbf{m}_2 \cap (Y^+ \cup Y^-)$). This suggests the possibility of searching for shortest implicants. To derive shortest implicants $\xi(\mathbf{m})$ of $\exists X : \varphi$, we turn to sorting networks [25], which are used to force the number of literals in $\xi(\mathbf{m})$ (i.e. its length) to be ℓ for increasing $\ell \in \{1, \ldots, k\}$. The value of a sorting network is that it can be applied to express the sum of k propositional variables [18] in no more than $12k(\lceil \log_2(k) \rceil + 1)$ ternary clauses where the sum is represented in unary fashion. By instantiating the outputs of a sorting network, a cardinality constraint can be obtained. For example, constraining the outputs of a 6-bit sorter to 110000 ensures that exactly two input bits are set. Such cardinality constraints can be imposed in conjunction with the formula $\tau(\psi)$ in order to force the discovery of the shortest (i.e. strongest) implicants first and thereby prevent the discovery of redundant implicants. The construction proceeds by introducing a set $Y^{\pm} = \{y_1^{\pm}, \ldots, y_k^{\pm}\}$ of fresh variables, which serve as inputs to a sorting network. Each variable $y_i^{\pm}$ indicates whether y_i^+ or y_i^- appears in the implicant, and we thus we constrain $\bigwedge_{i=1}^{k}(y_i^{\pm} \Leftrightarrow (y_i^+ \vee y_i^-))$ by introducing k clauses of the form $c_i = (\neg y_i^{\pm} \vee y_i^{+} \vee y_i^{-}) \wedge (y_i^{\pm} \vee \neg y_i^{+}) \wedge (y_i^{\pm} \vee \neg y_i^{-})$. Given a k-bit sorter σ with output variables $\{o_1, \ldots, o_k\}$, a formula whose models describe implicants of $\exists X : \varphi$ of length ℓ is obtained by augmenting $\tau(\psi)$ as follows:

$$\tau_\ell(\psi) = \tau(\psi) \wedge \sigma \wedge \left(\bigwedge_{i=1}^{k} c_i\right) \wedge \left(\bigwedge_{i=1}^{\ell} o_i\right) \wedge \left(\bigwedge_{i=\ell+1}^{k} \neg o_i\right)$$

Since $\tau_\ell(\psi)$ is unsatisfiable for $l \in \{1, \ldots, 4\}$, $\exists X : \varphi$ does not possess implicants shorter than 5. Testing $\tau_5(\psi)$ for satisfiability yields the model $\mathbf{m}'_1$ as above. Then, adding $\beta(\mathbf{m}'_1)$ to $\tau_5(\psi)$ to derive other implicants of length 5 yields an unsatisfiable formula. We thus proceed with $\tau_6(\psi) \wedge \beta(\mathbf{m}'_1)$ to give two implicants:

$$\begin{aligned} \xi(\mathbf{m}'_2) &= (y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6) \\ \xi(\mathbf{m}'_3) &= (y_1 \wedge \neg y_2 \wedge y_3 \wedge \neg y_4 \wedge y_5 \wedge \neg y_6) \end{aligned}$$

Since $\tau_6(\psi) \wedge \bigwedge_{i=1}^{3}(\beta(\mathbf{m}'_i))$ is unsatisfiable, we have $\bigvee_{i=1}^{3} \xi(\mathbf{m}'_i) = \exists X : \varphi$, and we have represented the projection in just 3 cubes. Note that although the clauses $(\bigwedge_{i=1}^{\ell} o_i) \wedge (\bigwedge_{i=\ell+1}^{k} \neg o_i)$ must be rescinded once all the implicants of length ℓ have been found, this sub-formula is itself a cube. The force of this is that SAT solvers support assumptions which are cubes. The assumption is added to the instance, thereby binding some variables, but these bindings are discarded once a model is found, in readiness for the next call to the solver. Conveniently, this lightweight version of incremental SAT is sufficient to support the above algorithm.

2.3 Over-Approximation by Dualisation

Recall that we are interested in obtaining CNF, whereas the construction we have presented so far yields formulae in DNF. Direct conversion of a formula in DNF to an equivalent one in CNF may increase the size of the formula exponentially. However, observe that since $\exists X : \varphi = \bigvee_{i=1}^{3} \xi(\mathbf{m}'_i)$, $\neg\exists X : \varphi = \neg \bigvee_{i=1}^{3} \xi(\mathbf{m}'_i) = \bigwedge_{i=1}^{3} \neg\xi(\mathbf{m}'_i)$; latter formula can be converted into CNF straightforwardly by pushing negations inward. We can thus reapply the above construction to infer implicants of $\neg\exists X : \varphi$. Given a cube ν such that $\nu \models \neg\exists X : \varphi$, the contrapositive holds, giving $\exists X : \varphi \models \neg\nu$. Therefore $\neg\nu$ over-approximates $\exists X : \varphi$, i.e. each model of $\exists X : \varphi$ is also a model of $\neg\nu$. In order to apply the above method on the dual of $\bigvee_{i=1}^{3} \xi(\mathbf{m}'_i)$, we start by negating the formula to give:

$$\neg\exists X : \varphi = \begin{cases} (y_1 \vee y_3 \vee \neg y_4 \vee y_5 \vee \neg y_6) \wedge (\neg y_1 \vee y_2 \vee y_3 \vee \neg y_4 \vee y_5 \vee \neg y_6) \wedge \\ (\neg y_1 \vee y_2 \vee \neg y_3 \vee y_4 \vee \neg y_5 \vee y_6) \end{cases}$$

Denote this formula by ω and apply τ to ω to give:

$$\tau(\omega) = \begin{cases} (y_1^- \vee y_3^- \vee y_4^+ \vee y_5^+ \vee y_6^+) \quad \wedge (y_1^+ \vee y_2^- \vee y_3^- \vee y_4^+ \vee y_5^- \vee y_6^+) \wedge \\ (y_1^+ \vee y_2^- \vee y_3^+ \vee y_4^- \vee y_5^+ \vee y_6^-) \wedge (\bigwedge_{i=1}^{6} \neg(y_i^+ \wedge y_i^-)) \end{cases}$$

We then solve $\tau_1(\omega)$, which is unsatisfiable: $\neg \bigvee_{i=1}^{3} \xi(\mathbf{m}'_i)$ does not posses implicants of length 1. Passing $\tau_2(\omega)$ to a SAT solver yields a model $\mathbf{m}''_1$ as follows:

$$\mathbf{m}''_1 = \begin{Bmatrix} y_1^+ \mapsto 0, y_2^+ \mapsto 1, y_3^+ \mapsto 0, y_4^+ \mapsto 0, y_5^+ \mapsto 0, y_6^+ \mapsto 0 \\ y_1^- \mapsto 0, y_2^- \mapsto 0, y_3^- \mapsto 0, y_4^- \mapsto 0, y_5^- \mapsto 0, y_6^- \mapsto 1 \end{Bmatrix}$$

We then extract a cube $\xi(\mathbf{m}_1'') = (y_2 \wedge \neg y_6)$ from $\mathbf{m}_1''$. From $\xi(\mathbf{m}_1'') \models \neg\exists X : \varphi$, we deduce $\exists X : \varphi \models \neg\xi(\mathbf{m}_1'')$ and $\neg\xi(\mathbf{m}_1'')$ is a clause. We add a blocking clause to suppress the cube as before and retrieve a model $\mathbf{m}_2''$ for $\tau_2(\omega) \wedge \beta(\mathbf{m}_1'')$, which induces a cube $\xi(\mathbf{m}_2'') = (y_3 \wedge y_6)$. Then $\exists X : \varphi \models \neg\xi(\mathbf{m}_1'') \wedge \neg\xi(\mathbf{m}_2'')$ and $\tau_2(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'')$ is unsatisfiable. We proceed with cubes of length 3 and solve $\tau_3(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'')$, which gives rise to a cube $\xi(\mathbf{m}_3'') = (\neg y_2 \wedge \neg y_5 \wedge \neg y_6)$. By adding blocking clauses and enumerating all cubes $\xi(\mathbf{m}_i'')$ for $i \in \{1, \ldots, m\}$, we could derive a CNF formula $\bigwedge_{i=1}^{m} \neg\xi(\mathbf{m}_i'')$ equivalent to $\exists X : \varphi$.

However, we can improve on this and produce a denser CNF representation by searching for a sub-cube of $\xi(\mathbf{m}_3'')$ which is itself an implicant of ω. To do this, let $N = (Y^+ \cup Y^-) \setminus \mathbf{m}_3'' = \{y_1^+, y_1^-, y_2^+, y_3^+, y_3^-, y_4^-, y_5^+, y_6^+\}$. We then solve $\tau_2(\omega)$ in conjunction with the cube $\bigwedge\{\neg y_i^+ \mid y_i^+ \in N \cap Y^+\} \wedge \bigwedge\{\neg y_i^- \mid y_i^- \in N \cap Y^-\}$ which we pass the solver as an assumption. The solver produces a model $\mathbf{m}_4''$ which defines $\xi(\mathbf{m}_4'') = (\neg y_5 \wedge \neg y_6)$; thus $\neg\xi(\mathbf{m}_4'') = (y_5 \vee y_6) \models \exists X : \varphi$. Since $\mathbf{m}_4'' \cap (Y^+ \cup Y^-) \subset \mathbf{m}_3'' \cap (Y^+ \cup Y^-)$, we have $\mathbf{m}_3'' \models \mathbf{m}_4''$ and $\xi(\mathbf{m}_3'') \models \xi(\mathbf{m}_4'')$. We thus discard $\xi(\mathbf{m}_3'')$ and proceed with $\tau_4(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'') \wedge \beta(\mathbf{m}_4'')$. Whenever a fresh cube is discovered, we apply the same strategy to weaken it to the most general one that still entails ω. It is interesting to note that an implicant of length ℓ can be generalised using at most $\lceil \log_2(\ell) \rceil$ calls a solver by applying dichotomic search (though we do not apply this technique because ℓ is typically small).

Repeatedly applying this generalise scheme we derive the following minimal (though not unique) CNF representation of $\exists X : \varphi$ in five more iterations:

$$\exists X : \varphi = \begin{cases} (\neg y_2 \vee y_6) \wedge (\neg y_3 \vee \neg y_6) \wedge (y_5 \vee y_6) & \wedge\ (y_3 \vee \neg y_5) \wedge \\ (y_4 \vee \neg y_6) \wedge (y_1 \vee y_6) \quad\ \wedge (\neg y_1 \vee \neg y_2) & \wedge\ (\neg y_4 \vee y_6) \end{cases}$$

Since the search is exhaustive, this is no longer an over-approximation of the projection, but equivalent to it. Our implementation of this algorithm using MiniSat takes 0.0012s and 0.0009s for the first and second stages of the algorithm (corresponding to Sections 2.2 and 2.3 respectively) thus taking 0.0021s overall.

2.4 Reprise and Reflection

One may wonder why not to enumerate prime implicants of $\neg\varphi$ directly as previously proposed [6]. To find an over-approximation $\neg\nu$ of $\exists X : \varphi$, put $\neg\varphi$ into CNF using a formula κ such that $\exists T : \kappa \equiv \neg\varphi$. Further, observe that $\nu \models \forall X : \exists T : \kappa$ iff $\neg\forall X : \exists T : \kappa \models \neg\nu$ iff $\exists X : \neg\exists T : \kappa \models \neg\nu$ iff $\exists X : \varphi \models \neg\nu$. Hence, to find an over-approximation of $\exists X : \varphi$, it suffices to find an implicant of $\forall X : \exists T : \kappa$. Since $\forall X : \exists T : \kappa \models \exists X : \exists T : \kappa$, each implicant of $\forall X : \exists T : \kappa$ is also an implicant of $\exists X : \exists T : \kappa$. This suggests enumerating each implicant ν of $\exists X : \exists T : \kappa$ and discarding those ν which are not implicants of $\forall X : \exists T : \kappa$, that is, those that fail the entailment check $\varphi \models \neg\nu$. However, this is hardly an "everyone a winner" strategy as this method produces very large numbers of spurious implicants that fail the entailment check. By combining model enumeration with the generation of prime implicants on the dual formula, our new method does not generate any spurious candidates, which explains performance improvements of several orders of magnitude (see Sect. 4).

3 Formal Correctness

Let Bool_V denote the class of propositional Boolean formulae over the set of variables V, which is partitioned into two disjoint subsets X and Y, i.e. $V = X \cup Y$ and $X \cap Y = \emptyset$. We shall consider the problem of computing an implicant of $\exists X : \varphi$, where the formula $\varphi \in \mathsf{Bool}_V$ is in CNF. The transformation is formalised as a map τ on the set of literals $\mathsf{Lit}_V = \{v, \neg v \mid v \in V\}$ over V. This map is, in turn, defined in terms of propositional variables $Y^+ = \{y^+ \mid y \in Y\}$ and $Y^- = \{y^- \mid y \in Y\}$ where $Y^+ \cap Y^- = \emptyset$ and $(Y^+ \cup Y^-) \cap V = \emptyset$.

Definition 1. The literal transformation map $\tau : \mathsf{Lit}_V \to \mathsf{Lit}_{X \cup Y^+ \cup Y^-}$ and its inverse $\tau^{-1} : \mathsf{Lit}_{X \cup Y^+ \cup Y^-} \to \mathsf{Lit}_V$ are defined as follows:

$$\tau(l) = \begin{cases} y^+ & \text{if } l = y \text{ and } y \in Y \\ y^- & \text{if } l = \neg y \text{ and } y \in Y \\ l & \text{otherwise} \end{cases} \qquad \tau^{-1}(l) = \begin{cases} y & \text{if } l = y^+ \\ \neg y & \text{if } l = y^- \\ l & \text{otherwise} \end{cases}$$

To lift τ to clauses, a clause is considered to be merely a set of literals. Then $\tau(C) = \{\tau(l) \mid l \in C\}$ for a clause $C \subseteq \mathsf{Lit}_V$. The literal transformation map τ is lifted to cubes and implicants (which is a particular type of cube) by likewise considering these to be sets of conjoined literals. The transformation relates cubes with literals drawn from Lit_V to cubes with literals drawn from $\mathsf{Lit}_{X \cup Y^+ \cup Y^-}$. We then define non-trivial cubes (which do not contain opposing literals) as below:

Definition 2

$$\mathsf{Cube}_V = \left\{ C \subseteq \mathsf{Lit}_V \,\middle|\, \forall v \in V : \{v, \neg v\} \not\subseteq C \ \right\}$$
$$\mathsf{Cube}_{X,Y} = \left\{ C \cup C' \,\middle|\, C \in \mathsf{Cube}_X \wedge C' \subseteq Y^+ \cup Y^- \wedge \forall y \in Y : \{y^+, y^-\} \not\subseteq C' \ \right\}$$

Note that a formula φ represented in CNF can be considered to be a set of implicitly conjoined clauses F. This is used to state the following equivalence result which asserts that implicants are preserved by the transformation τ:

Proposition 1 (Equivalence). Let $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ where $F \subseteq 2^{\mathsf{Lit}_V}$ and put $\varphi' = \bigwedge\{\bigvee \tau(C) \mid C \in F\}$. Then

- If $D \in \mathsf{Cube}_V$ and $(\bigwedge D) \models \varphi$ then $(\bigwedge \tau(D)) \models \varphi'$.
- If $D' \in \mathsf{Cube}_{X,Y}$ and $(\bigwedge D') \models \varphi'$ then $(\bigwedge \tau^{-1}(D')) \models \varphi$.

The following corollary of the above relates implicants with literals drawn from Lit_Y to the satisfiability of the transformed clause set:

Corollary 1. Suppose φ and φ' are defined as above. Then

- If $D \in \mathsf{Cube}_Y$ and $\bigwedge D \models \exists X : \varphi$ then $(\bigwedge \tau(D)) \wedge \varphi'$ is satisfiable.
- If $D' \in \mathsf{Cube}_{Y,\emptyset}$ and $(\bigwedge D') \wedge \varphi'$ is satisfiable then $(\bigwedge \tau^{-1}(D')) \models \exists X : \varphi$.

To state how to compute an image by enumerating implicants, the unusual notion of a blocking clause introduced in Sect. 2 is now formalised:

Definition 3. The mapping $\beta : \mathsf{Cube}_{Y,\emptyset} \rightarrow \mathsf{Cube}_{Y,\emptyset}$ is defined:

$$\beta(D') = \{y_i^- \mid y_i^+ \in D'\} \cup \{y_i^+ \mid y_i^- \in D'\}$$

Theorem 1 (correctness). Suppose φ and φ' are defined as above. Let $D'_1, \ldots, D'_\ell \in \mathsf{Cube}_{Y,\emptyset}$ be a sequence such that:

- $(\bigwedge_{l \in D'_k} l) \wedge \varphi' \wedge (\bigwedge_{i=1}^{k-1} (\bigvee_{l \in \beta(D'_i)} l))$ is satisfiable for all $k \in \{1, \ldots, \ell\}$ and
- $\varphi' \wedge (\bigwedge_{i=1}^{\ell} (\bigvee_{l \in \beta(D'_i)} l))$ is unsatisfiable.

Then $\bigvee_{i=1}^{\ell} \wedge \tau^{-1}(D'_i) = \exists X : \varphi$.

The following proposition dovetails with the theorem to show how a CNF representation of the projection can be derived in a two phase process. The corollary that follows is immediate and states that the computation of implicants, in the second phase at least, can be aborted prematurely without sacrificing correctness.

Proposition 2 (dualisation). Let $\psi = \bigvee_{i=1}^{\ell} (\bigwedge_{d \in D_i} d)$ where $D_1, \ldots, D_\ell \in \mathsf{Cube}_Y$. Further, let $\exists X : \varphi = \bigvee_{i=1}^{m} (\bigwedge_{e \in E_i} e)$ where $E_1, \ldots, E_m \in \mathsf{Cube}_Y$ and $\varphi = \bigwedge_{i=1}^{\ell} (\bigvee_{l \in D_i} \neg l)$. Then $\psi = \bigwedge_{i=1}^{m} (\bigvee_{l \in E_i} \neg l)$.

Corollary 2 (anytime). Let $\psi = \bigvee_{i=1}^{\ell} (\bigwedge_{d \in D_i} d)$ where $D_1, \ldots, D_\ell \in \mathsf{Cube}_Y$. Let $\bigwedge_{i=1}^{m} E_i \models \exists X : \varphi$ where $E_1, \ldots, E_m \in \mathsf{Cube}_Y$ and $\varphi = \bigwedge_{i=1}^{\ell} (\bigvee_{l \in D_i} \neg l)$. Then $\psi \models \bigwedge_{i=1}^{m} (\bigvee_{l \in E_i} \neg l)$.

The above results are presented in terms of any implicants, rather than prime implicants only. This is because, while the latter govern the rate of convergence, they do not affect correctness. Nevertheless, a prime implicant of an existentially quantified formula can be formulated as two satisfability conditions. To state the corollary, let $[\![\varphi]\!] \subseteq 2^V$ denote the set of models of the Boolean function φ. For example, if $V = \{x, y\}$ then $[\![x \vee y]\!] = \{\{x\}, \{y\}, \{x, y\}\}$.

Corollary 3. Suppose φ, φ' and $F \subseteq 2^{\mathsf{Lit}_V}$ are defined as above and put $\psi = \varphi' \wedge (\bigwedge_{y \in Y} (\neg y^+ \vee \neg y^-))$. Then $D \in \mathsf{Cube}_Y$ is a prime implicant of $\exists X : \varphi$ iff $D = \tau^{-1}(M^\star \cap (Y^+ \cup Y^-))$ where $M^\star \in [\![\psi]\!]$ and $|M^\star \cap (Y^+ \cup Y^-)| \leq |M \cap (Y^+ \cup Y^-)|$ for all $M \in [\![\psi]\!]$.

Note that ψ does not include any cardinality constraint on the set $M^\star \cap (Y^+ \cup Y^-)$, hence the need to define a prime implicant in terms of an implicant no longer than any other. The above result can straightforwardly be adapted to specify how an implicant of a given size can be defined as a SAT instance.

To conclude the elaborations on correctness, we observe that the greedy generation of prime implicants does not necessarily yield a minimal CNF formula. To see this, suppose $\varphi = (\neg w \wedge x \wedge y) \vee (\neg x \wedge \neg y \wedge \neg z)$ and consider $\exists X : \varphi$ where $X = \{x\}$. Clearly $\exists X : \varphi = D_1 \vee D_2$ where $D_1 = (\neg w \wedge y)$ and $D_2 = (\neg y \wedge \neg z)$. But also $\exists X : \varphi = E_1 \vee E_2 \vee E_3$ where $E_1 = (\neg w \wedge \neg z)$, $E_2 = (\neg w \wedge y \wedge z)$ and $E_3 = (w \wedge \neg y \wedge \neg z)$. Observe $|D_1| \leq |D_2|$ and likewise $|E_1| \leq |E_2| \leq |E_3|$, and indeed either CNF formulae can be generated, though the latter is sub-optimal.

4 Experiments

We have implemented the techniques described in this paper in C++ using MiniSat with the express aim of answering the following questions:

- What is the overhead of using primes compared to standard enumeration?
- How are the primes distributed in terms of size within the two phases of the algorithm, i.e. for DNF generation and CNF conversion?
- How does the method compare against BDD-based projection scheme, both in terms of the size of CNF formulae and the time required to produce them?

To answer these questions, we compared our technique against a hybrid SAT/BDD approach. We implemented our method on top of MiniSat v2.2. Cudd v2.4.2 was used for the BDD package since it offers direct support for enumerating the prime implicants of a BDD. We chose bitonic sorting for the sorting network, though smaller (albeit less regular) networks exist [25]. All experiments were performed on a 2.6 GHz MacBook Pro equipped with 4 GB of RAM.

4.1 Benchmarks

As benchmarks, we selected several circuits from the 74X and ISCAS-89 benchmark series as well as projection problems arising from range analysis of microcontroller code [3,5]. The 74X circuits include an ALU (74181), a carry-lookahead generator (74182), an adder (74283) and a magnitude comparator (74L85). The ALU is the hardest to analyse since it implements 16 different functions, depending on 4 control bits. The ISCAS-89 benchmarks consist of a traffic light controller (s298), two implementations of a 4 × 4 add-shift multiplier (s344 and s349), and a combinatorial circuit with randomly inserted flip-flops (s1196). All circuits were projected onto their input and output variables so as to express their semantics without reference to any intermediate variables.

The microcontroller code was exported from [mc]square [36] for the purpose of synthesising transfer functions [5] for propagating ranges across blocks of

Table 1. Information regarding the benchmark set; column φ contains the name of the formula as referred to later on, followed by information about the origin of the respective formula and its size; the benchmarks at the bottom are generated from blocks of ATmega16 binary code; for these benchmarks, column *info* contains the number of instructions and whether they were generated for set abstraction (*set*) or transfer function synthesis (*tf*).

φ	info	$\|V\|$	$\|\varphi\|$
74181	74x series	1001	2368
74182	74x series	227	526
74283	74x series	267	646
74L85	74x series	413	1084
add	3 (set)	74	119
increment	3 (set)	66	119
parity_mit	15 (set)	2066	6725
parity_swap	21 (set)	275	745
randerson	13 (set)	18658	61696
triple_swap	9 (set)	89	192

φ	info	$\|V\|$	$\|\varphi\|$
s298	ISCAS-89	1327	3164
s344	ISCAS-89	1665	3880
s349	ISCAS-89	1678	3914
s1196	ISCAS-89	5422	12870
adc	4 (tf)	19	290
admdswpcmp	11 (tf)	66	154
adsb2shad	8 (tf)	114	322
ilsh	5 (tf)	66	170
irsh	5 (tf)	66	170
iswp	8 (tf)	130	386

Atmel ATmega16 code. Transfer function synthesis is essentially an existential quantifier problem We also considered projection problems that arise when over-approximating the set of values that a register can take in a block (when a block is considered in isolation to those blocks that flow into it [3]). Table 1 presents the key statistics for each of these projection problems.

4.2 Projecting Using Prime Implicants

Table 2 presents the results for DNF generation (resp. CNF conversion) using prime implicants, giving the number of implicants (resp. clauses) in the resulting formulae and the time required to compute them. Analogous figures are given for the hybrid approach. It is interesting to see that for the circuits s344 and s349, only 512 implicants in DNF are generated, but exhaustive model enumeration yields 65792 disjuncts. This is because 256 out of 512 implicants are of length 12, and thus already cover a large number of models in the projection space. This suggests that our method can make model enumeration tractable where the classical approach fails. For other cases, as exemplified by the 74181 and s1196 circuits, our approach offers no clear advantage. However, it is important to see that transformation never seriously degrades performance; this is noteworthy because one cannot know the distribution of the primes up front.

The percental distribution of the lengths of clauses that arise in CNF conversion are depicted in Fig. 1. For reasons of space, graphs are given only for the 74X series (though these distributions are typical). For DNF generation the distributions are less interesting for these benchmarks, often consisting of a single spike, but sometimes consist of two spikes, as for s344 and s349 at lengths 12 and 20. It is in these latter cases that primes improve over classic model enumeration.

Table 2. Experimental results for projection using prime implicant enumeration and comparison to BDD-based method; the best results are emphasized

φ	$\|Y\|$	Primes					model		Hybrid		
		DNF		CNF		total	enum		BDD		total
		size	time	size	time	time	size	time	size	time	time
74181	22	16384	1.477	686	7.096	8.574	16384	1.421	**476**	**1.320**	2.798
74182	13	320	0.025	26	**0.009**	0.035	320	0.009	**23**	0.014	0.039
74283	14	512	0.022	**98**	0.147	0.169	512	0.023	270	**0.077**	0.099
74L85	14	2048	0.108	**144**	0.107	0.215	2048	0.092	145	**0.053**	0.162
s298	9	4	0.001	7	**0.003**	0.004	4	0.004	7	0.006	0.007
s344	20	512	0.068	16	**0.018**	0.087	65792	12.811	16	0.030	0.098
s349	20	512	0.070	16	**0.017**	0.088	65792	12.001	16	0.029	0.099
s1196	28	16384	11.182	**570**	**5.465**	16.653	16384	11.374	822	5.810	16.993
adder	16	256	0.007	16	**0.012**	0.020	1024	0.025	16	0.024	0.031
	24	1024	0.030	31	**0.054**	0.086	4096	0.117	**29**	0.090	0.120
increment	8	4	0.001	10	**0.001**	0.003	4	0.001	10	0.006	0.007
	16	256	0.004	14	**0.007**	0.012	256	0.003	14	0.010	0.014
	24	256	0.008	**32**	**0.024**	0.033	256	0.004	34	0.027	0.035
parity_mit	8	100	0.033	4	**0.001**	0.036	200	0.005	4	0.006	0.039
	16	12800	2.363	16	1.361	3.727	25600	5.227	**10**	**0.284**	2.647
	24	40960	8.543	**40**	6.316	14.875	51200	11.348	41	**1.155**	9.698
parity_swap	8	16	0.002	4	**0.001**	0.004	16	0.001	4	0.007	0.009
	16	256	0.008	12	**0.008**	0.017	256	0.010	12	0.011	0.019
	24	256	0.013	**37**	**0.038**	0.051	256	0.011	40	0.101	0.114
randerson	8	64	0.102	2	**0.001**	0.104	64	0.051	2	0.006	0.108
	16	256	0.136	14	**0.010**	0.147	256	0.089	14	0.013	0.149
	24	256	0.140	**27**	**0.023**	0.164	256	0.092	30	0.058	0.198
triple_swap	8	16	0.002	12	**0.001**	0.004	64	0.005	12	0.007	0.009
	16	512	0.013	**20**	0.029	0.042	512	0.009	22	**0.015**	0.028
adc	8	128	0.004	7	**0.002**	0.006	512	0.010	7	0.006	0.010
	16	128	0.005	**47**	**0.018**	0.023	512	0.012	52	0.036	0.041
	24	128	0.006	**80**	**0.047**	0.054	512	0.012	92	0.116	0.122
admdswpcmp	8	191	0.003	7	**0.003**	0.006	191	0.002	7	0.008	0.011
	16	191	0.007	**54**	**0.021**	0.029	191	0.005	60	0.049	0.057
	24	191	0.009	**56**	**0.025**	0.035	191	0.006	66	0.071	0.080
adsb2shad	16	154	0.008	**67**	**0.026**	0.034	512	0.012	71	0.089	0.097
	24	310	0.013	**124**	**0.045**	0.058	1024	0.021	129	0.095	0.108
ilsh	8	32	0.002	3	**0.001**	0.003	32	0.002	3	0.008	0.010
	16	256	0.008	13	**0.009**	0.017	256	0.005	13	0.011	0.019
	24	256	0.009	**44**	**0.023**	0.032	256	0.006	46	0.035	0.046
irsh	8	16	0.001	4	**0.001**	0.001	64	0.003	4	0.007	0.008
	16	16	0.002	22	**0.004**	0.006	64	0.003	**21**	0.009	0.011
	24	16	0.003	45	**0.012**	0.016	64	0.003	45	0.017	0.020
iswp	16	4096	0.103	16	0.235	0.339	4096	0.061	16	**0.075**	0.179
	24	4096	0.126	27	0.251	0.379	4096	0.073	27	**0.140**	0.266

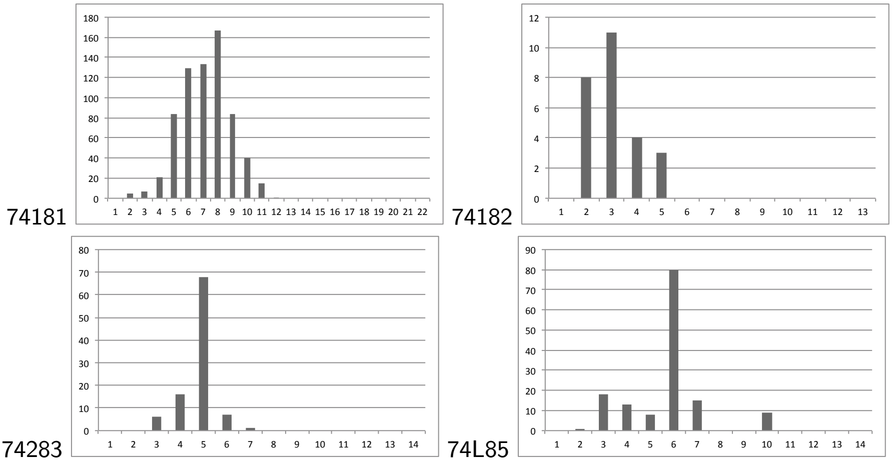


Fig. 1. Distribution of implicants by length for the 74X benchmarks

4.3 Projecting Using BDDs

In the hybrid SAT/BDD based approach, DNF to CNF conversion is realised with a BDD. To support this, Cudd provides a dedicated operation that computes prime implicants of a given BDD by finding a shortest path from the root to 1 leaves (though the lengths of the implicants and their number depend on the variable ordering). In terms of size of the resulting CNF formula, it is interesting to see that BDDs do not necessarily give the smallest representation; far from it.

In terms of running times, there is no clear winner: for the largest problem, s1196, the SAT-based approach is faster for CNF conversion whereas the BDD-based method is superior for 74181. However, we suspect that the balance may well shift towards SAT if solvers continue to advance in performance. Furthermore, the implementation of the SAT-based scheme required less than 100 lines of code which itself makes it attractive.

4.4 Generalising Implicants

During the development of the method described in this paper, we found that generalisation (as described in Sect. 2.3) had to be applied in tandem with search for a shortest implicant for the formula that is accompanied with the blocking clauses (as described in Sect. 2.2). Enumerating implicants without generalisation yields a much larger number of clauses; for the 74283 benchmark, 932 instead of 98. Strangely the runtimes are almost equal, which is the typical pattern. We conclude that generalisation is advisable since, though it does not improve the runtime, it does improve the density of the resulting CNF formula.

5 Related Work

The complexity of the shortest implicant problem for DNF formulae has been studied by Umans [38] who showed that it is $GC(\log_2(n), coNP)$-complete. Even though this result is not directly transferable to CNF, it suggests the parallel problem in CNF may be similarly difficult and thereby supports the application of SAT solvers to the derivation of shortest implicants.

5.1 Consensus Method and Resolution

The consensus method has been proposed [4,32,35] as a way of enumerating all the prime implicants of a propositional function in DNF. If f is in CNF, then it is straightforward to derive a DNF representation of $\neg f$, to which the consensus procedure can be applied to find its prime implicants. One might think that this provides a way to compute projection, but the key step of the consensus method combines two elementary conjunctions of $\neg f$, say, $x \wedge C$ and $(\neg x) \wedge D$, to form $C \wedge D$, which is isomorphic to resolution. Hence the consensus method shares the inefficiency problems associated with applying resolution to a formula in CNF.

5.2 Hybrid Methods and McMillan's Method

SAT has been used before to compute projections [26,29] as have BDDs [10,26,39]. Hybrid approaches that combine SAT solving and BDDs typically represent state sets as BDDs and express the transition relation in CNF [22,37], though some approaches combine BDDs and SAT solving in different ways. For example, Damiano and Kukula [16] substitute clauses with BDDs in a DPLL solver, Jin and Somenzi [23] combine BDDs and SAT solving using CNF to avoid explosion in the sizes of the resulting BDDs, whereas Aloul et al. [1] study the connection between CNF formulae and BDDs for good variable orderings. The approach of Cavada et al. [11] recursively computes quantifications for subtrees, which are then combined; SMT solving ensures consistency of the transformations.

McMillan [29] has shown how to perform universal projection for CTL modalities such as $\mathsf{AX}\varphi$ using DPLL-like enumeration and also explained how to represent an arbitrary Boolean encoding of φ in CNF without existential quantification. The key idea of his `toCNF(`φ`)` procedure [29, Sect. 3] is to deduce a clause from a satisfying assignment of φ whose complement rules out some cases that violate φ. His approach requires a modified DPLL-engine and resolution coupled with several heuristics — which literals to analyse, which variables to resolve on and suchlike — which strongly affect the performance of the approach [29, Sect. 2]. Our approach, in comparison, builds on top of an existing SAT library and is therefore both straightforward to implement and will immediately benefit from any improvement to the library itself. Nevertheless, we consider the SAT-based algorithm of McMillan to be an important work that has indeed found application in the predicate abstraction of hardware circuits [13] and post-image computation [12]. A variation on the McMillan algorithm is given by Sheng and Hsiao [37] who apply a success-driven rather than a conflict-driven search for models (recall that DPLL-style algorithms use a conflict-driven search). However, Sheng and Hsiao store their results in a BDD rather than generating a CNF formula.

5.3 Methods Based on Integer Linear Programming

Integer linear programming has been used to find shortest implications, as have SAT engines which have been modified to support inequalities [28]. In this work a transformation is described which is similar to τ. However, the work is not concerned with quantifier elimination, hence 0-1 variables are introduced for each variable in the formula rather than merely those in the projection space.

5.4 Methods Based on Primes and Cubes

Prime implicants have been directly applied to widening Boolean functions represented as ROBDDs [24]. By applying a recursive meta-product construction [15], collections of short primes can be used to derive an ROBDD that is an over-approximation of the input. Our work on applying SAT to projection was motivated by the empirical finding that collections of short primes often yield good approximations of Boolean formulae [24, Sect. 5.1].

Lahiri et al. [26] have described how to enumerate cubes in the projection space using SAT so as to perform image computation for predicate abstraction. Blocking clauses are chosen heuristically, though details of the heuristics are not given, and the approach does not guarantee to infer cubes of minimal size. This work was further developed by Lahiri et al. [27] who used DPLL(T)-based SMT solving to enumerate models. Each model is then stored in a BDD from which the results are extracted as disjunctions of prime implicants. They search for cubes $c_{1,k}, \ldots, c_{n,k}$ of increasing length k such that $\varphi \models \bigvee_{i=1}^{n} c_{i,k}$, which chimes with our approach. In contrast, however, we apply prime implicants in two different ways, that is, for enumerating cubes as well as clauses, so that our final quantifier-free formula is presented in CNF. To illustrate the conceptual difference between the methods, consider the benchmarks s344 and s349 from Sect. 4, for which DNF enumeration yields 256 cubes of lengths 12 and 22. The method of Lahiri et al. enumerates all intermediate cubes of lengths $13, \ldots, 21$ to converge onto $\exists X : \varphi$, whereas our approach leapfrogs these intermediate cubes by specifying the requirement of a cube of size k within the SAT instance itself. The MathSat SMT solver [7] uses an algorithm that also relies on a formula transformation similar to τ. However, rather than adding cardinality constraints to the SAT instance, they modified the solver so that it takes 0 decisions during SAT solving. Earlier approaches [17,19,34] to predicate abstraction invoke a solver for each cube c to discover if $\varphi \wedge c$ is satisfiable. To reduce the number of calls to the decision procedure, they start with small cubes, and only if $\varphi \wedge c$ is satisfiable, they proceed with cubes of the form $\varphi \wedge c \wedge d$ and $\varphi \wedge c \wedge \neg d$. This approach is based on a large number of SAT/SMT calls, typically requires many unsatisfiability proofs (which are often more difficult for SAT solvers to provide than find a model), and does not fit as well with incremental SAT [27, Sect. 3.2]. More recently, Monniaux [30] described a method for quantifier elimination called lazy model enumeration. The key idea of his algorithm is to derive a cube that implies a given formula, which is then generalised towards a weaker implicant. By way of comparison, our algorithm starts with implicants as short as possible; weakening is required by the encoding. Although similar in spirit, his algorithm proceeds diametrically opposed to ours, and moreover generates DNF.

6 Concluding Discussion

This paper advocates using SAT to eliminate existential quantifiers from formulae presented in CNF. The method is based on a two-phase approach: first, SAT solving is applied to enumerate the prime implicants of a quantified formula; second, cubes are translated into clauses to derive a CNF representation of the projection. The second phase is anytime in that it can be stopped early without compromising soundness. This can be considered a pragmatic response to the complexity of DNF to CNF conversion. As well as exploiting advances in incremental SAT and finessing the need to modify a solver, it provides an efficient way of storing projections without BDDs, whilst avoiding the blow-up in the number of intermediate clauses that comes with applying resolution.

Acknowledgements. The first author was supported, in part, by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication, German Research Foundation grant DFG EXC 89. The second author was funded, in part, by a Royal Society travel grant, reference TG092357, and a Royal Society Industrial Fellowship, reference IF081178. We thank Edd Barrett, Armin Biere, Stefan Kowalewski, Axel Simon and the anonymous referees for helpful comments.

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Faster SAT and Smaller BDDs via Common Function Structure. In: ICCAD, pp. 443–448 (2001)
2. Armstrong, T., Marriott, K., Schachte, P., Søndergaard, H.: Two Classes of Boolean Functions for Dependency Analysis. Sci. Comp. Program. 31(1), 3–45 (1998)
3. Barrett, E., King, A.: Range and Set Abstraction Using SAT. Electronic Notes in Theoretical Computer Science 267(1), 17–27 (2010)
4. Blake, A.: Canonical expressions in Boolean algebra. University of Chicago, Chicago (1938)
5. Brauer, J., King, A.: Automatic abstraction for intervals using boolean formulae. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 167–183. Springer, Heidelberg (2010)
6. Brauer, J., King, A.: Approximate quantifier elimination for propositional boolean formulae. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 73–88. Springer, Heidelberg (2011)

7. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
8. Bryant, R.: Boolean analysis of MOS circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6(4), 634–649 (1987)
9. Bryant, R.E.: A view from the engine room: Computational support for symbolic model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 145–149. Springer, Heidelberg (2008), http://www.slidefinder.net/m/mc25/7464626
10. Burch, J.R., Clarke, E.M., McMillan, K.L.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98, 142–170 (1992)
11. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing predicate abstractions by integrating BDDs and SMT solvers. In: FMCAD, pp. 69–76. IEEE Computer Society Press, Los Alamitos (2007)
12. Chauhan, P., Clarke, E.M., Kroening, D.: A SAT-based algorithm for reparameterization in symbolic simulation. In: DAC, pp. 524–529. ACM Press, New York (2004)
13. Clarke, E., Talupur, M., Veith, H., Wang, D.: SAT based predicate abstraction for hardware verification. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 78–92. Springer, Heidelberg (2004)
14. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
15. Coudert, O., Madre, J.C.: Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In: DAC, pp. 36–39. IEEE Computer Society Press, Los Alamitos (1992)
16. Damiano, R.F., Kukula, J.H.: Checking satisfiability of a conjunction of BDDs. In: DAC, pp. 818–823. ACM Press, New York (2003)
17. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
18. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. JSAT 2(1-4), 1–26 (2006)
19. Flanagan, C., Qadeer, S.: Predicate Abstraction for Software Verification. In: POPL, pp. 191–202. ACM Press, New York (2002)
20. Genaim, S., Giacobazzi, R., Mastroeni, I.: Modeling Secure Information Flow with Boolean Functions. In: IFIP WG 1.7, ACM Workshop on Issues in the Theory of Security, Barcelona, Spain, pp. 55–66 (2004)
21. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
22. Gupta, A., Yang, Z.-J., Ashar, P., Gupta, A.: SAT-based image computation with application in reachability analysis. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 354–371. Springer, Heidelberg (2000)
23. Jin, H., Somenzi, F.: CirCUs: A hybrid satisfiability solver. In: Holger Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 211–223. Springer, Heidelberg (2005)
24. Kettle, N., King, A., Strzemecki, T.: Widening rOBDDs with prime implicants. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 105–119. Springer, Heidelberg (2006)

25. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, London (1997)
26. 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)
27. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
28. Manquinho, V.M., Flores, P.F., Silva, J.P.M., Oliveira, A.L.: Prime Implicant Computation Using Satisfiability Algorithms. In: International Conference on Tools with Artificial Intelligence, pp. 232–239. IEEE Computer Society Press, Los Alamitos (1997)
29. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
30. Monniaux, D.: Quantifier elimination by lazy model enumeration. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 585–599. Springer, Heidelberg (2010)
31. Plaisted, D., Greenbaum, S.: A Structure-Preserving Clause Form Translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
32. Quine, W.V.: A way to simplify truth functions. American Mathematical Monthly 62(9), 627–631 (1955)
33. Read, R.C.: Everyone a Winner. Annals of Discrete Mathematics 2, 107–120 (1978)
34. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
35. Samson, E.W., Mills, B.E.: Circuit minimization: Algebra and algorithms for new Boolean canonical expressions. Technical Report TR, pp. 21–54. United States Air Force, Cambridge Research Lab (1954)
36. Schlich, B.: Model checking of software for microcontrollers. ACM Trans. Embedded Comput. Syst 9(4), article Number 36 (2010)
37. Sheng, S., Hsiao, M.S.: Efficient Preimage Computation Using A Novel Success-Driven ATPG. In: DATE, pp. 10822–10827. IEEE Computer Society Press, Los Alamitos (2003)
38. Umans, C.: The Minimum Equivalent DNF Problem and Shortest Implicants. In: FOCS, pp. 556–563. IEEE Computer Society Press, Los Alamitos (1998)
39. Weaver, S., Franco, J.V., Schlipf, J.S.: Extending Existential Quantification in Conjunctions of BDDs. JSAT 1(2), 89–110 (2006)

Efficient Analysis of Probabilistic Programs with an Unbounded Counter

Tomáš Brázdil[1,⋆], Stefan Kiefer[2,⋆⋆], and Antonín Kučera[1,⋆]

[1] Faculty of Informatics, Masaryk University, Czech Republic
{brazdil,kucera}@fi.muni.cz
[2] Department of Computer Science, University of Oxford, United Kingdom.
stefan.kiefer@cs.ox.ac.uk

Abstract. We show that a subclass of infinite-state probabilistic programs that can be modeled by probabilistic one-counter automata (pOC) admits an efficient quantitative analysis. In particular, we show that the expected termination time can be approximated up to an arbitrarily small relative error with polynomially many arithmetic operations, and the same holds for the probability of all runs that satisfy a given ω-regular property. Further, our results establish a powerful link between pOC and martingale theory, which leads to fundamental observations about quantitative properties of runs in pOC. In particular, we provide a "divergence gap theorem", which bounds a positive non-termination probability in pOC away from zero.

1 Introduction

In this paper we aim at designing *efficient* algorithms for analyzing basic properties of probabilistic programs operating on unbounded data domains that can be abstracted into a non-negative integer counter. Consider, e.g., the following recursive program *TreeEval* which evaluates a given AND-OR tree, i.e., a tree whose root is an AND node, all descendants of AND nodes are either leaves or OR nodes, and all descendants of OR nodes are either leaves or AND nodes.

```
procedure AND(node)
if node is a leaf
   then return node.value
else
   for each successor s of node do
      if OR(s) = 0 then return 0
   end for
   return 1
end if
```

```
procedure OR(node)
if node is a leaf
   then return node.value
else
   for each successor s of node do
      if AND(s) = 1 then return 1
   end for
   return 0
end if
```

[⋆] Tomáš Brázdil and Antonín Kučera are supported by the research center Institute for Theoretical Computer Science (ITI, project No. 1M0545) and by the Czech Science Foundation, grant No. P202/10/1469.

[⋆⋆] Stefan Kiefer is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 208–224, 2011.

Note that the program *TreeEval* evaluates a subtree only when necessary. In general, we cannot say anything about its expected termination time. If the input tree is infinite, the program may not even terminate, i.e., it may fail to evaluate the root node. Now assume that we *do* have some knowledge about the actual input domain of the program, which might have been gathered empirically:

– an AND node has about a descendants on average;
– an OR node has about o descendants on average;
– the length of a branch is b on average;
– the probability that a leaf evaluates to 1 is z.

Further, let us assume that the actual number of descendants and the actual length of a branch are *geometrically* distributed (which is a reasonably good approximation in many cases). Hence, the probability that an AND node has *exactly* n descendants is $(1 - x_a)^{n-1}x_a$ with $x_a = \frac{1}{a}$. Under these assumption, the behaviour of *TreeEval* is well-defined in the probabilistic sense, and we may ask the following questions:

1) Does the program terminate with probability one? If not, what is the termination probability?
2) If we restrict ourselves to terminating runs, what is the expected termination time?

These questions are not trivial, and at first glance it is not clear how to approach them. Apart of the expected termination time, which is a fundamental characteristic of terminating runs, we are also interested in the properties on *non-terminating* runs, specified by linear-time logics or automata on infinite words. Here, we ask for the probability of all runs satisfying a given linear-time property. Using the results of this paper, answers to such questions can be computed *efficiently* for a large class of programs, including the program *TreeEval*. More precisely, the first question about the probability of termination can be answered using the existing results [14]; the original contributions of this paper are efficient algorithms for computing answers to the remaining questions.

The abstract class of probabilistic programs considered in this paper corresponds to *probabilistic one-counter automata (pOC)*. Informally, a pOC has finitely many control states $p, q, \ldots$ that can store global data, and a single non-negative counter that can be incremented, decremented, and tested for zero. The dynamics of a given pOC is described by finite sets of *positive* and *zero* rules of the form $p \xrightarrow{x,c}_{>0} q$ and $p \xrightarrow{x,c}_{=0} q$, respectively, where p, q are control states, x is the *probability* of the rule, and $c \in \{-1, 0, 1\}$ is the *counter change* which must be non-negative in zero rules. A *configuration* $p(i)$ is given by the current control state p and the current counter value i. If i is positive/zero, then positive/zero rules can be applied to $p(i)$ in the natural way. Thus, every pOC determines an infinite-state Markov chain where states are the configurations and transitions are determined by the rules. As an example, consider a pOC model of the program *TreeEval*. We use the counter to abstract the stack of activation records. Since the procedures AND and OR alternate regularly in the stack, we keep just the current stack height in the counter, and maintain the "type" of the current procedure in the finite control (when we increase or decrease the counter, the "type" is swapped). The return values of the two procedures are also stored in the finite control. Thus, we obtain the following pOC model with 6 control states and 12 positive rules (zero rules are irrelevant and hence not shown).

/* if we have a leaf, return 0 or 1 */

$(\mathit{and,init}) \xrightarrow{yz,-1} (\mathit{or,return,1})$,

$(\mathit{and,init}) \xrightarrow{y(1-z),-1} (\mathit{or,return,0})$

/* otherwise, call OR */

$(\mathit{and,init}) \xrightarrow{(1-y),1} (\mathit{or,\ init})$

/* if OR returns 1, call another OR? */

$(\mathit{and,return,1}) \xrightarrow{(1-x_a),1} (\mathit{or,init})$

$(\mathit{and,return,1}) \xrightarrow{x_a,-1} (\mathit{or,return,1})$

/* if OR returns 0, return 0 immediately */

$(\mathit{and,return,0}) \xrightarrow{1,-1} (\mathit{or,return,0})$

/* if we have a leaf, return 0 or 1 */

$(\mathit{or,init}) \xrightarrow{yz,-1} (\mathit{and,return,1})$,

$(\mathit{or,init}) \xrightarrow{y(1-z),-1} (\mathit{and,return,0})$

/* otherwise, call AND */

$(\mathit{or,init}) \xrightarrow{(1-y),1} (\mathit{and,init})$

/* if AND returns 0, call another AND? */

$(\mathit{or,return,0}) \xrightarrow{(1-x_o),1} (\mathit{and,\ init})$

$(\mathit{or,return,0}) \xrightarrow{x_o,-1} (\mathit{and,return,0})$

/* if AND returns 1, return 1 immediately */

$(\mathit{or,return,1}) \xrightarrow{1,-1} (\mathit{and,return,1})$

The initial configuration is $(\mathit{and,init})(1)$, and the pOC terminates either in $(\mathit{or,return,0})(0)$ or $(\mathit{or,return,1})(0)$, which corresponds to evaluating the input tree to 0 and 1, respectively. We set $x_a := 1/a$, $x_o := 1/o$ and $y := 1/b$ in order to obtain the average numbers a, o, b from the beginning.

As we already indicated, pOC can model recursive programs operating on unbounded data structures such as trees, queues, or lists, assuming that the structure can be faithfully abstracted into a counter. Let us note that modeling general recursive programs requires more powerful formalisms such as *probabilistic pushdown automata (pPDA)* [12] or *recursive Markov chains (RMC)* [17]. However, as it is mentioned below, pPDA and RMC do not admit *efficient* quantitative analysis for fundamental reasons. Hence, we must inevitably sacrifice a part of pPDA modeling power to gain efficiency in algorithmic analysis, and pOC seem to be a good candidate.

The relevance of pOC is not limited just to recursive programs. As observed in [14], pOC are equivalent, in a well-defined sense, to discrete-time *Quasi-Birth-Death processes (QBDs)*, a well-established stochastic model that has been deeply studied since late 60s. Thus, the applicability of pOC extends to queuing theory, performance evaluation, etc., where QBDs are considered as a fundamental formalism. Very recently, games over (probabilistic) one-counter automata, also called "energy games", were considered in several independent works [9,10,4,3]. The study is motivated by optimizing the use of resources (such as energy) in modern computational devices.

Previous Work. In [12,17], it has been shown that the vector of termination probabilities in pPDA and RMC is the least solution of an effectively constructible system of quadratic equations. The termination probabilities may take irrational values, but can be effectively approximated up to an arbitrarily small absolute error $\varepsilon > 0$ in polynomial space by employing the decision procedure for the existential fragment of Tarski algebra (i.e., first order theory of the reals) [8]. Due to the results of [17], it is possible to approximate termination probabilities in pPDA and RMC "iteratively" by using the decomposed Newton's method. However, this approach may need exponentially many iterations of the method before it starts to produce one bit of precision per iteration [19]. Further, any non-trivial approximation of the non-termination probabilities is at least as hard as the SquareRootSum problem [17], whose exact complexity is a long-standing open question in exact numerical computations (the best known upper bound for SquareRootSum is PSPACE). Computing termination probabilities in pPDA and RMC up to a given *relative* error $\varepsilon > 0$, which is more relevant from the point of view of this paper, is *provably* infeasible because the termination probabilities can be doubly-exponentially small in the size of a given pPDA or RMC [17].

The expected termination time and the expected reward per transition in pPDA and RMC has been studied in [13]. In particular, it has been shown that the tuple of expected termination times is the least solution of an effectively constructible system of linear equations, where the (products of) termination probabilities are used as coefficients. Hence, the equational system can be represented only symbolically, and the corresponding approximation algorithm again employs the decision procedure for Tarski algebra. There also other results for pPDA and RMC, which concern model-checking problems for linear-time [15,16] and branching-time [7] logics, long-run average properties [5], discounted properties of runs [2], etc.

Our Contribution. In this paper, we build on the previously established results for pPDA and RMC, and on the recent results of [14] where is shown that the decomposed Newton method of [19] can be used to compute termination probabilities in pOC up to a given *relative* error $\varepsilon > 0$ in time which is *polynomial* in the size of pOC and $\log(1/\varepsilon)$, assuming the unit-cost rational arithmetic RAM (i.e., Blum-Shub-Smale) model of computation. Adopting the same model, we show the following:

1. The expected termination time in a pOC $\mathscr{A}$ is computable up to an arbitrarily small relative error $\varepsilon > 0$ in time polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$. Actually, we can even compute the expected termination time up to an arbitrarily small *absolute* error, which is a better estimate because the expected termination time is always at least 1.
2. The probability of all runs in a pOC $\mathscr{A}$ satisfying an ω-regular property encoded by a deterministic Rabin automaton $\mathcal{R}$ is computable up to an arbitrarily small relative error $\varepsilon > 0$ in time polynomial in $|\mathscr{A}|$, $|\mathcal{R}|$, and $\log(1/\varepsilon)$.

The crucial step towards obtaining these results is the construction of a suitable *martingale* for a given pOC, which allows to apply powerful results of martingale theory (such as the optional stopping theorem or Azuma's inequality, see, e.g., [20,21]) to the quantitative analysis of pOC. In particular, we use this martingale to establish the crucial *divergence gap theorem* in Section 4, which bounds a positive divergence probability in pOC away from 0. The divergence gap theorem is indispensable in analysing properties of non-terminating runs, and together with the constructed martingale provide generic tools for designing efficient approximation algorithms for other interesting quantitative properties of pOC.

Although our algorithms have polynomial worst-case complexity, the obtained bounds look complicated and it is not immediately clear whether the algorithms are practically usable. Therefore, we created a simple experimental implementation which computes the expected termination time for pOC, and used this tool to analyse the pOC model of the program *TreeEval*. The details are given in Section 5.

Due to space limits, we could not include most of the proofs into the main body of the paper. These can be found in a full version of this paper [6].

2 Definitions

We use $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Q}$, and $\mathbb{R}$ to denote the set of all integers, positive integers, non-negative integers, rational numbers, and real numbers, respectively. Let $\delta > 0$, $x \in \mathbb{Q}$, and $y \in \mathbb{R}$. We say that x approximates y up to a relative error δ, if either $y \neq 0$ and $|x - y|/|y| \leq \delta$, or $x = y = 0$. Further, we say that x approximates y up to an absolute error δ if $|x - y| \leq \delta$. We use standard notation for intervals, e.g., $(0, 1]$ denotes $\{x \in \mathbb{R} \mid 0 < x \leq 1\}$.

Given a finite set Q, we regard elements of $\mathbb{R}^Q$ as vectors over Q. We use boldface symbols like $\boldsymbol{u}, \boldsymbol{v}$ for vectors. In particular we write $\mathbf{1}$ for the vector whose entries are all 1. Similarly, elements of $\mathbb{R}^{Q \times Q}$ are regarded as square matrices.

Let $\mathcal{V} = (V, \rightarrow)$, where V is a non-empty set of vertices and $\rightarrow \subseteq V \times V$ a *total* relation (i.e., for every $v \in V$ there is some $u \in V$ such that $v \rightarrow u$). The reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$. A *finite path* in $\mathcal{V}$ of *length* $k \geq 0$ is a finite sequence of vertices $v_0, \ldots, v_k$, where $v_i \rightarrow v_{i+1}$ for all $0 \leq i < k$. The length of a finite path w is denoted by $\mathit{length}(w)$. A *run* in $\mathcal{V}$ is an infinite sequence w of vertices such that every finite prefix of w is a finite path in $\mathcal{V}$. The individual vertices of w are denoted by $w(0), w(1), \ldots$ The sets of all finite paths and all runs in $\mathcal{V}$ are denoted by $\mathit{FPath}_{\mathcal{V}}$ and $\mathit{Run}_{\mathcal{V}}$, respectively. The sets of all finite paths and all runs in $\mathcal{V}$ that start with a given finite path w are denoted by $\mathit{FPath}_{\mathcal{V}}(w)$ and $\mathit{Run}_{\mathcal{V}}(w)$, respectively. Let $U \subseteq V$. We say that U is *strongly connected* if $v \rightarrow^+ u$ for all $v, u \in U$ (here $v \rightarrow^+ u$ if there is a path of length greater than 1 from v to u). Further, we say that U is a *strongly connected component (SCC)* if $U \neq \emptyset$ is a maximal strongly connected subset of V, and U is a *bottom SCC (BSCC)* if for every $u \in U$ and every $u \rightarrow v$ we have that $v \in U$.

We assume familiarity with basic notions of probability theory, e.g., *probability space*, *random variable*, or the *expected value*. As usual, a *probability distribution* over a finite or countably infinite set X is a function $f : X \rightarrow [0, 1]$ such that $\sum_{x \in X} f(x) = 1$. We call f *positive* if $f(x) > 0$ for every $x \in X$, and *rational* if $f(x) \in \mathbb{Q}$ for every $x \in X$.

Definition 1. *A* Markov chain *is a triple* $\mathcal{M} = (S, \rightarrow, \mathit{Prob})$ *where* S *is a finite or countably infinite set of* states, $\rightarrow \subseteq S \times S$ *is a total* transition relation, *and Prob is a function that assigns to each state* $s \in S$ *a positive probability distribution over the outgoing transitions of* s*. As usual, we write* $s \xrightarrow{x} t$ *when* $s \rightarrow t$ *and* x *is the probability of* $s \rightarrow t$*.*

A Markov chain $\mathcal{M}$ can be also represented by its *transition matrix* $M \in [0, 1]^{S \times S}$, where $M_{s,t} = 0$ if $s \not\rightarrow t$, and $M_{s,t} = x$ if $s \xrightarrow{x} t$.

To every $s \in S$ we associate the probability space $(\mathit{Run}_{\mathcal{M}}(s), \mathcal{F}, \mathcal{P})$ of runs starting at s, where $\mathcal{F}$ is the σ-field generated by all *basic cylinders*, $\mathit{Run}_{\mathcal{M}}(w)$, where w is a finite path starting at s, and $\mathcal{P} : \mathcal{F} \rightarrow [0, 1]$ is the unique probability measure such that $\mathcal{P}(\mathit{Run}_{\mathcal{M}}(w)) = \prod_{i=1}^{\mathit{length}(w)} x_i$ where $w(i-1) \xrightarrow{x_i} w(i)$ for every $1 \leq i \leq \mathit{length}(w)$. If $\mathit{length}(w) = 0$, we put $\mathcal{P}(\mathit{Run}_{\mathcal{M}}(w)) = 1$.

Definition 2. *A* probabilistic one-counter automaton (pOC) *is a tuple,* $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$*, where*

- Q *is a finite set of* states*,*
- $\delta^{>0} \subseteq Q \times \{-1, 0, 1\} \times Q$ *and* $\delta^{=0} \subseteq Q \times \{0, 1\} \times Q$ *are the sets of* positive *and* zero rules *such that each* $p \in Q$ *has an outgoing positive rule and an outgoing zero rule;*
- $P^{>0}$ *and* $P^{=0}$ *are* probability assignments*, assigning to each* $p \in Q$ *a positive rational probability distribution over the outgoing rules in* $\delta^{>0}$ *and* $\delta^{=0}$*, resp., of* p*.*

In the following, we often write $p \xrightarrow{x,c}_{=0} q$ to denote that $(p, c, q) \in \delta^{=0}$ and $P^{=0}(p, c, q) = x$, and similarly $p \xrightarrow{x,c}_{>0} q$ to denote that $(p, c, q) \in \delta^{>0}$ and $P^{>0}(p, c, q) = x$. The size of $\mathscr{A}$, denoted by $|\mathscr{A}|$, is the length of the string which represents $\mathscr{A}$, where the probabilities of rules are written in binary. A *configuration* of $\mathscr{A}$ is an element of $Q \times \mathbb{N}_0$, written as $p(i)$. To $\mathscr{A}$ we associate an infinite-state Markov chain $\mathcal{M}_{\mathscr{A}}$ whose states are the configurations of $\mathscr{A}$, and for all $p, q \in Q$, $i \in \mathbb{N}$, and $c \in \mathbb{N}_0$ we have that $p(0) \xrightarrow{x} q(c)$ iff $p \xrightarrow{x,c}_{=0} q$, and $p(i) \xrightarrow{x} q(c)$ iff $p \xrightarrow{x,c-i}_{>0} q$. For all $p, q \in Q$, let

- $Run_{\mathscr{A}}(p{\downarrow}q)$ be the set of all runs in $\mathcal{M}_{\mathscr{A}}$ initiated in $p(1)$ that visit $q(0)$ and the counter stays positive in all configurations preceding this visit;
- $Run_{\mathscr{A}}(p{\uparrow})$ be the set of all runs in $\mathcal{M}_{\mathscr{A}}$ initiated in $p(1)$ where the counter never reaches zero.

We omit the "$\mathscr{A}$" in $Run_{\mathscr{A}}(p{\downarrow}q)$ and $Run_{\mathscr{A}}(p{\uparrow})$ when it is clear from the context, and we use $[p{\downarrow}q]$ and $[p{\uparrow}]$ to denote the probability of $Run(p{\downarrow}q)$ and $Run(p{\uparrow})$, respectively. Observe that $[p{\uparrow}] = 1 - \sum_{q \in Q}[p{\downarrow}q]$ for every $p \in Q$.

At various places in this paper we rely on the following proposition proven in [14] (recall that we adopt the unit-cost rational arithmetic RAM model of computation):

Proposition 3. *Let $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$ be a pOC, and $p, q \in Q$.*

- *The problem whether $[p{\downarrow}q] > 0$ is decidable in polynomial time.*
- *If $[p{\downarrow}q] > 0$, then $[p{\downarrow}q] \geq x_{\min}^{|Q|^3}$, where $x_{\min}$ is the least (positive) probability used in the rules of $\mathscr{A}$.*
- *The probability $[p{\downarrow}q]$ can be approximated up to an arbitrarily small relative error $\varepsilon > 0$ in a time polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$.*

Due to Proposition 3, the set $T^{>0}$ of all pairs $(p, q) \in Q \times Q$ satisfying $[p{\downarrow}q] > 0$ is computable in polynomial time.

3 Expected Termination Time

In this section we give an efficient algorithm which approximates the expected termination time in pOC up to an arbitrarily small relative (or even absolute) error $\varepsilon > 0$.

For the rest of this section, we fix a pOC $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$. For all $p, q \in Q$, let $R_{p{\downarrow}q} : Run(p(1)) \to \mathbb{N}_0$ be a random variable which to a given run w assigns either the least k such that $w(k) = q(0)$, or 0 if there is no such k. If $(p, q) \in T^{>0}$, we use $E(p{\downarrow}q)$ to denote the conditional expectation $\mathbb{E}[R_{p{\downarrow}q} \mid Run(p{\downarrow}q)]$. Note that $E(p{\downarrow}q)$ can be finite even if $[p{\downarrow}q] < 1$.

The first problem we have to deal with is that the expectation $E(p{\downarrow}q)$ can be infinite, as illustrated by the following example.

Example 4. Consider a simple pOC with only one control state p and two positive rules $(p, -1, p)$ and $(p, 1, p)$ that are both assigned the probability $1/2$. Then $[p{\downarrow}p] = 1$, and due to results of [13], $E(p{\downarrow}p)$ is the least solution (in $\mathbb{R}^+ \cup \{\infty\}$) of the equation $x = 1/2 + 1/2(1 + 2x)$, which is ∞.

We proceed as follows. First, we show that the problem whether $E(p{\downarrow}q) = \infty$ is decidable in polynomial time (Section 3.1). Then, we eliminate all infinite expectations, and show how to approximate the finite values of the remaining $E(p{\downarrow}q)$ up to a given absolute (and hence also relative) error $\varepsilon > 0$ efficiently (Section 3.2).

3.1 Finiteness of The Expected Termination Time

In this subsection we exhibit conditions that, given $(p,q) \in T^{>0}$, allow to decide in polynomial time whether $E(p{\downarrow}q)$ is finite. To state these conditions, we need some notions. Define sets $Pre^*(q(0))$ and $Post^*(p(1))$, where

- $Pre^*(q(0))$ consists of all $r(k)$ that can reach $q(0)$ along a run w in $\mathcal{M}_{\mathscr{A}}$ such that the counter stays positive in all configurations preceding the visit to $q(0)$;
- $Post^*(p(1))$ consists of all $r(k)$ that can be reached from $p(1)$ along a run w in $\mathcal{M}_{\mathscr{A}}$ where the counter stays positive in all configurations preceding the visit to $r(k)$.

Note that $q(0) \in Pre^*(q(0))$ and $p(1) \in Post^*(p(1))$. Further, define a finite-state Markov chain $\mathcal{X}$ with Q as set of states, and transition matrix $A \in [0,1]^{Q\times Q}$ given by $A_{p,q} = \sum_{(p,c,q)\in\delta^{>0}} P^{>0}(p,c,q)$. Given a BSCC $\mathscr{B}$ of $\mathcal{X}$, let $\boldsymbol{\alpha} \in (0,1]^{\mathscr{B}}$ be the *invariant distribution* of $\mathscr{B}$, i.e., the unique (row) vector satisfying $\boldsymbol{\alpha}A = \boldsymbol{\alpha}$ and $\boldsymbol{\alpha}\mathbf{1} = 1$ (see, e.g., [18, Theorem 5.1.2]). Further, we define the (column) vector $\boldsymbol{s} \in \mathbb{R}^{\mathscr{B}}$ of *expected counter changes* by $\boldsymbol{s}_p = \sum_{(p,c,q)\in\delta^{>0}} P^{>0}(p,c,q)\cdot c$ and the *trend* $t \in \mathbb{R}$ of $\mathscr{B}$ by $t = \boldsymbol{\alpha}\boldsymbol{s}$. Intuitively, the trend is the average counter increase per step. Note that t is easily computable in polynomial time. Now we can state the following theorem:

Theorem 5. *Let $(p,q) \in T^{>0}$. Let $x_{\min}$ denote the smallest nonzero probability in A. Then we have:*

(A) if q is not in a BSCC of $\mathcal{X}$, then $E(p{\downarrow}q) \le 5|Q| \,/\, x_{\min}^{|Q|+|Q|^3}$;
(B) if q is in a BSCC $\mathscr{B}$ of $\mathcal{X}$, then:
  (a) if $Pre^*(q(0)) \cap Post^*(p(1)) \cap \mathscr{B}\times\mathbb{N}$ is a finite set, then $E(p{\downarrow}q) \le 20|Q|^3/x_{\min}^{4|Q|^3}$;
  (b) if $Pre^*(q(0)) \cap Post^*(p(1)) \cap \mathscr{B}\times\mathbb{N}$ is an infinite set, then:
    (1) if $\mathscr{B}$ has trend $t \neq 0$, then $E(p{\downarrow}q) \le 85000|Q|^6/(x_{\min}^{5|Q|+|Q|^3}\cdot t^4)$;
    (2) if $\mathscr{B}$ has trend $t = 0$, then $E(p{\downarrow}q)$ is infinite.

One can check in polynomial time which case of Theorem 5 applies. In particular, due to [11], there are finite-state automata constructible in polynomial time recognizing the sets $Pre^*(q(0))$ and $Post^*(p(1))$. Hence, we can efficiently compute a finite-state automaton $\mathcal{F}$ recognizing the set $Pre^*(q(0)) \cap Post^*(p(1)) \cap \mathscr{B}\times\mathbb{N}$ and check whether the language accepted by $\mathcal{F}$ is finite. Thus we have the following corollary:

Corollary 6. *Let $(p,q) \in T^{>0}$. The problem whether $E(p{\downarrow}q)$ is finite is decidable in polynomial time.*

In the rest of this subsection we sketch a qualitative proof for Theorem 5; i.e., we sketch why $E(p{\downarrow}q)$ is infinite only in case (B.b.2). First assume case (A), i.e., q is not in a BSCC of $\mathcal{X}$. Then for all $s(\ell) \in Post^*(p(1))$, where $\ell \ge |Q|$, we have that $s(\ell)$ can reach a configuration outside $Pre^*(q(0))$ in at most $|Q|$ transitions. It follows that the probability of performing a path from $p(1)$ to $q(0)$ of length i decays exponentially in i, and hence $E(p{\downarrow}q)$ is finite.

Next assume case (B.a), i.e., $\mathscr{B}$ is a BSCC and $C := Pre^*(q(0)) \cap Post^*(p(1)) \cap \mathscr{B}\times\mathbb{N}$ is a finite set. It is easy to show that the expected time for a run in $Run(p{\downarrow}q)$ to reach $\mathscr{B}$ is finite. Once the run has reached $\mathscr{B}$ it basically moves within a Markov chain on C.

By assumption, C is finite (which implies, by a pumping argument, that $|C| \leq 3|Q|^3$). Consequently, after the run has reached $\mathscr{B}$, it reaches $q(0)$ in finite expected time.

Case (B.b) requires new non-trivial techniques. For the sake of simplicity, *from now on we assume that $Q = \mathscr{B}$* (the general case requires only slight modifications of the arguments presented below). We employ a generic observation which connects the study of pOC to martingale theory. Recall that a stochastic process $m^{(0)}, m^{(1)}, \ldots$ is a martingale if, for all $i \in \mathbb{N}$, $\mathbb{E}(|m^{(i)}|) < \infty$, and $\mathbb{E}(m^{(i+1)} \mid m^{(1)}, \ldots, m^{(i)}) = m^{(i)}$ almost surely. Let us fix an initial configuration $r(c) \in Q \times \mathbb{N}$. Our aim is to construct a suitable martingale over $Run(r(c))$. Let $p^{(i)}$ and $c^{(i)}$ be random variables which to every run $w \in Run(r(c))$ assign the control state and the counter value of the configuration $w(i)$, respectively. Note that if the vector $\boldsymbol{s}$ of expected counter changes is constant, i.e., $\boldsymbol{s} = \mathbf{1} \cdot t$ where t is the trend of $\mathcal{X}$, then we can define a martingale $m^{(0)}, m^{(1)}, \ldots$ simply by

$$m^{(i)} = \begin{cases} c^{(i)} \ - \ i \cdot t & \text{if } c^{(j)} \geq 1 \text{ for all } 0 \leq j < i; \\ m^{(i-1)} & \text{otherwise.} \end{cases}$$

Since $\boldsymbol{s}$ is generally not constant, we might try to "compensate" the difference among the individual control states by a suitable vector $\boldsymbol{v} \in \mathbb{R}^Q$. The next proposition shows that this is indeed possible.

Proposition 7. *There is a vector $\boldsymbol{v} \in \mathbb{R}^Q$ such that the stochastic process $m^{(0)}, m^{(1)}, \ldots$ defined by*

$$m^{(i)} = \begin{cases} c^{(i)} \ + \ \boldsymbol{v}_{p^{(i)}} \ - \ i \cdot t & \textit{if } c^{(j)} \geq 1 \textit{ for all } 0 \leq j < i; \\ m^{(i-1)} & \textit{otherwise} \end{cases}$$

is a martingale, where t is the trend of $\mathcal{X}$.

Moreover, the vector $\boldsymbol{v}$ satisfies $\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min} \leq 2|Q|/x_{\min}^{|Q|}$, where $x_{\min}$ is the smallest positive transition probability in $\mathcal{X}$, and $\boldsymbol{v}_{\max}$ and $\boldsymbol{v}_{\min}$ are the maximal and the minimal components of $\boldsymbol{v}$, respectively.

Due to Proposition 7, powerful results of martingale theory become applicable to pOC. In this paper, we use the constructed martingale to establish statements (iii) and (iv) of Theorem 5, by employing Azuma's inequality and the optional stopping theorem (see [20,21]). We also use the martingale to prove the crucial *divergence gap theorem* in Section 4. The range of possible applications of Proposition 7 is of course wider.

Assume now case (B.b.1), i.e., $t \neq 0$. For every $i \in \mathbb{N}$, let $Run(p{\downarrow}q, i)$ be the set of all $w \in Run(p{\downarrow}q)$ that visit $q(0)$ in i transitions, and let $[p{\downarrow}q, i]$ be the probability of $Run(p{\downarrow}q, i)$. We first show that there are $0 < a < 1$ and $h \in \mathbb{N}$ such that for all $i \geq h$ we have that $[p{\downarrow}q, i] \leq a^i$. Consider the martingale $m^{(0)}, m^{(1)}, \ldots$ over $Run(p(1))$ as defined in Proposition 7. A relatively straightforward computation reveals that for sufficiently large $h \in \mathbb{N}$ and all $i \geq h$ we have the following: If $t < 0$, then $[p{\downarrow}q, i] \leq \mathcal{P}\left(m^{(i)} - m^{(0)} \geq (i/2) \cdot (-t)\right)$, and if $t > 0$, then $[p{\downarrow}q, i] \leq \mathcal{P}\left(m^{(0)} - m^{(i)} \geq (i/2) \cdot t\right)$. In each step, the martingale value changes by at most $\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min} + t + 1$, where $\boldsymbol{v}$ is from Proposition 7. Hence, by applying Azuma's inequality (see [21]) we obtain the following (for all $t \neq 0$ and $i \geq h$):

$$[p{\downarrow}q, i] \quad \leq \quad \exp\left(-\frac{(i/2)^2 t^2}{2i(\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min} + t + 1)^2}\right) \quad = \quad a^i$$

Here $a = \exp\left(-t^2 \,/\, 8(\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min} + t + 1)^2\right)$. It follows that

$$E(p{\downarrow}q) \;=\; \sum_{i=1}^{\infty} i \cdot \frac{[p{\downarrow}q, i]}{[p{\downarrow}q]} \;\leq\; \frac{1}{[p{\downarrow}q]} \left(\sum_{i=1}^{h-1} i \cdot [p{\downarrow}q, i] + \sum_{i=h}^{\infty} i \cdot a^i \right) \;<\; \infty\,.$$

Finally assume case (B.b.2), i.e., $t = 0$. We need to show that $E(p{\downarrow}q) = \infty$. Let us introduce some notation. For every $k \in \mathbb{N}_0$, let $Q(k)$ be the set of all configurations where the counter value equals k. Let $p, q \in Q$ and $\ell, k \in \mathbb{N}_0$, where $\ell > k$. An *honest path from $p(\ell)$ to $q(k)$* is a finite path w from $p(\ell)$ to $q(k)$ such that the counter stays above k in all configurations of w except for the last one. We use $hpath(p(\ell), Q(k))$ to denote the set of all honest paths from $p(\ell)$ to some $q(k) \in Q(k)$. For a given $P \subseteq hpath(p(\ell), Q(k))$, the *expected length of an honest path in P* is defined as $\sum_{w \in P} \mathcal{P}(Run(w)) \cdot length(w)$. Using the martingale from Proposition 7 we show the following:

Proposition 8. *If $Pre^*(q(0))$ is infinite, then almost all runs initiated in an arbitrary configuration reach $Q(0)$. Moreover, there is $k_1 \in \mathbb{N}$ such that, for all $\ell \geq k_1$, the expected length of an honest path from $r(\ell)$ to $Q(0)$ is infinite.*

Proof (Sketch). Assume that $Pre^*(q(0))$ is infinite. The fact that almost all runs initiated in an arbitrary configuration reach $Q(0)$ follows from results of [4].

Consider an initial configuration $r(\ell)$ with $\ell + \boldsymbol{v}_r > \boldsymbol{v}_{\max}$. We will show that the expected length of an honest path from $r(\ell)$ to $Q(0)$ is infinite; i.e., we can take $k_1 := \lceil \boldsymbol{v}_{\max} - \boldsymbol{v}_{\min} + 1 \rceil$. Consider the martingale $m^{(0)}, m^{(1)}, \ldots$ defined in Proposition 7 over $Run(r(\ell))$. Note that as $t = 0$, the term $i \cdot t$ vanishes from the definition of the martingale.

Now let us fix $k \in \mathbb{N}$ such that $\ell + \boldsymbol{v}_r < \boldsymbol{v}_{\max} + k$ and define a *stopping time* τ (see e.g. [21]) which returns the first point in time in which either $m^{(\tau)} \geq \boldsymbol{v}_{\max} + k$, or $m^{(\tau)} \leq \boldsymbol{v}_{\max}$. A routine application of optional stopping theorem gives us the following

$$\mathcal{P}(m^{(\tau)} \geq \boldsymbol{v}_{\max} + k) \quad \geq \quad \frac{\ell + \boldsymbol{v}_r - \boldsymbol{v}_{\max}}{k + M}\,. \tag{1}$$

Denote by T the number of steps to hit $Q(0)$. Note that $m^{(\tau)} \geq \boldsymbol{v}_{\max} + k$ implies $c^{(\tau)} = m^{(\tau)} - \boldsymbol{v}_{p^{(\tau)}} \geq \boldsymbol{v}_{\max} + k - \boldsymbol{v}_{p^{(\tau)}} \geq k$, and thus also $T \geq k$, as at least k steps are required to decrease the counter value from k to 0. It follows that $\mathcal{P}(m^{(\tau)} \geq \boldsymbol{v}_{\max} + k) \leq \mathcal{P}(T \geq k)$. By putting this inequality together with the inequality (1) we obtain

$$\mathbb{E}[T] \;=\; \sum_{k \in \mathbb{N}} \mathcal{P}(T \geq k) \;\geq\; \sum_{k=\ell+1}^{\infty} \mathcal{P}(T \geq k) \;\geq\; \sum_{k=\ell+1}^{\infty} \frac{\ell + \boldsymbol{v}_r - \boldsymbol{v}_{\max}}{k + M} \;=\; \infty\,. \qquad \square$$

Further, we need the following observation about the structure of $\mathcal{M}_{\mathscr{A}}$, which holds also for non-probabilistic one-counter automata:

Proposition 9. *There is $k_2 \in \mathbb{N}$ such that for every configuration $r(\ell) \in Pre^*(q(0))$, where $\ell \geq k_2$, we have that if $r(\ell) \to r'(\ell')$, then $r'(\ell') \in Pre^*(q(0))$.*

To show that $E(p{\downarrow}q) = \infty$, it suffices to identify a subset $W \subseteq R(p{\downarrow}q)$ such that $\mathcal{P}(W) > 0$ and $\mathbb{E}[R_{p{\downarrow}q} \mid W] = \infty$. Now observe that if $Pre^*(q(0)) \cap Post^*(p(1))$ is infinite, there is a configuration $r(\ell) \in Pre^*(q(0))$ reachable from $p(1)$ along a finite path u such that $\ell \geq k_1 + k_2$, where k_1 and k_2 are the constants of Propositions 8 and 9.

Due to Proposition 8, the expected length of an honest path from $r(\ell - k_2)$ to $Q(0)$ is infinite. However, then also the expected length of an honest path from $r(\ell)$ to $Q(k_2)$ is infinite. This means that there is a state $s \in Q$ such that the expected length of an honest path from $r(\ell)$ to $s(k_2)$ in infinite. Further, it follows directly from Proposition 9 that $s(k_2) \in Pre^*(q(0))$ because there is an honest path from $r(\ell)$ to $s(k_2)$.

Now consider the set W of all runs w initiated in $p(1)$ that start with the finite path u, then follow an honest path from $r(\ell)$ to $s(k_2)$, and then follow an honest path from $s(k_2)$ to $q(0)$. Obviously, $\mathcal{P}(W) > 0$, and $\mathbb{E}[R_{p\downarrow q} \mid W] = \infty$ because the expected length of the middle subpath is infinite. Hence, $E(p{\downarrow}q) = \infty$ as needed.

3.2 Efficient Approximation of Finite Expected Termination Time

Let us denote by $T^{>0}_{<\infty}$ the set of all pairs $(p, q) \in T^{>0}$ satisfying $E(p{\downarrow}q) < \infty$. Our aim is to prove the following:

Theorem 10. *For all $(p, q) \in T^{>0}_{<\infty}$, the value of $E(p{\downarrow}q)$ can be approximated up to an arbitrarily small absolute error $\varepsilon > 0$ in time polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$.*

Note that if y approximates $E(p{\downarrow}q)$ up to an absolute error $1 > \varepsilon > 0$, then y approximates $E(p{\downarrow}q)$ also up to the relative error ε because $E(p{\downarrow}q) \geq 1$.

The proof of Theorem 10 is based on the fact that the vector of all $E(p{\downarrow}q)$, where $(p, q) \in T^{>0}_{<\infty}$, is the unique solution of a system of linear equations whose coefficients can be efficiently approximated (see below). Hence, it suffices to approximate the coefficients, solve the approximated equations, and then bound the error of the approximation using standard arguments from numerical analysis.

Let us start by setting up the system of linear equations for $E(p{\downarrow}q)$. For all $p, q \in T^{>0}$, we fix a fresh variable $V(p{\downarrow}q)$, and construct the following system of linear equations, $\mathcal{L}$, where the termination probabilities are treated as constants:

$$\begin{aligned} V(p{\downarrow}q) \;=\; & \sum_{(p,-1,q)\in\delta^{>0}} \frac{P^{>0}(p,-1,q)}{[p{\downarrow}q]} + \sum_{(p,0,t)\in\delta^{>0}} \frac{P^{>0}(p,0,t)\cdot[t{\downarrow}q]}{[p{\downarrow}q]}\cdot\Big(1 + V(t{\downarrow}q)\Big) \\ & + \sum_{(p,1,t)\in\delta^{>0}} \sum_{r\in Q} \frac{P^{>0}(p,1,t)\cdot[t{\downarrow}r]\cdot[r{\downarrow}q]}{[p{\downarrow}q]}\cdot\Big(1 + V(t{\downarrow}r) + V(r{\downarrow}q)\Big) \end{aligned}$$

It has been shown in [13] that the tuple of all $E(p{\downarrow}q)$, where $(p, q) \in T^{>0}$, is the least solution of $\mathcal{L}$ in $\mathbb{R}^+ \cup \{\infty\}$ with respect to component-wise ordering (where ∞ is treated according to the standard conventions). Due to Corollary 6, we can further simplify the system $\mathcal{L}$ by erasing the defining equations for all $V(p{\downarrow}q)$ such that $E(p{\downarrow}q) = \infty$ (note that if $E(p{\downarrow}q) < \infty$, then the defining equation for $V(p{\downarrow}q)$ in $\mathcal{L}$ cannot contain any variable $V(r{\downarrow}t)$ such that $E(r{\downarrow}t) = \infty$).

Thus, we obtain the system $\mathcal{L}'$. It is straightforward to show that the vector of all finite $E(p{\downarrow}q)$ is the *unique* solution of the system $\mathcal{L}'$ (see, e.g., Lemma 6.2.3 and Lemma 6.2.4 in [1]). If we rewrite $\mathcal{L}'$ into a standard matrix form, we obtain a system $\boldsymbol{V} = H\cdot\boldsymbol{V} + \boldsymbol{b}$, where H is a nonsingular nonnegative matrix, $\boldsymbol{V}$ is the vector of variables in $\mathcal{L}'$, and $\boldsymbol{b}$ is a vector. Further, we have that $\boldsymbol{b} = \mathbf{1}$, i.e., the constant coefficients are all 1. This follows from the following equality (see [12,17]):

$$\begin{aligned}[p{\downarrow}q] = \sum_{(p,-1,q)\in\delta^{>0}} P^{>0}(p,-1,q) \;+ \sum_{(p,0,t)\in\delta^{>0}} P^{>0}(p,0,t)\cdot[t{\downarrow}q] \\ + \sum_{(p,1,t)\in\delta^{>0}} \sum_{r\in Q} P^{>0}(p,1,t)\cdot[t{\downarrow}r]\cdot[r{\downarrow}q]\end{aligned} \tag{2}$$

Hence, $\mathcal{L}'$ takes the form $\boldsymbol{V} = H\cdot\boldsymbol{V}+\mathbf{1}$. Unfortunately, the entries of H can take irrational values and cannot be computed precisely in general. However, they can be approximated up to an arbitrarily small relative error using Proposition 3. Denote by G an approximated version of H. We aim at bounding the error of the solution of the "perturbed" system $\boldsymbol{V} = G\cdot\boldsymbol{V}+\mathbf{1}$ in terms of the error of G. To measure these errors, we use the l_∞ norm of vectors and matrices, defined as follows: For a vector $\boldsymbol{V}$ we have that $\|\boldsymbol{V}\| = \max_i |\boldsymbol{V}_i|$, and for a matrix M we have $\|M\| = \max_i \sum_j |M_{ij}|$. Hence, $\|M\| = \|M\cdot\mathbf{1}\|$ if M is nonnegative. We show the following:

Proposition 11. *Let* $b \geq \max\left\{E(p{\downarrow}q) \mid (p,q)\in T^{>0}_{<\infty}\right\}$. *Then for each* ε, *where* $0<\varepsilon<1$, *let* $\delta = \varepsilon/(12\cdot b^2)$. *If* $\|G-H\|\leq\delta$, *then the perturbed system* $\boldsymbol{V} = G\cdot\boldsymbol{V}+\mathbf{1}$ *has a unique solution* $\boldsymbol{F}$, *and in addition, we have that*

$$|E(p{\downarrow}q) - \boldsymbol{F}_{pq}| \quad\leq\quad \varepsilon \qquad \textit{for all } (p,q)\in T^{>0}_{<\infty}.$$

Here $\boldsymbol{F}_{pq}$ *is the component of* $\boldsymbol{F}$ *corresponding to the variable* $V(p{\downarrow}q)$.

The proof of Proposition 11 is based on estimating the size of the condition number $\kappa = \|1-H\|\cdot\|(1-H)^{-1}\|$ and applying standard results of numerical analysis.

The value of b in Proposition 11 can be estimated as follows: By Theorem 5, we have

$$E(p{\downarrow}q) \quad\leq\quad 85000\cdot|Q|^6/\left(x_{\min}^{6|Q|^3}\cdot t_{\min}^4\right) \qquad \text{for all } (p,q)\in T^{>0}_{<\infty},$$

where $t_{\min} = \min\{|t|\neq 0 \mid t \text{ is the trend in a BSCC of } \mathcal{X}\}$. Although b appears large, it is really the value of $\log(1/b)$ which matters, and it is still reasonable. Theorem 10 now follows by combining Propositions 11 and 3, because the approximated matrix G can be computed using a number of arithmetical operations which is polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$.

4 Quantitative Model-Checking of *Ω*-Regular Properties

In this section, we show that for every ω-regular property encoded by a deterministic Rabin automaton, the probability of all runs in a given pOC that satisfy the property can be approximated up to an arbitrarily small relative error $\varepsilon>0$ in polynomial time. This is achieved by designing and analyzing a new quantitative model-checking algorithm for pOC and ω-regular properties, which is *not* based on techniques developed for pPDA and RMC in [12,15,16].

Recall that a deterministic Rabin automaton (DRA) over a finite alphabet Σ is a deterministic finite-state automaton $\mathcal{R}$ with total transition function and *Rabin acceptance* *condition* $(E_1, F_1), \ldots, (E_k, F_k)$, where $k \in \mathbb{N}$, and all E_i, F_i are subsets of control states of $\mathcal{R}$. For a given infinite word w over Σ, let $\inf(w)$ be the set of all control states visited infinitely often along the unique run of $\mathcal{R}$ on w. The word w is accepted by $\mathcal{R}$ if there is $i \leq k$ such that $\inf(w) \cap E_i = \emptyset$ and $\inf(w) \cap F_i \neq \emptyset$.

Let Σ be a finite alphabet, $\mathcal{R}$ a DRA over Σ, and $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$ a pOC. A *valuation* is a function ν which to every configuration $p(i)$ of $\mathscr{A}$ assigns a unique letter of Σ. For simplicity, we assume that $\nu(p(i))$ depends only on the control state p (note that a "bounded" information about the current counter value can be encoded and maintained in the finite control of $\mathscr{A}$). Intuitively, the letters of Σ correspond to collections of predicates that are valid in a given configuration of $\mathscr{A}$. Thus, every run $w \in Run_{\mathscr{A}}(p(i))$ determines a unique infinite word $\nu(w)$ over Σ which is either accepted by $\mathcal{R}$ or not. The main result of this section is the following theorem:

Theorem 12. *For every $p \in Q$, the probability of all $w \in Run_{\mathscr{A}}(p(0))$ such that $\nu(w)$ is accepted by $\mathcal{R}$ can be approximated up to an arbitrarily small relative error $\varepsilon > 0$ in time polynomial in $|\mathscr{A}|$, $|\mathcal{R}|$, and* $\log(1/\varepsilon)$.

Our proof of Theorem 12 consists of three steps:

1. We show that the problem of our interest is equivalent to the problem of computing the probability of all accepting runs in pOC with Rabin acceptance condition.
2. We introduce a finite-state Markov chain $\mathcal{G}$ (with possibly irrational transition probabilities) such that the probability of all accepting runs in $\mathcal{M}_{\mathscr{A}}$ is equal to the probability of reaching a "good" BSCC in $\mathcal{G}$.
3. We show how to compute the probability of reaching a "good" BSCC in $\mathcal{G}$ with relative error at most ε in time polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$.

Let us note that Steps 1 and 2 are relatively simple, but Step 3 requires several insights. In particular, we cannot solve Step 3 without bounding a positive non-termination probability in pOC (i.e., a positive probability of the form $[p\uparrow]$) away from zero. This is achieved in our "divergence gap theorem" (i.e., Theorem 18), which is based on applying Azuma's inequality to the martingale constructed in Section 3.

Step 1. Let $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$ be a pOC. A *Rabin acceptance condition for* $\mathscr{A}$ is finite sequence $(\mathcal{E}_1, \mathcal{F}_1), \ldots, (\mathcal{E}_k, \mathcal{F}_k)$, where $\mathcal{E}_i, \mathcal{F}_i \subseteq Q$ for all $1 \leq i \leq k$. For every run $w \in Run_{\mathscr{A}}$, let $Q\text{-}\inf(w)$ be the set of all $p \in Q$ visited infinitely often along w. We use $Run_{\mathscr{A}}(p(0), acc)$ to denote the set of all *accepting runs* $w \in Run_{\mathscr{A}}(p(0))$ such that $Q\text{-}\inf(w) \cap \mathcal{E}_i = \emptyset$ and $Q\text{-}\inf(w) \cap \mathcal{F}_i \neq \emptyset$ for some $i \leq k$. Sometimes we also write $Run_{\mathscr{A}}(p(0), rej)$ to denote the set $Run_{\mathscr{A}}(p(0)) \setminus Run_{\mathscr{A}}(p(0), acc)$ of *rejecting* runs. Our next proposition says that the problem of computing/approximating the probability of all runs w in a given pOC that are accepted by a given DRA is efficiently reducible to the problem of computing/approximating the probability of all accepting runs in a given pOC with Rabin acceptance condition. The proof is simple (we just "synchronize" a given pOC with a given DRA).

Proposition 13. *Let Σ be a finite alphabet, $\mathscr{A}$ a pOC, ν a valuation, $\mathcal{R}$ a DRA over Σ, and $p(0)$ a configuration of $\mathscr{A}$. Then there is a pOC $\mathscr{A}'$ with Rabin acceptance condition and a configuration $p'(0)$ of $\mathscr{A}'$ constructible in polynomial time such that* *the probability of all $w \in Run_{\mathscr{A}}(p(0))$ where $v(w)$ is accepted by $\mathcal{R}$ is equal to the probability of all accepting $w \in Run_{\mathscr{A}'}(p'(0))$.*

For the rest of this section, we fix a pOC $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$ and a Rabin acceptance condition $(\mathcal{E}_1, \mathcal{F}_1), \ldots, (\mathcal{E}_k, \mathcal{F}_k)$ for $\mathscr{A}$. We show how to approximate the probability of $Run_{\mathscr{A}}(p(0), acc)$.

Step 2. Let $\mathcal{G}$ be a finite-state Markov chain, where $Q \times \{0,1\} \cup \{acc, rej\}$ is the set of states (the elements of $Q \times \{0,1\}$ are written as $q(i)$, where $i \in \{0,1\}$), and the transitions of $\mathcal{G}$ are defined as follows:

- $r(0) \xrightarrow{x} q(j)$ is a transition of $\mathcal{G}$ iff $r(0) \xrightarrow{x} q(j)$ is a transition of $\mathcal{M}_{\mathscr{A}}$;
- $r(1) \xrightarrow{x} q(0)$ iff $x = [r{\downarrow}q] > 0$;
- $r(1) \xrightarrow{x} acc$ iff $x = \mathcal{P}(Run_{\mathscr{A}}(r(1), acc) \cap Run_{\mathscr{A}}(r{\uparrow})) > 0$;
- $r(1) \xrightarrow{x} rej$ iff $x = \mathcal{P}(Run_{\mathscr{A}}(r(1), rej) \cap Run_{\mathscr{A}}(r{\uparrow})) > 0$;
- $acc \xrightarrow{1} acc$, $rej \xrightarrow{1} rej$;
- there are no other transitions.

Note that almost every $w \in Run_{\mathscr{A}}(p(0))$ has its "twin" $w' \in Run_{\mathcal{G}}(p(0))$, which is obtained from w as follows: each honest subpath in w of the form $r(1), \ldots, q(0)$ is replaced with a single transition $r(1) \to q(0)$ in w'; and if the counter is decreased to zero only finitely many times along w, then the last transition of the form $r(0) \to q(1)$ in w is replaced either with $r(0) \to acc$ or $r(0) \to rej$ in w', depending on whether w is accepting or rejecting (the rest of w is then replaced with loops on acc or rej).

A BSCC B of $\mathcal{G}$ is *good* if either $B = \{acc\}$, or there is $i \le k$ such that $\mathcal{E}_i \cap Q(B) = \emptyset$ and $\mathcal{F}_i \cap Q(B) \neq \emptyset$, where $Q(B)$ consists of all $r \in Q$ such that either $r(j) \in B$ for some $j \in \{0,1\}$, or there are $t(1), q(0) \in B$ such that $t(1) \to q(0)$ is a transition in $\mathcal{G}$ and $r(j) \in Pre^*(q(0)) \cap Post^*(t(1))$ for some $j \in \mathbb{N}_0$. For every $p \in Q$, let $Run_{\mathcal{G}}(p(0), good)$ be the set of all $w \in Run_{\mathcal{G}}(p(0))$ that visit a good BSCC of $\mathcal{G}$. The next proposition is obtained by a careful case analysis of accepting runs in $\mathcal{M}_{\mathscr{A}}$.

Proposition 14. *For every $p \in Q$ we have $\mathcal{P}(Run_{\mathscr{A}}(p(0), acc)) = \mathcal{P}(Run_{\mathcal{G}}(p(0), good))$.*

Step 3. Due to Proposition 14, the problem of our interest reduces to the problem of approximating the probability of visiting a good BSCC in the finite-state Markov chain $\mathcal{G}$. Since the termination probabilities in $\mathscr{A}$ can be approximated efficiently (see Proposition 3), the only problem with $\mathcal{G}$ is approximating the probabilities x and y in transitions of the form $p(1) \xrightarrow{x} acc$ and $p(1) \xrightarrow{y} rej$. Recall that x and y are the probabilities of all $w \in Run_{\mathscr{A}}(p{\uparrow})$ that are accepting and rejecting, respectively. A crucial observation is that almost all $w \in Run_{\mathscr{A}}(p{\uparrow})$ still behave accordingly with the underlying finite-state Markov chain $\mathcal{X}$ of $\mathscr{A}$ (see Section 3). More precisely, we have the following:

Proposition 15. *Let $p \in Q$. For almost all $w \in Run_{\mathscr{A}}(p{\uparrow})$ we have that w visits a BSCC B of $\mathcal{X}$ after finitely many transitions, and then it visits all states of B infinitely often.*

A BSCC B of $\mathcal{X}$ is *consistent* with the considered Rabin acceptance condition if there is $i \le k$ such that $B \cap \mathcal{E}_i = \emptyset$ and $B \cap \mathcal{F}_i \neq \emptyset$. If B is not consistent, it is *inconsistent*. An immediate corollary to Proposition 15 is the following:

Corollary 16. *Let $Run_{\mathscr{A}}(p(1), cons)$ and $Run_{\mathscr{A}}(p(1), inco)$ be the sets of all $w \in Run_{\mathscr{A}}(p(1))$ such that w visit a control state of some consistent and inconsistent BSCC of $\mathcal{X}$, respectively. Then*

- $\mathcal{P}(Run_{\mathscr{A}}(p(1), acc) \cap Run_{\mathscr{A}}(p\uparrow)) = \mathcal{P}(Run_{\mathscr{A}}(p(1), cons) \cap Run_{\mathscr{A}}(p\uparrow))$
- $\mathcal{P}(Run_{\mathscr{A}}(p(1), rej) \cap Run(p\uparrow)) = \mathcal{P}(Run_{\mathscr{A}}(p(1), inco) \cap Run_{\mathscr{A}}(p\uparrow))$

Due to Corollary 16, we can reduce the problem of computing the probabilities of transitions of the form $p(1) \xrightarrow{x} acc$ and $p(1) \xrightarrow{y} rej$ to the problem of computing the divergence probability in pOC. More precisely, we construct pOC's $\mathscr{A}_{cons}$ and $\mathscr{A}_{inco}$ which are the same as $\mathscr{A}$, except that for each control state q of an inconsistent (or consistent, resp.) BSCC of $\mathcal{X}$, all positive outgoing rules of q are replaced with $q \xrightarrow{1,-1}_{>0} q$. Then $x = \mathcal{P}(Run_{\mathscr{A}_{cons}}(p\uparrow))$ and $y = \mathcal{P}(Run_{\mathscr{A}_{inco}}(p\uparrow))$.

Due to [4], the problem whether a given divergence probability is positive (in a given pOC) is decidable in polynomial time. This means that the underlying graph of $\mathcal{G}$ is computable in polynomial time, and hence the sets G_0 and G_1 consisting of all states s of $\mathcal{G}$ such that $\mathcal{P}(Run_{\mathcal{G}}(s, good))$ is equal to 0 and 1, respectively, are constructible in polynomial time. Let G be the set of all states of $\mathcal{G}$ that are not contained in $G_0 \cup G_1$, and let $X_{\mathcal{G}}$ be the stochastic matrix of $\mathcal{G}$. For every $s \in G$ we fix a fresh variable V_s and the equation

$$V_s \quad = \quad \sum_{s' \in G} X_{\mathcal{G}}(s, s') \cdot V_{s'} \quad + \quad \sum_{s' \in G_1} X_{\mathcal{G}}(s, s')$$

Thus, we obtain a system of linear equations $\boldsymbol{V} = A\boldsymbol{V} + \boldsymbol{b}$ whose unique solution $\boldsymbol{V}^*$ in $\mathbb{R}$ is the vector of probabilities of reaching a good BSCC from the states of G. This system can also be written as $(I - A)\boldsymbol{V} = \boldsymbol{b}$. Since the elements of A and $\boldsymbol{b}$ correspond to (sums of) transition probabilities in $\mathcal{G}$, it suffices to compute the transition probabilities of $\mathcal{G}$ with a sufficiently small relative error so that the approximate A and $\boldsymbol{b}$ produce an approximate solution where the relative error of each component is bounded by the ε. By combining standard results for finite-state Markov chains with techniques of numerical analysis, we show the following:

Proposition 17. *Let $c = 2|Q|$. For every $s \in G$, let R_s be the probability of visiting a BSCC of $\mathcal{G}$ from s in at most c transitions, and let $R = \min\{R_s \mid s \in G\}$. Then $R > 0$ and if all transition probabilities in $\mathcal{G}$ are computed with relative error at most $\varepsilon R^3/8(c+1)^2$, then the resulting system $(I - A')\boldsymbol{V} = \boldsymbol{b}'$ has a unique solution $\boldsymbol{U}^*$ such that $|\boldsymbol{V}^*_s - \boldsymbol{U}^*_s|/\boldsymbol{V}^*_s \leq \varepsilon$ for every $s \in G$.*

Note that the constant R of Proposition 17 can be bounded from below by $x_t^{|Q|-1} \cdot x_n$, where

- $x_t = \min\{X_{\mathcal{G}}(s, s') \mid s, s' \in G\}$, i.e., x_t is the minimal probability that is either explicitly used in $\mathscr{A}$, or equal to some positive termination probability in $\mathscr{A}$;
- $x_n = \min\{X_{\mathcal{G}}(s, s') \mid s \in G, s' \in G_1\}$, i.e., x_n is the minimal probability that is either a positive termination probability in $\mathscr{A}$, or a positive non-termination probability in the pOC's $\mathscr{A}_{cons}$ and $\mathscr{A}_{inco}$ constructed above.

Now we need to employ the promised divergence gap theorem, which bounds a positive non-termination probability in pOC away from zero (for all $p, q \in Q$, we use $[p, q]$ to denote the probability of all runs w initiated in $p(1)$ that visit a configuration $q(k)$, where $k \geq 1$ and the counter stays positive in all configurations preceding this visit).

Theorem 18. *Let $\mathscr{A} = (Q, \delta^{=0}, \delta^{>0}, P^{=0}, P^{>0})$ be a pOC and X the underlying finite-state Markov chain of $\mathscr{A}$. Let $p \in Q$ such that $[p\uparrow] > 0$. Then there are two possibilities:*

1. *There is $q \in Q$ such that $[p, q] > 0$ and $[q\uparrow] = 1$. Hence, $[p\uparrow] \geq [p, q]$.*
2. *There is a BSCC $\mathscr{B}$ of X and a state q of $\mathscr{B}$ such that $[p, q] > 0$, $t > 0$, and $\boldsymbol{v}_q = \boldsymbol{v}_{\max}$ (here t is the trend, $\boldsymbol{v}$ is the vector of Proposition 7, and $\boldsymbol{v}_{\max}$ is the maximal component of $\boldsymbol{v}$; all of these are considered in $\mathscr{B}$). Further, $[p\uparrow] \geq [p, q]t^3/12(2(\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min}) + 4)^3$.*

Hence, denoting the relative precision $\varepsilon R^3/8(c+1)^2$ of Proposition 17 by δ, we obtain that $\log(1/\delta)$ is bounded by a polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$. Further, the transition probabilities of $\mathcal{G}$ can be approximated up to the relative error δ in time polynomial in $|\mathscr{A}|$ and $\log(1/\varepsilon)$ by approximating the termination probabilities of $\mathscr{A}$ (see Proposition 3). This proves Theorem 12.

5 Experimental Results, Future Work

We have implemented a prototype tool in the form of a Maple worksheet[1], which allows to compute the termination probabilities of pOC and the conditional expected termination times. Our tool employs Newton's method to approximate the termination probabilities within a sufficient accuracy so that the expected termination time is computed with absolute error (at most) one by solving the linear equation system from Section 3.2.

We applied our tool to the pOC model of the program *TreeEval* (see Section 1) for various values of the parameters. The following table shows the results. We also show the associated termination probabilities, rounded to three digits. We write $[a{\downarrow}0]$ etc. to abbreviate $[(\mathit{and,init}){\downarrow}(\mathit{or,return,0})]$ etc., and $[a{\downarrow}]$ for $[a{\downarrow}0] + [a{\downarrow}1]$.

	$[a{\downarrow}]$	$[a{\downarrow}0]$	$[a{\downarrow}1]$	$E[a{\downarrow}0]$	$E[a{\downarrow}1]$
$z = 0.5, y = 0.4, x_a = 0.2, x_o = 0.2$	0.800	0.500	0.300	11.000	7.667
$z = 0.5, y = 0.4, x_a = 0.2, x_o = 0.4$	0.967	0.667	0.300	104.750	38.917
$z = 0.5, y = 0.4, x_a = 0.2, x_o = 0.6$	1.000	0.720	0.280	20.368	5.489
$z = 0.5, y = 0.4, x_a = 0.2, x_o = 0.8$	1.000	0.732	0.268	10.778	2.758
$z = 0.5, y = 0.5, x_a = 0.1, x_o = 0.1$	0.861	0.556	0.306	11.400	5.509
$z = 0.5, y = 0.5, x_a = 0.2, x_o = 0.1$	0.931	0.556	0.375	23.133	20.644
$z = 0.5, y = 0.5, x_a = 0.3, x_o = 0.1$	1.000	0.546	0.454	83.199	111.801
$z = 0.5, y = 0.5, x_a = 0.4, x_o = 0.1$	1.000	0.507	0.493	12.959	21.555
$z = 0.2, y = 0.4, x_a = 0.2, x_o = 0.2$	0.810	0.696	0.115	7.827	6.266
$z = 0.3, y = 0.4, x_a = 0.2, x_o = 0.2$	0.811	0.636	0.175	8.928	6.783
$z = 0.4, y = 0.4, x_a = 0.2, x_o = 0.2$	0.808	0.571	0.236	10.005	7.258
$z = 0.5, y = 0.4, x_a = 0.2, x_o = 0.2$	0.800	0.500	0.300	11.000	7.667

[1] Available at `http://www.comlab.ox.ac.uk/people/stefan.kiefer/pOC.mws`

We believe that other interesting quantities and numerical characteristics of pOC, related to both finite paths and infinite runs, can also be efficiently approximated using the methods developed in this paper. An efficient implementation of the associated algorithms would result in a verification tool capable of analyzing an interesting class of infinite-state stochastic programs, which is beyond the scope of currently available tools limited to finite-state systems only.

References

1. Brázdil, T.: Verification of Probabilistic Recursive Sequential Programs. PhD thesis, Masaryk University, Faculty of Informatics (2007)
2. Brázdil, T., Brožek, V., Holeček, J., Kučera, A.: Discounted properties of probabilistic pushdown automata. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 230–242. Springer, Heidelberg (2008)
3. Brázdil, T., Brožek, V., Etessami, K.: One-counter stochastic games. In: Proceedings of FST&TCS 2010. LIPIcs, vol. 8, pp. 108–119. Schloss Dagstuhl (2010)
4. Brázdil, T., Brožek, V., Etessami, K., Kučera, A., Wojtczak, D.: One-counter Markov decision processes. In: Proceedings of SODA 2010, pp. 863–874. SIAM, Philadelphia (2010)
5. Brázdil, T., Esparza, J., Kučera, A.: Analysis and prediction of the long-run behavior of probabilistic sequential programs with recursion. In: Proceedings of FOCS 2005, pp. 521–530. IEEE Computer Society Press, Los Alamitos (2005)
6. Brázdil, T., Kiefer, S., Kučera, A.: Efficient analysis of probabilistic programs with an unbounded counter. CoRR, abs/1102.2529 (2011)
7. Brázdil, T., Kučera, A., Stražovský, O.: On the decidability of temporal properties of probabilistic pushdown automata. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 145–157. Springer, Heidelberg (2005)
8. Canny, J.: Some algebraic and geometric computations in PSPACE. In: Proceedings of STOC 1988, pp. 460–467. ACM Press, New York (1988)
9. Chatterjee, K., Doyen, L.: Energy parity games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
10. Chatterjee, K., Doyen, L., Henzinger, T., Raskin, J.-F.: Raskin. Generalized mean-payoff and energy games. In: Proceedings of FST&TCS 2010. LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl (2010)
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
12. Esparza, J., Kučera, A., Mayr, R.: Model-checking probabilistic pushdown automata. In: Proceedings of LICS 2004, pp. 12–21. IEEE Computer Society Press, Los Alamitos (2004)
13. Esparza, J., Kučera, A., Mayr, R.: Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In: Proceedings of LICS 2005, pp. 117–126. IEEE Computer Society Press, Los Alamitos (2005)
14. Etessami, K., Wojtczak, D., Yannakakis, M.: Quasi-birth-death processes, tree-like QBDs, probabilistic 1-counter automata, and pushdown systems. In: Proceedings of 5th Int. Conf. on Quantitative Evaluation of Systems (QEST 2008). IEEE Computer Society Press, Los Alamitos (2008)
15. Etessami, K., Yannakakis, M.: Algorithmic verification of recursive probabilistic state machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 253–270. Springer, Heidelberg (2005)

16. Etessami, K., Yannakakis, M.: Checking LTL properties of recursive Markov chains. In: Proceedings of 2nd Int. Conf. on Quantitative Evaluation of Systems (QEST 2005), pp. 155–165. IEEE Computer Society Press, Los Alamitos (2005)
17. Etessami, K., Yannakakis, M.: Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 340–352. Springer, Heidelberg (2005)
18. Kemeny, J.G., Snell, J.L.: Finite Markov Chains. D. Van Nostrand Company (1960)
19. Kiefer, S., Luttenberger, M., Esparza, J.: On the convergence of Newton's method for monotone systems of polynomial equations. In: Proceedings of STOC 2007, pp. 217–226. ACM Press, New York (2007)
20. Rosenthal, J.S.: A first look at rigorous probability theory. World Scientific, Singapore (2006)
21. Williams, D.: Probability with Martingales. Cambridge University Press, Cambridge (1991)

Model Checking Algorithms for CTMDPs

Peter Buchholz[1], Ernst Moritz Hahn[2], Holger Hermanns[2], and Lijun Zhang[3]

[1] Technical University of Dortmund, Computer Science, Germany
[2] Saarland University, Computer Science, Germany
[3] Technical University of Denmark, DTU Informatics, Denmark

Abstract. Continuous Stochastic Logic (CSL) can be interpreted over continuous-time Markov decision processes (CTMDPs) to specify quantitative properties of stochastic systems that allow some external control. Model checking CSL formulae over CTMDPs requires then the computation of optimal control strategies to prove or disprove a formula. The paper presents a conservative extension of CSL over CTMDPs—with rewards—and exploits established results for CTMDPs for model checking CSL. A new numerical approach based on uniformization is devised to compute time bounded reachability results for time dependent control strategies. Experimental evidence is given showing the efficiency of the approach.

1 Introduction

Model checking of continuous-time Markov chains (CTMCs) is a well established approach to prove or disprove quantitative properties for a wide variety of systems [1, 2]. If the system can be controlled by some external entity, then continuous-time Markov decision processes (CTMDPs) [3, 4] rather than CTMCs are the natural extension to be used for modeling, possibly enriched with rewards.

In this paper we formulate the model checking problem of the logic CSL—with reward extensions—in terms of decision problems in CTMDPs. The most challenging model checking subproblem for this logic is to compute the minimum/maximum reward with which a CSL formula holds. The problem contains as a specific case the problem of computing the time or time-interval bounded reachability probability in CTMDPs, a problem that has received considerable attention recently [5–10].

We introduce a numerical algorithm based on uniformization to compute, and approximate, the minimum/maximum *gain vector* per state (can be interpreted as rewards and/or costs) for a finite interval $[0, T]$ that is the key for model checking CSL formulae. The method we present is an adaption and extension of a recent algorithm [11] to compute the accumulated reward in a CTMDP over a finite interval. It works in a backward manner by starting with some initial gain vector $\mathbf{g}_T$ at time $t = T$, then it determines the optimal decision at t, and then assumes that the optimal decision is deterministic for a small interval $(t', t]$. The gain vector can then be computed for the whole interval. Afterwards, the optimal action at t' is determined, and the procedure is repeated until we arrive at $t = 0$. The correctness follows from the celebrated result by Miller [12] showing that an optimal policy exists, and only a finite number of switches of the actions is needed for describing it. It returns a control strategy that maximizes or minimizes a reward measure over a finite or an infinite time horizon.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 225–242, 2011.

If reward values are *zero*, and we have the appropriate *initial value* for the gain vector $\mathbf{g}_T$, the problem can be exploited to arrive at an uniformization based approach for the computation of time bounded reachability probabilities within time T. It can easily be generalized to the maximal reachability for a finite interval $[t_0, T]$, which is the key element of checking the probabilistic operator in CSL. Moreover, by computing the gain vector between $[t_0, T]$ with $t_0 > 0$, and followed by a probabilistic reachability analysis for the interval $[0, t_0]$, we are able to compute the minimum/maximum gain vector for $[t_0, T]$: this gives us then a complete CSL model checking algorithm for CTMDPs.

Contribution. This paper provides a full CSL model checking algorithm for CTMDPs with rewards. We show that the problem, for both probabilistic operator and various reward properties, can be reduced to the computation of accumulated rewards within time T, which allows us to exploit a deep insight by Miller [12]. This then provides both theoretical and practical insights: (i) on the theoretical side, we have that all maximal (or minimal) values arising in model checking can be obtained by finite memory policies, (ii) on the practical side, we exploit recent algorithmic advances [11] to arrive at an efficient approximation algorithm—providing upper and lower bounds—based on the well known notion of uniformization. We also provide experimental evidence showing the efficiency of the new numerical approach. The improvements over the state-of-the-art are dramatic, and resemble the milestones in approximate CTMC model checking research, which was initially resorting to discretization [13], but got effective—and mainstream technology—only through the use of uniformization [2].

Organization of the paper. Section 2 provides the basic definitions. Section 3 introduces the logic CSL and shows how CSL formulae can be interpreted in terms of minimal/maximal rewards gained in CTMDPs. Afterwards, in Section 4, the basic model checking approach is presented. The key step of model checking is the computation of an appropriate gain vector. Section 5 introduces a new algorithm based on uniformization to compute the gain vector. Then the performance of the new model checking algorithm is evaluated by means of some examples in Section 6. Section 7 discusses related work, and the paper is concluded in Section 8.

2 Basic Definitions

In this section we define CTMDPs as our basic model class and formulate the general problem of computing maximal/minimal instantaneous and accumulated rewards. The following notations are mainly taken from [12] and are used similarly in [11].

Definition 1 (CTMDP). *A* continuous-time Markov decision process (CTMDP) *is a tuple* $\mathcal{C} = (\mathcal{S}, \mathcal{D}, \mathbf{Q}^{\mathbf{d}})$ *where*

- $\mathcal{S} = \{1, \ldots, n\}$ *is a finite set of states,*
- $\mathcal{D} = \times_{s=1}^{n} \mathcal{D}_s$ *where* $\mathcal{D}_s$ *is a finite set of decisions that can be taken in state* $s \in \mathcal{S}$*,*
- $\mathbf{Q}^{\mathbf{d}}$ *is an* $n \times n$ *generator matrix of a continuous-time Markov chain for each decision vector* $\mathbf{d}$ *of length* n *with* $\mathbf{d}(s) \in \mathcal{D}_s$*.*

A CTMDP with reward *is a pair* $(\mathcal{C}, \mathbf{r})$ *where* $\mathcal{C}$ *is a CTMDP and* $\mathbf{r}$ *is a nonnegative (column) reward vector of length* n*.*

Sometimes we additionally define the initial distribution $\mathbf{p}_0$ of a CTMDP, which is a row vector of length n that defines a probability distribution over the set of states $\mathcal{S}$.

We consider a time interval $[0, T]$ with $T > 0$. Let Ω denote the set of all (right continuous) step functions on $[0, T]$ into $\mathcal{S}$, and let $\mathcal{F}$ denote the σ-algebra [12] of the sets in the space Ω generated by the sets $\{\omega \in \Omega \mid \omega(t) = s\}$ for all $t \leq T$ and $s \in \mathcal{S}$.

The notation $\mathbf{d} \in \mathcal{D}$, or the variant with an index, is used for decision vectors. A *policy* π (also known as *scheduler* or *adversary*) is a mapping from $[0, T]$ into $\mathcal{D}$, and $\mathbf{d}_t$ is the corresponding decision vector at time $t \in [0, T]$, i.e., $\mathbf{d}_t(s)$ is the decision taken if the system is in state s at time t. We require that π is a measurable function where measurable means Lebesgue measurable [12, 14]. For a measurable policy π, the CTMDP with initial distribution $\mathbf{p}_0$ induces the *probability space* $(\Omega, \mathcal{F}, P^\pi_{\mathbf{p}_0})$. If we have an initial state s (i.e. $\mathbf{p}_0(s) = 1$), we write P^π_s instead of $P^\pi_{\mathbf{p}_0}$.

Let $\mathcal{M}$ be the set of all measurable policies on $[0, T]$. A policy π is *piecewise constant* if there exist some $m < \infty$ and $0 = t_0 < t_1 < t_2 < \ldots < t_{m-1} < t_m = T < \infty$ such that $\mathbf{d}_t = \mathbf{d}_{t'}$ for $t, t' \in (t_k, t_{k+1}]$ $(0 \leq k < m)$. The policy is *stationary* if $m = 1$.

For a given policy $\pi \in \mathcal{M}$, define a matrix $\mathbf{V}^\pi_{t,u}$ with $0 \leq t \leq u \leq T$ by the following differential equations:

$$\frac{\mathrm{d}}{\mathrm{d}u}\mathbf{V}^\pi_{t,u} = \mathbf{V}^\pi_{t,u}\mathbf{Q}^{\mathbf{d}_u} \tag{1}$$

with the initial condition $\mathbf{V}^\pi_{t,t} = \mathbf{I}$. Element (i, j) of this matrix contains the probability that the CTMDP under policy π is in state j at time u when it has been in state i at time t [12]. We use the notation $\mathbf{V}^\pi_t$ for $\mathbf{V}^\pi_{0,t}$. Knowing the initial distribution $\mathbf{p}_0$ at time 0, the distribution at time t equals $\mathbf{p}^\pi_t = \mathbf{p}_0\mathbf{V}^\pi_t$ with $0 \leq t \leq T$.

Let $(\mathcal{C}, \mathbf{r})$ be a CTMDP with reward, and $G \subseteq \mathcal{S}$ a set of states of our interests. Define as $\mathbf{r}|_G$ the vector which results from assigning zero rewards to non-G states, namely $\mathbf{r}|_G(s) = \mathbf{r}(s)$ if $s \in G$ and 0 otherwise. For $t \leq T$, let $\mathbf{g}^\pi_{t,T}|_G$ be a column vector of length n defined by:

$$\mathbf{g}^\pi_{t,T}|_G = \mathbf{V}^\pi_{t,T}\mathbf{g}_T + \int_t^T \mathbf{V}^\pi_{\tau,T}\mathbf{r}|_G \,\mathrm{d}\tau \tag{2}$$

where $\mathbf{g}_T$ is the initial gain vector at time T, independent of the policies. The second part is the accumulated gain vector through G-states between $[t, T]$. Intuitively, it contains in position $s \in \mathcal{S}$ the expected reward accumulated until time T, if the CTMDP is at time t in state s and policy π is chosen. In most of the cases, T is fixed and clear from the context, then we skip it and write $\mathbf{g}^\pi_t|_G$ instead. Moreover, $|_G$ will also be skipped in case $G = \mathcal{S}$. As we will see later, $\mathbf{g}_T$ will be initialized differently for different model checking problems but is independent of π. For a given initial vector $\mathbf{p}_0$ the expected reward under policy π equals $\mathbf{p}_0\mathbf{g}^\pi_0$.

3 Continuous Stochastic Logic

To specify quantitative properties we use a conservative extension of the logic *Continuous Stochastic Logic* (CSL) introduced in [1, 2], here interpreted over CTMDPs. We relate the model checking of CSL formulae to the computation of minimal/maximal gain vectors in CTMDPs.

3.1 CSL

Let $\mathtt{I}, \mathtt{J}$ be non-empty closed intervals on $\mathbb{R}_{\geq 0}$ with rational bounds. The syntax of the CSL formulae is defined as follows:

$$\Phi := a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathbb{P}_{\mathtt{J}}(\Phi \,\mathsf{U}^{\mathtt{I}}\Phi) \mid \mathbb{S}_{\mathtt{J}}(\Phi) \mid \mathbb{I}^{t}_{\mathtt{J}}(\Phi) \mid \mathbb{C}^{\mathtt{I}}_{\mathtt{J}}(\Phi)$$

where $a \in AP$, and $t \geq 0$. We use Φ, Ψ for CSL formulae, and use the abbreviations $\mathit{true} = a \vee \neg a$, $\Diamond^{\mathtt{I}}(\Phi) = \mathit{true}\ \mathsf{U}^{\mathtt{I}}\Phi$, and $\mathbb{C}^{\leq t}_{\mathtt{J}}(\Phi)$ for $\mathbb{C}^{[0,t]}_{\mathtt{J}}(\Phi)$. We refer to $\Phi\ \mathsf{U}^{\mathtt{I}}\Psi$ as a (CSL) path formula.

Except for the rightmost two operators, this logic agrees with CSL on CTMCs [2]. It should however be noted that $\mathbb{S}_{\mathtt{J}}(\Phi)$ refers to the long-run average reward gained in Φ-states, which coincides with the CTMC interpretation of CSL for a reward structure of constant 1. The rightmost two operators are inspired by the discussion in [15]. $\mathbb{I}^{t}_{\mathtt{J}}(\Phi)$ specifies that the instantaneous reward at time t in Φ-states is in the interval $\mathtt{J}$. $\mathbb{C}^{\mathtt{I}}_{\mathtt{J}}(\Phi)$ in turn accumulates (that is, integrates) the instantaneous reward gained over the interval $\mathtt{I}$ and specifies it to be in $\mathtt{J}$.[1]

The semantics of CSL formulae are interpreted over the states of the given reward CTMDP $(\mathcal{C}, \mathbf{r})$. Formally, the pair (s, Φ) belongs to the relation $\models_{(\mathcal{C},\mathbf{r})}$, denoted by $s \models_{(\mathcal{C},\mathbf{r})} \Phi$, if and only if Φ is true at s. The index is omitted whenever clear from the context. We need to introduce some additional notation. For state s, let α_s be the Dirac distribution with $\alpha_s(s) = 1$ and 0 otherwise. For a formula Φ, let $\mathit{Sat}(\Phi)$ denote the set of states satisfying Φ, moreover, we let $\mathbf{r}|_{\Phi}$ denote $\mathbf{r}|_{\mathit{Sat}(\Phi)}$. The relation $\models$ is defined as follows:

- Probabilistic Operator: $s \models \mathbb{P}_{\mathtt{J}}(\Phi\ \mathsf{U}^{\mathtt{I}}\Psi)$ iff for all policies π, it holds:
$$P^{\pi}_{s}(\{\omega \in \Omega \mid \omega \models \Phi\ \mathsf{U}^{\mathtt{I}}\Psi\}) \in \mathtt{J}$$
where $\omega \models \Phi\ \mathsf{U}^{\mathtt{I}}\Psi$ iff $\exists t \in \mathtt{I}.\ \omega(t) \models \Psi \wedge \forall 0 \leq t' < t.\ \omega(t') \models \Phi$.
- Instantaneous reward: $s \models \mathbb{I}^{t}_{\mathtt{J}}(\Phi)$ iff it holds that $\mathbf{p}^{\pi}_{t} \cdot \mathbf{r}|_{\Phi} \in \mathtt{J}$ for all policies π, where $\mathbf{p}^{\pi}_{t} = \alpha_s \mathbf{V}^{\pi}_{t}$ is the distribution at time t under π, starting with state s.
- Cumulative reward: $s \models \mathbb{C}^{[t,T]}_{\mathtt{J}}(\Phi)$ iff it holds that $(\alpha_s \mathbf{V}^{\pi}_{t}) \cdot \mathbf{g}^{\pi}_{t,T}|_{\mathit{Sat}(\Phi)} \in \mathtt{J}$ for all policies π, where $\mathbf{g}^{\pi}_{t,T}|_{\mathit{Sat}(\Phi)}$ is the gain vector under π as defined in Eqn. (2), with terminal condition $\mathbf{g}_T = 0$.
- Long-run average reward: $s \models \mathbb{S}_{\mathtt{J}}(\Phi)$ iff it holds that $\lim_{T\to\infty} \frac{1}{T} \cdot (\alpha_s \cdot \mathbf{g}^{\pi}_{0,T}|_{\Phi}) \in \mathtt{J}$ for all policies π. This is the average reward gained in an interval with a length going to infinity. In case $\mathbf{r}(s) = 1$ for all $s \in \mathcal{S}$, we refer to $\mathbb{S}$ also as the *steady state probability operator*.

The reward CTMDP satisfies a formula if the initial state does. A few remarks are in order. To simplify the presentation we have skipped the probabilistic next state operator $\mathbb{P}_{\mathtt{J}}(\mathsf{X}^{\mathtt{I}}\,\Phi)$. Recently, the policy classes depending on the whole history, including the complete sequence of visited states, action, sojourn time, has been considered for CTMDPs. This seemingly more powerful class of policies is known to be as powerful as the piecewise constant policies considered in this paper, as shown in [8, 9].

[1] For readers familiar with the PRISM tool notation, $\mathtt{R}_{\mathtt{J}}[\mathtt{C}^{\leq t}]$ corresponds to $\mathbb{C}^{\leq t}_{\mathtt{J}}(\mathit{true})$, $\mathtt{R}_{\mathtt{J}}[\mathtt{I}^{=t}]$ to $\mathbb{I}^{t}_{\mathtt{J}}(\mathit{true})$, and $\mathtt{R}_{\mathtt{J}}[\mathtt{S}]$ to $\mathbb{S}_{\mathtt{J}}(\mathit{true})$, respectively, for CTMCs with rewards.

3.2 Optimal Values and Policies

Our semantics is based on resolving the nondeterministic choices by policies. Obviously, checking probabilistic and reward properties amounts to computing, or approximating, the corresponding optimal values. For the probabilistic operator $\mathbb{P}_{\mathtt{J}}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$, we define

$$P_s^{max}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) := \sup_{\pi\in\mathcal{M}} P_s^{\pi}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi), \qquad P_s^{min}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) := \inf_{\pi\in\mathcal{M}} P_s^{\pi}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$$

as the *maximal (and minimal) probability* of reaching a Ψ-state along Φ-states. Then, $s \models \mathbb{P}_{\mathtt{J}}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$ iff $P_s^{max}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) \leq \sup \mathtt{J}$ and $P_s^{min}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) \geq \inf \mathtt{J}$. In case the condition is true, i.e., $\Phi = true$, we refer to it simply as *reachability probability*.

The defined extreme probabilities P_s^{max} and P_s^{min} are also referred to as the *optimal values*. A policy π is called *optimal*, with respect to $\mathbb{P}_{\mathtt{J}}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$, if it achieves the optimal values, i.e., if $P_s^{\pi}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) = P_s^{max}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$ or $P_s^{\pi}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi) = P_s^{min}(\Phi \,\mathsf{U}^{\mathtt{I}}\Psi)$.

The optimal values and policies are also defined for reward properties in a similar way. Briefly, we define:

- $R_s^{max}(\mathbb{I}^t\,\Phi) = \sup_{\pi\in\mathcal{M}}(\mathbf{p}_t^{\pi}\cdot \mathbf{r}|_{\Phi})$ for instantaneous reward,
- $R_s^{max}(\mathbb{C}^{[t,T]}\,\Phi) = \sup_{\pi\in\mathcal{M}}((\alpha_s\mathbf{V}_t^{\pi})\cdot \mathbf{g}_{t,T}^{\pi}|_{Sat(\Phi)})$ for cumulative reward, and
- $R_s^{max}(\mathbb{S}\,\Phi) = \sup_{\pi\in\mathcal{M}}\left(\lim_{T\to\infty}\frac{1}{T}\left(\alpha_s\cdot \mathbf{g}_{0,T}^{\pi}|_{Sat(\Phi)}\right)\right)$ for long-run average reward.

For the long-run average reward the optimal policy is stationary, which can be computed using a dynamic programming algorithm for average rewards as for example presented in [4]. The optimal policies achieving the supremum (or infimum) for instantaneous and cumulative rewards are piecewise constant, which will become clear in the next section.

4 Model Checking Algorithm

Given a CTMDP $(\mathcal{C}, \mathbf{r})$ with reward, a state s, and a CSL formula Φ, the model checking problem asks whether $s \models \Phi$ holds. In this section we present a model checking approach where the basic step consists in characterizing the gain vector for the computation of $R_s^{max}(\mathbb{C}^{\mathtt{I}}\,\Phi)$, $P_s^{max}(\Phi\mathsf{U}^{\mathtt{I}}\Psi)$, and $R_s^{max}(\mathbb{I}^t\,\Phi)$ (Of course, the same holds for the minimal gain vector, which is skipped). The corresponding numerical algorithms shall be presented in the next section.

4.1 Optimal Gain Vector for $R_s^{max}(\mathbb{C}^{\mathtt{I}}\; true)$

Our goal is to obtain the vector $\mathbf{g}_0^*$ that corresponds to the maximal gain that can be achieved by choosing an optimal policy in $[0, T]$. Stated differently, for a given $\mathbf{p}_0$, we aim to find a policy π^* which maximizes the gain vector in the interval $[0, T]$ in all elements. It can be shown [12] that this policy is independent of the initial probability vector and we need to find π^* such that

$$\pi^* = \arg\max_{\pi\in\mathcal{M}}\left(\mathbf{V}_T^{\pi}\mathbf{g}_T + \int_0^T \mathbf{V}_t^{\pi}\mathbf{r}\,\mathrm{d}t \quad \text{in all elements}\right). \tag{3}$$

Moreover, the maximal gain vector is denoted by $\mathbf{g}_0^* := \mathbf{g}_0^{\pi^*}$, with $|_G$ omitted as $G = \mathcal{S}$.

The problem of maximizing the accumulated reward of a finite CTMDP in a finite interval $[0, T]$ has been analyzed for a long time. The basic result can be found in [12] and is more than 40 years old. Further results and extensions can be found in [14]. The paper of Miller [12] introduces the computation of a policy π^* which maximizes the accumulated reward in $[0, T]$. The following theorem summarizes the main results of [12], adapted to our setting with a non-zero terminal gain vector $\mathbf{g}_T$:

Theorem 1 (Theorem 1 and 6 of [12]). *Let $(\mathcal{C}, \mathbf{r})$ be a CTMDP with reward, $T > 0$, and let $\mathbf{g}_T$ be the terminal condition of the gain vector. A policy is optimal if it maximizes for almost all $t \in [0, T]$*

$$\max_{\pi \in \mathcal{M}} \left(\mathbf{Q}^{\mathbf{d}_t} \mathbf{g}_t^\pi + \mathbf{r} \right) \text{ where } -\frac{\mathrm{d}}{\mathrm{d}t} \mathbf{g}_t^\pi = \mathbf{Q}^{\mathbf{d}_t} \mathbf{g}_t^\pi + \mathbf{r} \, . \tag{4}$$

There exists a piecewise constant policy $\pi \in \mathcal{M}$ that maximizes the equations.

In [12], the terminal condition $\mathbf{g}_T$ is fixed to the zero vector which is sufficient for the problem considered there. The corresponding proofs can be adapted in a straightforward way for the non-zero $\mathbf{g}_T$. We will see later that a non-zero terminal condition allows us to treat various reachability probabilities as they occur in model checking problems. Recall the vector $\mathbf{g}_t^\pi$ describes the gain at time t, i.e., $\mathbf{g}_t^\pi(s)$ equals the expected reward gained at time T if the CTMDP is in state s at time t and policy π is applied in the interval $[t, T]$. Miller presents a constructive proof of Theorem 1 which defines the following sets for some measurable policy $\pi \in \mathcal{M}$ with gain vector $\mathbf{g}_t^\pi$ at time t.

$$\begin{aligned}
\mathcal{F}_1(\mathbf{g}_t^\pi) &= \left\{ \mathbf{d} \in \mathcal{D} \mid \mathbf{d} \text{ maximizes } \mathbf{q}_{\mathbf{d}}^{(1)} \right\} \, , \\
\mathcal{F}_2(\mathbf{g}_t^\pi) &= \left\{ \mathbf{d} \in \mathcal{F}_1(\mathbf{g}_t) \mid \mathbf{d} \text{ maximizes } -\mathbf{q}_{\mathbf{d}}^{(2)} \right\} \, , \\
&\dots \\
\mathcal{F}_j(\mathbf{g}_t^\pi) &= \left\{ \mathbf{d} \in \mathcal{F}_{j-1}(\mathbf{g}_t) \mid \mathbf{d} \text{ maximizes } (-1)^{j-1} \mathbf{q}_{\mathbf{d}}^{(j)} \right\}
\end{aligned}$$

where

$$\begin{aligned}
&\mathbf{q}_{\mathbf{d}}^{(1)} = \mathbf{Q}^{\mathbf{d}} \mathbf{g}_t^\pi + \mathbf{r} \, , \; \mathbf{q}_{\mathbf{d}}^{(j)} = \mathbf{Q}^{\mathbf{d}} \mathbf{q}^{(j-1)} \text{ and} \\
&\mathbf{q}^{(j-1)} = \mathbf{q}_{\mathbf{d}}^{(j-1)} \text{ for any } \mathbf{d} \in \mathcal{F}_{j-1} (j = 2, 3, \dots)
\end{aligned}$$

The following theorem results from [12, Lemma 3 and 4].

Theorem 2. *If $\mathbf{d} \in \mathcal{F}_{n+1}(\mathbf{g}_t^\pi)$ then $\mathbf{d} \in \mathcal{F}_{n+k}(\mathbf{g}_t^\pi)$ for all $k > 1$.*

Let π be a measurable policy in $(t', T]$ and assume that $\mathbf{d} \in \mathcal{F}_{n+1}(\mathbf{g}_t^\pi)$ for $t' < t < T$, then exists some ε ($0 < \varepsilon \leq t - t'$) such that $\mathbf{d} \in \mathcal{F}_{n+1}(\mathbf{g}_{t''}^\pi)$ for all $t'' \in [t - \varepsilon, t]$.

We define a *selection procedure* that selects the lexicographically largest vector $\mathbf{d}$ from $\mathcal{F}_{n+1}$ which implies that we define some lexicographical ordering on the vectors $\mathbf{d}$. Then, the algorithm can be defined to get the optimal value with respect to cumulative reward (see [12]), which is presented in Algorithm 1. Let $\mathbf{g}_{t_0}^*$ denote the gain vector at $t = t_0 \geq 0$ and π^* the piecewise constant policy resulting from $\textsc{Optimal}(\mathcal{C}, \mathbf{r}, t_0, T, \mathbf{0})$ of the above algorithm. For the case $t_0 = 0$, the optimal gain for a given initial state s equals then $\alpha_s \mathbf{g}_0^*$. According to the Bellman equations [4] the restriction of the policy

Algorithm 1. $\text{OPTIMAL}(\mathcal{C}, \mathbf{r}, t_0, T, \mathbf{g}_T)$: Deciding optimal value and policy

```
1. Set $t' = T$ ;
2. Select $\mathbf{d}_{t'}$ using $\mathbf{g}_{t'}$ from $\mathcal{F}_{n+1}(\mathbf{g}_{t'})$ as described ;
3. Obtain $\mathbf{g}_t$ for $0 \leq t \leq t'$ by solving
       $-\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{g}_t = \mathbf{r} + \mathbf{Q}^{\mathbf{d}_{t'}}\mathbf{g}_t$
   with terminal condition $\mathbf{g}_{t'}$ ;
4. Set $t'' = \inf\{t : \mathbf{d}_t$ satisfies the selection procedure in $(t'', t']\}$ ;
5. If $t'' > t_0$ go to 2. with $t' = t''$. Otherwise, terminate and return the gain vector $\mathbf{g}^*_{t_0}$
   at $t = t_0$ and the resulting piecewise constant policy $\pi^*$ ;
```

π^* to the interval $(t, T]$ $(0 < t < T)$ results in an optimal policy with gain vector $\mathbf{g}^*_t$. Observe that Algorithm 1 is not implementable as it is described here, since step 4. cannot be effectively computed. We shall present algorithms to approximate or compute bounds for the optimal gain vector in Section 5.

4.2 Cumulative Reward $R_s^{max}(\mathbb{C}^{\leq t}\,\Phi)$

For computing $R_s^{max}(\mathbb{C}^{\leq t}\,\Phi)$, we have the terminal gain vector $\mathbf{g}_T = \mathbf{0}$. Let $\mathbf{g}^*_0$ denote the gain vector at $t = 0$ and π^* the piecewise constant policy resulting from $\text{OPTIMAL}(\mathcal{C}, \mathbf{r}|_{\Phi}, 0, T, \mathbf{0})$ of the above algorithm. The optimal cumulative reward for a given initial state s equals then $R_s^{max}(\mathbb{C}^{\leq t}\,\Phi) = \alpha_s \mathbf{g}^*_0$.

4.3 Probabilistic Operator $P_s^{max}(\Phi\ \mathsf{U}^{\mathrm{I}}\Psi)$

Let $(\mathcal{C}, \mathbf{0})$ be a CTMDP with zero rewards, $T > 0$. We consider the computation of $P_s^{max}(\Phi\ \mathsf{U}^{\mathrm{I}}\Psi)$, which will be discussed below.

Intervals of the Form $I = [0, T]$. In this case, as for CTMCs [2], once a state satisfying $\neg\Phi \vee \Psi$ has been reached, the future behaviors becomes irrelevant. Thus, these states can be made absorbing by removing all outgoing transitions, without altering the reachability probability. Let $Sat(\Phi)$ denote the set of states satisfying Φ. Applying Theorem 1 for zero-rewards $\mathbf{r} = \mathbf{0}$, with a terminal gain vector $\mathbf{g}_T$, we get directly:

Corollary 1. *Let $\Phi\ \mathsf{U}^{[0,T]}\Psi$ be a CSL path formula with $T > 0$. Let $(\mathcal{C}, \mathbf{0})$ be a CTMDP with zero rewards such that $Sat(\neg\Phi \vee \Psi)$ states are absorbing. Moreover, let $\mathbf{g}_T$ be the terminal gain vector with $\mathbf{g}_T(s) = 1$ if $s \in Sat(\Psi)$ and 0 otherwise. A policy is optimal (w.r.t. $P_s^{max}(\Phi\ \mathsf{U}^{[0,T]}\Psi)$) if it maximizes for almost all $t \in [0, T]$,*

$$\max_{\pi \in \mathcal{M}} \left(\mathbf{Q}^{\mathbf{d}_t}\mathbf{g}^{\pi}_t\right) \ \textit{where}\ -\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{g}^{\pi}_t = \mathbf{Q}^{\mathbf{d}_t}\mathbf{g}^{\pi}_t\ . \tag{5}$$

There exists a piecewise constant policy $\pi \in \mathcal{M}$ that maximizes the equations.

The following lemma shows that the optimal gain vector obtained by the above corollary can be used directly to obtain the maximal reachability probability:

Lemma 1. *Let* $\mathbf{g}_T$ *be the terminal gain vector with* $\mathbf{g}_T(s) = 1$ *if* $s \in Sat(\Psi)$ *and* 0 *otherwise. Assume the procedure* $\text{OPTIMAL}(\mathcal{C}, \mathbf{0}, 0, T, \mathbf{g}_T)$ *returns the optimal policy* π^* *and the corresponding optimal gain vector* $\mathbf{g}_0^*$*. Then, it holds* $P_s^{max}(\Phi \, \mathsf{U}^{[0,T]} \Psi) = \alpha_s \mathbf{g}_0^*$.

Proof. Since $\mathbf{r} = \mathbf{0}$, Eqn. (3) reduces to $\pi^* = \arg\max_{\pi \in \mathcal{M}} (\mathbf{V}_T^{\pi} \mathbf{g}_T^{\pi}$ in all elements). By definition, it is $\mathbf{g}_0^* = \mathbf{V}_T^{\pi^*} \mathbf{g}_T$, which is maximal in all elements. Moreover, since $Sat(\neg\Phi \vee \Psi)$-states are absorbing, the maximal transient probability is the same as the maximal time bounded reachability. Thus, $\mathbf{g}_0^*(s)$ is the maximal probability of reaching $Sat(\Psi)$ within T, along $Sat(\Phi)$-states from s, as $\mathbf{g}_0^*$ is maximal in all elements. Thus, $P_s^{max}(\Phi \, \mathsf{U}^{\mathrm{I}} \Psi) = \alpha_s \mathbf{g}_0^*$. □

Intervals of the form $I = [t_0, T]$ with $t_0 > 0$ and $T \geq t_0$. Let us review the problem of computing an optimal gain vector of a finite CTMDP in a finite interval $[0, T]$ from a new angle. Assume that an optimal policy is known for $[t_0, T]$ and $\mathbf{a}_{[t_0,T]}$ is the optimal gain vector at t_0, then the problem is reduced to finding an extension of the policy in $[0, t_0)$ which means to solve the following maximization problem:

$$\mathbf{g}_0^* = \max_{\pi \in \mathcal{M}} \left(\mathbf{V}_{t_0}^{\pi} \mathbf{a}_{[t_0,T]} \right) . \tag{6}$$

The problem can be easily transferred in the problem of computing the reachability probability for some interval $[t_0, T]$, after a modification of the CTMDP. Essentially, a two step approach has to be taken. As we have seen in Algorithm 1, the optimal policy to maximize the reward is computed in a backwards manner. First the optimal policy is computed for the interval $[t_0, T]$ with respect to the maximal probability $P_s^{max}(\Phi \, \mathsf{U}^{[0,T-t_0]} \Psi)$, using the CTMDP where states from $Sat(\neg\Phi \vee \Psi)$ are made absorbing. This policy defines the vector $\mathbf{a}_{[t_0,T]} = \mathbf{g}_{t_0}$: this is adapted appropriately—by setting the element to 0 for states satisfying $\neg\Phi$—which is then used as terminal condition to extend the optimal policy to $[0, t_0)$ on the original CTMDP.

Let $\mathcal{C}[\Phi]$ denote the CTMDP with states in $Sat(\Phi)$ made absorbing, and let $\mathbf{Q}[\Phi]$ denote the corresponding modified $\mathbf{Q}$-matrix in $\mathcal{C}[\Phi]$. The following corollary summarizes Theorem 1 when it is adopted to the interval bounded reachability probability.

Corollary 2. *Let* $(\mathcal{C}, \mathbf{0})$ *be a CTMDP with zero rewards* $\mathbf{r} = \mathbf{0}$, $t_0 > 0$ *and* $T \geq t_0$. *Let* $\Phi \, \mathsf{U}^{[t_0,T]} \Psi$ *be a path formula, and* $\mathbf{g}_T$ *be the terminal gain vector with* $\mathbf{g}_T(s) = 1$ *if* $s \in Sat(\Psi)$ *and* 0 *otherwise. A policy is optimal (w.r.t.* $P_s^{max}(\Phi \, \mathsf{U}^{[t_0,T]} \Psi)$*) if it*

- *maximizes for almost all* $t \in [t_0, T]$

$$\max_{\pi \in \mathcal{M}} \left(\mathbf{Q}_1^{\mathbf{d}_t} \mathbf{g}_t^{\pi} \right) \textit{ where } -\frac{\mathrm{d}}{\mathrm{d}t} \mathbf{g}_t^{\pi} = \mathbf{Q}_1^{\mathbf{d}_t} \mathbf{g}_t^{\pi} \, . \tag{7}$$

with $\mathbf{Q}_1 := \mathbf{Q}[\neg\Phi \vee \Psi]$ *and initial condition at* T *given by* $\mathbf{g}_T$*. Note that the vector* $\mathbf{g}_{t_0}^*$ *is uniquely determined by the above equation.*

- *maximizes for almost all* $t \in [0, t_0]$*:*

$$\max_{\pi \in \mathcal{M}} \left(\mathbf{Q}_2^{d_t} \mathbf{g}_t^\pi \right) \text{ where } -\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{g}_t^\pi = \mathbf{Q}_2^{\mathbf{d}_t} \mathbf{g}_t^\pi$$

with $\mathbf{Q}_2 := \mathbf{Q}[\neg\Phi]$*, and initial condition at* t_0 *given by* $\mathbf{g}'$ *defined by:* $\mathbf{g}'(s) = \mathbf{g}^*_{t_0}(s)$ *if* $s \models \Phi$*, and* 0 *otherwise.*

There exists a piecewise constant policy $\pi \in \mathcal{M}$ *that maximizes the equations.*

Notice that the corollary holds for the special case $\Phi = \mathit{true}$ and $t_0 = T$, what we get is also called the *maximal transient probability* of being at $\mathit{Sat}(\Psi)$ at exact time T, namely $\mathbf{V}_T^\pi$ with terminal condition $\mathbf{g}_T$. Now we can achieve the maximal interval bounded reachability probability:

Lemma 2. *Let* $\mathbf{g}_T$ *be as defined in Corollary 2. Assume the procedure* $\textsc{Optimal}(\mathcal{C}[\neg\Phi \vee \Psi], \mathbf{0}, t_0, T, \mathbf{g}_T)$ *returns the optimal policy* $\pi^*_{t_0}$ *and the corresponding optimal gain vector* $\mathbf{g}^*_{t_0}$*. Let* $\mathbf{g}'$ *be defined by* $\mathbf{g}'(s) = \mathbf{g}^*_{t_0}(s)$ *if* $s \models \Phi$*, and* 0 *otherwise.*

Assume the procedure $\textsc{Optimal}(\mathcal{C}[\neg\Phi], \mathbf{0}, 0, t_0, \mathbf{g}')$ *returns the optimal policy* π^* *(extending the policy* $\pi^*_{t_0}$*) and the corresponding optimal gain vector* $\mathbf{g}^*_0$*. Then, it holds* $P_s^{max}(\Phi \,\mathsf{U}^{[t_0,T]} \Psi) = \alpha_s \mathbf{g}^*_0$.

Proof. The optimal gain at time t_0 is obtained by $\mathbf{g}^*_{t_0}$, by Lemma 1. For all $t \leq t_0$, Φ must be satisfied by the semantics for the path formula, thus $\mathbf{g}^*_{t_0}$ is replaced with $\mathbf{g}'$ as initial vector for the following computation. Thus, $\mathbf{g}^*_0 = \mathbf{V}^{\pi^*}_{t_0} \mathbf{g}'$ is maximal in all elements, and $\mathbf{g}^*_0(s)$ is the maximal probability of reaching $\mathit{Sat}(\Psi)$ from s within $[t_0, T]$, along $\mathit{Sat}(\Phi)$ states. Thus, $P_s^{max}(\Phi \,\mathsf{U}^{[t_0,T]} \Psi) = \alpha_s \mathbf{g}^*_0$. □

4.4 Interval Cumulative Reward $R_s^{max}(\mathbb{C}^{\mathrm{I}}\ \Phi)$

The maximal interval cumulative reward $R_s^{max}(C^{\mathrm{I}}\ \Phi)$ can now be handled by combining the cumulative rewards and reachability property. Assume that $I = [t_0, T]$ with $t_0 > 0$ and $T \geq t_0$. As before, we can first compute the cumulative reward between $[t_0, T]$ by $\mathbf{a}_{[t_0,T]} := \textsc{Optimal}(\mathcal{C}, \mathbf{r}|_\Phi, t_0, T, \mathbf{0})$ (see (6)). So $\mathbf{a}_{[t_0,T]}$ is the maximal cumulative reward between $[t_0, T]$, and the problem now is reduced to finding an extension of the policy in $[0, t_0)$ such that $\mathbf{g}^*_0 = \max_{\pi \in \mathcal{M}} \left(\mathbf{V}^\pi_{t_0} \mathbf{a}_{[t_0,T]} \right)$, which can be seen as reachability probability with terminal condition $\mathbf{a}_{[t_0,T]}$. This value can be computed by $\textsc{Optimal}(\mathcal{C}, \mathbf{0}, 0, t_0, \mathbf{a}_{[t_0,T]})$.

4.5 Instantaneous Reward $R_s^{max}(\mathbb{I}^t\ \Phi)$

Interestingly, the maximal instantaneous reward $\sup_\pi(\mathbf{p}_t^\pi \cdot \mathbf{r}|_\Phi)$ can be obtained directly by $\textsc{Optimal}(\mathcal{C}, \mathbf{0}, 0, t, \mathbf{r}|_\Phi)$. Intuitively, we have a terminal condition given by the reward vector $\mathbf{r}$, and afterwards, it behaves very similar to the same as probabilistic reachability for intervals of the form $[t, t]$.

We have shown that the CSL model checking problem reduces to the procedure $\textsc{Optimal}(\mathcal{C}, \mathbf{r}, t_0, T, \mathbf{g}_T)$. By Theorem 1, an optimal policy existis. Thus, the established connection to the paper by Miller gives another very important implication: namely the existence of finite memory schedulers (each for a nested state subformula) for the CSL formula.

5 Computational Approaches

We now present an improved approach for approximating OPTIMAL such that the error of the final result can be adaptively controlled. It is based on uniformization [16] for CTMCs and its recent extension to CTMDPs with rewards [11], which, in our notation, treats the approximation of $R_s^{max}(\mathbb{C}^{[0,T]}\ true)$.

The optimal policy $\mathbf{g}_t$ and vector are approximated from T backwards to 0 or t_0, starting with some vector $\mathbf{g}_T$ which is known exactly or for which bounds $\underline{\mathbf{g}}_T \le \mathbf{g}_T \le \overline{\mathbf{g}}_T$ are known. Observe that for a fixed $\mathbf{d}$ in $(t-\delta, t]$ we can compute $\mathbf{g}_{t-\delta}$ from $\mathbf{g}_t$ as

$$\mathbf{g}_{t-\delta}^{\mathbf{d}} = e^{\delta \mathbf{Q}^{\mathbf{d}}}\mathbf{g}_t + \int_{\tau=0}^{\delta} e^{\tau \mathbf{Q}^{\mathbf{d}}}\mathbf{r}\,\mathrm{d}\tau = \sum_{k=0}^{\infty}\frac{\left(\mathbf{Q}^{\mathbf{d}}\delta\right)^k}{k!}\mathbf{g}_t + \int_{\tau=0}^{\delta}\sum_{k=0}^{\infty}\frac{\left(\mathbf{Q}^{\mathbf{d}}\tau\right)^k}{k!}\mathbf{r}\,\mathrm{d}\tau. \quad (8)$$

We now solve (8) via uniformization [16] and show afterwards how upper and lower bounds for the optimal gain vector can be computed. Let $\alpha_{\mathbf{d}} = \max_{i\in\mathcal{S}}\left(|\mathbf{Q}^{\mathbf{d}}(i,i)|\right)$ and $\alpha = \max_{\mathbf{d}\in\mathcal{D}}(\alpha_{\mathbf{d}})$. Then we can define the following two stochastic matrices for every decision vector $\mathbf{d}$:

$$\mathbf{P}^{\mathbf{d}} = \mathbf{Q}^{\mathbf{d}}/\alpha_{\mathbf{d}} + \mathbf{I} \text{ and } \bar{\mathbf{P}}^{\mathbf{d}} = \mathbf{Q}^{\mathbf{d}}/\alpha + \mathbf{I}. \quad (9)$$

Define the following function to determine the Poisson probabilities in the uniformization approach.

$$\beta(\alpha\delta, k) = e^{-\alpha\delta}\frac{(\alpha\delta)^k}{k!} \text{ and } \zeta(\alpha\delta, K) = \left(1 - \sum_{l=0}^{K}\beta(\alpha\delta, l)\right). \quad (10)$$

Eqns. (9) and (10), combined with the uniformization approach (8), can be used to derive (see [11]) the following sequences of vectors:

$$\mathbf{g}_{t-\delta}^{\mathbf{d}} = \sum_{k=0}^{\infty}\left(\mathbf{P}^{\mathbf{d}}\right)^k\left(\beta(\alpha_{\mathbf{d}}\delta, k)\mathbf{g}_t + \frac{\zeta(\alpha_{\mathbf{d}}\delta, k)}{\alpha_{\mathbf{d}}}\mathbf{r}\right). \quad (11)$$

Assume that bounds $\underline{\mathbf{g}}_t \le \mathbf{g}_t^* \le \overline{\mathbf{g}}_t$ are known and define

$$\begin{aligned}
&\underline{\mathbf{v}}^{(k)} = \mathbf{P}^{\mathbf{d}_t}\underline{\mathbf{v}}^{(k-1)},\ \underline{\mathbf{w}}^{(k)} = \mathbf{P}^{\mathbf{d}_t}\underline{\mathbf{w}}^{(k-1)} \text{ and}\\
&\overline{\mathbf{v}}^{(k)} = \max_{\mathbf{d}\in\mathcal{D}}\left(\bar{\mathbf{P}}^{\mathbf{d}}\overline{\mathbf{v}}^{(k-1)}\right),\ \overline{\mathbf{w}}^{(k)} = \max_{\mathbf{d}\in\mathcal{D}}\left(\bar{\mathbf{P}}^{\mathbf{d}}\overline{\mathbf{w}}^{(k-1)}\right)\\
&\text{with } \underline{\mathbf{v}}^{(0)} = \underline{\mathbf{g}}_t,\ \overline{\mathbf{v}}^{(0)} = \overline{\mathbf{g}}_t,\ \underline{\mathbf{w}}^{(0)} = \overline{\mathbf{w}}^{(0)} = \mathbf{r}.
\end{aligned} \quad (12)$$

If not stated otherwise, we compute $\underline{\mathbf{v}}^k, \underline{\mathbf{w}}^{(k)}$ with $\mathbf{P}^{\mathbf{d}_t}$ where $\mathbf{d}_t$ is the lexicographically smallest vector from $\mathcal{F}_{n+1}(\underline{\mathbf{g}}_t)$. Observe that $\underline{\mathbf{v}}^{(k)}, \underline{\mathbf{w}}^{(k)}$ correspond to a concrete policy that uses decision vector $\mathbf{d}_t$ in the interval $(t-\delta, t]$. Vectors $\overline{\mathbf{v}}^{(k)}, \overline{\mathbf{w}}^{(k)}$ describe some strategy where the decisions depend on the number of transitions which is an ideal case that cannot be improved by any realizable policy. Notice that for zero rewards for probabilistic reachability, we have $\underline{\mathbf{w}}^{(0)} = \overline{\mathbf{w}}^{(0)} = \mathbf{r} = \mathbf{0}$. From the known bounds for $\mathbf{g}_t^*$, new bounds for $\mathbf{g}_{t-\delta}^*$ can then be computed as follows (see [11, Theorem 3]):

$$\begin{array}{l}
\underline{\mathbf{g}}_{t-\delta}^{K} = \sum\limits_{k=0}^{K} \left(\beta(\alpha_{\mathbf{d}}\delta, k)\underline{\mathbf{v}}^{(k)} + \frac{\zeta(\alpha_{\mathbf{d}}\delta,k)}{\alpha_{\mathbf{d}}}\underline{\mathbf{w}}^{(k)}\right) + \zeta(\alpha_{\mathbf{d}}\delta, K)\min_{s\in\mathcal{S}}\left(\underline{\mathbf{v}}^{(K)}(s)\right)\mathbb{1} + \\
\quad \left(\delta\zeta(\alpha_{\mathbf{d}}\delta, K) - \frac{K+1}{\alpha_{\mathbf{d}}}\zeta(\alpha_{\mathbf{d}}\delta, K+1)\right)\min_{s\in\mathcal{S}}\left(\underline{\mathbf{w}}^{(K)}(s)\right)\mathbb{1} \qquad \leq \\
\mathbf{g}_{t-\delta}^{*} \leq \sum\limits_{k=0}^{K} \left(\beta(\alpha\delta, k)\overline{\mathbf{v}}^{(k)} + \frac{\zeta(\alpha\delta,k)}{\alpha}\overline{\mathbf{w}}^{(k)}\right) + \zeta(\alpha\delta, K)\max_{s\in\mathcal{S}}\left(\overline{\mathbf{v}}^{(K)}(s)\right) \qquad + \\
\quad \left(\delta\zeta(\alpha\delta, K) - \frac{K+1}{\alpha}\zeta(\alpha\delta, K+1)\right)\max_{s\in\mathcal{S}}\left(\overline{\mathbf{w}}^{(K)}(s)\right)\mathbb{1} \quad = \quad \overline{\mathbf{g}}_{t-\delta}^{K}.
\end{array} \tag{13}$$

where $\mathbb{1}$ is a column vector of ones with length n. Before we formulate an algorithm based on the above equation, we analyze the spread of the bounds. If $\underline{\mathbf{g}}_t$ and $\overline{\mathbf{g}}_t$ are upper and lower bounding vectors used for the computation of $\underline{\mathbf{g}}_{t-\delta}^{K}$ and $\overline{\mathbf{g}}_{t-\delta}^{K}$, then $\|\overline{\mathbf{g}}_t - \underline{\mathbf{g}}_t\| \leq \|\overline{\mathbf{g}}_{t-\delta}^{K} - \underline{\mathbf{g}}_{t-\delta}^{K}\|$ and the additional spread results from the truncation of the Poisson probabilities

$$\begin{aligned}
\varepsilon_{trunc}(t, \delta, K) = {} & \zeta(\alpha\delta, K)\max_{i\in\mathcal{S}}\left(\overline{\mathbf{v}}^{(K)}(i)\right) - \zeta(\alpha_{\mathbf{d}}\delta, K)\min_{i\in\mathcal{S}}\left(\underline{\mathbf{v}}^{(K)}(i)\right) \\
& + \left(\delta\zeta(\alpha\delta, K) - \frac{(K+1)\zeta(\alpha\delta, K+1)}{\alpha}\right)\max_{s\in\mathcal{S}}\left(\overline{\mathbf{w}}^{(K)}(s)\right) \\
& - \left(\delta\zeta(\alpha_{\mathbf{d}}\delta, K) - \frac{(K+1)\zeta(\alpha_{\mathbf{d}}\delta, K+1)}{\alpha_{\mathbf{d}}}\right)\min_{s\in\mathcal{S}}\left(\underline{\mathbf{w}}^{(K)}(s)\right)
\end{aligned} \tag{14}$$

and the difference due to the different decisions, denoted by $\varepsilon_{succ}(t, \delta, K) =: \varepsilon^{*}$, is,

$$\varepsilon^{*} = \left\|\sum_{k=0}^{K}\left(\beta(\alpha\delta, k)\overline{\mathbf{v}}^{(k)} + \frac{\zeta(\alpha\delta, t)}{\alpha}\overline{\mathbf{w}}^{(k)} - \beta(\alpha_{\mathbf{d}}\delta, k)\underline{\mathbf{v}}^{(k)} - \frac{\zeta(\alpha_{\mathbf{d}}\delta, t)}{\alpha_{\mathbf{d}}}\underline{\mathbf{w}}^{(k)}\right)\right\| \tag{15}$$

where $\mathbf{d}$ is the decision vector chosen by the selection procedure using $\underline{\mathbf{g}}_t$. As shown in [11] the local error of a step of length δ is in $O(\delta^2)$ such that theoretically the global error goes to 0 for $\delta \to 0$. Observe that $\varepsilon_{trunc}(t, \delta, K) \leq \varepsilon_{trunc}(t, \delta, K + 1)$, $\varepsilon_{succ}(t, \delta, K) \geq \varepsilon_{succ}(t, \delta, K + 1)$ and

$$\begin{aligned}
\varepsilon(t, \delta, K) \quad &= \varepsilon_{trunc}(t, \delta, K) + \varepsilon_{succ}(t, \delta, K) \leq \\
\varepsilon(t, \delta, K + 1) &= \varepsilon_{trunc}(t, \delta, K + 1) + \varepsilon_{succ}(t, \delta, K + 1).
\end{aligned}$$

With these ingredients we can define an adaptive algorithm that computes $\underline{\mathbf{g}}_{t_0}$, $\overline{\mathbf{g}}_{t_0}$ ($t_0 \leq T$) and a policy π to reach a gain vector of at least $\underline{\mathbf{g}}_{t_0}$ such that $\underline{\mathbf{g}}_{t_0} \leq \mathbf{g}_{t_0}^{*} \leq \overline{\mathbf{g}}_{t_0}$ and $\|\overline{\mathbf{g}}_{t_0} - \underline{\mathbf{g}}_{t_0}\|_{\infty} \leq \varepsilon$ for the given accuracy $\varepsilon > 0$.

Algorithm 2 computes bounds for the gain vector with a spread of less than ε, if the time steps become not too small ($< \delta_{\min}$). Parameter ω determines the fraction of the error resulting from truncation of the Poisson probabilities and $K_{\max}$ defines the number of intermediate vectors that are stored. The decision vector for the interval $(t_i, t_{i-1}]$ is stored in $\mathbf{c}_i$. Observe that $t_i < t_{i-1}$ since the computations in the algorithm start at T and end at t_0. The policy defined by the time point t_i and vectors $\mathbf{c}_i$ guarantees a gain vector which is elementwise larger or equal to $\underline{\mathbf{g}}_{t_0}^{*}$. Parameter $\delta_{\min}$ is used as a lower bound for the time step to avoid numerical underflows. If the Poisson probabilities

Algorithm 2. $\textsc{Uniform}(\mathcal{C}, \mathbf{r}, t_0, T, \underline{\mathbf{g}}_T, \overline{\mathbf{g}}_T, \omega, K_{\max}, \varepsilon)$: Bounding vectors for $\mathbf{g}^*_{t_0}$

```
 1. initialize $i = 0$ and $t = T$ ;
 2. set $stop = false$, $K = 1$ and $\underline{\mathbf{v}}^{(0)} = \underline{\mathbf{g}}_t, \overline{\mathbf{v}}^{(0)} = \overline{\mathbf{g}}_t, \underline{\mathbf{w}}^{(0)} = \overline{\mathbf{w}}^{(0)} = \mathbf{r}$ ;
 3. select $\mathbf{d}_t$ from $\mathcal{F}_{n+1}(\mathbf{g}_t)$ as described and if $i = 0$ let $\mathbf{c}_0 = \mathbf{d}_t$ ;
 4. repeat
 5.   compute $\underline{\mathbf{v}}^{(K)}, \overline{\mathbf{v}}^{(K)}, \underline{\mathbf{w}}^{(K)}, \overline{\mathbf{w}}^{(K)}$ using (12);
 6.   find $\delta = \max\left(\arg\max_{\delta' \in [0,t]}\left(\varepsilon_{trunc}(t, \delta', K) \le \frac{\omega\delta'}{T-t_0}\varepsilon\right), \min(\delta_{\min}, t - t_0)\right)$;
 7.   compute $\varepsilon_{trunc}(t, \delta, K)$ and $\varepsilon_{succ}(t, \delta, K)$ using (14,15) ;
 8.   if $\varepsilon_{trunc}(t, \delta, K) + \varepsilon_{succ}(t, \delta, K) > \frac{T-t+\delta}{T-t_0}\varepsilon$ then
 9.     reduce $\delta$ until
        $\varepsilon_{trunc}(t, \delta, K) + \varepsilon_{succ}(t, \delta, K) \le \frac{T-t+\delta}{T-t_0}\varepsilon$
        or $\delta = \min(\delta_{\min}, t - t_0)$ and set $stop = true$ ;
10.   else
11.     $K = K + 1$;
12. until $stop$ or $K = K_{\max} + 1$ ;
13. compute $\underline{\mathbf{g}}_{t-\delta}$ from $\underline{\mathbf{v}}^{(k)}, \underline{\mathbf{w}}^{(k)}$ and $\overline{\mathbf{g}}_{t-\delta}$ from $\overline{\mathbf{v}}^{(k)}, \overline{\mathbf{w}}^{(k)}$ $(k = 0, \ldots, K)$ using
    (13);
14. if $\mathbf{d}_t \ne \mathbf{c}_i$ then $\mathbf{c}_{i+1} = \mathbf{d}_t$, $t_i = t - \delta$ and $i = i + 1$ ;
15. if $t - t_0 = \delta$ then terminate else go to 2. with $t = t - \delta$ ;
```

are computed with the algorithm from [17], then all computations are numerically stable and use only positive values. A non-adaptive version of the algorithm can be realized by fixing the number of iterations used in the loop between 4. and 12.

To verify a property that requires a reward to be smaller than some threshold value, the computed upper bound has to be smaller than the threshold. If the lower bound is larger than the required value, then the property is disproved, if the threshold lies between lower and upper bound, no decision about the property is possible.

6 Case Studies

We implemented our model checking algorithm in an extension of the probabilistic model checker MRMC [18]. In addition, we implemented a method to compute long-run average state probabilities [3]. The implementation is written in C, using sparse matrices. Parallelism is not exploited. All experiments are performed on an Intel Core 2 Duo P9600 with 2.66 GHz and 4 GB of RAM running on Linux.

6.1 Introductory Example

We consider a simple example taken from [19], which is shown in Figure 1. We consider a single atomic proposition s_4 which holds only in state s_4.

First we analyze the property $\mathbb{P}_{<x}(\Diamond^{[0,T]} s_4)$ for state s_1. In this case, state s_4 is made absorbing by removing the transition from s_4 to s_1 (shown as a dashed line in the figure), as discussed in Subs. 4.3. Table 1 contains the results and efforts to compute the maximal reachability probabilities for $T = 4$ and 7 with the adaptive and non-adaptive variant of the uniformization approach. The time usage is given in seconds. It can be seen that the adaptive version is much more efficient and should be the method of choice in this example. The value of ε that is required to prove $\mathbb{P}_{<x}(\Diamond^{[0,T]} s_4)$ depends on x. E.g., if $T = 4$ and $x = 0.672$, then $\varepsilon = 10^{-4}$ is sufficient whereas $\varepsilon = 10^{-3}$ would not allow one to prove or disprove the property.

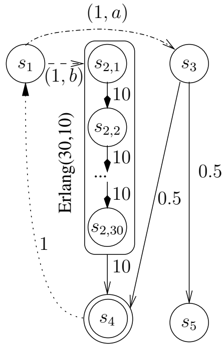


Fig. 1. A CTMDP

Table 1. Bounds for the probability of reaching s_4 in $[0, T]$, i.e., $P^{max}_{s_1}(\Diamond^{[0,T]} s_4)$

		Uniformization $K = 5$					Uniformization $K_{\max} = 20, \omega = 0.1$				
T	ε	lower	upper	steps	iter	time	lower	upper	steps	iter	time
4.0	10^{-4}	0.671701	0.671801	720	3600	0.03	0.671772	0.671803	211	774	0.02
4.0	10^{-5}	0.671771	0.671781	5921	29605	0.10	0.671778	0.671781	2002	5038	0.09
4.0	10^{-6}	0.671778	0.671779	56361	281805	0.87	0.671778	0.671779	19473	40131	0.63
7.0	10^{-4}	0.982746	0.982846	1283	6415	0.04	0.982836	0.982846	364	1333	0.04
7.0	10^{-5}	0.982835	0.982845	10350	51750	0.22	0.982844	0.982845	3463	8098	0.19
7.0	10^{-6}	0.982844	0.982845	97268	486340	1.64	0.982845	0.982845	33747	68876	1.50

Table 2. Bounds for reaching s_4 in $[3, 7]$, i.e., $P^{max}_{s_1}(\Diamond^{[t_0,T]} s_4)$

	$\varepsilon = 1.0e-3$				$\varepsilon = 6.0e-4$			
ε_1	time bounded prob.		$iter_1$	$iter_2$	time bounded prob.		$iter_1$	$iter_2$
$9.0e-4$	0.97170	0.97186	207	90	–	–	–	–
$5.0e-4$	0.97172	0.97186	270	89	0.97176	0.97185	270	93
$1.0e-4$	0.97175	0.97185	774	88	0.97178	0.97185	774	91
$1.0e-5$	0.97175	0.97185	5038	88	0.97179	0.97185	5038	91

To compute the result for $\mathbb{P}_{<x}(\Diamond^{[t_0,T]} s_4)$, the two step approach is used. We consider the interval $[3, 7]$. Thus, in a first step the vector $\mathbf{a}_{[3,7]}$ is computed from the CTMDP where s_4 is made absorbing. Then the resulting vectors $\underline{\mathbf{g}}_3$ and $\overline{\mathbf{g}}_3$ are used as terminal conditions to compute $\underline{\mathbf{g}}_0$ and $\overline{\mathbf{g}}_0$ from the original process including the transition between s_4 and s_1. Apart from the final error bound ε for the spread between $\underline{\mathbf{g}}_0$ and $\overline{\mathbf{g}}_0$, an additional error bound ε_1 $(< \varepsilon)$ has to be defined which defines the spread between $\underline{\mathbf{g}}_3$ and $\overline{\mathbf{g}}_3$. Table 2 includes some results for different values of ε and ε_1. The column headed with $iter_i$ $(i = 1, 2)$ contains the number of iterations of the i-th phase. It can be seen that for this example, the first phase requires more effort such that ε_1 should be chosen only slightly smaller than ε to reduce the overall number of iterations.

Here, it is important to take time-dependent policies to arrive at truly maximal reachability probabilities. The maximal value obtainable for time-abstract policies (using a recent algorithm for CTMDPs [6, 18]) are 0.584284 (versus 0.6717787) for a time bound of 4.0, and 0.9784889 (versus 0.9828449) for a time bound of 7.0.

6.2 Work Station Cluster

As a more complex example, we consider a fault-tolerant workstation cluster (FTWC), in the form considered in [18]. Time bounded reachability analysis for this model was thus far restricted to time-abstract policies [18], using a dediated algorithm for uniform CTMDPs [5]. In a uniform CTMDP (including the one studied here) rate sums are identical across states and nondeterministic choices, and this can be exploited in the algorithm. The general design of the workstation cluster is shown in Fig. 2. It consists of two sub-clusters which are connected via a backbone. There are N workstations in each sub-cluster which are connected together in a star-topology with a switch as central node. The switches provide additionally the interface to the backbone. Each of the components in the fault-tolerant workstation cluster can break down (fail) with a given rate and then needs to be repaired before becoming available again. There is a single repair unit for the entire cluster, not depicted in the figure, which is only capable of repairing one failed component at a time, with a rate depending on the component. When multiple components are down, there is a non-deterministic decision to be taken which of the failed components is to be repaired next.

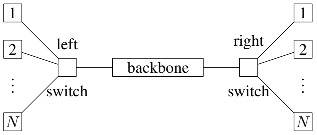


Fig. 2. FTWC structure

We say that our system provides *premium* service whenever at least N workstations are operational. These workstations have to be connected to each other via operational switches. When the number of operational workstations in one sub-cluster is below N, premium quality can be ensured by an operational backbone under the condition that there are at least N operational workstations in total. We consider these properties:

$P1$: Probability to reach non-premium service within time T: $\mathbb{P}_{<x}(\Diamond^{[0,T]}\neg premium)$,
$P2$: Steady-state probability of having non-premium service: $\mathbb{S}_{<x}(\neg premium)$,
$P3$: Steady-state probability of being in a state where the probability to reach non-premium service within time T is above $\frac{1}{2}$: $\mathbb{S}_{<x}(\neg\mathbb{P}_{<\frac{1}{2}}(\Diamond^{[0,T]}\neg premium))$.

Results and statistics are reported in Table 3. For $P1$, we also give numbers for time-abstract policy-based computation exploiting model uniformity [5]. We chose $\varepsilon = 10^{-6}$ and $K_{\max} = 70$. As we see, for $P1$ the probabilities obtained using time-abstract and general policies agree up to ε, thus time-abstract policies seem sufficient to obtain maximal reachability probabilities for this model and property, opposed to the previous example. Our runtime requirements are higher than what is needed for the time-abstract policy class, if exploiting uniformity [5]. Without uniformity exploitation [6], the time-abstract computation times are worse by a factor of 100 to 100,000 compared to our analysis (yielding the same probability result, not shown in the table). However, even for the largest models and time bounds considered, we were able to obtain precise

Table 3. Statistics for the FTWC analysis. For $N = 16$, $N = 64$ and $N = 128$, the state space cardinality is $10130, 151058$ and 597010, respectively.

		P1		P1 time-abstract		P2	P3	
$\downarrow N$	$T_{\rightarrow}$	500h	5000h	500h	5000h		500h	5000h
16	time	1s	9s	0s	1s	0s	1s	9s
	prob.	0.0381333	0.3243483	0.0381323	0.3243474	0.0003483	0.0003483	0.0003526
64	time	21s	3m 28s	3s	7s	14s	33s	3m 31s
	prob.	0.1228243	0.7324406	0.1228233	0.7324401	0.0012808	0.0018187	1.0
128	time	2m 46s	34m 5s	13s	40s	1m 30s	4m 8s	35m 9s
	prob.	0.1837946	0.8698472	0.1837937	0.8698468	0.0020517	0.0037645	1.0

results within reasonable time, which shows the practical applicability of the method. Long-run properties $P2$ and nested variation $P3$ can be handled in a similar amount of time, compared to $P1$.

6.3 Further Empirical Evaluation

Further empirical evaluations can be found at

`http://depend.cs.uni-saarland.de/tools/ctmdp.`

The results are generally consistent with the above experiments. As an example, Table 4 lists some runtimes for the European Train Control System (ETCS) case [20]. Details for the model can be found on the website. The property considered is $P_{<x}(\Diamond^{[0,T]}\,\mathit{unsafe})$, corresponding to the maximal probability that a train must break within T hours of operation. The model consists of "#tr." trains, that are affected by failures. Failure delay distributions are given by Erlang distributions with "#ph." phases. As can be seen, the algorithm for time dependent scheduler analysis is slower than the simpler time-independent analysis, but scales rather smoothly.

Table 4. ETCS Runtimes

			time-dep.		time-abs.	
#tr.	#ph.	#states	10h	180h	10h	180h
3	5	21722	5s	1m 22s	2s	22s
3	10	56452	14s	3m 41s	4s	1m 1s
4	5	15477	4s	59s	1s	16s
4	10	59452	15s	4m 2s	5s	1m 8s

7 Related Work

Our paper builds on the seminal paper of Miller [12]: the problem studied there can be considered as the reward operator $\mathbb{C}^{[0,T]}_{[0,x]}(\mathit{true})$. Time-bounded reachability for CTMDPs in the context of model checking has been studied, restricted to uniform CTMDPs and for a restricted, time-abstract, class of policies [5]. These results have later been extended to non-uniform stochastic games [6]. Time-abstract policies are strictly less powerful than time-dependent ones [5], considered here and in [7].

Our logic is rooted in [15]. Restricting to CTMCs with or without rewards, the semantics coincides with the standard CSL semantics, as in [15]. However, it is interesting to note that our semantics is defined without refering to *timed paths*, in contrast to established work (e.g. [2]). This twist enables a drastically simplified presentation. The logic in [15] has a more general probabilistic operator of the form $\mathbb{P}_{\mathtt{J}}(\Phi\ \mathsf{U}^{\mathtt{I}}_{\mathtt{K}}\Psi)$ which allows one to constrain the reward accumulated prior to satisfying Ψ to lie in the interval $\mathtt{K}$. Our framework can not be applied directly to those properties, which we consider as interesting future work.

So far, the common approach to obtain the optimal gain vector proceeds via an approximate *discretization* using a fixed interval of length h, instead of computing t'' as in Algorithm 1. As shown in [12] and also for a slightly different problem in [21], this approach converges towards the optimal solution for $h \rightarrow 0$. Let λ be the maximal exit rate in matrix $\mathbf{Q}^{\mathbf{d}}$ for some decision vector $\mathbf{d}$. For probabilistic reachability with interval $[0, T]$, namely $\mathbb{P}^{max}_{s}(\Diamond^{[0,T]}\Phi)$, the number of steps is shown to be bounded by $\mathcal{O}((\lambda T)^2/\varepsilon)$ in [9], to guarantee global accuracy ε. Recently, this bound was further improved to $\mathcal{O}(\lambda T/\varepsilon)$ [10].

The approach presented here is much more efficient than the discretization technique in [9, 10]. As an example we reconsider our introductory example. Discretization requires $iter \approx \lambda T/\varepsilon$ iterations to reach a global accuracy of ε. For $\lambda = 10$, $T = 4$ and $\varepsilon = 0.001$, uniformization requires 201 iterations whereas the discretization approach would need about $40,000$ iterations. For $T = 7$ and for $\varepsilon = 10^{-6}$, uniformization needs $68,876$ iterations, whereas discretization requires about $70,000,000$ iterations to arrive at comparable accuracy, thus the difference is a factor of 1000.

8 Conclusions

The paper presents a new approach to model checking CSL formulae over CTMDPs. A computational approach based on uniformization enables the computation of time bounded reachability probabilities and rewards accumulated during some finite interval. It is shown how these values can be used to prove or disprove CSL formulae. The proposed uniformization technique allows one to compute results with a predefined accuracy that can be chosen with respect to the CSL formula that has to be proved. The improvements resemble the milestones in approximate CTMC model checking research, which was initially resorting to discretization [13], but got effective only through the use of uniformization [2].

The uniformization algorithm approximates, apart from the bounds for the gain vector, also a policy that reaches the lower bound gain vector. This policy is not needed for model checking a CSL formula but it is, of course, of practical interest since it describes a control strategy which enables a system to obtain the required gain—up to ε.

Finally, we note that the current contribution of our paper can be combined with three-value CSL model checking by Katoen *et al* [22], to attenuate the well-known robustness problem of nested formulae in stochastic model checking. For the inner probabilistic state formulae, our algorithm will compute the corresponding probability—up to ε. Using the method in [22] we obtain a three-valued answer, either yes/no, or ”don’t-know”. Then, if we come to the outermost probabilistic operator, we will compute an upper and lower bound of the probabilities. We get a three-valued answer again. In case of a don’t-know answer for a state we want to check, we can reduce ε to decrease the number of don’t-know states for the inner probabilistic formulae.

Acknowledgement. Ernst Moritz Hahn and Holger Hermanns are partially supported by the DFG/NWO Bilateral Research Programme ROCKS, by the DFG as part of SFB/TR 14 AVACS, and by the EC FP-7 programme under grant agreement no. 214755 - QUASIMODO. Lijun Zhang is partially supported by MT-LAB, a VKR Centre of Excellence.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Model-checking continuous-time Markov chains. ACM Trans. Comput. Log. 1, 162–170 (2000)
2. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. 29, 524–541 (2003)
3. Howard, R.A.: Dynamic Programming and Markov Processes. John Wiley and Sons, Inc., Chichester (1960)
4. Bertsekas, D.P.: Dynamic Programming and Optimal Control. Athena Scientific, Belmont (2005)
5. Baier, C., Hermanns, H., Katoen, J.P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theor. Comput. Sci. 345, 2–26 (2005)
6. Brázdil, T., Forejt, V., Krcal, J., Kretínský, J., Kucera, A.: Continuous-time stochastic games with time-bounded reachability. In: FSTTCS. LIPIcs, vol. 4, pp. 61–72 (2009)
7. Neuhäußer, M.R., Stoelinga, M., Katoen, J.-P.: Delayed nondeterminism in continuous-time markov decision processes. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 364–379. Springer, Heidelberg (2009)
8. Rabe, M., Schewe, S.: Finite optimal control for time-bounded reachability in CTMDPs and continuous-time Markov games. In: CoRR, pp. 1004–4005 (2010)
9. Neuhäußer, M.R., Zhang, L.: Time-bounded reachability probabilities in continuous-time Markov decision processes. In: QEST (2010)
10. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Computing maximum reachability probabilities in Markovian timed automata. Technical report, RWTH Aachen (2010)
11. Buchholz, P., Schulz, I.: Numerical analysis of continuous time Markov decision processes over finite horizons. Computers & Operations Research 38, 651–659 (2011)
12. Miller, B.L.: Finite state continuous time Markov decision processes with a finite planning horizon. SIAM Journal on Control 6, 266–280 (1968)
13. Baier, C., Katoen, J.-P., Hermanns, H.: Approximate symbolic model checking of continuous-time markov chains (Extended abstract). In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, p. 146. Springer, Heidelberg (1999)
14. Lembersky, M.R.: On maximal rewards and ε-optimal policies in continuous time Markov decision chains. The Annals of Statistics 2, 159–169 (1974)
15. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: On the logical characterisation of performability properties. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, p. 780. Springer, Heidelberg (2000)
16. Gross, D., Miller, D.: The randomization technique as a modeling tool and solution procedure for transient Markov processes. Operations Research 32, 926–944 (1984)
17. Fox, B.L., Glynn, P.W.: Computing Poisson probabilities. Comm. ACM 31, 440–445 (1988)
18. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: QEST, pp. 167–176 (2009)

19. Zhang, L., Neuhäußer, M.R.: Model checking interactive markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)
20. Böde, E., et al.: Compositional performability evaluation for statemate. In: QEST, pp. 167–178 (2006)
21. Martin-Löfs, A.: Optimal control of a continuous-time Markov chain with periodic transition probabilities. Operations Research 15, 872–881 (1967)
22. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007)

Quantitative Synthesis for Concurrent Programs[⋆],[⋆⋆]

Pavol Černý[1], Krishnendu Chatterjee[1], Thomas A. Henzinger[1],
Arjun Radhakrishna[1], and Rohit Singh[2]

[1] IST Austria
[2] IIT Bombay

Abstract. We present an algorithmic method for the quantitative, performance-aware synthesis of concurrent programs. The input consists of a nondeterministic *partial program* and of a *parametric performance model*. The nondeterminism allows the programmer to omit which (if any) synchronization construct is used at a particular program location. The performance model, specified as a weighted automaton, can capture system architectures by assigning different costs to actions such as locking, context switching, and memory and cache accesses. The quantitative synthesis problem is to automatically resolve the nondeterminism of the partial program so that both correctness is guaranteed and performance is optimal. As is standard for shared memory concurrency, correctness is formalized "specification free", in particular as race freedom or deadlock freedom. For worst-case (average-case) performance, we show that the problem can be reduced to 2-player graph games (with probabilistic transitions) with quantitative objectives. While we show, using game-theoretic methods, that the synthesis problem is NEXP-complete, we present an algorithmic method and an implementation that works efficiently for concurrent programs and performance models of practical interest. We have implemented a prototype tool and used it to synthesize finite-state concurrent programs that exhibit different programming patterns, for several performance models representing different architectures.

1 Introduction

A promising approach to the development of correct concurrent programs is *partial program synthesis*. The goal of the approach is to allow the programmer to specify a part of her intent declaratively, by specifying which conditions, such as linearizability or deadlock freedom, need to be maintained. The synthesizer then constructs a program that satisfies the specification (see, for example, [17,16,19]). However, quantitative considerations have been largely missing from previous

[⋆] This work was partially supported by the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 and S11407-N23 (RiSE), the EU NOE Grant ArtistDesign, and a Microsoft faculty fellowship.

[⋆⋆] Full version available at `http://arxiv.org/abs/1104.4306v1`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 243–259, 2011.

frameworks for partial synthesis. In particular, there has been no possibility for a programmer to ask the synthesizer for a program that is not only correct, but also *efficient* with respect to a specific performance model. We show that providing a quantitative performance model that represents the architecture of the system on which the program is to be run can considerably improve the quality and, therefore, potential usability of synthesis.

Motivating Examples. *Example 1.* Consider a *producer-consumer* program, where k producer and k consumer threads access a buffer of n cells. The programmer writes a partial program implementing the procedures that access the buffer as if writing the sequential version, and specifies that at each control location a global lock or a cell-local lock can be taken. It is easy to see that there are at least two different ways of implementing correct synchronization. The first is to use a global lock, which locks the whole buffer. The second is to use cell-local locks, with each thread locking only the cell it currently accesses. The second program allows more concurrent behavior and is better in many settings. However, if the cost of locks is high (relative to the other operations), the global-locking approach is more efficient. In our experiments on a desktop machine, the global-locking implementation out-performed the cell-locking implementation by a factor of 3 in certain settings.

```
1: while(true) {
2:   lver=gver; ldata=gdata;
3:   n  = choice(1..10);
4:   i = 0;
5:   while (i < n) {
6:    work(ldata); i++;
7:   }
8:   if (trylock(lock)) {
9    if (gver==lver) {
10:      gdata = ldata;
11:      gver = lver+1;
12:      unlock(lock);
13:    } else {unlock(lock)}
14:} }
```

Fig. 1. Example 2

Example 2. Consider the program in Figure 1. It uses classic conflict resolution mechanism used for optimistic concurrency. The shared variables are `gdata`, on which some operation (given by the function `work()`) is performed repeatedly, and `gver`, the version number. Each thread has local variables `ldata` and `lver` that store local copies of the shared variables. The data is read (line 2) and operated on (line 6) without acquiring any locks. When the data is written back, the shared data is locked (line 8), and it is checked (using the version number, line 9) that no other thread has changed the data since it has been read. If the global version number has not changed, the new value is written to the shared memory (line 10), and the global version number is increased (line 11). If the global version number has changed, the whole procedure is retried. The number of operations (calls to `work`) performed optimistically without writing back to shared memory can influence the performance significantly. For approaches that perform many operations before writing back, there can be many retries and the performance can drop. On the other hand, if only a few operations are performed optimistically, the data has to be written back often, which also can lead to a performance drop. Thus, the programmer would like to leave the task of finding the optimal number of operations to be performed optimistically to the synthesizer. This is done via the choice statement (line 4).

The Partial Program Resolution Problem. Our aim is to synthesize concurrent programs that are both correct and optimal with respect to a performance model. The input for partial program synthesis consists of (1) a finite-state partial program, (2) a performance model, (3) a model of the scheduler, and (4) a correctness condition. A *partial program* is a finite-state concurrent program that includes nondeterministic choices which the synthesizer has to resolve. A program is *allowed* by a partial program if it can be obtained by resolving the nondeterministic choices. The second input is a *parametric performance model*, given by a weighted automaton. The automaton assigns different costs to actions such as locking, context switching, and memory and cache access. It is a flexible model that allows the assignment of costs based on past sequences of actions. For instance, if a context switch happens soon after the preceding one, then its cost might be lower due to cache effects. Similarly, we can use the automaton to specify complex cost models for memory and cache accesses. The performance model can be fixed for a particular architecture and, hence, need not be constructed separately for every partial program. The third input is the *scheduler*. Our schedulers are state-based, possibly probabilistic, models which support flexible scheduling schemes (e.g., a thread waiting for a long time may be scheduled with higher probability). In performance analysis, average-case analysis is as natural as worst-case analysis. For the average-case analysis, probabilistic schedulers are needed. The fourth input, the *correctness condition*, is a safety condition. We use "specification-free" conditions such as data-race freedom or deadlock-freedom. The output of synthesis is a program that is (a) allowed by the partial program, (b) correct with respect to the safety condition, and (c) has the best performance of all the programs satisfying (a) and (b) with respect to the performance and scheduling models.

Quantitative Games. We show that the partial program resolution problem can be reduced to solving *imperfect information* (stochastic) graph games with quantitative (limit-average or mean-payoff) objectives. Traditionally, imperfect information graph games have been studied to answer the question of existence of general, *history-dependent* optimal strategies, in which case the problem is undecidable for quantitative objectives [8]. We show that the partial program resolution problem gives rise to the question (not studied before) whether there exist *memoryless* optimal strategies (i.e. strategies that are independent of the history) in imperfect information games. We establish that the memoryless problem for imperfect information games (as well as imperfect information stochastic games) is Np-complete, and prove that the partial program resolution problem is Nexp-complete for both average-case and worst-case performance based synthesis. We present several techniques that overcome the theoretical difficulty of Nexp-hardness in cases of programs of practical interest: (1) First, we use a lightweight static analysis technique for efficiently eliminating parts of the strategy tree. This reduces the number of strategies to be examined significantly. We then examine each strategy separately and, for each strategy, obtain a (perfect information) Markov decision process (EDP). For MDPs, efficient strategy improvement algorithms exist, and require solving Markov chains. (2)

Second, Markov chains obtained from concurrent programs typically satisfy certain progress conditions, which we exploit using a forward propagation technique together with Gaussian elimination to solve Markov chains efficiently. (3) Our third technique is to use an abstraction that preserves the value of the quantitative (limit-average) objective. An example of such an abstraction is the classical data abstraction.

Experimental Results. In order to evaluate our synthesis algorithm, we implemented a prototype tool and applied it to four finite-state examples that illustrate basic patterns in concurrent programming. In each case, the tool automatically synthesized the optimal correct program for various performance models that represent different architectures. For the producer-consumer example, we synthesized a program where two producer and two consumer threads access a buffer with four cells. The most important parameters of the performance model are the cost l of locking/unlocking and the cost c of copying data from/to shared memory. If the cost c is higher than l (by a factor 100:1), then the fine-grained locking approach is better (by 19 percent). If the cost l is equal to c, then the coarse-grained locking is found to perform better (by 25 percent). Referring back to the code in Figure 1, for the optimistic concurrency example and a particular performance model, the analysis found that increasing n improves the performance initially, but after a small number of increments the performance started to decrease. We measured the running time of the program on a desktop machine and observed the same phenomenon.

Related Work. Synthesis from specifications is a classical problem [6,7,15]. More recently, sketching, a technique where a partial implementation of a program is given and a correct program is generated automatically, was introduced [17] and applied to concurrent programs [16]. However, none of the above approaches consider performance-aware algorithms for sketching; they focus on qualitative synthesis without any performance measure. We know of two works where quantitative synthesis was considered. In [2,3] the authors study the synthesis of sequential systems from temporal-logic specifications. In [19,5] fixed optimization criteria (such as preferring short atomic sections or fine-grained locks) are considered. Optimizing these measures may not lead to optimal performance on all architectures. None of the cited approaches use the framework of imperfect information games, nor parametric performance models.

2 The Quantitative Synthesis Problem

2.1 Partial Programs

In this section we define threads, partial programs, programs and their semantics. We start with the definitions of guards and operations.

Guards and operations. Let L, G, and I be finite sets of variables (representing local, global (shared), and input variables, respectively) ranging over finite domains. A *term* t is either a variable in L, G, or I, or t_1 *op* t_2, where

t_1 and t_2 are terms and *op* is an operator. Formulas are defined by the following grammar, where t_1 and t_2 are terms and *rel* is a relational operator: $e := t_1 \mathit{rel}\ t_2 \mid e \wedge e \mid \neg e$. *Guards* are formulae over L, G, and I. *Operations* are simultaneous assignments to variables in $L \cup G$, where each variable is assigned a term over L, G, and I.

Threads. A *thread* is a tuple $\langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$, with: (a) a finite set of control locations Q and an initial location q_0; (b) L, G and I are as before; (c) an initial valuation of the variables ρ_0; and (d) a set δ of transition tuples of the form (q, g, a, q'), where q and q' are locations from Q, and g and a are *guards* and *operations* over variables in L, G and I.

The set of locations $\mathit{Sk}(c)$ of a thread $c = \langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$ is the subset of Q containing exactly the locations where δ is non-deterministic, i.e., locations where there exists a valuation of variables in L, G and I, for which there are multiple transitions whose guards evaluate to true.

Partial programs and programs. A *partial program* M is a set of threads that have the same set of global variables G and whose initial valuation of variables in G is the same. Informally, the semantics of a partial program is a parallel composition of threads. The set G represents the shared memory. A *program* is a partial program, in which the set $\mathit{Sk}(c)$ of each thread c is empty. A program P is *allowed* by a partial program M if it can be obtained by removing the outgoing transitions from sketch locations of all the threads of M, so that the transition function of each thread becomes deterministic.

The guarded operations allow us to model basic concurrency constructs such as locks (for example, as variables in G and locking/unlocking is done using guarded operations) and compare-and-set. As partial program defined as a collection of fixed threads, thread creation is not supported.

Semantics. A *transition system* is a tuple $\langle S, A, \Delta, s_0 \rangle$ where S is a finite set of states, A is a finite set of actions, $\Delta \subseteq S \times A \times S$ is a set of transitions and s_0 is the initial state. The semantics of a partial program M is given in terms of a transition system (denoted as $\mathsf{Tr}(M)$). Given a partial program M with n threads, let $\mathcal{C} = \{1, \ldots, n\}$ represent the set of threads of M.

- *State space.* Each state $s \in S$ of $\mathsf{Tr}(M)$ contains input and local variable valuations and locations for each thread in $\mathcal{C}$, and a valuation of the global variables. In addition, it contains a value $\sigma \in \mathcal{C} \cup \{*\}$, indicating which (if any) thread is currently scheduled. The initial state contains the initial locations of all threads and the initial valuations ρ_0, and the value $*$ indicating that no thread is currently scheduled.
- *Transition.* The transition function Δ defines interleaving semantics for partial programs. There are two types of transitions: thread transitions, that model one step of a scheduled thread, and environment transitions, that model input from the environment and the scheduler. For every $c \in \mathcal{C}$, there exists a thread transition labeled c from a state s to a state s' if and only if there exists a transition (q, g, a, q') of c such that (i) $\sigma = c$ in s (indicating that c is scheduled) and $\sigma = *$ in s', (ii) the location of c is q in s and q' in s', (iii) the guard g evaluates to true in s, and (iv) the valuation of

local variables of c and global variables in s is obtained from the valuation of variables in s' by performing the operation a. There is an environment transition labeled c from state s to state s' in $\mathsf{Tr}(M)$ if and only if (i) the value σ in s is $*$ and the value σ in s' is c and (ii) the valuations of variables in s and s' differ only in input variables of the thread c.

2.2 The Performance Model

We define a flexible and expressive performance model via a weighted automaton that specifies costs of actions. A *performance automaton* W is a tuple $W = (Q_W, \Sigma, \delta, q_0, \gamma)$, where Q_W is a set of states, Σ is a finite alphabet, $\delta : Q_W \times \Sigma \to Q_W$ is a transition relation, q_0 is an initial location and γ is a cost function $\gamma : Q_W \times \Sigma \times Q_W \to \mathbb{Q}$. The labels in Σ represent (concurrency related) actions that incur costs, while the values of the function γ specify these costs. The symbols in Σ are matched with the actions performed by the system to which the performance measures are applied. A special symbol $ot \in \Sigma$ denotes that none of the tracked actions occurred. The costs that can be specified in this way include for example the cost of locking, the access to the (shared) main memory or the cost of context switches.

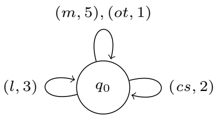


Fig. 2. Perf. aut

An example specification that uses the costs mentioned above is the automaton W in Figure 2. The automaton describes the costs for locking (l), context switching (cs), and main memory access (m). Specifying the costs via a weighted automaton is more general than only specifying a list of costs. For example, automaton based specification enables us to model a cache, and the cost of reading from a cache versus reading from the main memory, as shown in Figure 5 in Section 5. Note that the performance model can be fixed for a particular architecture. This eliminates the need to construct a performance model for the synthesis of each partial program.

2.3 The Partial Program Resolution Problem

Weighted probabilistic transition system (WPTS). A *probabilistic transition system* (PTS) is a generalization of a transition system with a probabilistic transition function. Formally, let $\mathcal{D}(S)$ denote the set of probability distributions over S. A PTS consists of a tuple $\langle S, A, \Delta, s_0 \rangle$ where S, A, s_0 are defined as for transition systems, and $\Delta : S \times A \to \mathcal{D}(S)$ is probabilistic, i.e., given a state and an action, it returns a probability distribution over successor states. A WPTS consists of a PTS and a weight function $\gamma : S \times A \times S \to \mathbb{Q} \cup \{\infty\}$ that assigns costs to transitions. An *execution* of a weighted probabilistic transition system is an infinite sequence of the form $(s_0 a_0 s_1 a_2 \ldots)$ where $s_i \in S$, $a_i \in A$, and $\Delta(s_i, a_i)(s_{i+1}) > 0$, for all $i \geq 0$. We now define boolean and quantitative objectives for WPTS.

Safety objectives. A *safety objective* Safety_B is defined by a set B of "bad" states and requires that states in B are never present in an execution. An execution $e = (s_0a_0s_1a_2\ldots)$ is *safe* (denoted by $e \in \mathrm{Safety}_B$) if $s_i \notin B$, for all $i \geq 0$.

Limit-average and limit-average safety objectives. The *limit-average* objective assigns a real-valued quantity to every infinite execution e. We have $\mathrm{LimAvg}_\gamma(s_0a_0s_1a_1s_2\ldots) = \limsup_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n} \gamma((s_i, a, s_{i+1}))$ if there are no infinite cost transitions, and ∞ otherwise. The *limit-average safety* objective (defined by γ and B) is a *lexicographic* combination of a safety and a limit-average objective: $\mathrm{LimAvg}_\gamma^B(e) = \mathrm{LimAvg}_\gamma(e)$, if $e \in \mathrm{Safety}_B$, and ∞ otherwise. Limit-average safety objectives can be reduced to limit-average objectives by making states in B absorbing (states with only self-loop transitions) and assigning the self-loop transitions the weight ∞.

Value of WPTS. Given a WPTS T with weight function γ, a *policy* $pf : (S \times A)^* \times S \to A$ is a function that given a sequence of states and actions chooses an action. A policy pf defines a unique probability measure on the executions and let $E^{pf}(\cdot)$ be the associated expectation measure. Given a WPTS T with weight function γ, and a policy pf, the value $Val(T, \gamma, \mathrm{Safety}_B, pf)$ is the expected value $E^{pf}(\mathrm{LimAvg}_\gamma^B)$ of the limit-average safety objective. The *value* of the WPTS is the supremum value over all policy functions, i.e., $Val(T, \gamma, \mathrm{Safety}_B) = \sup_{pf} Val(T, \gamma, \mathrm{Safety}_B, pf)$.

Schedulers. A *scheduler* has a finite set of internal memory states Q_{Sch}. At each step, it considers all the active threads and chooses one either (i) nondeterministically (nondeterministic schedulers) or (ii) according to a probability distribution (probabilistic schedulers), which depends on the current internal memory state.

Composing a program with a scheduler and a performance model. In order to evaluate the performance of a program, we need to take into account the scheduler and the performance model. Given a program P, a scheduler Sch, and a performance model W, we construct a WPTS, denoted $\mathsf{Tr}(P, \mathrm{Sch}, W)$, with a weight function γ as follows. A state s of $\mathsf{Tr}(P, \mathrm{Sch}, W)$ is composed of a state of the transition system of P ($\mathsf{Tr}(P)$), a state of the scheduler Sch and a state of the performance model W. The transition function matches environment transitions of $\mathsf{Tr}(P)$ with the scheduler transitions (which allows the scheduler to schedule threads) and it matches thread transitions with the performance model transitions. The weight function γ assigns costs to edges as given by the weighted automaton W. Furthermore, as the limit average objective is defined only for infinite executions, for terminating safe executions of the program we add an edge back to the initial state. The value of the limit average objective function of the infinite execution is the same as the average over the original finite execution. Note that the performance model can specify a locking cost, while the program model does not specifically mention locking. We thus need to specifically designate which shared memory variables are used for locking.

Correctness. We restrict our attention to safety conditions for correctness. We illustrate how various correctness conditions for concurrent programs can be modelled as Safety objectives: (a) *Data-race freedom.* Data-races occur when two or more threads access the same shared memory location and one of the accesses is a write access. We can check for absence of data-races by denoting as unsafe states those in which there exist two enabled transitions (with at least one being a write) accessing a particular shared variable, from different threads. (b) *Deadlock freedom.* One of the major problems of synchronizing programs using blocking primitives such as locks is that deadlocks may arise. A deadlock occurs when two (or more) threads are waiting for each other to finish an operation. Deadlock-freedom is a safety property. The unsafe states are those where there exists two or more threads with each one waiting for a resource held by the next one.

Value of a program and of a partial program. For P, Sch, W as before and Safety_B is a safety objective, we define the value of the program using the composition of P, Sch and W as: $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B) = \mathit{Val}(\mathsf{Tr}(P, \mathrm{Sch}, W), \gamma, \mathrm{Safety}_B)$. For be a partial program M, let $\mathcal{P}$ be the set of all allowed programs. The value of M, $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) = \min_{P \in \mathcal{P}} \mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$.

Partial Program resolution problem. The central technical questions we address are as follows: (1) The *partial program resolution optimization problem* consists of a partial program M, a scheduler Sch, a performance model W and a safety condition Safety_B, and asks for a program P allowed by the partial program M such that the value $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is minimized. Informally, we have: (i) if the value $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is ∞, then no safe program exists; (ii) if it is finite, then the answer is the optimal safe program, i.e., a correct program that is optimal with respect to the performance model. The *partial program resolution decision problem* consists of the above inputs and a rational threshold λ, and asks whether $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) \leq \lambda$.

3 Quantitative Games on Graphs

Games for synthesis of controllers and sequential systems from specifications have been well studied in literature. We show how the partial program resolution problems can be reduced to quantitative imperfect information games on graphs. We also show that the arising technical questions on game graphs is different from the classical problems on quantitative graph games.

3.1 Imperfect Information Games for Partial Program Resolution

An *imperfect information stochastic game graph* is a tuple $\mathcal{G} = \langle S, A, \mathrm{En}, \Delta, (S_1, S_2), O, \eta, s_0 \rangle$, where S is a finite set of states, A is a finite set of actions, $\mathrm{En} : S \to 2^A \setminus \emptyset$ is a function that maps every state s to the non-empty set of actions enabled at s, and s_0 is an initial state. The transition function $\Delta : S \times A \to \mathcal{D}(S)$ is a probabilistic function which maps a state s and an enabled action a to the probability distribution $\Delta(s, a)$ over the successor states. The sets (S_1, S_2) define a partition of S into Player-1 and Player-2 states, respectively; and the function $\eta : S \to O$ maps every state to an observation from the finite observation set O. We refer to these as ImpIn $2\frac{1}{2}$-player game graphs: ImpIn for imperfect information, 2 for the two players and $\frac{1}{2}$ for the probabilistic transitions.

Special cases. We also consider ImpIn 2-player games (no randomness), perfect information games (no partial information), MDPs (only one enabled action for Player 1 states) and Markov chains (only one enabled action for all states) as special cases of ImpIn $2\frac{1}{2}$-player games. For full definitions, see [4].

The informal semantics for an imperfect information game is as follows: the game starts with a token being placed on the initial state. In each step, Player 2 can observe the exact state s in which the token is placed whereas, Player 1 can observe only $\eta(s)$. If the token is in S_1 (resp. S_2), Player 1 (resp. Player 2) chooses an action a enabled in s. The token is then moved to a successor of s based on the distribution $\Delta(s, a)$.

A strategy for Player 1 (Player 2) is a "recipe" that chooses an action for her based on the history of observations (states). Memoryless Player 1 (Player 2) strategies are those which choose an action based only on the current observation (state). We denote the set of Player 1 and Player 2 strategies by Σ and Γ, respectively, and the set of Player 1 and Player 2 memoryless strategies by Σ^M and Γ^M, respectively.

Probability space and objectives Given a pair of Player 1 and Player 2 strategies (σ, τ), it is possible to define a unique probability measure $\Pr^{\sigma,\tau}(\cdot)$ over the set of paths of the game graph. For details, refer to any standard work on $2\frac{1}{2}$-player stochastic games (for example, [18]).

In a graph game, the goal of Player 1, i.e., the *objective* is given as a boolean or quantitative function from paths in the game graph to either $\{0, 1\}$, or $\mathbb{R}$. We consider only the LimAvg-Safety objectives defined in Section 2. Player 1 tries to maximize the expected value of the objective. The *value* of a Player 1 strategy σ is defined as $\mathit{ValGame}(f, \mathcal{G}, \sigma) = \sup_{\tau \in \Gamma} \mathbb{E}^{\sigma,\tau}[f]$ and the value of the game is defined as $\mathit{ValGame}(f, \mathcal{G}) = \inf_{\sigma \in \Sigma} \mathit{ValGame}(f, \mathcal{G}, \sigma)$.

For a more detailed exposition on ImpIn $2\frac{1}{2}$-player graph games and the formal definition of strategies, objectives, and values, see [4].

Decision problems. Given a game graph $\mathcal{G}$, an objective f and a rational threshold $q \in \mathbb{Q}$, the general decision problem (resp. memoryless decision problem) asks if there is a Player 1 strategy (resp. memoryless strategy) σ with $\mathit{ValGame}(f, \mathcal{G}, \sigma) \leq q$. Similarly, the value problem (memoryless value problem) is to compute $\inf_{\sigma \in \Sigma} \mathit{ValGame}(f, \mathcal{G}, \sigma)$ ($\min_{\sigma \in \Sigma^M} \mathit{ValGame}(f, \mathcal{G}, \sigma)$ resp.). Traditional game theory study always considers the general decision problem which is undecidable for limit-average objectives [8] in imperfect information games.

Theorem 1. *[8] The decision problems for* LimAvg *and* LimAvg-Safety *objectives are undecidable for* ImpIn $2\frac{1}{2}$*- and* ImpIn 2*-player game graphs.*

However, we show here that the partial program resolution problems reduce to the memoryless decision problem for imperfect information games.

Theorem 2. *Given a partial program M, a scheduler* Sch*, a performance model W, and a correctness condition ϕ, we construct an exponential-size* ImpIn $2\frac{1}{2}$*-player game graph $\mathcal{G}^p_M$ with a* LimAvg-Safety *objective such that the memoryless value of $\mathcal{G}^p_M$ is equal to* $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety})$.

The proof relies on a construction of a game graph similar to the product of a program, a scheduler and a performance model. Player 2 chooses the thread to be scheduled and Player 1 resolves the nondeterminism when the scheduled thread c is in a location in $Sk(c)$. The crucial detail is that Player 1 can observe only the location of the thread and not the valuations of the variables. This partial information gives us a one-one correspondence between the memoryless strategies of Player 1 and programs allowed by the partial program.

3.2 Complexity of ImpIn Games and Partial-Program Resolution

We establish complexity bounds for the relevant memoryless decision problems and use them to establish upper bounds for the partial program resolution problem. We also show a matching lower bound. First, we state a theorem on complexity of MDPs.

Theorem 3. *[9] The memoryless decision problem for* LimAvg-Safety *objectives can be solved in polynomial time for MDPs.*

Theorem 4. *The memoryless decision problems for* Safety*,* LimAvg*, and* LimAvg-Safety *objectives are* NP*-complete for* ImpIn $2\frac{1}{2}$*- and* ImpIn *2-player game graphs.*

For the lower bound we show a reduction from 3SAT problem and for the upper bound we use memoryless strategies as polynomial witness and Theorem 3 for polynomial time verification procedure.

Remark 1. The NP-completeness of the memoryless decision problems rules out the existence of the classical strategy improvement algorithms as their existence implies existence of randomized sub-exponential time algorithms (using the techniques of [1]), and hence a strategy improvement algorithm would imply a randomized sub-exponential algorithm for an NP-complete problem.

Theorem 5. *The partial-program resolution decision problem is* NEXP*-complete for both nondeterministic and probabilistic schedulers.*

Proof. (a) The NEXP upper bound follows by an exponential reduction to the ImpIn games' memoryless decision problem (Theorem 2), and by Theorem 4.

(b) We reduce the NEXP-hard problem *succinct 3*-SAT (see [14]) to the partial program resolution problem to show NEXP-hardness. The idea is to construct a two thread partial program (shown in Figure 3) where Thread 1 chooses a clause from the formula and Thread 2 will determine the literals in the clause and then, enters an error state if the clause is not satisfied.

Given an instance of succinct 3-SAT over variables $v_1, \ldots v_M$, i.e., a circuit $\mathcal{Q}$ which takes pairs (i, j) and returns the j^{th} literal in the i^{th} clause. Thread 1 just changes the global variable i, looping through all clause indices.

```
GLOBALS: var i;

THREAD 1:
while (true)
     i = (i + 1) mod N;

THREAD 2:
choice: {
     val[v1] = true;
     val[v1] = false;
}
...

while (true)
     l1 = compute_Q(i,1);
     l2 = compute_Q(i,2);
     l3 = compute_Q(i,3);
     if(not (val[l1] ∨
             val[l2] ∨
             val[l3]))
          assert(false);
```

Fig. 3. The reduction of succinct 3-SAT to partial program resolution

Thread 2 will first non-deterministically choose a valuation $\mathcal{V}$ for all literals. It then does the following repeatedly: (a) Read global i, (b) Compute the i^{th} clause by solving the circuit value problem for $\mathcal{Q}$ with $(i, 1)$, $(i, 2)$ and $(i, 3)$ as inputs. This can be done in polynomial time. (c) If the i^{th} clause is not satisfied with the valuation $\mathcal{V}$, it goes to an error state.

To show the validity of the reduction: (i) Given a satisfying valuation for $\mathcal{Q}$, choosing that valuation in the first steps of Thread 2 will obviously generate a safe program. (ii) Otherwise, for every valuation $\mathcal{V}$ chosen in the partial program, there exists a clause (say k) which is not satisfied. We let Thread 1 run till i becomes equal to k and then let Thread 2 run. The program will obviously enter the error state. Note that the result is independent of schedulers (non-deterministic or probabilistic), and performance models (as it uses only safety objectives). □

4 Practical Solutions for Partial-Program Resolution

Algorithm 1. Strategy Elimination

```
Input: $M$: partial program;
  $W$: performance model;
  Sch: scheduler;
  Safety: safety condition
Output: $Candidates$: Strategies
  $StrategySet \leftarrow$ CompleteTree($M$)
  {A complete strategy tree}
  $Candidates \leftarrow \emptyset$
  while $StrategySet \neq \emptyset$ do
    Choose $Tree$ from $StrategySet$
    $\sigma \leftarrow$Root($Tree$)
    if PartialCheck($\sigma$, Safety) then
      $StrategySet =$
        $StrategySet \cup$ children($Tree$)
    if $Tree$ is singleton then
      $Candidates = Candidates \cup \{\sigma\}$
  return $Candidates$
```

We present practical solutions for the computationally hard (Nexp-complete) partial-program resolution problem.

Strategy elimination. We present the general strategy enumeration scheme for partial program resolution. We first introduce the notions of a partial strategy and strategy tree.

Partial strategy and strategy trees. A *partial memoryless strategy* for Player 1 is a partial function from observations to actions. A *strategy tree* is a finite branching tree labelled with partial memoryless strategies of Player 1 such that: (a) Every leaf node is labelled with a complete strategy; (b) Every node is labelled with a unique partial strategy; and (c) For any parent-child node pair, the label of the child (σ_2) is a proper extension of the label of parent (σ_1), i.e., $\sigma_1(o) = \sigma_2(o)$ when both are defined and the domain of σ_2 a proper superset of σ_1. A complete strategy tree is one where all Player 1 memoryless strategies are present as labels.

In the strategy enumeration scheme, we maintain a set of candidate strategy trees and check each one for partial correctness. If the root label of the tree fails the partial correctness check, then remove the whole tree from the set. Otherwise, we replace it with the children of the root node. The initial set is a single complete strategy tree. In practice, the choice of this tree can be instrumental in the efficiency of partial correctness checks. Trees which first fix the choices that help the partial correctness check to identify an incorrect partial strategy are more useful. The partial program resolution scheme is shown in Algorithm 1, and the details are presented in the full version.

The `PartialCheck` function checks for the partial correctness of partial strategies, and returns "Incorrect" if it is able to prove that all strategies compatible with the input are unsafe, or it returns "Don't know". In practice, for the partial correctness checks the following steps can be used: (a) checking of lock discipline to prevent deadlocks; and (b) simulation of the partial program on small inputs; The result of the scheme is a set of candidate strategies for which we evaluate full correctness and compute the value.

Evaluation of a memoryless strategy. Fixing a memoryless Player 1 strategy in a ImplIn $2\frac{1}{2}$-player game for partial program resolution gives us (i) a non-deterministic transition system in the case of a non-deterministic scheduler, or (ii) an MDP in case of probabilistic schedulers. These are perfect-information games and hence, can be solved efficiently. In case (i), we use a standard min-mean cycle algorithm (for example, [12]) to find the value of the strategy . In case (ii), we focus on solving Markov chains with limit-average objectives efficiently. Markov chains arise from MDPs due to two reasons: (1) In many cases, program input can be abstracted away using data abstraction and the problem is reduced to solving a LimAvg Markov Chain. (2) The most efficient algorithm for LimAvg MDPs is the strategy improvement algorithm [9], and each step of the algorithm involves solving a Markov chain (for standard techniques, see [9]).

In practice, a large fraction of concurrent programs are designed to ensure progress condition called *lock-freedom* [10]. Lock-freedom ensures that some thread always makes progress in a finite number of steps. This leads to Markov chains with a directed-acyclic tree like structure with only few cycles introduced to eliminate finite executions as mentioned in Section 2. We present a *forward propagation* technique to compute stationary probabilities for these Markov chains. Computing the stationary distribution for a Markov chain involves solving a set of linear equalities using Gaussian elimination. For Markov chains that satisfy the special structure, we speed up the process by eliminating variables in the tree by forward propagating the root variable. Using this technique, we were able to handle the special Markov chains of up to 100,000 states in a few seconds in the experiments.

Quantitative probabilistic abstraction. To improve the performance of the synthesis, we use standard abstraction techniques. However, for the partial program resolution problem we require abstraction that also preserves quantitative objectives such as LimAvg and LimAvg-Safety. We show that an extension of probabilistic bisimilarity [13] with a condition for weight function preserves the quantitative objectives.

Quantitative probabilistic bisimilarity. A binary equivalence relation $\equiv$ on the states of a MDP is a *quantitative probabilistic bisimilarity* relation if (a) $s \equiv s'$ iff s and s' are both safe or both unsafe; (b) $\forall s \equiv s', a \in A : \sum_{t \in C} \Delta(s,a)(t) = \sum_{t \in C} \Delta(s',a)(t)$ where C is an equivalence class of $\equiv$; and (c) $s \equiv s' \wedge t \equiv t' \implies \gamma(s,a,s') = \gamma(t,a,t')$. The states s and s' are *quantitative probabilistic bisimilar* if $s \equiv s'$.

A *quotient* of an MDP $\mathcal{G}$ under quantitative probabilistic bisimilarity relation $\equiv$ is an MDP $({}^{\mathcal{G}}/_{\equiv})$ where the states are the equivalence classes of $\equiv$ and: (i) $\gamma(C,a,C') = \gamma(s,a,s')$ where $s \in C$ and $s' \in C'$, and (ii) $\Delta(C,a)(C') = \sum_{t' \in C'} \Delta'(s,a)(t)$ where $s \in C$.

Theorem 6. *Given an MDP $\mathcal{G}$, a quantitative probabilistic bisimilarity relation $\equiv$, and a limit-average safety objective f, the values in $\mathcal{G}$ and ${}^{\mathcal{G}}/_{\equiv}$ coincide.*

Consider a standard abstraction technique, *data abstraction*, which erases the value of given variables. We show that under certain syntactic restrictions (namely, that the abstracted variables do not appear in any guard statements), the equivalence relation given by the abstraction is a quantitative probabilistic bisimilarity relation and thus is a sound abstraction with respect to any limit-average safety objective. We also consider a less coarse abstraction, *equality and order abstraction*, which preserves equality and order relations among given variables. This abstraction defines a quantitative probabilistic bisimilarity relation under the syntactic condition that the guards test only for these relations, and no arithmetic is used on the abstracted variables.

5 Experiments

We describe the results obtained by applying our prototype implementation of techniques described above on four examples. In the examples, obtaining a correct program is not difficult and we focus on the synthesis of optimal programs.

The partial programs were manually abstracted (using the data and order abstractions) and translated into PROMELA, the input language of the SPIN model checker [11]. The abstraction step was straightforward and could be automated. The transition graphs were generated using SPIN. Then, our tool constructed the game graph by taking the product with the scheduler and performance model. The resulting game was solved for the LimAvg-Safety objectives using techniques from Section 4. The examples we considered were small (each thread running a procedure with 15 to 20 lines of code). The synthesis time was under a minute for all but one case (Example 2 with larger values of n), where it was under five minutes. The experiments were run on a dual-core 2.5Ghz machine with 2GB of RAM. For all examples, the tool reports normalized performance metrics where higher values indicate better performance.

Example 1. We consider the producer-consumer example described in Section 1, with two consumer and two producer threads. The partial program models a four slot concurrent buffer which is operated on by producers and consumers. Here, we try to synthesize lock granularity. The synthesis results are presented in Table 1.

Table 1. Performance of shared buffers under various locking strategies: LC and CC are the locking cost and data copying cost

LC: CC	Granularity	Performance
1:100	Coarse	1
	Medium	1.15
	Fine	1.19
1:20	Coarse	1
	Medium	1.14
	Fine	1.15
1:10	Coarse	1
	Medium	1.12
	Fine	1.12
1:2	Coarse	1
	Medium	1.03
	Fine	0.92
1:1	Coarse	1
	Medium	0.96
	Fine	0.80

The most important parameters in the performance model are the cost of locking/unlocking l and the cost c of copying data from/to shared memory. If c was higher than l (by 100:1), then the fine-grained locking approach is better (by 19 percent), and is the result of synthesis. If the cost l is equal to c, then the coarse-grained locking approach was found to perform better (by 25 percent), and thus the coarse-grained program is the result of the synthesis.

Example 2. We consider the optimistic concurrency example described in detail in Section 1. In the code (Figure 1), the number of operations performed optimistically is controlled by the variable `n`. We synthesized the optimal `n` for various performance models and the results are summarized in Table 2. We were able to find correspondence between our models and the program behavior on a desktop machine: (a) We observed that the graph of performance-vs-`n` has a local maximum when we tested the partial program on the desktop. In our experiments, we were able to find parameters for the performance model which have similar performance-vs-`n` curves. (b) Furthermore, by changing the cost of locking operations on a desktop, by introducing small delays during locks, we were able to observe performance results similar to those produced by other performance model parameters.

Example 3. We synthesize the optimal number of threads for work sharing (pseudocode in the full version). For independent operations, multiple threads utilize multiple processors more efficiently. However, for small number of operations, thread initialization cost will possibly overcome any performance gain. The experimental results are summarized in Figure 4. The x- and y- axes measure the initialization cost and performance, respectively. Each plot in the graph is for a different number of threads. The two graphs (a) and (b) are for a different amounts of work to be shared (the length of the array to be operated was varied between 16, and 32). Further graphs are in the full version. As it can be seen from the figure, for smaller amounts of work, spawning fewer threads is usually better. However, for larger amounts of work, greater number of threads outperforms smaller number of threads, even in the presence of higher

Table 2. Optimistic performance: WC, CC, and LWO are the work cost, lock cost, and the length of the work operation

WC : LC	LWO	Performance for `n`				
		1	2	3	4	5
20:1	1	1.0	1.049	1.052	1.048	1.043
20:1	2	1.0	0.999	0.990	0.982	0.976
10:1	1	1.0	1.134	1.172	1.187	1.193
10:1	2	1.0	1.046	1.054	1.054	1.052

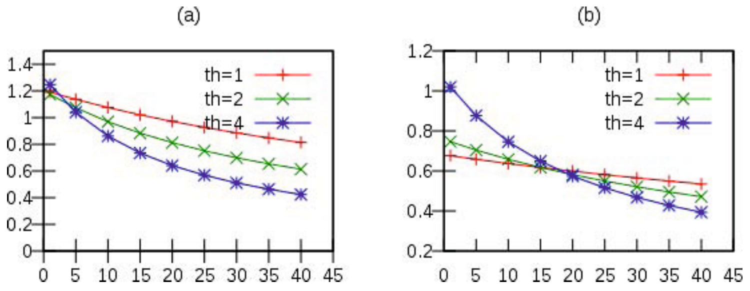


Fig. 4. Work sharing for initialization costs and thread counts: More work is shared in case (b) than case (a)

initialization costs. The code was run on a desktop (with scaled parameters) and similar results were observed.

Example 4. We study the effects of processor caches on performance using a simple performance model for caches. A cache line is modeled as in Figure 5. It assigns differing costs to read and write actions if the line is cached or not. The performance model is the synchronous product of one such automata per memory line. The only actions in the performance model after the synchronous product (caches synchronize on `evict` and `flush`) are `READ` and `WRITE` actions. These actions are matched with the transitions of the partial program.

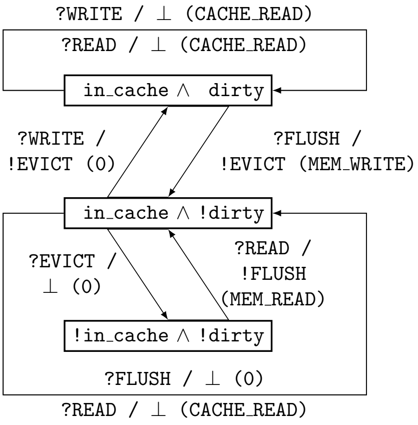


Fig. 5. Perf. aut. for Example 4

The partial program is a pessimistic variant of Figure 1 (pseudocode in full version). Increasing `n`, i.e., the number of operations performed under locks, increases the temporal locality of memory accesses and hence, increase in performance is expected. We observed the expected results in our experiments. For instance, increasing `n` from 1 to 5 increases the performance by a factor of 2.32 and increasing `n` from to 10 gives an additional boost of about 20%. The result of the synthesis is the program with $n = 10$.

6 Conclusion

Summary. Our main contributions are: (1) we developed a technique for synthesizing concurrent programs that are both correct and *optimal*; (2) we introduced a parametric performance model providing a flexible framework for specifying performance characteristics of architectures; (3) we showed how to apply imperfect-information games to the synthesis of concurrent programs and established the complexity for the game problems that arise in this context (4) we developed and implemented practical techniques to efficiently solve partial-program synthesis, and we applied the resulting prototype tool to several examples that illustrate common patterns in concurrent programming.

Future work. Our approach examines every correct strategy. There is thus the question whether there exists a practical algorithm that overcomes this limitation. Also, we did not consider the question which solution(s) to present to the programmer in case there is a number of correct strategies with the same performance. Furthermore, one could perhaps incorporate some information on the expected workload to the performance model. There are several other future research directions: one is to consider the synthesis of programs that access concurrent data structures; another is to create benchmarks from which performance automata can be obtained automatically.

References

1. Björklund, H., Sandberg, S., Vorobyov, S.: A discrete subexponential algorithm for parity games. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 663–674. Springer, Heidelberg (2003)
2. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
3. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
4. Chatterjee, K., Doyen, L., Gimbert, H., Henzinger, T.A.: Randomness for free. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 246–257. Springer, Heidelberg (2010)
5. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI, pp. 304–315 (2008)
6. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians (1962)
7. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. Workshop on Logic of Programs, pp. 52–71 (1981)
8. Degorre, A., Doyen, L., Gentilini, R., Raskin, J.-F., Toruńczyk, S.: Energy and mean-payoff games with imperfect information. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 260–274. Springer, Heidelberg (2010)
9. Filar, J., Vrieze, K.: Competitive Markov decision processes (1996)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, Amsterdam (2008)
11. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
12. Karp, R.: A characterization of the minimum cycle mean in a digraph. Discrete Mathematics 23, 309–311 (1978)
13. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. In: POPL, pp. 344–352 (1989)

14. Papadimitriou, C.: Computational Complexity. Addison-Wesley Publishing, Reading (1994)
15. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
16. Solar-Lezama, A., Jones, C., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
17. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294 (2005)
18. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS, pp. 327–338 (1985)
19. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)

Symbolic Algorithms for Qualitative Analysis of Markov Decision Processes with Büchi Objectives[⋆,⋆⋆]

Krishnendu Chatterjee[1], Monika Henzinger[2], Manas Joglekar[3], and Nisarg Shah[3]

[1] IST Austria
[2] University of Vienna
[3] IIT Bombay

Abstract. We consider Markov decision processes (MDPs) with ω-regular specifications given as parity objectives. We consider the problem of computing the set of *almost-sure* winning states from where the objective can be ensured with probability 1. The algorithms for the computation of the almost-sure winning set for parity objectives iteratively use the solutions for the almost-sure winning set for Büchi objectives (a special case of parity objectives). Our contributions are as follows: First, we present the first subquadratic symbolic algorithm to compute the almost-sure winning set for MDPs with Büchi objectives; our algorithm takes $O(n \cdot \sqrt{m})$ symbolic steps as compared to the previous known algorithm that takes $O(n^2)$ symbolic steps, where n is the number of states and m is the number of edges of the MDP. In practice MDPs often have constant out-degree, and then our symbolic algorithm takes $O(n \cdot \sqrt{n})$ symbolic steps, as compared to the previous known $O(n^2)$ symbolic steps algorithm. Second, we present a new algorithm, namely *win-lose* algorithm, with the following two properties: (a) the algorithm iteratively computes subsets of the almost-sure winning set and its complement, as compared to all previous algorithms that discover the almost-sure winning set upon termination; and (b) requires $O(n \cdot \sqrt{K})$ symbolic steps, where K is the maximal number of edges of strongly connected components (scc's) of the MDP. The win-lose algorithm requires symbolic computation of scc's. Third, we improve the algorithm for symbolic scc computation; the previous known algorithm takes linear symbolic steps, and our new algorithm improves the constants associated with the linear number of steps. In the worst case the previous known algorithm takes $5 \cdot n$ symbolic steps, whereas our new algorithm takes $4 \cdot n$ symbolic steps.

1 Introduction

Markov decision processes. The model of systems in verification of probabilistic systems are *Markov decision processes (MDPs)* that exhibit both probabilistic and nondeterministic behavior [12]. MDPs have been used to model and solve control problems for stochastic systems [10]: there, nondeterminism represents the freedom of the controller to choose a control action, while the probabilistic component of the behavior describes the system response to control actions. MDPs have also been adopted

[⋆] This work was partially supported by FWF NFN Grant S11407-N23 (RiSE) and a Microsoft faculty fellowship.

[⋆⋆] Details omitted due to lack of space available in [3].

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 260–276, 2011.

as models for concurrent probabilistic systems [6], probabilistic systems operating in open environments [18], and under-specified probabilistic systems [1]. A *specification* describes the set of desired behaviors of the system, which in the verification and control of stochastic systems is typically an ω-regular set of paths. The class of ω-regular languages extends classical regular languages to infinite strings, and provides a robust specification language to express all commonly used specifications, such as safety, liveness, fairness, etc [21]. Parity objectives are a canonical way to define such ω-regular specifications. Thus MDPs with parity objectives provide the theoretical framework to study problems such as the verification and control of stochastic systems.

Qualitative and quantitative analysis. The analysis of MDPs with parity objectives can be classified into qualitative and quantitative analysis. Given an MDP with parity objective, the *qualitative analysis* asks for the computation of the set of states from where the parity objective can be ensured with probability 1 (almost-sure winning). The more general *quantitative analysis* asks for the computation of the maximal probability at each state with which the controller can satisfy the parity objective.

Importance of qualitative analysis. The qualitative analysis of MDPs is an important problem in verification that is of interest irrespective of the quantitative analysis problem. There are many applications where we need to know whether the correct behavior arises with probability 1. For instance, when analyzing a randomized embedded scheduler, we are interested in whether every thread progresses with probability 1 [8]. Even in settings where it suffices to satisfy certain specifications with probability $p < 1$, the correct choice of p is a challenging problem, due to the simplifications introduced during modeling. For example, in the analysis of randomized distributed algorithms it is quite common to require correctness with probability 1 (see, e.g., [16,15,20]). Furthermore, in contrast to quantitative analysis, qualitative analysis is robust to numerical perturbations and modeling errors in the transition probabilities, and consequently the algorithms for qualitative analysis are combinatorial. Finally, for MDPs with parity objectives, the best known algorithms and all algorithms used in practice first perform the qualitative analysis, and then performs a quantitative analysis on the result of the qualitative analysis [6,7,5]. Thus qualitative analysis for MDPs with parity objectives is one of the most fundamental and core problems in verification of probabilistic systems. One of the key challenges in probabilistic verification is to obtain efficient and symbolic algorithms for qualitative analysis of MDPs with parity objectives, as symbolic algorithms allow to handle MDPs with a large state space.

Previous results. The qualitative analysis for MDPs with parity objectives is achieved by iteratively applying solutions of the qualitative analysis of MDPs with Büchi objectives [6,7,5]. The qualitative analysis of an MDP with a parity objective with d priorities can be achieved by $O(d)$ calls to an algorithm for qualitative analysis of MDPs with Büchi objectives, and hence we focus on the qualitative analysis of MDPs with Büchi objectives. The classical algorithm for qualitative analysis for MDPs with Büchi objectives works in $O(n \cdot m)$ time, where n is the number of states, and m is the number of edges of the MDP [6,7]. The classical algorithm can be implemented symbolically, and it takes at most $O(n^2)$ symbolic steps. An improved algorithm for the problem was given in [4] that works in $O(m \cdot \sqrt{m})$ time. The algorithm of [4] crucially depends on maintaining the same number of edges in certain forward searches. Thus the algorithm needs to explore edges of the graph explicitly and is inherently non-symbolic. In the literature, there is no symbolic subquadratic algorithm for qualitative analysis of MDPs with Büchi objectives.

Our contribution. In this work our main contributions are as follows.

1. We present a new and simpler subquadratic algorithm for qualitative analysis of MDPs with Büchi objectives that runs in $O(m \cdot \sqrt{m})$ time, and show that the algorithm can be implemented symbolically. The symbolic algorithm takes at most $O(n \cdot \sqrt{m})$ symbolic steps, and thus we obtain the first symbolic subquadratic algorithm. In practice, MDPs often have constant out-degree: for example, see [9] for MDPs with large state space but constant number of actions, or [10,17] for examples from inventory management where MDPs have constant number of actions (the number of actions correspond to the out-degree of MDPs). For MDPs with constant out-degree our new symbolic algorithm takes $O(n \cdot \sqrt{n})$ symbolic steps, as compared to $O(n^2)$ symbolic steps of the previous best known algorithm.
2. All previous algorithms for the qualitative analysis of MDPs with Büchi objectives iteratively discover states that are guaranteed to be not almost-sure winning, and only when the algorithm terminates the almost-sure winning set is discovered. We present a new algorithm (namely *win-lose* algorithm) that iteratively discovers both states in the almost-sure winning set and its complement. Thus if the problem is to decide whether a given state s is almost-sure winning, and the state s is almost-sure winning, then the win-lose algorithm can stop at an intermediate iteration unlike all the previous algorithms. Our algorithm works in time $O(\sqrt{K_E} \cdot m)$ time, where K_E is the maximal number of edges of any scc of the MDP (in this paper we write scc for maximal scc). We also show that the win-lose algorithm can be implemented symbolically, and it takes at most $O(\sqrt{K_E} \cdot n)$ symbolic steps.
3. Our win-lose algorithm requires to compute the scc decomposition of a graph in $O(n)$ symbolic steps. The scc decomposition problem is one of the most fundamental problem in the algorithmic study of graph problems. The symbolic scc decomposition problem has many other applications in verification: for example, checking emptiness of ω-automata, and bad-cycle detection problems in model checking, see [2] for other applications. An $O(n \cdot \log n)$ symbolic step algorithm for scc decomposition was presented in [2], and the algorithm was improved in [11]. The algorithm of [11] is a linear symbolic step scc decomposition algorithm that requires at most $\min\{ 5 \cdot n, 5 \cdot D \cdot N + N \}$ symbolic steps, where D is the diameter of the graph, and N is the number of scc's of the graph. We present an improved version of the symbolic scc decomposition algorithm. Our algorithm improves the constants of the number of the linear symbolic steps. Our algorithm requires at most $\min\{ 3 \cdot n + N, 5 \cdot D^* + N \}$ symbolic steps, where D^* is the sum of the diameters of the scc's of the graph. Thus, in the worst case, the algorithm of [11] requires $5 \cdot n$ symbolic steps, whereas our algorithm requires $4 \cdot n$ symbolic steps. Moreover, the number of symbolic steps of our algorithm is always bounded by the number of symbolic steps of the algorithm of [11] (i.e. our algorithm is never worse).

Our experimental results show that our new algorithms perform better than the previous known algorithms both for qualitative analysis of MDPs with Büchi objectives and symbolic scc computation.

2 Definitions

Markov decision processes (MDPs). A *Markov decision process (MDP)* $G = ((S, E), (S_1, S_P), \delta)$ consists of a directed graph (S, E), a partition (S_1,S_P) of the *finite* set S of states, and a probabilistic transition function δ: $S_P \to \mathcal{D}(S)$, where $\mathcal{D}(S)$ denotes the set of probability distributions over the state space S. The states in S_1 are the *player*-1 states, where player 1 decides the successor state, and the states in S_P are the *probabilistic (or random)* states, where the successor state is chosen according to the probabilistic transition function δ. We assume that for $s \in S_P$ and $t \in S$, we have $(s, t) \in E$ iff $\delta(s)(t) > 0$, and we often write $\delta(s, t)$ for $\delta(s)(t)$. For a state $s \in S$, we write $E(s)$ to denote the set $\{ t \in S \mid (s, t) \in E \}$ of possible successors. For technical convenience we assume that every state in the graph (S, E) has at least one outgoing edge, i.e., $E(s) \neq \emptyset$ for all $s \in S$.

Plays and strategies. An infinite path, or a *play*, of the game graph G is an infinite sequence $\omega = \langle s_0, s_1, s_2, \ldots \rangle$ of states such that $(s_k, s_{k+1}) \in E$ for all $k \in \mathbb{N}$. We write Ω for the set of all plays, and for a state $s \in S$, we write $\Omega_s \subseteq \Omega$ for the set of plays that start from the state s. A *strategy* for player 1 is a function σ: $S^* \cdot S_1 \to \mathcal{D}(S)$ that chooses the probability distribution over the successor states for all finite sequences $\boldsymbol{w} \in S^* \cdot S_1$ of states ending in a player-1 state (the sequence represents a prefix of a play). A strategy must respect the edge relation: for all $\boldsymbol{w} \in S^*$ and $s \in S_1$, if $\sigma(\boldsymbol{w} \cdot s)(t) > 0$, then $t \in E(s)$. A strategy is *deterministic (pure)* if it chooses a unique successor for all histories (rather than a probability distribution), otherwise it is *randomized*. Player 1 follows the strategy σ if in each player-1 move, given that the current history of the game is $\boldsymbol{w} \in S^* \cdot S_1$, she chooses the next state according to $\sigma(\boldsymbol{w})$. We denote by Σ the set of all strategies for player 1. A *memoryless* player-1 strategy does not depend on the history of the play but only on the current state; i.e., for all $\boldsymbol{w}, \boldsymbol{w'} \in S^*$ and for all $s \in S_1$ we have $\sigma(\boldsymbol{w} \cdot s) = \sigma(\boldsymbol{w'} \cdot s)$. A memoryless strategy can be represented as a function σ: $S_1 \to \mathcal{D}(S)$, and a pure memoryless strategy can be represented as $\sigma : S_1 \to S$.

Once a starting state $s \in S$ and a strategy $\sigma \in \Sigma$ is fixed, the outcome of the MDP is a random walk ω_s^σ for which the probabilities of events are uniquely defined, where an *event* $\mathcal{A} \subseteq \Omega$ is a measurable set of plays. For a state $s \in S$ and an event $\mathcal{A} \subseteq \Omega$, we write $\Pr_s^\sigma(\mathcal{A})$ for the probability that a play belongs to $\mathcal{A}$ if the game starts from the state s and player 1 follows the strategy σ.

Objectives. We specify *objectives* for the player 1 by providing a set of *winning* plays $\Phi \subseteq \Omega$. We say that a play ω *satisfies* the objective Φ if $\omega \in \Phi$. We consider *ω-regular objectives* [21], specified as parity conditions. We also consider the special case of Büchi objectives.

- *Büchi objectives.* Let T be a set of target states. For a play $\omega = \langle s_0, s_1, \ldots \rangle \in \Omega$, we define $\mathrm{Inf}(\omega) = \{ s \in S \mid s_k = s$ for infinitely many $k \}$ to be the set of states

that occur infinitely often in ω. The Büchi objective requires that some state of T be visited infinitely often, and defines the set of winning plays $\text{Büchi}(T) = \{ \omega \in \Omega \mid \text{Inf}(\omega) \cap T \neq \emptyset \}$.
- *Parity objectives.* For $c, d \in \mathbb{N}$, we write $[c..d] = \{ c, c+1, \ldots, d \}$. Let p: $S \to [0..d]$ be a function that assigns a *priority* $p(s)$ to every state $s \in S$, where $d \in \mathbb{N}$. The *parity objective* is defined as $\text{Parity}(p) = \{ \omega \in \Omega \mid \min\big(p(\text{Inf}(\omega))\big) \text{ is even } \}$. In other words, the parity objective requires that the minimum priority visited infinitely often is even. In the sequel we will use Φ to denote parity objectives.

Qualitative analysis: almost-sure winning. Given a player-1 objective Φ, a strategy $\sigma \in \Sigma$ is *almost-sure winning* for player 1 from the state s if $\Pr_s^\sigma(\Phi) = 1$. The *almost-sure winning set* $\langle\langle 1 \rangle\rangle_{almost}(\Phi)$ for player 1 is the set of states from which player 1 has an almost-sure winning strategy. The qualitative analysis of MDPs correspond to the computation of the almost-sure winning set for a given objective Φ. It follows from the results of [6,7] that for all MDPs and all reachability and parity objectives, if there is an almost-sure winning strategy, then there is a memoryless almost-sure winning strategy. The qualitative analysis of MDPs with parity objectives is achieved by iteratively applying the solutions of qualitative analysis for MDPs with Büchi objectives [7,5], and hence in this work we will focus on qualitative analysis for Büchi objectives.

Theorem 1 ([6,7]). *For all MDPs G and all parity objectives Φ, there exists a pure memoryless strategy σ_* such that for all $s \in \langle\langle 1 \rangle\rangle_{almost}(\Phi)$ we have* $\Pr_s^{\sigma_*}(\Phi) = 1$.

Scc and bottom scc. Given a graph $G = (S, E)$, a set C of states is an scc if for all $s, t \in C$ there is a path from s to t going through states in C. In sequel we write scc for maximal scc. An scc C is a bottom scc if for all $s \in C$ all out-going edges are in C, i.e., $E(s) \subseteq C$.

Markov chains, closed recurrent sets. A Markov chain is a special case of MDP with $S_1 = \emptyset$, and hence for simplicity a Markov chain is a tuple $((S, E), \delta)$ with a probabilistic transition function $\delta : S \to \mathcal{D}(S)$, and $(s, t) \in E$ iff $\delta(s, t) > 0$. A *closed recurrent* set C of a Markov chain is a bottom scc in the graph (S, E). Let $\mathcal{C} = \bigcup_{C \text{ is closed recurrent}} C$. It follows from the results on Markov chains [14] that for all $s \in S$, the set $\mathcal{C}$ is reached with probability 1 in finite time, and for all C such that C is closed recurrent, for all $s \in C$ and for all $t \in C$, if the starting state is s, then the state t is visited infinitely often with probability 1.

Markov chain from a MDP and memoryless strategy. Given a MDP $G = ((S, E), (S_1, S_P), \delta)$ and a memoryless strategy $\sigma_* : S_1 \to \mathcal{D}(S)$ we obtain a Markov chain $G' = ((S, E'), \delta')$ as follows: $E' = E \cap (S_P \times S) \cup \{ (s, t) \mid s \in S_1, \sigma_*(s)(t) > 0 \}$; and $\delta'(s, t) = \delta(s, t)$ for $s \in S_P$, and $\delta'(s, t) = \sigma(s)(t)$ for $s \in S_1$ and $t \in E(s)$. We will denote by G_{σ_*} the Markov chain obtained from an MDP G by fixing a memoryless strategy σ_* in the MDP.

Symbolic encoding of an MDP. All algorithms of the paper will only depend on the graph (S, E) of the MDP and the partition (S_1, S_P), and not on the probabilistic transition function δ. Thus the symbolic encoding of an MDP is obtained as the standard encoding of a transition system (with an OBDD [19]), with one additional bit, and the bit denotes whether a state belongs to S_1 or S_P.

3 Symbolic Algorithms for Büchi Objectives

In this section we will present a new improved algorithm for the qualitative analysis of MDPs with Büchi objectives, and then present a symbolic implementation of the algorithm. Thus we obtain the first symbolic subquadratic algorithm for the problem. We start with the notion of *attractors* that is crucial for our algorithm.

Random and player 1 attractor. Given an MDP G, let $U \subseteq S$ be a subset of states. The *random attractor* $Attr_R(U)$ is defined inductively as follows: $X_0 = U$, and for $i \geq 0$, let $X_{i+1} = X_i \cup \{ s \in S_P \mid E(s) \cap X_i \neq \emptyset \} \cup \{ s \in S_1 \mid E(s) \subseteq X_i \}$. In other words, X_{i+1} consists of (a) states in X_i, (b) player-1 states whose all successors are in X_i and (c) random states that have at least one edge to X_i. Then $Attr_R(U) = \bigcup_{i \geq 0} X_i$. The definition of *player-1 attractor* $Attr_1(U)$ is analogous and is obtained by exchanging the role of random states and player 1 states in the above definition.

Property of attractors. Given an MDP G, and set U of states, let $A = Attr_R(U)$. Then from A player 1 cannot force to avoid U, in other words, for all states in A and for all player 1 strategies, the set U is reached with positive probability. For $A = Attr_1(U)$ there is a player 1 memoryless strategy to ensure that the set U is reached with certainty. The computation of random and player 1 attractor is the computation of alternating reachability and can be achieved in $O(m)$ time [13], and can be achieved in $O(n)$ symbolic steps.

3.1 A New Subquadratic Algorithm

The classical algorithm for computing the almost-sure winning set in MDPs with Büchi objectives has $O(n \cdot m)$ running time, and the symbolic implementation of the algorithm takes at most $O(n^2)$ symbolic steps. A subquadratic algorithm, with $O(m \cdot \sqrt{m})$ running time, for the problem was presented in [4]. The algorithm of [4] uses a mix of backward exploration and forward exploration. Every forward exploration step consists of executing a set of DFSs (depth first searches) simultaneously for a specified number of edges, and must maintain the exploration of the same number of edges in each of the DFSs. The algorithm thus depends crucially on maintaining the number of edges traversed explicitly, and hence the algorithm has no symbolic implementation. In this section we present a new subquadratic algorithm to compute $\langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. The algorithm is simpler as compared to the algorithm of [4] and we will show that our new algorithm can be implemented symbolically. Our new algorithm has some similar ideas as the algorithm of [4] in mixing backward and forward exploration, but the key difference is that the new algorithm never stops the forward exploration after a certain number of edges, and hence need not maintain the traversed edges explicitly. Thus the new algorithm is simpler, and our correctness and running time analysis proofs are different. We show that our new algorithm works in $O(m \cdot \sqrt{m})$ time, and requires at most $O(n \cdot \sqrt{m})$ symbolic steps.

Improved algorithm for almost-sure Büchi. Our algorithm iteratively removes states from the graph, until the almost-sure winning set is computed. At iteration i, we denote the remaining subgraph as (S_i, E_i), where S_i is the set of remaining states, E_i is the set of remaining edges, and the set of remaining target states as T_i (i.e., $T_i = S_i \cap T$). The set of states removed will be denoted by Z_i, i.e., $S_i = S \setminus Z_i$. The algorithm will ensure that (a) $Z_i \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$; and (b) for all $s \in S_i \cap S_P$ we have $E(s) \cap Z_i = \emptyset$. In every iteration the algorithm identifies a set Q_i of states such that there is no path from Q_i to the set T_i. Hence clearly $Q_i \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. By the random attractor property from $Attr_R(Q_i)$ the set Q_i is reached with positive probability against any strategy for player 1. The algorithm maintains the set L_{i+1} of states that were removed from the graph since (and including) the last iteration of Case 1, and the set J_{i+1} of states that lost an edge to states removed from the graph since the last iteration of Case 1. Initially $L_0 := J_0 := \emptyset$, $Z_0 := \emptyset$, and let $i := 0$ and we describe the iteration i of our algorithm, and we call our algorithm ImprAlgo (Improved Algorithm) and the formal pseudocode is in [3].

1. *Case 1.* If $((|J_i| \geq \sqrt{m})$ or $i = 0)$, then
   (a) Let Y_i be the set of states that can reach the current target set T_i (this can be computed in $O(m)$ time by a graph reachability algorithm).
   (b) Let $Q_i := S_i \setminus Y_i$, i.e., there is no path from Q_i to T_i.
   (c) $Z_{i+1} := Z_i \cup Attr_R(Q_i)$. The set $Attr_R(Q_i)$ is removed from the graph.
   (d) The set L_{i+1} is the set of states removed from the graph in this iteration (i.e., $L_{i+1} := Attr_R(Q_i)$) and J_{i+1} be the set of states in the remaining graph with an edge to L_{i+1}.
   (e) If Q_i is empty, the algorithm stops, otherwise $i := i + 1$ and go to the next iteration.
2. *Case 2.* Else $(|J_i| \leq \sqrt{m})$, then
   (a) We do a lock-step search from every state s in J_i as follows: we do a DFS from s and (a) if the DFS tree reaches a state in T_i, then we stop the DFS search from s; and (b) if the DFS is completed without reaching a state in T_i, then we stop the entire lock-step search, and all states in the DFS tree are identified as Q_i. The set $Attr_R(Q_i)$ is removed from the graph and $Z_{i+1} := Z_i \cup Attr_R(Q_i)$. If DFS searches from all states s in J_i reach the set T_i, then the algorithm stops.
   (b) The set L_{i+1} is the set of states removed from the graph since the last iteration of Case 1 (i.e., $L_{i+1} := L_i \cup Attr_R(Q_i)$, where Q_i is the DFS tree that stopped without reaching T_i in the previous step of this iteration) and J_{i+1} be the set of states in the remaining graph with an edge to L_{i+1}, i.e., $J_{i+1} := (J_i \setminus Attr_R(Q_i)) \cup X_i$, where X_i is the subset of states of S_i with an edge to $Attr_R(Q_i)$.
   (c) $i := i + 1$ and go to the next iteration.

The proof of Lemma 1 is available in [3]. We then present the running time analysis.

Lemma 1. *Algorithm* ImprAlgo *correctly computes the set* $\langle\langle 1 \rangle\rangle_{almost}(\textit{Büchi}(T))$.

Lemma 2. *Given an MDP G with m edges, Algorithm* ImprAlgo *takes* $O(m \cdot \sqrt{m})$ *time.*

Proof. The total work of the algorithm, when Case 1 is executed, over all iterations is at most $O(\sqrt{m} \cdot m)$: this follows because between two iterations of Case 1 at least $\sqrt{m}$ edges must have been removed from the graph (since $|J_i| \geq \sqrt{m}$ everytime Case 1 is executed other than the case when $i = 0$), and hence Case 1 can be executed at most $m/\sqrt{m} = \sqrt{m}$ times. Since each iteration can be achieved in $O(m)$ time, the $O(m \cdot \sqrt{m})$ bound for Case 1 follows. We now show that the total work of the algorithm, when Case 2 is executed, over all iterations is at most $O(\sqrt{m} \cdot m)$. The argument is as follows: consider an iteration i such that Case 2 is executed. Then we have $|J_i| \leq \sqrt{m}$. Let Q_i be the DFS tree in iteration i while executing Case 2, and let $E(Q_i) = \cup_{s \in Q_i} E(s)$. The lock-step search ensures that the number of edges explored in this iteration is at most $|J_i| \cdot |E(Q_i)| \leq \sqrt{m} \times |E(Q_i)|$. Since Q_i is removed from the graph we *charge* the work of $\sqrt{m} \cdot |E(Q_i)|$ to edges in $E(Q_i)$, charging work $\sqrt{m}$ to each edge. Since there are at most m edges, the total charge of the work over all iterations when Case 2 is executed is at most $O(m \cdot \sqrt{m})$. Note that if instead of $\sqrt{m}$ we would have used a bound k in distinguishing Case 1 and Case 2, we would have achieved a running time bound of $O(m^2/k + m \cdot k)$, which is optimized by $k = \sqrt{m}$. ∎

Theorem 2. *Given an MDP G and a set T of target states, the algorithm* ImprAlgo *correctly computes the set $\langle\langle 1 \rangle\rangle_{almost}$(Büchi$(T)$) in time $O(m \cdot \sqrt{m})$.*

3.2 Symbolic Implementation of ImprAlgo

In this subsection we will a present symbolic implementation of each of the steps of algorithm ImprAlgo. The symbolic algorithm depends on the following symbolic operations that can be easily achieved with an Obdd implementation. For a set $X \subseteq S$ of states, let

$$\mathsf{Pre}(X) = \{ s \in S \mid E(s) \cap X \neq \emptyset \}; \qquad \mathsf{Post}(X) = \{ t \in S \mid t \in \textstyle\bigcup_{s \in X} E(s) \};$$
$$\mathsf{CPre}(X) = \{ s \in S_P \mid E(s) \cap X \neq \emptyset \} \cup \{ s \in S_1 \mid E(s) \subseteq X \}.$$

In other words, $\mathsf{Pre}(X)$ is the predecessors of states in X; $\mathsf{Post}(X)$ is the successors of states in X; and $\mathsf{CPre}(X)$ is the set of states Y such that for every random state in Y there is a successor in X, and for every player 1 state in Y all successors are in Y.

We now present a symbolic version of ImprAlgo. For the symbolic version the basic steps are as follows: (i) Case 1 of the algorithm is same as Case 1 of ImprAlgo, and (ii) Case 2 is similar to Case 2 of ImprAlgo, and the only change in Case 2 is instead of lock-step search exploring the same number of edges, we have lock-step search that executes the same number of symbolic steps. The details of the symbolic implementation are as follows, and we will refer to the algorithm as SymbImprAlgo.

1. *Case 1.* In Case 1(a) we need to compute reachability to a target set T. The symbolic implementation is standard and done as follows: $X_0 = T$ and $X_{i+1} := X_i \cup \mathsf{Pre}(X_i)$ until $X_{i+1} = X_i$. The computation of the random attractor is also standard and is achieved as above replacing Pre by CPre. It follows that every iteration of Case 1 can be achieved in $O(n)$ symbolic steps.
2. *Case 2.* For analysis of Case 2 we present a symbolic implementation of the lock-step forward search. The lock-step ensures that each search executes the same number of symbolic steps. The implementation of the forward search from a state s in

iteration i is achieved as follows: $P_0 := \{ s \}$ and $P_{j+1} := P_j \cup \mathsf{Post}(P_j)$ unless $P_{j+1} = P_j$ or $P_j \cap T_i \neq \emptyset$. If $P_j \cap T_i \neq \emptyset$, then the forward search is stopped from s. If $P_{j+1} = P_j$ and $P_j \cap T_i = \emptyset$, then we have identified that there is no path from states in P_j to T_i.

3. *Symbolic computation of cardinality of sets.* The other key operation required by the algorithm is determining whether the size of set J_i is at least $\sqrt{m}$ or not. The details of this symbolic operation is in [3].

Correctness and runtime analysis. The correctness of SYMBIMPRALGO is established following the correctness arguments for algorithm IMPRALGO. We now analyze the worst case number of symbolic steps. The total number of symbolic steps executed by Case 1 over all iterations is $O(n \cdot \sqrt{m})$ since between two executions of Case 1 at least $\sqrt{m}$ edges are removed, and every execution is achieved in $O(n)$ symbolic steps. The work done for the symbolic cardinality computation is charged to the edges already removed from the graph, and hence the total number of symbolic steps over all iterations for the size computations is $O(m)$. We now show that the total number of symbolic steps executed over all iterations of Case 2 is $O(n \cdot \sqrt{m})$. The analysis is achieved as follows. Consider an iteration i of Case 2, and let the number of states removed in the iteration be n_i. Then the number of symbolic steps executed in this iteration for each of the forward search is at most n_i, and since $|J_i| \leq \sqrt{m}$, it follows that the number of symbolic steps executed is at most $n_i \cdot \sqrt{m}$. Since we remove n_i states, we *charge* each state removed from the graph with $\sqrt{m}$ symbolic steps for the total $n_i \cdot \sqrt{m}$ symbolic steps. Since there are at most n states, the total charge of symbolic steps over all iterations is $O(n \cdot \sqrt{m})$. Thus it follows that we have a symbolic algorithm to compute the almost-sure winning set for MDPs with Büchi objectives in $O(n \cdot \sqrt{m})$ symbolic steps.

Theorem 3. *Given an MDP G and a set T of target states, the symbolic algorithm* SYMBIMPRALGO *correctly computes $\langle\langle 1 \rangle\rangle_{almost}(\textit{Büchi}(T))$ in $O(n \cdot \sqrt{m})$ symbolic steps.*

Remark 1. In many practical cases, MDPs have constant out-degree and hence we obtain a symbolic algorithm that works in $O(n \cdot \sqrt{n})$ symbolic steps, as compared to the previous known (symbolic implementation of the classical) algorithm that takes $O(n^2)$ symbolic steps.

4 The Win-Lose Algorithm

All the algorithms known for computing the almost-sure winning set (including the algorithms presented in the previous section) iteratively compute the set of states from where it is guaranteed that there is no almost-sure winning strategy for the player. The almost-sure winning set is discovered only when the algorithm stops. In this section, first we will present an algorithm that iteratively computes two sets W_1 and W_2, where W_1 is a subset of the almost-sure winning set, and W_2 is a subset of the complement of the almost-sure winning set. The algorithm has $O(K \cdot m)$ running time, where K is the size of the maximal strongly connected component (scc) of the graph of the MDP.

We then present an improved version of the algorithm, using the techniques to obtain IMPRALGO from the classical algorithm, and finally present the symbolic implementation of the new algorithm.

4.1 The Basic Win-Lose Algorithm

The basic steps of the new algorithm are as follows. The algorithm maintains W_1 and W_2, that are guaranteed to be subsets of the almost-sure winning set and its complement respectively. Initially $W_1 = \emptyset$ and $W_2 = \emptyset$. We also maintain that $W_1 = \mathit{Attr}_1(W_1)$ and $W_2 = \mathit{Attr}_R(W_2)$. We denote by W the union of W_1 and W_2. We describe an iteration of the algorithm and we will refer to the algorithm as the WINLOSE algorithm (formal pseudocode in [3]).

1. *Step 1.* Compute the scc decomposition of the remaining graph of the MDP, i.e., scc decomposition of the MDP graph induced by $S \setminus W$.
2. *Step 2.* For every bottom scc C in the remaining graph: if $C \cap \mathsf{Pre}(W_1) \neq \emptyset$ or $C \cap T \neq \emptyset$, then $W_1 = \mathit{Attr}_1(W_1 \cup C)$; else $W_2 = \mathit{Attr}_R(W_2 \cup C)$, and the states in W_1 and W_2 are removed from the graph.

The stopping criterion is as follows: the algorithm stops when $W = S$. Observe that in each iteration, a set C of states is included in either W_1 or W_2, and hence W grows in each iteration.

Correctness of the algorithm. Note that in Step 2 we ensure that $\mathit{Attr}_1(W_1) = W_1$ and $\mathit{Attr}_R(W_2) = W_2$, and hence in the remaining graph there is no state of player 1 with an edge to W_1 and no random state with an edge to W_2. We show by induction that after every iteration $W_1 \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$ and $W_2 \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. The base case (with $W_1 = W_2 = \emptyset$) follows trivially. We prove the inductive case considering the following two cases.

1. Consider a bottom scc C in the remaining graph such that $C \cap \mathsf{Pre}(W_1) \neq \emptyset$ or $C \cap T \neq \emptyset$. Consider the randomized memoryless strategy σ for the player that plays all edges in C uniformly at random, i.e., for $s \in C$ we have $\sigma(s)(t) = \frac{1}{|E(s) \cap C|}$ for $t \in E(s) \cap C$. If $C \cap \mathsf{Pre}(W_1) \neq \emptyset$, then the strategy ensures that W_1 is reached with probability 1, since $W_1 \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$ by inductive hypothesis it follows $C \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. Hence $\mathit{Attr}_1(W_1 \cup C) \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. If $C \cap T \neq \emptyset$, then since there is no edge from random states to W_2, it follows that under the randomized memoryless strategy σ, the set C is a closed recurrent set of the resulting Markov chain, and hence every state is visited infinitely often with probability 1. Since $C \cap T \neq \emptyset$, it follows that $C \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$, and hence $\mathit{Attr}_1(W_1 \cup C) \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$.
2. Consider a bottom scc C in the remaining graph such that $C \cap \mathsf{Pre}(W_1) = \emptyset$ and $C \cap T = \emptyset$. Then consider any strategy for player 1: (a) If a play starting from a state in C stays in the remaining graph, then since C is a bottom scc, it follows that the play stays in C with probability 1. Since $C \cap T = \emptyset$ it follows that T is never visited. (b) If a play leaves C (note that C is a bottom scc of the remaining graph and not the original graph, and hence a play may leave C), then since $C \cap \mathsf{Pre}(W_1) = \emptyset$, it follows that the play reaches W_2, and by hypothesis

$W_2 \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. In either case it follows that $C \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. It follows that $Attr_R(W_2 \cup C) \subseteq S \setminus \langle\langle 1 \rangle\rangle_{almost}(\text{Büchi}(T))$. The correctness of the algorithm follows as when the algorithm stops we have $W_1 \cup W_2 = S$ and running time analysis is given in [3].

Theorem 4. *Given an MDP with a Büchi objective, the* WINLOSE *algorithm iteratively computes the subsets of the almost-sure winning set and its complement, and in the end correctly computes the set $\langle\langle 1 \rangle\rangle_{almost}(Büchi(T))$ and the algorithm runs in time $O(K_S \cdot m)$, where K_S is the maximum number of states in an scc of the graph of the MDP.*

4.2 Improved WINLOSE Algorithm and Symbolic Implementation

Improved WINLOSE algorithm. The improved version of the WINLOSE algorithm performs a forward exploration to obtain a bottom scc like Case 2 of IMPRALGO. At iteration i, we denote the remaining subgraph as (S_i, E_i), where S_i is the set of remaining states, and E_i is the set of remaining edges. The set of states removed will be denoted by Z_i, i.e., $S_i = S \setminus Z_i$, and Z_i is the union of W_1 and W_2. In every iteration the algorithm identifies a set C_i of states such that C_i is a bottom scc in the remaining graph, and then it follows the steps of the WINLOSE algorithm. We will consider two cases. The algorithm maintains the set L_{i+1} of states that were removed from the graph since (and including) the last iteration of Case 1, and the set J_{i+1} of states that lost an edge to states removed from the graph since the last iteration of Case 1. Initially $J_0 := L_0 := Z_0 := W_1 := W_2 := \emptyset$, and let $i := 0$ and we describe the iteration i of our algorithm. We call our algorithm IMPRWINLOSE (formal pseudocode in [3]).

1. *Case 1.* If $((|J_i| \geq \sqrt{m})$ or $i = 0)$, then
   (a) Compute the scc decomposition of the remaining graph.
   (b) For each bottom scc C_i, if $C_i \cap T \neq \emptyset$ or $C_i \cap \mathsf{Pre}(W_1) \neq \emptyset$, then $W_1 := Attr_1(W_1 \cup C_i)$, else $W_2 := Attr_R(W_2 \cup C_i)$.
   (c) $Z_{i+1} := W_1 \cup W_2$. The set $Z_{i+1} \setminus Z_i$ is removed from the graph.
   (d) The set L_{i+1} is the set of states removed from the graph in this iteration and J_{i+1} be the set of states in the remaining graph with an edge to L_{i+1}.
   (e) If Z_i is S, the algorithm stops, otherwise $i := i + 1$ and go to the next iteration.
2. *Case 2.* Else $(|J_i| \leq \sqrt{m})$, then
   (a) Consider the set J_i to be the set of vertices in the graph that lost an edge to the states removed since the last iteration that executed Case 1.
   (b) We do a lock-step search from every state s in J_i as follows: we do a DFS from s, until the DFS stops. Once the DFS stops we have identified a bottom scc C_i.
   (c) If $C_i \cap T \neq \emptyset$ or $C_i \cap \mathsf{Pre}(W_1) \neq \emptyset$, then $W_1 := Attr_1(W_1 \cup C_i)$, else $W_2 := Attr_R(W_2 \cup C_i)$.
   (d) $Z_{i+1} := W_1 \cup W_2$. The set $Z_{i+1} \setminus Z_i$ is removed from the graph.
   (e) The set L_{i+1} is the set of states removed from the graph since the last iteration of Case 1 and J_{i+1} be the set of states in the remaining graph with an edge to L_{i+1}.
   (f) If $Z_i = S$, the algorithm stops, otherwise $i := i + 1$ and go to the next iteration.

Correctness and running time. The correctness of the algorithm follows from the correctness of the WINLOSE algorithm. The running time analysis of the algorithm is similar to IMPRALGO algorithm, and this shows the algorithm runs in $O(m \cdot \sqrt{m})$ time. Applying the IMPRWINLOSE algorithm bottom up on the scc decomposition of the MDP gives us a running time of $O(m \cdot \sqrt{K_E})$, where K_E is the maximum number of edges of an scc of the MDP.

Theorem 5. *Given an MDP with a Büchi objective, the* IMPRWINLOSE *algorithm iteratively computes the subsets of the almost-sure winning set and its complement, and in the end correctly computes the set* $\langle\langle 1 \rangle\rangle_{almost}(\textit{Büchi}(T))$ *and the algorithm runs in time* $O(\sqrt{K_E} \cdot m)$*, where* K_E *is the maximum number of edges in an scc of the graph of the MDP.*

Symbolic implementation. The symbolic implementation of IMPRWINLOSE algorithm is obtained in a similar fashion as SYMBIMPRALGO was obtained from IMPRALGO. The only additional step required is the symbolic scc computation. It follows from the results of [11] that scc decomposition can be computed in $O(n)$ symbolic steps. In the following section we will present an improved symbolic scc computation algorithm.

Corollary 1. *Given an MDP with a Büchi objective, the symbolic* IMPRWINLOSE *algorithm (*SYMBIMPRWINLOSE*) iteratively computes the subsets of the almost-sure winning set and its complement, and in the end correctly computes the set* $\langle\langle 1 \rangle\rangle_{almost}(\textit{Büchi}(T))$ *and the algorithm runs in* $O(\sqrt{K_E} \cdot n)$ *symbolic steps, where* K_E *is the maximum number of edges in an scc of the graph of the MDP.*

Remark 2. It is clear from the complexity of the WINLOSE and IMPRWINLOSE algorithms that they would perform better for MDPs where the graph has many small scc's, rather than few large ones.

5 Improved Symbolic SCC Algorithm

A symbolic algorithm to compute the scc decomposition of a graph in $O(n \cdot \log n)$ symbolic steps was presented in [2]. The algorithm of [2] was based on forward and backward searches. The algorithm of [11] improved the algorithm of [2] to obtain an algorithm for scc decomposition that takes at most linear amount of symbolic steps. In this section we present an improved version of the algorithm of [11] that improves the constants of the number of linear symbolic steps required. We first describe the main ideas of the algorithm of [11] and then present our improved algorithm. The algorithm of [11] improves the algorithm of [2] by maintaining the right order for forward sets. The notion of *spine-sets* and *skeleton of a forward set* was designed for this purpose.

Spine-sets and skeleton of a forward set. Let $G = (S, E)$ be a directed graph. Consider a finite path $\tau = (s_0, s_1, \ldots, s_\ell)$, such that for all $0 \leq i \leq \ell - 1$ we have $(s_i, s_{i+1}) \in E$. The path is *chordless* if for all $0 \leq i < j \leq \ell$ such that $j - i > 1$, there is no edge from s_i to s_j. Let $U \subseteq S$. The pair (U, s) is a *spine-set* of G iff G contains a chordless path whose set of states is U that ends in s. For a state s, let $\mathsf{FW}(s)$ denote the set of states that is reachable from s (i.e., reachable by a forward search from s). The set (U, t) is a *skeleton of* $\mathsf{FW}(s)$ iff t is a state in $\mathsf{FW}(s)$ whose distance from s is maximum and U is the set of states on a shortest path from s to t. The following lemma was shown in [11] establishing relation of skeleton of forward set and spine-set.

Lemma 3 ([11]). *Let* $G = (S, E)$ *be a directed graph, and let* $\mathsf{FW}(s)$ *be the forward set of* $s \in S$*. The following assertions hold: (1) If* (U, t) *is a skeleton of a forward-set* $\mathsf{FW}(s)$*, then* $U \subseteq \mathsf{FW}(s)$*. (2) If* (U, t) *is a skeleton of* $\mathsf{FW}(s)$*, then* (U, t) *is a spine-set in* G*.*

The intuitive idea of the algorithm. The algorithm of [11] is a recursive algorithm, and in every recursive call the scc of a state s is determined by computing $\mathsf{FW}(s)$, and then identifying the set of states in $\mathsf{FW}(s)$ having a path to s. The choice of the state to be processed next is guided by the implicit inverse order associated with a possible spine-set. This is achieved as follows: whenever a forward-set $\mathsf{FW}(s)$ is computed, a skeleton of such a forward set is also computed. The order induced by the skeleton is then used for the subsequent computations. Thus the symbolic steps performed to compute $\mathsf{FW}(s)$ is distributed over the scc computation of the states belonging to a skeleton of $\mathsf{FW}(s)$. The key to establish the linear complexity of symbolic steps is the amortized analysis. We now present the main procedure SCCFIND and the main sub-procedure SKELFWD of the algorithm from [11].

Procedures SCCFIND **and** SKELFWD**.** The main procedure of the algorithm is SCCFIND that calls SKELFWD as a sub-procedure. The input to SCCFIND is a graph (S, E) and (A, B), where either $(A, B) = (\emptyset, \emptyset)$ or $(A, B) = (U, \{ s \})$, where (U, s) is a spine-set. If S is $\emptyset$, then the algorithm stops. Else, (a) if (A, B) is $(\emptyset, \emptyset)$, then the procedure picks an arbitrary s from S and proceeds; (b) otherwise, the sub-procedure SKELFWD is invoked to compute the forward set of s together with the skeleton (U', s') of such a forward set. The SCCFIND procedure has the following local variables: FWSet, NewSet, $\mathsf{NewState}$ and SCC. The variable FWSet that maintains the forward set, whereas NewSet and $\mathsf{NewState}$ maintain U' and $\{ s' \}$, respectively. The variable SCC is initialized to s, and then augmented with the scc containing s. The partition of the scc's is updated and finally the procedure is recursively called over:

1. the subgraph of (S, E) is induced by $S \setminus \mathsf{FWSet}$ and the spine-set of such a subgraph is obtained from $(U, \{ t \})$ by subtracting SCC;
2. the subgraph of (S, E) induced by $\mathsf{FWSet} \setminus \mathsf{SCC}$ and the spine-set of such a subgraph obtained from $(\mathsf{NewSet}, \mathsf{NewState})$ by subtracting SCC.

The SKELFWD procedure takes as input a graph (S, E) and a state s, first it computes the forward set $\mathsf{FW}(s)$, and second it computes the skeleton of the forward set. The forward set is computed by symbolic breadth first search, and the skeleton is computed with a stack. The detailed pseudocodes are in [3]. We will refer to this algorithm of [11] as SYMBOLICSCC. The following result was established in [11]: for the proof of the constant 5, refer to the appendix of [11] and the last sentence explicitly claims that every state is charged at most 5 symbolic steps.

Theorem 6 ([11]). *Let* $G = (S, E)$ *be a directed graph. The algorithm* SYMBOLICSCC *correctly computes the scc decomposition of* G *in* $\min\{ 5 \cdot |S|, 5 \cdot D(G) \cdot N(G) + N(G) \}$ *symbolic steps, where* $D(G)$ *is the diameter of* G*, and* $N(G)$ *is the number of scc's in* G*.*

Improved symbolic algorithm. We now present our improved symbolic scc algorithm and refer to the algorithm as IMPROVEDSYMBOLICSCC. Our algorithm mainly modifies the sub-procedure SKELFWD. The improved version of SKELFWD procedure takes an additional input argument Q, and returns an additional output argument that is stored as a set P by the calling SCCFIND procedure. The calling function passes the set U as Q. The way the output P is computed is as follows: at the end of the forward search we have the following assignment: $P := \mathsf{FWSet} \cap Q$. After the forward search, the skeleton of the forward set is computed with the help of a stack. The elements of the stacks are sets of states stored in the forward search. The spine set computation is similar to SKELFWD, the difference is that when elements are popped of the stack, we check if there is a non-empty intersection with P, if so, we break the loop and return. Moreover, for the backward searches in SCCFIND we initialize SCC by P rather than s. We refer to the new sub-procedure as IMPROVEDSKELFWD (detailed pseudocode in [3]).

Correctness. Since s is the last element of the spine set U, and P is the intersection of a forward search from s with U, it means that all elements of P are both reachable from s (since P is a subset of $\mathsf{FW}(s)$) and can reach s (since P is a subset of U). It follows that P is a subset of the scc containing s. Hence not computing the spine-set beyond P does not change the future function calls, i.e., the value of U', since the omitted parts of NewSet are in the scc containing s. The modification of starting the backward search from P does not change the result, since P will anyway be included in the backward search. So the IMPROVEDSYMBOLICSCC algorithm gives the same result as SYMBOLICSCC, and the correctness follows from Theorem 6.

Symbolic steps analysis. We present two upper bounds on the number of symbolic steps of the algorithm. Intuitively following are the symbolic operations that need to be accounted for: (1) when a state is included in a spine set for the first time in IMPROVEDSKELFWD sub-procedure which has two parts: the first part is the forward search and the second part is computing the skeleton of the forward set; (2) when a state is already in a spine set and is found in forward search of IMPROVEDSKELFWD and (3) the backward search for determining the scc. We now present the number of symbolic steps analysis for IMPROVEDSYMBOLICSCC.

1. There are two parts of IMPROVEDSKELFWD, (i) a forward search and (ii) a backward search for skeleton computation of the forward set. For the backward search, we show that the number of steps performed equals the size of NewSet computed. One key idea of the analysis is the proof where we show that a state becomes part of spine-set at most once, as compared to the algorithm of [11] where a state can be part of spine-set at most twice. Because, when it is already part of a spine-set, it will be included in P and we stop the computation of spine-set when an element of P gets included. We now split the analysis in two cases: (a) states that are included in spine-set, and (b) states that are not included in spine-set.
   (a) We charge one symbolic step for the backward search of IMPROVEDSKELFWD (spine-set computation) to each element when it first gets inserted in a spine-set. For the forward search, we see that the number of steps performed is the size of spine-set that would have been computed if we did not stop the skeleton computation. But by stopping it, we are only omitting states that are part of the scc. Hence we charge one symbolic step to each state getting inserted into

spine-set for the first time and each state of the scc. Thus, a state getting inserted in a spine-set is charged two symbolic steps (for forward and backward search) of IMPROVEDSKELFWD the first time it is inserted.

(b) A state not inserted in any spine-set is charged one symbolic step for backward search which determines the scc.

Along with the above symbolic steps, one step is charged to each state for the forward search in IMPROVEDSKELFWD at the time its scc is being detected. Hence each state gets charged at most three symbolic steps. Besides, for computing NewState, one symbolic step is required per scc found. Thus the total number of symbolic steps is bounded by $3 \cdot |S| + N(G)$, where $N(G)$ is the number of scc's of G.

2. Let D^* be the sum of diameters of the scc's in a G. Consider a scc with diameter d. In any scc the spine-set is a shortest path, and hence the size of the spine-set is bounded by d. Thus the three symbolic steps charged to states in spine-set contribute to at most $3 \cdot d$ symbolic steps for the scc. Moreover, the number of iterations of forward search of IMPROVEDSKELFWD charged to states belonging to the scc being computed are at most d. And the number of iterations of the backward search to compute the scc is also at most d. Hence, the two symbolic steps charged to states not in any spine-set also contribute at most $2 \cdot d$ symbolic steps for the scc. Finally, computation of NewSet takes one symbolic step per scc. Hence we have $5 \cdot d + 1$ symbolic steps for a scc with diameter d. We thus obtain an upper bound of $5D^* + N(G)$ symbolic steps.

It is straightforward to argue that the number of symbolic steps of IMPROVEDSCCFIND is at most the number of symbolic steps of SCCFIND. The detailed pseudocode and running time analysis is presented in [3].

Theorem 7. *Let $G = (S, E)$ be a directed graph. The algorithm* IMPROVEDSYMBOLICSCC *correctly computes the scc decomposition of G in* $\min\{\ 3 \cdot |S| + \cdot N(G), 5 \cdot D^*(G) + N(G)\ \}$ *symbolic steps, where $D^*(G)$ is the sum of diameters of the scc's of G, and $N(G)$ is the number of scc's in G.*

Remark 3. Observe that in the worst case SCCFIND takes $5 \cdot n$ symbolic steps, whereas IMPROVEDSCCFIND takes at most $4 \cdot n$ symbolic steps. Thus our algorithm improves the constant of the number of linear symbolic steps required for symbolic scc decomposition.

6 Experimental Results

In this section we present our experimental results. We first present the results for symbolic algorithms for MDPs with Büchi objectives and then for symbolic scc decomposition. We present the results for symbolic steps comparison and the running time comparison is similar (see [3]).

Symbolic algorithm for MDPs with Büchi objectives. We implemented all the symbolic algorithms (including the classical one) and ran the algorithms on randomly generated graphs. If we consider arbitrarily randomly generated graphs, then in most cases it

Table 1. The symbolic steps required by symbolic algorithms for MDPs with Büchi objectives

Number of states	Classical	SymbImprAlgo	SymbImprWinLose
5000	16508	3382	4007
10000	57438	6807	7146
20000	121376	11110	12519

Table 2. The symbolic steps required for scc computation

Number of states	Algorithm from [11]	Our Algorithm	Percentage Improvement
10000	1045	877	16.06
25000	2642	2262	14.38
50000	6298	5398	14.27

gives rise to trivial MDPs. Hence we generated large number of MDP graphs randomly, first chose the ones where all the algorithms required the most number of symbolic steps, and then considered random graphs obtained by small uniform perturbations of them. Our results of average symbolic steps required are shown in Table 1 and show that the new algorithms perform significantly better than the classical algorithm.

Symbolic scc computation. We implemented the symbolic scc decomposition algorithm from [11] and our new symbolic algorithm. We ran the algorithms on randomly generated graphs. Again arbitrarily randomly generated graphs in many cases gives rise to graphs that are mostly disconnected or completely connected. Hence we generated random graphs by first constructing a topologically sorted order of the scc's and then adding edges randomly respecting the topologically sorted order. Our results of average symbolic steps are shown in Table 2 and shows that our new algorithm performs better (around 15% improvement).

References

1. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
2. Bloem, R., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in n log n symbolic steps. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 37–54. Springer, Heidelberg (2000)
3. Chatterjee, K., Henzinger, M., Joglekar, M., Shah, N.: Symbolic algorithms for qualitative analysis of Markov decision processes with Büchi objectives. In: CoRR (2011), `http://arxiv.org/abs/1104.3348`
4. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Simple stochastic parity games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
5. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: SODA 2004, pp. 121–130. SIAM, Philadelphia (2004)

6. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
7. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University (1997)
8. de Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code-aware resource management. In: EMSOFT 2005. ACM Press, New York (2005)
9. de Alfaro, L., Roy, P.: Magnifying-lens abstraction for markov decision processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 325–338. Springer, Heidelberg (2007)
10. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Heidelberg (1997)
11. Gentilini, R., Piazza, C., Policriti, A.: Computing strongly connected components in a linear number of symbolic steps. In: SODA, pp. 573–582 (2003)
12. Howard, H.: Dynamic Programming and Markov Processes. MIT Press, Washington (1960)
13. Immerman, N.: Number of quantifiers is better than number of tape cells. Journal of Computer and System Sciences 22, 384–406 (1981)
14. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. D. Van Nostrand Company (1966)
15. Kwiatkowska, M., Norman, G., Parker, D.: Verifying randomized distributed algorithms with prism. In: Workshop on Advances in Verification, WAVE 2000 (2000)
16. Pogosyants, A., Segala, R., Lynch, N.: Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. Distributed Computing 13(3), 155–186 (2000)
17. Puterman, M.L.: Markov Decision Processes. John Wiley and Sons, Chichester (1994)
18. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, MIT, Technical Report MIT/LCS/TR-676 (1995)
19. Somenzi, F.: Colorado university decision diagram package (1998), `http://vlsi.colorado.edu/pub/`
20. Stoelinga, M.I.A.: Fun with FireWire: Experiments with verifying the IEEE1394 root contention protocol. In: Formal Aspects of Computing (2002)
21. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages. Beyond Words, vol. 3, ch. 7, pp. 389–455. Springer, Heidelberg (1997)

Smoothing a Program Soundly and Robustly*

Swarat Chaudhuri[1] and Armando Solar-Lezama[2]

[1] Rice University
[2] MIT

Abstract. We study the foundations of *smooth interpretation*, a recently-proposed program approximation scheme that facilitates the use of local numerical search techniques (e.g., gradient descent) in program analysis and synthesis. While the popular techniques for local optimization works well only on relatively smooth functions, functions encoded by real-world programs are infested with discontinuities and local irregular features. Smooth interpretation attenuates such features by taking the convolution of the program with a Gaussian function, effectively replacing discontinuous switches in the program by continuous transitions. In doing so, it extends to programs the notion of *Gaussian smoothing*, a popular signal-processing technique used to filter noise and discontinuities from signals.

Exact Gaussian smoothing of programs is undecidable, so algorithmic implementations of smooth interpretation must necessarily be approximate. In this paper, we characterize the approximations carried out by such algorithms. First, we identify three correctness properties—*soundness*, *robustness*, and *β-robustness*—that an approximate smooth interpreter should satisfy. In particular, a smooth interpreter is sound if it computes an abstraction of a program's "smoothed" semantics, and robust if it has arbitrary-order derivatives in the input variables at every point in its input space. Second, we describe the design of an approximate smooth interpreter that provably satisfies these properties. The interpreter combines program abstraction using a new domain with symbolic calculation of convolution.

1 Introduction

Smooth interpretation [4] is a recently-proposed program transformation permitting more effective use of numerical optimization in automated reasoning about programs. Many problems in program analysis and synthesis can be framed as optimization questions—examples include finding program parameters so that the resultant program behavior is *as close as possible* to a specification [4], or the generation of tests that maximize the number of times a certain operation is executed [1]. But rarely are such problems solvable using off-the-shelf numerical optimization engines. This is because search spaces arising in real-world programs are rife with discontinuities—on such spaces, local search algorithms like gradient descent or Newton iteration find themselves unable to converge on a good solution. It is this predicament that smooth interpretation tries to resolve.

* This work was supported by NSF CAREER Award #0953507 and the MIT CSAIL.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 277–292, 2011.

To aid numerical search over the input space of a program P, smooth interpretation transforms P into a smooth mathematical function. For example, if the semantics of P (viewed as a function $P(x_1, x_2)$ of inputs x_1 and x_2) is the discontinuous map in Fig. 1-(a), the "smoothed" version of P will typically have semantics as in Fig. 1-(b). More precisely, smooth interpretation extends to programs the notion of *Gaussian smoothing* [12], an elementary signal-processing technique for filtering out noise and discontinuities from ill-behaved real-world signals. To perform Gaussian smoothing of a signal, one takes the *convolution* of the signal with a Gaussian function. Likewise, to smooth a program, we take the convolution of the denotational semantics of P with a Gaussian function.

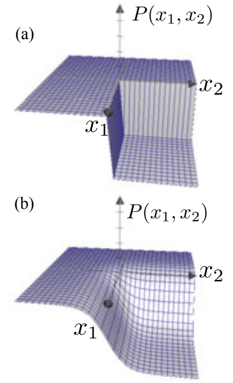


Fig. 1. (a) A discontinuous program. (b) After smoothing.

The smoothing transformation is parameterized by a value β which controls the degree of smoothing. Numerical search algorithms will converge faster and find better local minima when β is high and the function is very smooth, but a bigger β also introduces imprecision, as the minima that is found when using a big β may be far from the real minima. The numerical optimization algorithm from [4] addresses this by starting with a high value of β then reducing it every time a minima is found. When β is reduced, the location of the last minima is used as a starting point for a new round of numerical search.

In our previous work, we showed the effectiveness of this approach for the problem of embedded controller synthesis. Specifically, we showed that the algorithm defined above could find optimal parameters for interesting controllers where simple numerical search would fail. But the benefits of the technique are not limited to parameter synthesis; smooth interpretation constitutes a wholly new form of program approximation, and is likely to have broad impact by opening the door to a wide array of applications of numerical optimization in program analysis.

Smooth interpretation exhibits many parallels with program abstraction, but it also introduces some new and important concerns. The goal of this paper is to understand these concerns by analyzing the foundations of smooth interpretation. In particular, we seek to characterize the approximations that must be made by algorithmic implementations of program smoothing given that computing the exact Gaussian convolution of an arbitrary program is undecidable.

Our concrete contributions are the following:

1. We identify three correctness properties that an algorithmic (and therefore approximate) implementation of smooth interpretation should ideally satisfy: *soundness*, *robustness*, and *β-robustness*.

   While the notion of soundness here is related to the corresponding notion in program abstraction, the two notions are semantically quite different: a sound smooth interpreter computes an abstraction of a "smoothed" semantics of programs. As for robustness, this property states that the function computed by an approximate smooth interpreter has a linearly bounded derivative at every point in its input space—i.e., that even at the points in the input space where P is discontinuous or non-differentiable, the smoothed version of P is only as "steep" as a quadratic function. This property allows gradient-based optimization techniques to be applied to the program—indeed, many widely used algorithms for gradient-based nonlinear optimization [10,7] are known to perform best under such a guarantee.

As for β-robustness, this property demands that the output of the smooth interpreter has a small partial derivative in β. The property is important to the success of the iterative algorithm described above, which relies on repeated numerical search with progressively smaller values of β. Without β-robustness, it would be possible for the approximation to change dramatically with small changes to β, making the algorithm impractical.

2. We give a concrete design for an approximate smooth interpreter (called $\mathit{Smooth}_P(\mathbf{x}, \beta)$) that satisfies the above properties. The framework combines symbolic computation of convolution with abstract interpretation: when asked to smooth a program P, $\mathit{Smooth}_P(\mathbf{x}, \beta)$ uses an abstract interpreter to approximate P by a pair of simpler programs P_{inf} and P_{sup}, then performs a symbolic computation on these approximations. The result of this computation is taken as the semantics of the smoothed version of P.

   The soundness of our method relies on the insight that Gaussian convolution is a monotone map on the pointwise partial order of vector functions. We establish robustness and β-robustness under a weak assumption about β, by bounding the derivative of a convolution. Thus, the techniques used to prove our analysis correct are very different from those in traditional program analysis. Also, so far as we know, the abstract domain used in our construction is new to the program analysis literature.

The paper is structured as follows. In Sec. 2, we recapitulate the elements of smooth interpretation and set up the programming language machinery needed for our subsequent development. In Sec. 3, we introduce our correctness requirements for smoothing; in Sec. 4, we present our framework for smooth interpretation. Sec. 5 studies the properties of interpreters derived from this framework. Our discussion of related work, as well as our conclusions, appear in Sec. 6.

2 Smooth Interpretation

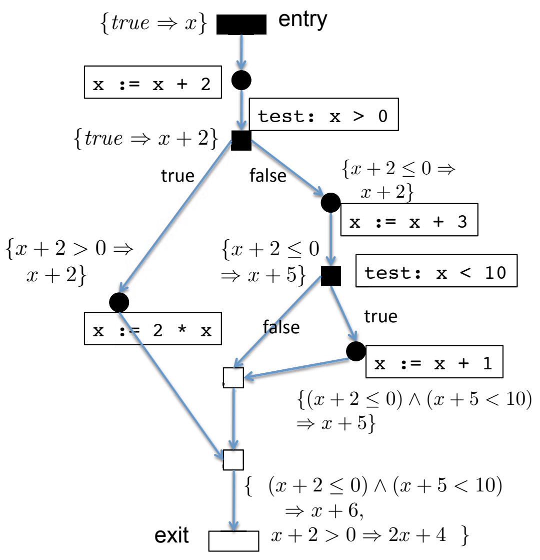
We begin by fixing, for the rest of the paper, a simple language of programs. Our programs are written in a flow-graph syntax [8], and maintain their state in k real-valued variables named $\mathtt{x_1}$ through $\mathtt{x_k}$.

Formally, let *Re* denote the set of linear arithmetic expressions over $\mathtt{x_1}, \ldots, \mathtt{x_k}$, encoding linear transformations of the type $\mathbb{R}^k \rightarrow \mathbb{R}^k$. Also, let *Be* the set of boolean expressions of the form $Q > 0$ or $Q \geq 0$, where $Q \in \mathbb{R}^k \rightarrow \mathbb{R}$. A program P in our language is a directed graph. Nodes of this graph can be of five types:

- An *entry node* has one outgoing edge and no incoming edge, and an *exit node* has one incoming edge and no outgoing edge. A program has a single entry node and a single exit node.

- An *assignment node* has a single incoming edge and single outgoing edge. Each assignment node u is labeled with an expression $E \in \mathit{Re}$. Intuitively, this expression is the r-value of the assignment.
- A *test node* u has one incoming edge and two outgoing edges (known as the true and false-edges), and is labeled by a boolean expression $\mathit{Test}(u) \in \mathit{Be}$. Intuitively, u is a conditional branch.
- A *junction node* has a single outgoing edge and two incoming edges.

For example, Fig. 2 depicts a simple program over a single variable.



$\{true \Rightarrow x\}$
entry
x := x + 2
test: x > 0
$\{true \Rightarrow x+2\}$
true
false
$\{x+2 \leq 0 \Rightarrow x+2\}$
x := x + 3
$\{x+2 > 0 \Rightarrow x+2\}$
$\{x+2 \leq 0 \Rightarrow x+5\}$
test: x < 10
true
false
x := 2 * x
x := x + 1
$\{(x+2 \leq 0) \wedge (x+5 < 10) \Rightarrow x+5\}$
$\{ (x+2 \leq 0) \wedge (x+5 < 10) \Rightarrow x+6, x+2 > 0 \Rightarrow 2x+4 \}$
exit

Fig. 2. A program and its collecting semantics

Semantics. The intuitive operational semantics of P is that it starts executing at its entry node, taking transitions along the edges, and terminates at the exit node. For our subsequent development, however, a denotational semantics as well as an abstract-interpretation-style collecting semantics are more appropriate than an operational one. Now we define these semantics.

Let a *state* of P be a vector $\mathbf{x} = \langle x_1, \ldots, x_k \rangle \in \mathbb{R}^k$, where each x_i captures the value of the variable $\mathtt{x_i}$. For each arithmetic expression $E \in \mathit{Re}$, the denotational semantics $[\![E]\!](\mathbf{x}) : \mathbb{R}^k \to \mathbb{R}^k$ produces the value of E at the state $\mathbf{x}$. For each $B \in \mathit{Be}$, the denotation function $[\![B]\!]$ produces $[\![B]\!](\mathbf{x}) = 0$ if B is false at the state $\mathbf{x}$, and 1 otherwise.

To define the semantics of P, we need some more machinery. Let a *guarded linear expression* be an expression of the form

$$\texttt{if } B \texttt{ then } F \texttt{ else } \mathbf{0},$$

where B is a conjunction of linear inequalities over the variables $x_1, \ldots, x_k$, and F is a linear arithmetic expression. We abbreviate the above expression by the notation $(B \Rightarrow F)$, and lift the semantic function $[\![\circ]\!]$ to such expressions:

$$[\![B \Rightarrow F]\!](\mathbf{x}) = \texttt{if } [\![B]\!](\mathbf{x}) \texttt{ then } [\![F]\!](\mathbf{x}) \texttt{ else } \mathbf{0} = [\![B]\!](\mathbf{x}) \cdot [\![F]\!](\mathbf{x}).$$

The *collecting semantics* of P is given by a map Ψ_P that associates with each node u of P a set $\Psi_P(u)$ of guarded linear expressions. Intuitively, each such expression captures the computation carried out along a path in P ending at u. As the program P can have "loops," $\Psi_P(u)$ is potentially infinite.

In more detail, we define the sets $\Psi_P(u)$ so that they form the least fixpoint of a monotone map. For each node u of P, $\Psi_P(u)$ is the least set of guarded linear expressions satisfying the following conditions:

- If u is the entry node and its out-edge goes to v, then $\{(\mathit{true} \Rightarrow \mathbf{x})\} \subseteq \Psi_P(v)$.
- Suppose u is an assignment node labeled by E and its outgoing edge leads to v. Let $\circ$ be the usual composition operator over expressions. Then for all $(B \Rightarrow F) \in \Psi_P(u)$, the expression $B \Rightarrow (E \circ F)$ is in $\Psi_P(v)$.
- Suppose u is a branch node, and let v_t and v_f respectively be the targets of the true- and false-edges out of it. Let $Q_t = \mathit{Test}(u) \circ F$ and $Q_f = (\neg\, \mathit{Test}(u)) \circ F$; then, for all $(B \Rightarrow F) \in \Psi_P(u)$, we have
$$(B \wedge Q_t) \Rightarrow F \in \Psi_P(v_t) \qquad (B \wedge Q_f) \Rightarrow F \in \Psi_P(v_f).$$
- If u is a junction node and its outgoing edge leads to v, then $\Psi_P(u) \subseteq \Psi_P(v)$.

For example, consider the program in Fig. 2. Most nodes u of this program are labeled with $\Psi_P(u)$. We note that one of the three control flow paths to the exit node *ex* is infeasible; as a result $\Psi_P(ex)$ has two formulas rather than three.

The denotational semantics $[\![P]\!]$ of P is now defined using the above collecting semantics. Let *ex* be the exit node of P. We define:

$$[\![P]\!](\mathbf{x}) = \sum_{(B \Rightarrow F) \in \Psi_P(ex)} [\![B]\!](\mathbf{x}) \cdot [\![F]\!](\mathbf{x}).$$

Intuitively, for any $\mathbf{x} \in \mathbb{R}^k$, $[\![P]\!](\mathbf{x})$ is the output of P on input $\mathbf{x}$.

Smoothed semantics. Now we recall the definition [4] of the *smoothed semantics* of programs, which is the semantics that smooth interpretation seeks to compute. To avoid confusion, the previously defined semantics is from now on known as the *crisp semantics*.

Let $\beta > 0$ be a real-valued *smoothing parameter*, and let $\mathcal{N}(\mathbf{x}, \beta)$ be the joint density function of k independent normal variables, each with mean 0 and standard deviation β. In more detail, letting $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ as before, $\mathcal{N}(\mathbf{x}, \beta)$ is given by $\mathcal{N}(\mathbf{x}, \beta) = \frac{1}{(2\pi\beta^2)^{k/2}} e^{-\frac{\sum_{i=1}^{k} x_i^2}{2\beta^2}} = \prod_{i=0..k} \mathcal{N}(x_i)$.

The *smoothed semantics* $\overline{[\![P]\!]}_\beta : \mathbb{R}^k \to \mathbb{R}^k$ of a program P with respect to β is obtained by taking the *convolution* of $\mathcal{N}$ and the crisp semantics of P as shown be the following equation.

$$\begin{aligned} \overline{[\![P]\!]}_\beta(\mathbf{x}) &= \textstyle\int_{\mathbf{r} \in \mathbb{R}^k} [\![P]\!](\mathbf{r})\ \mathcal{N}(\mathbf{x} - \mathbf{r}, \beta)\ d\mathbf{r} \\ &= \textstyle\sum_{(B \Rightarrow F) \in \Psi_P(ex)} \int_{\mathbf{r} \in \mathbb{R}^k} [\![B \Rightarrow F]\!](\mathbf{r})\ \mathcal{N}(\mathbf{x} - \mathbf{r}, \beta)\ d\mathbf{r}. \end{aligned} \tag{1}$$

As before, *ex* refers to the exit node of P. Note that because convolution is a commutative operator, we have the property

$$\overline{[\![P]\!]}_\beta(\mathbf{x}) = \int_{\mathbf{r} \in \mathbb{R}^k} [\![P]\!](\mathbf{x} - \mathbf{r})\ \mathcal{N}(\mathbf{r}, \beta)\ d\mathbf{r}.$$

When β is clear from context, we denote $\overline{\llbracket P \rrbracket}_\beta$ by $\overline{\llbracket P \rrbracket}$, and $\mathcal{N}(\mathbf{x}, \beta)$ by $\mathcal{N}(\mathbf{x})$.

One of the properties of smoothing is that even if $\llbracket P \rrbracket$ is highly discontinuous, $\overline{\llbracket P \rrbracket}$ is a smooth mathematical function that has all its derivatives defined at every point in $\mathbb{R}^k$. The smoothing parameter β can be used to control the extent to which $\llbracket P \rrbracket$ is smoothed by the above operation—the higher the value of β, the greater the extent of smoothing.

Example 1. Consider a program P over one variable x such that $\llbracket P \rrbracket(x) = \texttt{if}\ x > a\ \texttt{then}\ 1\ \texttt{else}\ 0$, where $a \in \mathbb{R}$. Let erf be the Gauss error function. We have

$$\begin{aligned}\overline{\llbracket P \rrbracket}(x) &= \textstyle\int_{-\infty}^{\infty} \llbracket P \rrbracket(y) \mathcal{N}(x-y)\ dy = 0 + \int_a^{\infty} \mathcal{N}(x-y)\ dy \\ &= \textstyle\int_0^{\infty} \frac{1}{\sqrt{2\pi}\beta}\ e^{-(y-x+a)^2/2\beta^2} dy = \frac{1+\mathrm{erf}(\frac{x-a}{\sqrt{2}\beta})}{2}.\end{aligned}$$

Figure 3-(a) plots the crisp semantics $\llbracket P \rrbracket$ of P with $a = 2$, as well as $\overline{\llbracket P \rrbracket}$ for $\beta = 0.5$ and $\beta = 3$. While $\llbracket P \rrbracket$ has a discontinuous "step," $\overline{\llbracket P \rrbracket}$ is a smooth S-shaped curve, or a *sigmoid*. Note that as we decrease the "tuning knob" β, the sigmoid $\overline{\llbracket P \rrbracket}$ becomes steeper and steeper, and at the limit, approaches $\llbracket P \rrbracket$.

Now consider the program P' such that $\llbracket P' \rrbracket(x) = \texttt{if}\ a < x < c\ \texttt{then}\ 1\ \texttt{else}\ 0$, where $a, c \in \mathbb{R}$ and $a < c$. The "bump-shaped" functions obtained by smoothing P' are plotted in Figure 3-(b) (here $a = -5$, $c = 5$, and β has two values 0.5 and 2). Note how the discontinuities are smoothed.

Now we consider an even more interesting program, one that is key to the main results of this paper. Consider $\llbracket P'' \rrbracket(x) = \llbracket B \Rightarrow F \rrbracket(x)$, where B is the boolean expression $a < x < b$ for constants a, b, and $F(x) = \alpha \cdot x + \gamma$ for constants α, γ. In this case, the smoothed semantics of P'' can be evaluated symbolically as follows:

$$\begin{aligned}\overline{\llbracket P'' \rrbracket}(x) &= \int_a^b (\alpha y + \gamma)\ \mathcal{N}(x-y)\ dy = \alpha \int_a^b y\ \mathcal{N}(x-y)\ dy + \gamma \int_a^b \mathcal{N}(x-y)\ dy \\ &= \frac{(\alpha\ x + \gamma) \cdot (\mathrm{erf}(\frac{b-x}{\sqrt{2}\beta}) - \mathrm{erf}(\frac{a-x}{\sqrt{2}\beta}))}{2} + \frac{\beta\ \alpha\ (e^{-(a-x)^2/2\beta^2} - e^{-(b-x)^2/2\beta^2})}{\sqrt{2\pi}}.\end{aligned}$$

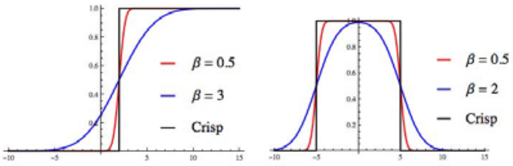


Fig. 3. (a) A sigmoid (b) A bump

Smooth interpretation. We use the phrase "*smooth interpreter* of a program P" to refer to an algorithm that can execute P according to the smoothed semantics $\overline{\llbracket \circ \rrbracket}_\beta$. The primary application of smooth interpretation is to enable the use of numerical search for parameter optimization problems. For example, suppose our goal is to compute an input $\mathbf{x}$ for which $\llbracket P \rrbracket(\mathbf{x})$ is minimal. For a vast number of real-world programs, discontinuities in $\llbracket P \rrbracket$ would preclude the use of local numerical methods in this minimization problem. By eliminating such discontinuities, smooth interpretation can make numerical search practical.

In a way, smooth interpretation is to numerical optimization what abstraction is to model checking: a mechanism to approximate challenging local transitions in a program-derived search space. But the smoothed semantics that we use is very different from the collecting semantics used in abstraction. Rather, the smoothed semantics of P can be seen to be the expectation of a probabilistic semantics of P.

Consider an input $\mathbf{x} \in \mathbb{R}^k$ of P, and suppose that, before executing P, we randomly perturb $\mathbf{x}$ following a k-D normal distribution with independent components and standard deviation β. Thus, the input of P is now a random variable X following a normal distribution $\mathcal{N}$ with mean $\mathbf{x}$ and similar shape as the one used in the perturbation. Now let us execute P on X with crisp semantics. Consider the *expectation* of the output $[\![P]\!](\mathsf{X})$: $\mathbf{Exp}\,[[\![P]\!](\mathsf{X})] = \int_{-\infty}^{\infty} [\![P]\!](\mathbf{r})\ \mathcal{N}(\mathbf{x}-\mathbf{r})\ d\mathbf{r} = \overline{[\![P]\!]}(\mathbf{x})$. Here $\overline{[\![P]\!]}$ is computed using the Gaussian $\mathcal{N}$. In other words, the smoothed semantics of P is the expected crisp semantics of P under normally distributed perturbations to the program inputs.

Now that we have defined how a smooth interpreter must *behave*, consider the question of how to algorithmically implement such an interpreter. On any input $\mathbf{x}$, an idealized smooth interpreter for P must compute $\overline{[\![P]\!]}_\beta(\mathbf{x})$—in other words, integrate the semantics of P over a real space. This problem is of course undecidable in general; therefore, any algorithmic implementation of a smooth interpreter must necessarily be *approximate*. But when do we judge such an approximation to be "correct"? Now we proceed to answer this question.

3 Approximate Smooth Interpreters and their Correctness

In this section, we define three correctness properties for algorithmic implementations of smooth interpretation: *soundness*, *robustness*, and *β-robustness*. While an algorithm for smooth interpretation of a program must necessarily be approximate, these desiderata impose limits on the approximations that it makes.

Formally, we let an *approximate smooth interpreter* $\mathit{Smooth}_P(\mathbf{x}, \beta)$ for P be an algorithm with two inputs: an input $\mathbf{x} \in \mathbb{R}^k$ and a smoothing parameter $\beta \in \mathbb{R}^+$. Given these, Smooth_P returns a symbolic representation of a set $\mathbf{Y} \subseteq \mathbb{R}^k$. To avoid notation-heavy analytic machinery, we restrict the sets returned by Smooth_P to be *intervals* in $\mathbb{R}^k$. Recall that such an interval is a Cartesian product $\langle [l_1, u_1], \ldots, [l_k, u_k] \rangle$ of intervals over $\mathbb{R}$; the interval can also be represented more conveniently as a pair of vectors $[\langle l_1, \ldots, l_k \rangle, \langle u_1, \ldots, u_k \rangle]$; from now on, we denote the set of all such intervals by $\mathbf{I}$.

Soundness. Just as a traditional static analysis of P is sound if it computes an abstraction of the crisp semantics of P, an approximate smooth interpreter is sound if it abstracts the smoothed semantics of P. In other words, the interval returned by Smooth_P on any input $\mathbf{x}$ bounds the output of the idealized smoothed version of P on $\mathbf{x}$. We define:

Definition 1 (Soundness). *An approximate smooth interpreter $\mathit{Smooth}_P : \mathbb{R}^k \times \mathbb{R}^+ \to \mathbf{I}$ is* sound *iff for all $\mathbf{x} \in \mathbb{R}^k$, $\overline{[\![P]\!]}_\beta(\mathbf{x}) \in \mathit{Smooth}_P(\mathbf{x}, \beta)$.*

Robustness. A second requirement, critical for smooth interpreters but less relevant in abstract interpretation, is *robustness.* This property asserts that for all $\mathbf{x}$ and β, $\mathit{Smooth}_P(\mathbf{x}, \beta)$ has derivatives of all orders, and that further, its partial derivative with respect to the component scalars of $\mathbf{x}$ is small—i.e., bounded by a function linear in $\mathbf{x}$ and β.

The first of the two requirements above simply asserts that $\mathit{Smooth}_P(\mathbf{x}, \beta)$ computes a smooth function, in spite of all the approximations carried out for computability. The rationale behind the second requirement is apparent when we consider smooth interpretation in the broader context of local numerical optimization of programs. A large number of numerical search routines work by sampling the input space under the expectation that the derivative around each sample point is well defined. Our requirement guarantees this.

In fact, by putting a linear bound on the derivative of Smooth_P, we give a stronger guarantee: the absence of regions of extremely steep descent that can lead to numerical instability. Indeed, robustness implies that even at the points in the input space where P is discontinuous, the gradient of Smooth_P is Lipschitz-continuous—i.e., Smooth_P is only as "steep" as a quadratic function. Many algorithms for nonlinear optimization [10,7] demand a guarantee of this sort for best performance. As we will see later, we can implement a smooth interpreter that is robust (under a weak additional assumption) even in our strong sense. Let us now define:

Definition 2 (Robustness). *$\mathit{Smooth}_P(\mathbf{x}, \beta)$ is* robust *if $\frac{\partial}{\partial x_i}\mathit{Smooth}_P(\mathbf{x}, \beta)$ exists at all $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ and for all i, and there exists a* linear *function $K(\mathbf{x}, \beta)$ in $\mathbf{x}$ that satisfies $\|\frac{\partial}{\partial x_i}\mathit{Smooth}_P(\mathbf{x}, \beta)\| \leq K(\mathbf{x}, \beta)$ for all $\mathbf{x}, i, \beta$.*

The definition above abuses notation somewhat because, as you may recall, $\mathit{Smooth}_P(\mathbf{x}, \beta)$ actually produces a Cartesian product of intervals, as opposed to a real number; so the derivative $\frac{\partial}{\partial x_i}\mathit{Smooth}_P(\mathbf{x}, \beta)$ is actually a pair of vectors $[\langle \frac{\partial}{\partial x_i} l_1, \ldots, \frac{\partial}{\partial x_i} l_k \rangle, \langle \frac{\partial}{\partial x_i} u_1, \ldots, \frac{\partial}{\partial x_i} u_k \rangle]$. The measure for such a pair of vectors is a simple Euclidian measure that adds the squares of each of the components.

β-robustness. Another correctness property for an approximate smooth interpreter is that it produces functions that have small partial derivatives with respect to the smoothing parameter β. In more detail, the derivative $\frac{\partial \mathit{Smooth}_P(\mathbf{x}, \beta)}{\partial \beta}$ must be bounded by a function linear in $\mathbf{x}$ and β. We consider this property important because of the way our numerical search algorithm from [4] uses smoothing: starting with a large β and progressively reducing it, improving the quality of the approximation in a way akin to the abstraction-refinement loop in program verification. The property of β-robustness guarantees that the functions optimized in two successive iterations of this process are not wildly different. In other words, the optima of one iteration of the process do not become entirely suboptimal in the next iteration.

Formally, we define the property of β-robustness as follows:

Definition 3 (Robustness in β). *$\mathit{Smooth}_P(\mathbf{x}, \beta)$ is* robust in β *if the partial derivative $\frac{\partial}{\partial \beta}\mathit{Smooth}_P(\mathbf{x}, \beta)$ exists for all $\beta > 0$, and there is a linear function $K(\mathbf{x}, \beta)$ such that for all $\mathbf{x}$ and β, $\|\frac{\partial}{\partial \beta}\mathit{Smooth}_P(\mathbf{x}, \beta)\| \leq K(\mathbf{x}, \beta)$.*

4 Designing a Smooth Interpreter

Now we present a framework for approximate smooth interpretation that satisfies the correctness properties defined in the previous section. We exploit the fact that under certain restrictions on a program P, it is possible to build an *exact* smooth interpreter for P—i.e., an algorithm that computes the smoothed semantics $\overline{[\![P]\!]}$ exactly. The idea behind our construction is to approximate P by programs for which exact smooth interpreters can be constructed.

Let us consider guarded linear expressions (defined in Sec. 2); recall that for nodes u of P, $\Psi_P(u)$ is a possibly-infinite set of guarded linear expressions $(B \Rightarrow F)$. By Eqn. (1), computing the exact smoothed semantics of P amounts to computing a sum of Gaussian convolutions of guarded linear expressions.

Unfortunately, the convolution integral of a general guarded linear expression does not have a clean symbolic solution. We overcome this problem by abstracting each such expression using an *interval-guarded linear expression* $B_{int} \Rightarrow F$, where B_{int} is an interval in $\mathbb{R}^k$ obtained through a Cartesian abstraction of B. From an argument as in Example 1, if an expression is interval-guarded and linear, then its exact smoothed semantics can be computed in closed-form.

The above strategy alone is not enough to achieve convergence, given that $\Psi_P(u)$ can be infinite. Hence we use *pairs* of interval-guarded linear expressions of the form $\langle (B_{int} \Rightarrow F_{sup}), (B_{int} \Rightarrow F_{inf}) \rangle$ to abstract unbounded sets of linear expressions guarded by subintervals of B_{int}. Such a tuple is known as an *interval-guarded bounding expression*, and abbreviated by the notation $\langle B_{int}, F_{sup}, F_{inf} \rangle$.

To see what such an abstraction means, let us define the *pointwise ordering relation* $\preceq$ among functions of type $\mathbb{R}^k \rightarrow \mathbb{R}^k$ as follows: $F_1 \preceq F_2$ iff for all $\mathbf{x} \in \mathbb{R}^k$, we have $F_1(\mathbf{x}) \leq F_2(\mathbf{x})$. We lift this function to arithmetic expressions E, letting $E_1 \preceq E_2$ iff $[\![E_1]\!] \preceq [\![E_2]\!]$. We guarantee that if $\langle B_{int}, F_{sup}, F_{inf} \rangle$ abstracts a set S of interval-guarded linear expressions, then for all $(B \Rightarrow F) \in S$, we have

$$(B_{int} \Rightarrow F_{inf}) \preceq (B \Rightarrow F) \preceq (B_{int} \Rightarrow F_{sup}).$$

In fact, rather than tracking just a single interval-guarded bounding expression, an abstract state in our framework tracks bounded sets of such expressions. Using this abstraction, it is possible to approximate the semantics $[\![P]\!]$ of P by two programs P_{sup} and P_{inf}, whose semantics can be represented as a sum of a

1. Given a program P for which an approximate smooth interpreter is to be constructed, use abstract interpretation to obtain programs P_{sup} and P_{inf} such that:
   - P_{sup} and P_{inf} are interval-guarded linear programs.
   - $[\![P_{inf}]\!] \preceq [\![P]\!] \preceq [\![P_{sup}]\!]$ (here $\preceq$ is the pointwise ordering over functions).
2. Construct symbolic representations of $\overline{[\![P_{sup}]\!]}_\beta$ and $\overline{[\![P_{inf}]\!]}_\beta$.
3. Let the approximate smooth interpreter for P be a function that on any $\mathbf{x}$ and β, returns the Cartesian interval $[\overline{[\![P_{inf}]\!]}_\beta(\mathbf{x}), \overline{[\![P_{sup}]\!]}_\beta(\mathbf{x})]$.

Fig. 4. The approximate smooth interpretation algorithm

bounded number of terms, each term being an interval-guarded linear expression. (We call such programs *interval-guarded linear programs.*)

The smoothed semantics of P_{sup} and P_{inf} can be computed in closed form, leading to an approximate smooth interpreter that is sketched in Fig. 4. In the rest of this section, we complete the above algorithm by describing the abstract interpreter in Step (1) and the analytic calculation in Step (2).

Step 1: Abstraction Using Interval-Guarded Bounding Expressions

Abstract domain. An *abstract state* σ in our semantics is either a bounded-sized set of interval-guarded bounding expressions (we let N be a bound on the number of elements in such a set), or the special symbol $\top$. The set of all abstract states is denoted by $\mathcal{A}$.

As usual, we define a partial order over the domain $\mathcal{A}$. Before defining this order, let us define a partial order $\trianglelefteq$ over the universe IGB that consists of all interval-guarded bounding expressions, as well as $\top$. For all σ, we have $\sigma \trianglelefteq \top$. We also have:

$$\begin{aligned}\langle B, F_{sup}, F_{inf}\rangle \trianglelefteq \langle B', F'_{sup}, F'_{inf}\rangle \quad \text{iff} \quad & B \Rightarrow B' \text{ and}\\ & (B' \wedge B) \Rightarrow (F_{sup} \preceq F'_{sup}) \wedge (F'_{inf} \preceq F_{inf}) \text{ and}\\ & (B' \wedge \neg B) \Rightarrow (\mathbf{0} \preceq F'_{sup}) \wedge (F'_{inf} \preceq \mathbf{0}).\end{aligned}$$

Intuitively, in the above, B is a subinterval of B', and the expressions $(B' \Rightarrow F'_{inf})$ and $(B' \Rightarrow F'_{sup})$ define more relaxed bounds than $(B \Rightarrow F_{inf})$ and $(B \Rightarrow F_{sup})$.

Note that $\trianglelefteq$ is *not* a lattice relation—e.g., the interval-guarded expressions $(1 < x < 2) \Rightarrow 1$ and $(3 < x < 4) \Rightarrow 5$ do not have a unique least upper bound. However, it is easy to give an algorithm $\sqcup_{IGB}$ that, given $H_1, H_2 \in IGB$, returns a *minimal*, albeit nondeterministic, upper bound H of H_1 and H_2 (i.e., $H_1 \trianglelefteq H$, $H_2 \trianglelefteq H$, and there is no $H' \neq H$ such that $H' \trianglelefteq H$, $H_1 \trianglelefteq H'$, and $H_2 \trianglelefteq H'$).

Now can we define the partial order $\sqsubseteq$ over $\mathcal{A}$ that we use for abstract interpretation. For $\sigma_1, \sigma_2 \in \mathcal{A}$, we have $\sigma_1 \sqsubseteq \sigma_2$ iff either $\sigma_2 = \top$, or if for all $H \in \sigma_1$, there exists $H' \in \sigma_2$ such that $H \trianglelefteq H'$.

Once again, we can construct an algorithm $\sqcup_{\mathcal{A}}$ that, given $\sigma_1, \sigma_2 \in \mathcal{A}$, returns a minimal upper bound σ for σ_1 and σ_2. If σ_1 or σ_2 equals $\top$, the algorithm simply returns $\top$. Otherwise, it executes the following program:

1. Let $\sigma' := \sigma_1 \cup \sigma_2$.
2. While $|\sigma'| > N$, repeatedly: (a) Nondeterministically select two elements $H_1, H_2 \in \sigma'$; (b) assign $\sigma' := \sigma' \setminus \{H_1, H_2\} \cup (H_1 \sqcup_{IGB} H_2)$;
3. Return σ'.

Abstraction. The *abstract semantics* of the program P is given by a map $\Psi_P^{\#}$ that associates an abstract state $\Psi_P^{\#}(u)$ with each node u of P. To define this semantics, we need some more machinery. First, we need a way to capture the effect of an assignment node labeled by an expression E, on an abstract state σ. To this end, we define a notation $(E \circ \sigma)$ that denotes the composition of E and σ. We have $E \circ \top = \top$ for all E. If $\sigma \neq \top$, then we have $(E \circ \sigma) = \{\langle B, (E \circ F_{sup}), (E \circ F_{inf})\rangle : \langle B, F_{sup}, F_{inf}\rangle \in \sigma\}$. Applied to the abstract state σ, the assignment produces the abstract state $(E \circ \sigma)$.

Second, we must be able to propagate an abstract state σ through a test node labeled by the boolean expression C. To achieve this, we define, for each abstract state σ and boolean expression C, the composition $(C \circ \sigma)$ of C and σ. In fact, we begin by defining the composition of $(C \circ H)$, where $H = \langle B, F_{sup}, F_{inf}\rangle$ is an interval-guarded boolean expression.

The idea here is that if a test node is labeled by C and H reaches the node, then $(C \circ H)$ is propagated along the true-branch. For simplicity, let us start with the case $B = \mathit{true}$. Clearly, $(C \circ H)$ should be of the form $\langle B', F_{sup}, F_{inf}\rangle$ for some B'. To see what B' should be, consider the scenario in Fig. 5, which shows the points $F_{sup}(\mathbf{x})$ and $F_{inf}(\mathbf{x})$ for a fixed $\mathbf{x} \in \mathbb{R}^2$. For all F such that $F_{inf} \preceq F \preceq F_{sup}$, the point $F(\mathbf{x})$ must lie within the dashed box. So long as an "extreme point" of this box satisfies the constraint C (the region to the left of the inclined line), $\mathbf{x}$ should satisfy B'.

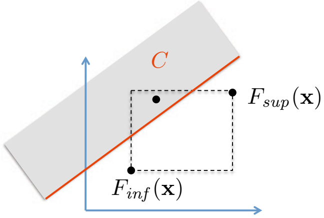


Fig. 5. Propagation through test C

We need some more notation. For each function $F : \mathbb{R}^k \to \mathbb{R}^k$, let us define a collection of "components" $F^1, \ldots, F^k$ such that for all i, $F^i(\mathbf{x}) = (F(\mathbf{x}))(i)$. A collection of component functions $F_1, \ldots, F_k : \mathbb{R}^k \to \mathbb{R}$ can be "combined" into a function $F = \langle F_1, \ldots, F_k\rangle : \mathbb{R}^k \to \mathbb{R}^k$, where for all $\mathbf{x}$, $F(\mathbf{x}) = \langle F_1(\mathbf{x}), \ldots, F_k(\mathbf{x})\rangle$. The extreme points of our dashed box can now be seen to be obtained by taking all possible combinations of components from F_{sup} and F_{inf} and combining those functions. Then the property that some extreme point of the dashed box of Fig. 5 satisfies C is captured by the formula

$$(C \circ \langle F^1_{sup}, F^2_{sup}\rangle) \vee (C \circ \langle F^1_{sup}, F^2_{inf}\rangle) \vee (C \circ \langle F^1_{inf}, F^2_{sup}\rangle) \vee (C \circ \langle F^1_{inf}, F^2_{inf}\rangle).$$

The above can now be generalized to k-dimensions and the case $B \neq \mathit{true}$. The composition of C with an interval-guarded boolean expression $\langle B, F_{sup}, F_{inf}\rangle$ is defined to be $C \circ \langle B, F_{sup}, F_{inf}\rangle = \langle B', F_{sup}, F_{inf}\rangle$, where

$$B' = B \wedge \left(\bigvee_{G^i \in \{F^i_{sup}, F^i_{inf}\}} (C \circ \langle G^1, \ldots, G^k\rangle) \right).$$

Let us now lift this composition operator to abstract states. We define $C \circ \top = \top$ for all C. For all $\sigma \neq \top$, we have $(C \circ \sigma) = \{C \circ \langle B, F_{sup}, F_{inf}\rangle : \langle B, F_{sup}, F_{inf}\rangle \in \sigma\}$. Finally, for any boolean expression C, let us define $C^{\#}$ to be an interval that overapproximates C.

The abstract semantics $\Psi^{\#}_P$ of P is now defined using the algorithm in Figure 6. We note that $\Psi^{\#}_P$ can have different values depending on the sequence of nondeterministic choices made by our upper-bound operators. However, *every* resolution of such choices leads to an abstract semantics that can support a sound and robust approximate smooth interpreter. Consequently, from now on,

1. If u is the entry node of P and its outgoing edge leads to v, then assign $\Psi^{\#}_P(v) := \{\langle \mathit{true}, \mathbf{x}, \mathbf{x}\rangle\}$. Assign $\Psi^{\#}_P(v') = \emptyset$ for every other node v'.
2. Until fixpoint, repeat:
   (a) If u is an assignment node labeled by E and its outgoing edge leads to v, then assign $\Psi^{\#}_P(v) := \Psi^{\#}_P(v) \sqcup (E \circ \Psi^{\#}_P(u))$.
   (b) Suppose u is a branch node; then let v_t and v_f respectively be the targets of the true- and false-edges out of it, and let $Q_t = \mathit{Test}(u)$ and $Q_f = \neg \mathit{Test}(u)$
   $$\begin{aligned}\Psi^{\#}_P(v_t) &:= \Psi^{\#}_P(v_t) \sqcup \{Q_t \circ \langle B, F_{\mathit{sup}}, F_{\mathit{inf}}\rangle : \langle B, F_{\mathit{sup}}, F_{\mathit{inf}}\rangle \in \Psi^{\#}_P(u)\}\\ \Psi^{\#}_P(v_f) &:= \Psi^{\#}_P(v_f) \sqcup \{Q_f \circ \langle B, F_{\mathit{sup}}, F_{\mathit{inf}}\rangle : \langle B, F_{\mathit{sup}}, F_{\mathit{inf}}\rangle \in \Psi^{\#}_P(u)\}\end{aligned}$$
   (c) If u is a junction node with an out-edge to v, then $\Psi^{\#}_P(v) := \Psi^{\#}_P(u) \sqcup \Psi^{\#}_P(v)$.

Fig. 6. Algorithm to compute $\Psi^{\#}_P$

we will ignore the fact that $\Psi^{\#}_P$ actually represents a family of maps, and instead view it as a function of P.

Widening. As our abstract domain is infinite, our fixpoint computation does not guarantee termination. For termination of abstract interpretation, our domain needs a widening operator [8]. For example, one such operator $\bigtriangledown$ can be defined as follows.

First we define $\bigtriangledown$ on interval-guarded bounding expressions. Let us suppose $\langle B_w, F''_{\mathit{sup}}, F''_{\mathit{inf}}\rangle = \langle B, F_{\mathit{sup}}, F_{\mathit{inf}}\rangle \bigtriangledown \langle B', F'_{\mathit{sup}}, F'_{\mathit{inf}}\rangle$. Then we have:

- $B_w = B \bigtriangledown_{\mathit{int}} B'$, where $\bigtriangledown_{\mathit{int}}$ is the standard widening operator for the interval domain [8].
- F''_{sup} is a minimal function in the pointwise order $\preceq$ such that for all $\mathbf{x} \in B_w$, we have $F''_{\mathit{sup}}(\mathbf{x}) \geq (B \Rightarrow F_{\mathit{sup}})(\mathbf{x})$ and $F''_{\mathit{sup}}(\mathbf{x}) \geq (B' \Rightarrow F'_{\mathit{sup}})(\mathbf{x})$.
- F''_{inf} is a maximal function such that for all $\mathbf{x} \in B_w$, we have $F''_{\mathit{sup}}(\mathbf{x}) \leq (B \Rightarrow F_{\mathit{inf}})(\mathbf{x})$ and $F''_{\mathit{inf}}(\mathbf{x}) \leq (B' \Rightarrow F'_{\mathit{inf}})(\mathbf{x})$.

This operator is now lifted to abstract states in the natural way.

Computing P_{sup} and P_{inf}. Now we can compute the interval-guarded linear programs P_{sup} and P_{inf} that bound P. Let ex be the exit node of P, and let $\Psi^{\#}_P(\mathit{ex}) = \{\langle B^1, F^1_{\mathit{sup}}, F^1_{\mathit{inf}}\rangle, \dots, \langle B^n, F^n_{\mathit{sup}}, F^n_{\mathit{inf}}\rangle\}$ for some $n \leq N$. Then, the symbolic representation of the semantics $[\![P_{\mathit{sup}}]\!]$ and $[\![P_{\mathit{inf}}]\!]$ of P_{sup} and P_{inf} is as follows:

$$[\![P_{\mathit{sup}}]\!] = \textstyle\sum_i^N [\![B^i \Rightarrow F^i_{\mathit{sup}}]\!] \qquad\qquad [\![P_{\mathit{inf}}]\!] = \textstyle\sum_i^N [\![B^i \Rightarrow F^i_{\mathit{inf}}]\!].$$

Step 2: Symbolic Convolution of Interval-Guarded Linear Programs

Now we give a closed-form expression for the smoothed semantics of P_{sup} (the case of $[\![P_{\mathit{inf}}]\!]$ is symmetric). We have:

$$\overline{[\![P_{\mathit{sup}}]\!]}(\mathbf{x}) = \int_{\mathbf{r}\in\mathbb{R}^k} [\![P_{\mathit{sup}}]\!](\mathbf{r})\, \mathcal{N}(\mathbf{x}-\mathbf{r})\, d\mathbf{r} = \sum_i \int_{\mathbf{r}\in\mathbb{R}^k} [\![B^i \Rightarrow F^i_{\mathit{sup}}]\!](\mathbf{r})\, \mathcal{N}(\mathbf{x}-\mathbf{r})\, d\mathbf{r}.$$

To solve this integral, we first observe that by our assumptions, $\mathcal{N}$ is the joint distribution of k univariate *independent* Gaussians with the same standard deviation β. Therefore, we can split $\mathcal{N}$ into a product of univariate Gaussians $\mathcal{N}_1, \ldots, \mathcal{N}_k$, where the Gaussian $\mathcal{N}_j$-th ranges over x_j. Letting r_j be the j-th component of $\mathbf{r}$, we have:

$$\int_{\mathbf{r}\in\mathbb{R}^k} [\![B^i \Rightarrow F^i_{sup}]\!](\mathbf{r})\ \mathcal{N}(\mathbf{x}-\mathbf{r})\ d\mathbf{r} = \\ \int_{-\infty}^{\infty} \cdots \left(\int_{-\infty}^{\infty} [\![B^i \Rightarrow F^i_{sup}]\!](\mathbf{r})\ \mathcal{N}_1(x_1 - r_1)\ dr_1\right) \ldots\ \mathcal{N}_k(x_k - r_k) dr_k.$$

Thus, due to independence of the x_j's, it is possible to reduce the vector integral involved in convolution to a sequence of integrals over one variable. Each of these integrals will be of the form $\int_{-\infty}^{\infty} [\![B^i \Rightarrow F^i_{sup}]\!](\mathbf{r})\ \mathcal{N}_j(x_j - r_j)\ dr_j$. Projecting the interval B^i over the axis for x_j leaves us with an integral like the one we solved analytically in Example 1. This allows us to represent the result of smooth interpretation as a finite sum of closed form expressions.

5 Properties of the Interpreter

Now we prove that the algorithm as defined above actually satisfies the properties claimed in Sec. 3. We first establish that the approximate smooth interpreter presented in this paper is sound. Next we show that under a weak assumption about β, it is robust and β-robust as well.

Theorem 1. *The approximate smooth interpreter of Figure 4 is sound.*

Proof: In order to prove soundness, we need to prove that $\overline{[\![P]\!]}_\beta(\mathbf{x}) \in \mathit{Smooth}_P$ $(\mathbf{x}, \beta)$. This follows directly from the soundness of abstract interpretation thanks to a property of Gaussian convolution: that it is a monotone map on the pointwise partial order $\preceq$ of functions between $\mathbb{R}^k$.

First, we have seen how to use abstract interpretation to produce two programs P_{inf} and P_{sup} such that $[\![P_{inf}]\!] \preceq [\![P]\!] \preceq [\![P_{sup}]\!]$.

Now, we defined $\mathit{Smooth}_P(\mathbf{x}, \beta)$ as the interval $[\overline{[\![P_{inf}]\!]}(\mathbf{x}), \overline{[\![P_{sup}]\!]}(\mathbf{x})]$; therefore, all we have to do to prove soundness is to show that $\overline{[\![P_{inf}]\!]} \preceq \overline{[\![P]\!]}_\beta \preceq \overline{[\![P_{sup}]\!]}$. In other words, we need to show that Gaussian smoothing preserves the ordering among functions.

This follows directly from a property of convolution. Let $F \preceq G$ for functions $F, G : \mathbb{R}^k \to \mathbb{R}^k$, and $H : \mathbb{R}^k \to \mathbb{R}^k$ be any function satisfying $H(\mathbf{x}) > 0$ for all $\mathbf{x}$ (note that the Gaussian function satisfies this property). Also, let F_H and G_H be respectively the convolutions of F and H, and G and H. Then we have $F' \preceq H'$.

Theorem 2. *For every constant $\epsilon > 0$, the approximate smooth interpreter of Figure 4 is robust in the region $\beta > \epsilon$.*

Proof: To prove robustness of $Smooth_P$, it suffices to show that both $\overline{[\![P_{inf}]\!]}$ and $\overline{[\![P_{sup}]\!]}$ satisfy the robustness condition. We focus on proving $\overline{[\![P_{sup}]\!]}$ robust, since the proof for $\overline{[\![P_{inf}]\!]}$ is symmetric. Now, we know that $[\![P_{sup}]\!]$ has the form $[\![P_{sup}]\!](\mathbf{x}) = \sum_{i=0}^{N} B_i(\mathbf{x})\, F_i(\mathbf{x})$, where $B_i(\mathbf{x})$ is an interval in $\mathbb{R}^k$ and $F_i(\mathbf{x})$ is a linear function. Hence we have:

$$[\![\overline{P_{sup}}]\!](\mathbf{x}) = \int_{\mathbf{r}\in\mathbb{R}^k} (\sum_{i=0}^{N} B_i(\mathbf{r}) \cdot F_i(\mathbf{r})) \cdot \mathcal{N}(\mathbf{r}-\mathbf{x},\beta)\ d\mathbf{r}.$$

where $\mathcal{N}$ is the joint density function of k independent 1-D Gaussians. It is easy to see that this function is differentiable arbitrarily many times in x_j. We have:

$$\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial x_j} = \int_{\mathbf{r}\in\mathbb{R}^k} (\sum_{i=0}^{N} B_i(\mathbf{r}) \cdot F_i(\mathbf{r})) \cdot (\frac{\partial\mathcal{N}(\mathbf{r}-\mathbf{x},\beta)}{\partial x_j})\, d\mathbf{r}$$

Now note that $\frac{\partial\mathcal{N}(\mathbf{r}-\mathbf{x},\beta)}{\partial x_j} = \frac{(r_j - x_j)}{\beta^2} \cdot \mathcal{N}(\mathbf{r}-\mathbf{x},\beta)$. Hence,

$$\begin{aligned}\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial x_j} &= \int_{\mathbf{r}\in\mathbb{R}^k} (\frac{1}{\beta^2}\sum_{i=0}^{N} B_i(\mathbf{r}) \cdot F_i(\mathbf{r}) \cdot (r_j - x_j)) \cdot \mathcal{N}(\mathbf{r}-\mathbf{x},\beta)\ d\mathbf{r} \\ &= (\frac{1}{\beta^2}\sum_{i=0}^{N} \int_{\mathbf{r}\in B_i} F_i(\mathbf{r}) \cdot (r_j - x_j) \cdot \mathcal{N}(\mathbf{r}-\mathbf{x},\beta)\ d\mathbf{r} \qquad (2)\end{aligned}$$

Each F_i is a linear function of type $\mathbb{R}^k \to \mathbb{R}^k$, so for each $n < k$, we have $(F_i(\mathbf{r}))(n) = (\sum_{l=0}^{k-1} \alpha_{i,l,n} \cdot r_l) + \gamma_{i,n}$ for constants $\alpha_{i,j,n}$ and $\gamma_{i,n}$. Substituting in Eqn. 2, we find that the n-th coordinate of the vector-valued expression $\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial x_j}$ can be expanded into a linear sum of terms as below:

$$\frac{1}{\beta^2}\int_{r_j=a}^{b} \mathcal{N}(r_j - x_j,\beta)\ dr_j \qquad (3)$$

$$\frac{1}{\beta^2}\int_{r_j=a}^{b} (r_j - x_j) \cdot \mathcal{N}(r_j - x_j,\beta)\ dr_j \qquad (4)$$

$$\frac{1}{\beta^2}\int_{r_j=a}^{b} r_j \cdot \mathcal{N}(r_j - x_j,\beta)\ dr_j \qquad (5)$$

$$\frac{1}{\beta^2}\int_{r_j=a}^{b} r_j \cdot (r_j - x_j) \cdot \mathcal{N}(r_j - x_j,\beta)\ dr_j \qquad (6)$$

In order to show that $\left|\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial x_j}\right|$ is bounded by a linear function, we first observe that for all $\beta > \epsilon$, the multiplier $\frac{1}{\beta^2}$ is bounded by a constant. Now we show that the each integrals in the above is bounded by a function that is linear in x_j, even when $a = -\infty$ or $b = \infty$.

It is easy to see that of these integrals, the first two are bounded in value for any value of x_j; this follows from the fact that $\mathcal{N}$ decays exponentially as its first argument goes to infinity. The last integral will also be bounded as a function of x_j, while the second to last function can grow linearly with x_j. It follows that $\left|\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial x_j}\right|$ can grow only linearly with $\mathbf{x}$ and β. Hence $\overline{[\![P_{sup}]\!]}$ is robust.

Theorem 3. *For every constant $\epsilon > 0$, the approximate smooth interpreter from Figure 4 is β-robust in the region $\beta > \epsilon$.*

Proof: As in the proof of Theorem 2, it suffices to only consider the derivative of P_{sup} (w.r.t. β this time)—the case for P_{inf} is symmetric. From the definition of smoothing, we have:

$$\begin{aligned}\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial\beta} &= \int_{\mathbf{r}\in\mathbb{R}^k}(\sum_{i=0}^{N} B_i(\mathbf{r})\cdot F_i(\mathbf{r}))\cdot\frac{\partial\mathcal{N}(\mathbf{r}-\mathbf{x},\beta)}{\partial\beta}\ d\mathbf{r}\\ &= \int_{\mathbf{r}\in\mathbb{R}^k}(\sum_{i=0}^{N} B_i(\mathbf{r})\cdot F_i(\mathbf{r}))\cdot(\frac{\|\mathbf{r}-\mathbf{x}\|^2}{\beta^3}-\frac{k}{\beta}))\cdot\mathcal{N}(\mathbf{r}-\mathbf{x},\beta)\ d\mathbf{r}\\ &= \sum_{i=0}^{N}\int_{\mathbf{r}\in B_i} F_i(\mathbf{r})\cdot(\frac{\|\mathbf{r}-\mathbf{x}\|^2}{\beta^3}-\frac{k}{\beta})\cdot\mathcal{N}(\mathbf{r}-\mathbf{x},\beta)\ d\mathbf{r}. \qquad (7)\end{aligned}$$

Just like in the proof of robustness, we can decompose the integral into a linear sum of terms of one of the following forms:

$$\frac{1}{\beta^3}\int_{r_j=a}^{b}(r_j-x_j)^2\cdot\mathcal{N}(r_j-x_j,\beta)\ dr_j \qquad (8)$$

$$\frac{1}{\beta^3}\int_{r_j=a}^{b} r_j\cdot(r_j-x_j)^2\cdot\mathcal{N}(r_j-x_j,\beta)\ dr_j \qquad (9)$$

$$\frac{1}{\beta}\int_{r_j=a}^{b} r_j\cdot\mathcal{N}(r_j-x_j,\beta)\ dr_j \qquad (10)$$

$$\frac{1}{\beta}\int_{r_j=a}^{b}\mathcal{N}(r_j-x_j,\beta)\ dr_j \qquad (11)$$

The first and last terms are clearly bounded by constants. The third term is similar to the term we saw in the proof of robustness, and is bounded by a linear function; in fact, when a and b go to infinity, the term corresponds to the mean of a Gaussian centered at x. As for the second term, it is bounded when a and b are bounded, but will grow as $O(x_j)$ when a or b are $-\infty$ or ∞ respectively. Putting the facts together, we note that $\left|\frac{\partial[\![\overline{P_{sup}}]\!](\mathbf{x})}{\partial\beta}\right|$ is bounded by a linear function of $\mathbf{x}$ and β, which completes the proof of β-robustness.

6 Related Work and Conclusion

In a recent paper [4], we introduced smooth interpretation as a program approximation that facilitates more effective use of numerical optimization in reasoning about programs. While the paper showed the effectiveness of the method, it left open a lot of theoretical questions. For example, while we implemented and empirically evaluated a smooth interpreter, could we also formally characterize smooth interpretation? How did program smoothing relate to program abstraction? In the present paper, we have answered the above questions.

Regarding related work, Gaussian smoothing is ubiquitous in signal and image processing [12]. Also, the idea of using smooth, robust approximations to enable optimization of non-smooth functions has been previously studied in the optimization community [11]. However, these approaches are technically very different from ours, and we were the first to propose and find an application for Gaussian smoothing of programs written in a general-purpose programming language. As for work in the software engineering community, aside from a cursory note by DeMillo and Lipton [9], there does not seem to be any prior proposal here for the use of "smooth" models of programs (although some recent work in program verification studies continuity properties [2,3] of programs).

The abstract interpretation used in our smoothing framework is closely related to a large body of prior work on static analysis, in particular analysis using intervals [8], interval equalities [6], and interval polyhedra [5]. However, so far as we know, there is no existing abstract domain that can conditionally bound the denotational semantics of a program from above and below.

References

1. Burnim, J., Juvekar, S., Sen, K.: Wise: Automated test generation for worst-case complexity. In: ICSE, pp. 463–473 (2009)
2. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: POPL (2010)
3. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving programs robust (2011)
4. Chaudhuri, S., Solar-Lezama, A.: Smooth interpretation. In: PLDI (2010)
5. Chen, L., Miné, A., Wang, J., Cousot, P.: Interval polyhedra: An abstract domain to infer interval linear relationships. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 309–325. Springer, Heidelberg (2009)
6. Chen, L., Miné, A., Wang, J., Cousot, P.: An abstract domain to discover interval linear equalities. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 112–128. Springer, Heidelberg (2010)
7. Chen, X.: Convergence of the BFGS method for LC convex constrained optimization. SIAM J. Control and Optim. 14, 2063 (1996)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
9. DeMillo, R., Lipton, R.: Defining software by continuous, smooth functions. IEEE Transactions on Software Engineering 17(4), 383–384 (1991)
10. Hager, W.W., Zhang, H.: A survey of nonlinear conjugate gradient methods. Pacific Journal of Optimization 2(1), 35–58 (2006)
11. Nesterov, Y.: Smooth minimization of non-smooth functions. Mathematical Programming 103(1), 127–152 (2005)
12. Russ, J.: The image processing handbook. CRC Press, Boca Raton (2007)

A Specialization Calculus for Pruning Disjunctive Predicates to Support Verification

Wei-Ngan Chin[1], Cristian Gherghina[1], Răzvan Voicu[1],
Quang Loc Le[1], Florin Craciun[1], and Shengchao Qin[2]

[1] Department of Computer Science, National University of Singapore
[2] School of Computing, Teesside University

Abstract. Separation logic-based abstraction mechanisms, enhanced with user-defined inductive predicates, represent a powerful, expressive means of specifying heap-based data structures with strong invariant properties. However, expressive power comes at a cost: the manipulation of such logics typically requires the unfolding of disjunctive predicates which may lead to expensive proof search. We address this problem by proposing a *predicate specialization* technique that allows efficient symbolic pruning of infeasible disjuncts inside each predicate instance. Our technique is presented as a calculus whose derivations preserve the satisfiability of formulas, while reducing the subsequent cost of their manipulation. Initial experimental results have confirmed significant speed gains from the deployment of predicate specialization. While specialization is a familiar technique for code optimization, its use in program verification is new.

1 Introduction

Abstraction mechanisms are important for modelling and analyzing programs. Recent developments allow richer classes of properties to be expressed via user-defined predicates for capturing commonly occurring patterns of program properties. Separation logic-based abstraction mechanisms represent one such development. As an example, the following predicate captures an abstraction of a sorted doubly-linked list.

$$\begin{array}{l}
\mathbf{data}\ \mathtt{node}\ \{\ \mathtt{int\ val};\ \mathtt{node\ prev};\ \mathtt{node\ next};\ \} \\
\mathtt{dll(root,p,n,S)} \equiv \mathtt{root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{S{=}\{\}} \\
\ \vee\ \exists \mathtt{v,q,S_1} \cdot \mathtt{root{\mapsto}node(v,p,q)} * \mathtt{dll(q,root,n{-}1,S_1)} \\
\quad \wedge \mathtt{S} = \mathtt{S_1{\cup}\{v\}} {\wedge} \forall \mathtt{a{\in}S_1} \cdot \mathtt{v{\leq}a} \qquad \mathbf{inv}\ \mathtt{n{\geq}0};
\end{array}$$

In this definition `root` denotes a pointer into the list, `n` the length of the list, `S` represents its set of values, whereas `p` denotes a backward pointer from the first node of the doubly-linked list. The invariant $\mathtt{n{\geq}0}$ must hold for all instances of this predicate.

We clarify the following points. Firstly, spatial conjunction, denoted by the symbol $*$, provides a concise way of describing disjoint heap spaces. Secondly, this abstraction mechanism is inherently infinite, due to recursion in predicate definition. Thirdly, a predicate definition is capable of capturing multiple features of the data structure it models, such as its size and set of values. While this richer set of features can enhance the precision of a program analysis, it inevitably leads to larger disjunctive formulas.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 293–309, 2011.

This paper is concerned with a novel way of handling disjunctive formulas, in conjunction with abstraction via user-defined predicates. While disjunctive forms are natural and expressive, they are major sources of redundancy and inefficiency. The goal of this paper is to ensure that disjunctive predicates can be efficiently supported in a program analysis setting, in general, and program verification setting, in particular.

To achieve this, we propose a *specialization calculus* for disjunctive predicates that supports symbolic pruning of infeasible states within each predicate instance. This allows for the implementation of both *incremental pruning* and *memoization* techniques. As a methodology, predicate specialization is not a new concept, since general specialization techniques have been extensively used in the optimization of logic programs [18,17,11]. The novelty of our approach stems from applying specialization to a new domain, namely program verification, with its focus on pruning infeasible disjuncts, rather than a traditional focus on propagating static information into callee sites. This new use of specialization yields a fresh approach towards optimising program verification. This approach has not been previously explored, since pervasive use of user-defined predicates in analysis and verification has only become popular recently (e.g. [14]). Our key contributions are:

- We propose a *new specialization calculus* that leads to more effective program verification. Our calculus specializes proof obligations produced in the program verification process, and can be used as a preprocessing step before the obligations are fed into third party theorem provers or decision procedures.
- We adapt *memoization* and *incremental pruning* techniques to obtain an optimized version of the specialization calculus.
- We present a prototype implementation of our specialization calculus, integrated into an existing program verification system. The use of our specializer yields significant reductions in verification times, especially for larger problems.

Section 2 illustrates the technique of specializing disjunctive predicates. Section 3 introduces the necessary terminology. Section 4 presents our calculus for specializing disjunctive predicates and outlines its formal properties. Section 5 presents inference mechanisms for predicate definitions to support our specialization calculus. Section 6 presents experimental results which show multi-fold improvement to verification times for larger problems. Section 7 discusses related work, prior to a short conclusion.

2 Motivating Example

Program states that are built from predicate abstractions are more concise, but may require properties that are hidden inside predicates. As an example, consider :

$$\texttt{dll}(\texttt{x},\texttt{p}_1,\texttt{n},\texttt{S}_1) * \texttt{dll}(\texttt{y},\texttt{p}_2,\texttt{n},\texttt{S}_2) \wedge \texttt{x}{\neq}\texttt{null}$$

This formula expresses the property that the two doubly-linked lists pointed to by $\texttt{x}$ and $\texttt{y}$ have the same length. Ideally, we should augment our formula with the property: $\texttt{y}{\neq}\texttt{null}$, $\texttt{n}{>}0$, $\texttt{S}_1{\neq}\{\}$ and $\texttt{S}_2{\neq}\{\}$, currently hidden inside the two predicate instances but may be needed by the program verification tasks at hand.

A naive approach would be to unfold the two predicate instances, but this would blow up the number of disjuncts to four, as shown:

$$\begin{array}{l}
\mathtt{x{=}null} \wedge \mathtt{y{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{S_1{=}\{\}} \wedge \mathtt{S_2{=}\{\}} \wedge \mathtt{x{\neq}null} \\
\vee\ \mathtt{y{\mapsto}node(v_2,p_2,q_2)} * \mathtt{dll(q_2,y,n{-}1,S_4)} \wedge \mathtt{x{=}null} \\
\quad \wedge\, \mathtt{S_1{=}\{\}} \wedge \mathtt{n{=}0} \wedge \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null} \\
\vee\ \mathtt{x{\mapsto}node(v_1,p_1,q_1)} * \mathtt{dll(q_1,x,n{-}1,S_3)} \wedge \mathtt{y{=}null} \wedge \mathtt{n{=}0} \\
\quad \wedge\, \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3} \wedge \mathtt{S_2{=}\{\}} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null} \\
\vee\ \mathtt{x{\mapsto}node(v_1,p_1,q_1)} * \mathtt{y{\mapsto}node(v_2,p_2,q_2)} * \mathtt{dll(q_1,x,n{-}1,S_3)} \\
\quad * \mathtt{dll(q_2,y,n{-}1,S_4)} \wedge \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3} \wedge \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null}
\end{array}$$

As contradictions occur in the first three disjuncts, we can simplify our formula to:

$$\begin{array}{l}
\mathtt{x{\mapsto}node(v_1,p_1,q_1)} * \mathtt{y{\mapsto}node(v_2,p_2,q_2)}\ *\ \mathtt{dll(q_1,x,n{-}1,S_3)} \\
\quad *\ \mathtt{dll(q_2,y,n{-}1,S_4)} \wedge \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3}\ \wedge\ \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null}
\end{array}$$

After removing infeasible disjuncts, the propagated properties are exposed in the above more *specialized* formula. However, this naive approach has the shortcoming that unfolding leads to an increase in the number of disjuncts handled, and its associated costs.

A better approach would be to avoid predicate unfolding, but instead apply predicate specialization to prune infeasible disjuncts and propagate hidden properties. Given a predicate $\mathtt{pred}(\cdots)$ that is defined by $\mathtt{k}$ disjuncts, we shall denote each of its specialized instances by $\mathtt{pred}(\cdots)@\mathtt{L}$, where $\mathtt{L}$ denotes a subset of the disjuncts, namely $\mathtt{L} \subseteq \{\mathtt{1}...k\}$, that have not been pruned. Initially, we can convert each predicate instance $\mathtt{pred}(\cdots)$ to its most general form $\mathtt{pred}(\cdots)@\{\mathtt{1}...k\}$, while adding the basic invariant of the predicate to its context. As an illustration, we may view the definition of $\mathtt{dll}$ as a predicate with two disjuncts, labelled informally by $\mathtt{1}$: and $\mathtt{2}$: prior to each of its disjuncts, as follows:

$$\begin{array}{l}
\mathtt{dll(root,p,n,S)} \equiv \mathtt{1{:}(root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{S{=}\{\})} \\
\ \vee\ \mathtt{2{:}(root{\mapsto}node(v,p,q)} * \mathtt{dll(q,root,n{-}1,S_1)}\ \wedge\ \mathtt{S} = \mathtt{S_1{\cup}\{v\}} \wedge \forall \mathtt{a{\in}S_1} \cdot \mathtt{v{\leq}a)}
\end{array}$$

We may convert each $\mathtt{dll}$ predicate by adding its invariant $\mathtt{n{\geq}0}$, as follows:

$$\mathtt{dll(x,p,n,S)} \implies \mathtt{dll(x,p,n,S)@\{1,2\}} \wedge \mathtt{n{\geq}0}$$

With our running example, this would lead to the following initial formula after the same invariant $\mathtt{n{\geq}0}$ (from the two predicate instances) is added.

$$\mathtt{dll(x,p_1,n,S_1)@\{1,2\}} * \mathtt{dll(y,p_2,n,S_2)@\{1,2\}} \wedge \mathtt{x{\neq}null} \wedge \mathtt{n{\geq}0}$$

This predicate may be further specialized with the help of its context by pruning away disjuncts that are found to be infeasible. Each such pruning would allow more states to be propagated by the specialized predicate. By using the context, $\mathtt{x{\neq}null}$, we can specialize the first predicate instance to $\mathtt{dll(x,p_1,n,S_1)@\{2\}}$ since this context contradicts the first disjunct of the $\mathtt{dll}$ predicate. With this specialization, we may strengthen the context with a propagated state, namely $\mathtt{n{>}0} \wedge \mathtt{S_1{\neq}\{\}}$, that is implied by its specialized instance, as follows:

$$\mathtt{dll(x,p_1,n,S_1)@\{2\}} * \mathtt{dll(y,p_2,n,S_2)@\{1,2\}}\ \wedge\ \mathtt{x{\neq}null} \wedge \mathtt{n{>}0} \wedge \mathtt{S_1{\neq}\{\}}$$

Note that $\mathtt{n{\geq}0}$ is removed when a stronger constraint $\mathtt{n{>}0}$ is added. The new constraint $\mathtt{n{>}0}$ now triggers a pruning of the second predicate instance, since its first disjunct can be shown to be infeasible. This leads to a specialization of the second predicate, with more propagation of atomic formulas, as follows:

$$
\begin{array}{lll}
\mathit{pred} & ::= p(v^*) \equiv \Phi \;\; [\mathit{inv}\;\; \pi] & \\
\Phi & ::= \bigvee\; (\exists w^* {\cdot} \sigma)^* & \sigma ::= \kappa {\wedge} \pi \\
\kappa & ::= \texttt{emp} \mid v {\mapsto} c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 & \\
\pi & ::= \alpha \mid \pi_1 {\wedge} \pi_2 & \\
\alpha & ::= \beta \mid \neg\beta & \\
\beta & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid a{\leq}0 \mid a{=}0 & \\
a & ::= k \mid k{\times}v \mid a_1 + a_2 \mid \mathit{max}(a_1,a_2) \mid \mathit{min}(a_1,a_2) & \\
\mathit{where} & p \textit{ is a predicate name};\; v, w \textit{ are variable names}; & \\
 & c \textit{ is a data type name};\; k \textit{ is an integer constant}; & \\
 & \kappa \textit{ represents heap formulas};\; \pi \textit{ represents pure formulas}; & \\
 & \beta \textit{ represents atomic interpreted predicates} &
\end{array}
$$

Fig. 1. The Unannotated Specification Language

$$
\begin{array}{l}
\texttt{dll}(\texttt{x},\texttt{p}_1,\texttt{n},\texttt{S}_1)@\{2\} * \texttt{dll}(\texttt{y},\texttt{p}_2,\texttt{n},\texttt{S}_2)@\{2\} \\
\quad \wedge\; \texttt{x}{\neq}\texttt{null} \wedge \texttt{n}{>}0 \wedge \texttt{S}_1{\neq}\{\} \wedge \texttt{y}{\neq}\texttt{null} \wedge \texttt{S}_2{\neq}\{\}
\end{array}
$$

In a nutshell, the goal of our approach is to apply aggressive specialization to our predicate instances, without the need to resort to predicate unfolding, in the hope that infeasible disjuncts are pruned, where possible. In the process, our specialization technique is expected to propagate states that are consequences of each of the specialized predicate instances. We expect this proposal to support more efficient manipulation of program states, whilst keeping the original abstractions intact where possible.

3 Formal Preliminaries

Our underlying computation model is a state machine with a countable set of variables and a heap, which is a partial mapping from addresses to values.

Fig. 1 defines the syntax of our (unannotated) specification language. We denote sequences of variables $v_1, \ldots, v_n$ by the notation v^*, and by β atomic interpreted predicates such as equality and disequality of program variables and arithmetic expressions. Conjunctions of (possibly negated) atomic predicates form *pure* formulas, which we denote by the symbol π. Heap formulas, denoted by κ, model the configuration of the heap. They rely on two important components: data constructors $c(v^*)$, which model simple data records (e.g. the node of a tree), and inductively defined predicates, which are generated by the non-terminal *pred* in Fig. 1.

Definition 1 (Heap Formula and Predicate Definition). *A* heap formula κ *is either the symbol* emp*, denoting the empty heap, or a formula of the form* $v{\mapsto}c(v^*)$*, denoting a singleton heap, or a predicate* $p(v^*)$*, or finally, a formula of the form* $\kappa_1 * \kappa_2$*, where* κ_1 *and* κ_2 *are heap formulas, and* $*$ *is the separating conjunction connective. Predicates are defined inductively as the equivalence between a predicate symbol* $p(v^*)$*, and disjunctions of formulas of the form* $\exists w^* \cdot (\kappa \wedge \pi)$*, where variables* v^* *may appear free. Predicate definitions may be augmented with invariants specified by the* inv *keyword.*

$spred ::= p(v^*) \equiv \hat{\Phi}; \mathcal{I}; \mathcal{R}$
$\hat{\Phi} ::= \bigvee\ (\exists w^* \cdot \hat{\sigma} \mid C)^* \qquad \hat{\sigma} ::= \hat{\kappa} \wedge \pi$
$\hat{\kappa} ::= \texttt{emp} \mid v {\mapsto} c(v^*) \mid p(v^*)@L\#R \mid \hat{\kappa}_1 * \hat{\kappa}_2$
where p, v, w, c, π *denote the same as in Fig. 1;*
$\mathcal{I}$ *is a family of invariants*;
$\mathcal{R}$ *is a set of pruning conditions*.
C is a pure formula denoting a context computed in the specialization process.

Fig. 2. The Annotated Specification Language

Unannotated formulas become annotated in the specialization process. Fig. 2 defines the syntax of the *annotated* specification language. Annotated predicate definitions are generated by the nonterminal *spred*.

Definition 2 (Annotated Predicates and Formulas). *Given a predicate definition* $p(v^*) \equiv \bigvee\ (\exists w^* \cdot \kappa \wedge \pi)^*$*, the corresponding* annotated predicate definition *has the form* $p(v^*) \equiv \bigvee\ (\exists w^* \cdot \hat{\kappa} \wedge \pi \mid C)^*; \mathcal{I}; \mathcal{R}$*, where* $\mathcal{I}$ *is a family of invariants, and* $\mathcal{R}$ *is a set of pruning conditions. Each disjunct* $\exists w^* \cdot \hat{\kappa} \wedge \pi \mid C$ *now contains the annotated counterpart* $\hat{\kappa}$ *of* κ*, and is augmented with a context C, which is a pure formula for which* $C \rightarrow \pi$ *always holds. Intuitively, C captures also the consequences of the specialized states of* $\hat{\kappa}$*. An* annotated formula *is a formula where all the predicate instances are annotated. An* annotated predicate *instance is of the form* $p(v^*)@L\#R$*, where* $L \subseteq \{1,..,n\}$ *is a set of labels denoting the unpruned disjuncts, and where* $R \subseteq \mathcal{R}$ *is a set of remaining pruning conditions. The* set of invariants $\mathcal{I}$ *is of the form* $\{(L {\rightarrow} \pi_L) \mid \emptyset {\subset} L {\subseteq} \{1,..,n\}\}$*. For each set of labels L,* π_L *represents the invariant for the specialized predicate instance* $p(v^*)@L$*. For a given annotated predicate instance* $p(v^*)@L\#R$*, it is possible for* $L = \emptyset$*. When this occurs, it denotes that none of the predicate's disjuncts are satisfiable. Moreover, we have* $\pi_{\emptyset} {=} false$ *which will contribute towards a false state (or contradiction) for its given context.*

Definition 3 (Pruning Condition). *A* pruning condition *is a pair between an atomic predicate instance* α *and a set of labels L, written* $\alpha {\leftarrow} L$*. Its intuitive meaning is that the disjuncts in L should be kept if* α *is satisfiable in the current context. The symbol* $\mathcal{R}$ *denotes a finite set of such pruning conditions.*

Given a predicate definition $p(v^*) \equiv \bigvee_{i=1}^{n} (\exists w^* \cdot \hat{\sigma_i} \mid C_i);\ \mathcal{I};\ \mathcal{R}$, we call $\texttt{D}_i =_{df} (\exists w^* \cdot \hat{\sigma_i} \mid C_i)$ the i^{th} disjunct of p ; i will be called the *label* of its disjunct. We shall use $\texttt{D}_i$ freely as the i^{th} disjunct of the predicate at hand whenever there is no risk of confusion. We employ a notion of *closure* for a given conjunctive formula. Consider a formula $\pi(w^*) = \exists v^* \cdot \alpha_1 \wedge \cdots \wedge \alpha_m$, where α_i are atomic predicates, and variables w^* appear free. We denote by $\texttt{S} = closure(\pi(w^*))$ a set of atomic predicates (over the free variables w^*) such that each element $\alpha \in \texttt{S}$ is entailed by $\pi(w^*)$. Some of the variables w^* may appear free in α but *not* v^*. To ensure this closure set be finite, we also impose a requirement that weaker atomic constraints are never present in the same set, as follows:

$\forall \alpha_i \in \mathtt{S} \cdot \neg(\exists \alpha_j \in \mathtt{S} \cdot i{\neq}j \wedge \alpha_i \implies \alpha_j)$. Ideally, *closure*$(\pi(w^*))$ contains *all* stronger atomic formulas entailed by $\pi(w^*)$, though depending on the abstract domain used, this set may not be computable. A larger closure set leads to more aggressive pruning.

Our specialization calculus (Sec 4) is based on the annotated specification language. We have an initialization and inference process (Sec 5) to automatically generate all annotations (including $\mathcal{I}, \mathcal{R}$) that are required by specialization. For simplicity of presentation, we only include normalized linear arithmetic constraints in our language. Our system currently supports both arbitrary linear arithmetic constraints, as well as set constraints. This is made possible by integrating the Omega [19] and MONA solvers [9] into the system. In principle, the system may support arbitrary constraint domains, provided that a suitable solver is available for the domain of interest. Such a solver should be capable of handling conjunctions efficiently, as well as computing approximations of constraints that convert disjunctions into conjunctions (e.g. hulling).

4 A Specialization Calculus

Our specialization framework detects infeasible disjuncts in predicate definitions without explicitly unfolding them, and computes a corresponding strengthening of the pure part while preserving satisfiability. We present this as a calculus consisting of *specialization rules* that can be applied exhaustively to convert a non-specialized annotated formula[1] into a fully specialized one, with stronger pure parts, that can be subsequently extracted and passed on to a theorem prover for satisfiability/entailment checking. Apart from being syntactically correct, annotated formulas must satisfy the following well-formedness conditions.

Definition 4 (Well-formedness). *For each annotated predicate* $\mathtt{p}(\mathtt{v}^*)@L\#R$ *in the formula at hand, assuming the definition* $p(v^*){\equiv}\bigvee_{i=1}^{n} \mathtt{D_i};\mathcal{I};\mathcal{R}$*, we have that (a)* $L \subseteq \{1,\ldots,n\}$ *; (b)* $R \subseteq \mathcal{R}$ *; and (c) forall* $\alpha{\leftarrow}L_0 \in R$ *we have* $L \cap L_0 \neq \emptyset$*.*

Definition 5 (Specialization Step). *A* specialization step *has the form* $\hat{\Phi}_1 \mid C_1 \xrightarrow{sf} \hat{\Phi}_2 \mid C_2$*, and denotes the relation allows the annotated formula* $\hat{\Phi}_1$ *with context* C_1 *to be transformed into a more specialized formula* $\hat{\Phi}_2$ *with context* C_2*.*

Our calculus produces specialization steps, which are applied in sequence, exhaustively, to produce *fully specialized* formulas (a formal definition of such formulas will be given below). Relation $\xrightarrow{sf}$ depends on relation $\xrightarrow{sp}$, which produces predicate specialization steps defined by the following:

Definition 6 (Predicate Specialization Step). *A* predicate specialization *step has form*

$$(1) \qquad p(v^*)@L_1\#R_1 \mid C_1 \xrightarrow{sp} p(v^*)@L_2\#R_2 \mid C_2.$$

and signifies that annotated predicate $p(v^*)@L_1\#R_1 \mid C_1$ *can be specialized into* $p(v^*)@L_2\#R_2 \mid C_2$*, where* $L_2 \subseteq L_1$*,* $R_2 \subset R_1$*, and* C_2 *is stronger than* C_1*.*

Here, the sets L_1 and L_2 denote sets of disjuncts of $p(v^*)$ that have not been detected to be infeasible. Each specialization step aims at detecting new infeasible disjuncts and removing them during the transformation. Thus L_2 is expected to be a subset of L_1.

[1] The conversion of non-annotated formulas into annotated ones shall be presented in Sec. 5.

$$[\underline{\textsc{sp}-[\textsc{filter}]}]$$

$$\frac{R_f = \{(\alpha\leftarrow L_0) \mid (\alpha\leftarrow L_0)\in R,\ (L\cap L_0=\emptyset)\vee(C \Longrightarrow \alpha)\}}{C, L \vdash R \xrightarrow{\text{filter}} (R - R_f)}$$

$$[\underline{\textsc{sp}-[\textsc{prune}]}]$$

$$\frac{\begin{array}{c} C\wedge\alpha \Longrightarrow \texttt{false} \quad (\alpha\leftarrow L_0)\in R \quad L\cap L_0 \neq \emptyset \quad L_2 = L - L_0 \\ C_1 = \mathcal{I}nv(p(v^*), L_2) \qquad C\wedge C_1, L_2 \vdash R \xrightarrow{\text{filter}} R_1 \end{array}}{p(v^*)@L\#R \mid C \xrightarrow{\text{sp}} p(v^*)@L_2\#R_1 \mid C\wedge C_1}$$

$$[\underline{\textsc{sp}-[\textsc{finish}]}]$$

$$\frac{C, L \vdash R \xrightarrow{\text{filter}} \emptyset \qquad R \neq \emptyset}{p(v^*)@L\#R \mid C \xrightarrow{\text{sp}} p(v^*)@L\#\emptyset \mid C}$$

Fig. 3. Single-step Predicate Specialization

The sets of pruning conditions R_1 and R_2 may be redundant, but are instrumental in making specialization efficient. They record incremental changes to the state of the specializer, and represent information that would be expensive to re-compute at every step. Essentially, a pruning condition $\alpha\leftarrow L_0$ states that whenever $\neg\alpha$ is entailed by the current context, the disjuncts whose labels are in L_0 can be pruned. The initial set of pruning conditions is derived when converting formulas into annotated formulas, and is formally discussed in Section 5.

In a nutshell, each specialization step of the form (1) detects (if possible) a pruning condition $\alpha\leftarrow L_0 \in R$ such that if $\neg\alpha$ is entailed by the current context, then the disjuncts whose labels occur in L_0 are infeasible and can be pruned. Given the notations in (1), this is achieved by setting $L_2 = L_1 - L_0$. Subsequently, the current set of pruning conditions is reduced to contain only elements of the form $\alpha'\leftarrow L_0'$ such that $L_0' \cap L_2 \neq \emptyset$. Thus, the well-formedness of the annotated formula is preserved

A key aspect of specialization is that context strengthening helps reveal and prune *mutually infeasible* disjuncts in groups of predicates, which leads to a more aggressive optimization as compared to the case where predicates are specialized in isolation.

Definition 7 (Fully Specialized Formula; Complete Specialization). *An annotated formula is* fully specialized *w.r.t a context when all its annotated predicates have empty pruning condition sets. If the initial pruning condition sets are computed using a notion of* strongest closure*, then for each predicate in the fully specialized formula, all the remaining labels in the predicate's label set denote feasible disjuncts with respect to the current context, and in that sense, the specialization is* complete.

This procedure is formalized in the calculus rules given in Figures 3 and 4. Figure 3 defines the predicate specialization relation $\xrightarrow{\text{sp}}$. This relation has two main components: the one represented by the rule $[\underline{\textsc{sp}-[\textsc{filter}]}]$, which restores the well-formedness of an annotated predicate, and the one represented by the rule $[\underline{\textsc{sp}-[\textsc{prune}]}]$, which detects infeasible disjuncts and removes the corresponding labels from the annotation. A third rule, $[\underline{\textsc{sp}-[\textsc{finish}]}]$ produces the fully specialized predicate.

$$\left[\underline{\text{SF}-[\text{PRUNE}]}\right]$$
$$\frac{p(v^*)@L\#R \mid C \xrightarrow{\text{sp}} p(v^*)@L_2\#R_2 \mid C_2}{p(v^*)@L\#R * \hat{\kappa} \mid C \xrightarrow{\text{sf}} p(v^*)@L_2\#R_2 * \hat{\kappa} \mid C_2}$$

$$\left[\underline{\text{SF}-[\text{CASE}-\text{SPLIT}]}\right]$$
$$\frac{\vdash C \Longrightarrow \alpha_1 \vee \alpha_2 \quad \vdash \alpha_1 \wedge \alpha_2 \Longrightarrow \texttt{false} \qquad \forall i \in \{1,2\} \cdot \hat{\kappa} \mid C \wedge \alpha_i \xrightarrow{\text{sf}} \hat{\kappa}_i \mid C_i}{\hat{\kappa} \mid C \xrightarrow{\text{sf}} (\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2)}$$

$$\left[\underline{\text{SF}-[\text{OR}]}\right]$$
$$\frac{\hat{\kappa}_1 \mid C_1 \xrightarrow{\text{sf}} \hat{\kappa}_3 \mid C_3}{(\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2) \xrightarrow{\text{sf}} (\hat{\kappa}_3 \mid C_3) \vee (\hat{\kappa}_2 \mid C_2)}$$

Fig. 4. Single-step Formula Specialization

The predicate specialization relation can be weaved into the formula specialization relation given in Fig. 4. The first rule, $\left[\underline{\text{SF}-[\text{PRUNE}]}\right]$, defines the part of the $\xrightarrow{\text{sf}}$ relation which picks a predicate in a formula and transforms it using the $\xrightarrow{\text{sp}}$ relation, leaving the rest of the predicates unchanged. However, the transformation of the predicate's context is incorporated into the transformation of the formula's context. This rule realizes the potential for cross-specialization of predicates, eliminating disjuncts of different predicates that are mutually unsatisfiable.

The rule $\left[\underline{\text{SF}-[\text{CASE}-\text{SPLIT}]}\right]$ allows further specialization via case analysis. It defines the part of the $\xrightarrow{\text{sf}}$ relation that produces two instances of the same formula, joined in a disjunction, each of the new formulas having a stronger context. Each stronger context is produced by conjunction with an atom α_i, $i \in \{1,2\}$, with the requirement that the two atoms be disjoint and their disjunction cover the original context C. This rule is instrumental in guaranteeing that all predicates reach a fully specialized status. Indeed, whenever an annotated predicate has a pruning condition $\alpha \leftarrow L_0$, such that α is not entailed by the context C, yet $\alpha \wedge C$ is satisfiable, the only way to further specialize the predicate is by case analysis with the atoms α and $\neg\alpha$. Finally, the rule $\left[\underline{\text{SF}-[\text{OR}]}\right]$ handles formulas with multiple disjunctions.

In the remainder of this section, we formalize the notion that our calculus produces terminating derivations, and is sound and complete.

Property 1. *Relations $\xrightarrow{sp}$ and $\xrightarrow{sf}$ preserve well-formedness. Thus, given two annotated predicate instances* $\mathtt{p(v^*)}@L_1\#R_1$ *and* $\mathtt{p(v^*)}@L_2\#R_2$*, if*

$$\mathtt{p(v^*)}@L_1\#R_1 \mid C_1 \xrightarrow{sp} \mathtt{p(v^*)}@L_2\#R_2 \mid C_2$$

can be derived from the calculus, and $\mathtt{p(v^*)}@L_1\#R_1$ *is well-formed, then* $\mathtt{p(v^*)}@L_2\#R_2$ *is well-formed as well. Moreover, for all annotated formulas $\hat{\Phi}_1$ and $\hat{\Phi}_2$, if*

$$\hat{\Phi}_1 \mid C_1 \xrightarrow{sf} \hat{\Phi}_2 \mid C_2$$

can be derived from the calculus, and $\hat{\Phi}_1$ is well-formed, then $\hat{\Phi}_2$ is well formed as well.

A *predicate specialization sequence* is a sequence of annotated predicates such that each pair of consecutive predicates is in the relation $-sp\rightarrow$. A *formula specialization sequence* is a sequence of annotated formulas such that each pair of consecutive formulas is in the $-sf\rightarrow$ relation.

Definition 8 (Canonical Specialization Sequence). *A canonical specialization sequence is a formula specialization sequence where (a) the first element is well-formed; (b) specialization rules are applied exhaustively ; (c) the* $[\underline{\mathbf{SF}-[\mathbf{CASE}-\mathbf{SPLIT}]}]$ *relation is only applied as a last resort (i.e. when no other relation is applicable); and (d) the case analysis atoms for the* $[\underline{\mathbf{SF}-[\mathbf{CASE}-\mathbf{SPLIT}]}]$ *relation must be of the form* α, $\neg\alpha$, *where* $\alpha{\leftarrow}L_0$ *is a pruning condition occurring in an annotated predicate* $\mathtt{p}(\mathtt{v}^*)@L\#R$ *of the formula, such that* $L \cap L_0 \neq \emptyset$*.*

Property 2 (Termination). *All canonical specialization sequences are finite and produce either fully specialized formulas, or formulas whose context is unsatisfiable.*

Property 3 (Soundness). *The* $-sp\rightarrow$ *and* $-sf\rightarrow$ *relations preserve satisfiability. Thus, if* $\mathtt{p}(\mathtt{v}_{0..n})@L_1\#R_1 \mid C_1 \ -sp\rightarrow \ \mathtt{p}(\mathtt{v}_{0..n})@L_2\#R_2 \mid C_2$ *can be derived from the calculus, then for all heaps* h *and stacks* s, $s,h \models \mathtt{p}(\mathtt{v}^*)@L_1\#R_1 \mid C_1$ *iff* $s,h \models \mathtt{p}(\mathtt{v}^*)@L_2\#R_2 \mid C_2$*. Moreover, if* $\hat{\Phi}_1 \mid C_1 -sf\rightarrow \hat{\Phi}_2 \mid C_2$ *can be derived from the calculus, then* $s,h \models \hat{\Phi}_1|C_1$ *iff* $s,h \models \hat{\Phi}_2|C_2$*.*

We note here that the set *R* does not play a role in the way an annotated predicate is interpreted. Mishandling *R* (as long as no elements are added) may result in lack of termination or incompleteness, but does not affect soundness.

Finally, we address the issue of completeness. This property, however, is dependent on how "complete" the conversion of a predicate into its annotated form is. Thus, we shall first give an ideal characterization of such a conversion, after which we shall endeavour to prove the completeness property. Realistic implementations of this conversion shall be discussed in Section 5.

Definition 9 (Strongest Closure). *The strongest closure of unannotated formula* Φ*, denoted* $sclosure(\Phi)$*, is the largest set of atoms* α *with the following properties: (a) for all stacks* s*,* $s \models \alpha$ *whenever there exists* h *such that* $s,h \models \Phi$*, and (b) there exists no atom* α' *strictly stronger than* α *– that is, it is not the case that for all* s*,* $s \models \alpha$ *whenever* $s \models \alpha'$*. For practical and termination reasons, we shall assume only closures which return finite sets in our formulation.*

In our conversion of an unannotated predicate definition for $p(v^*)$ into the annotated definition $p(v^*){\equiv}\bigvee_{i=1}^{n}\mathtt{D}_i;\mathcal{I};\mathcal{R}$, we compute the following sets: $\mathtt{G}_i = sclosure(\mathtt{D}_i{\wedge}\pi)$, for $i = 1,..,n$, $\mathtt{H}_L = \{\alpha \,|\, \text{forall } i \in L, \text{exists } \alpha' \in \mathtt{G}_i \text{ s.t. forall } s, s \models \alpha' \text{ whenever } s \models \alpha\}$ and $\mathcal{I} = \{L{\rightarrow}\pi \,|\, L{\subseteq}\{1...n\},\ \pi = \bigwedge_{\alpha\in\mathtt{H}_L}\alpha\}$, and $\mathcal{R} = \{\alpha{\leftarrow}L \,|\, L \text{ is the largest set s.t. } \alpha{\in}\bigcap_{i\in L}\mathtt{G}_i\}$. Moreover, we introduce the notation $\mathcal{I}nv(p(v^*),L) = \pi_L$, where $(L{\rightarrow}\pi_L){\in}\mathcal{I}$. This notation is necessary in applying the rule $[\underline{\mathbf{SP}-[\mathbf{PRUNE}]}]$.

In practice, either the assumption holds, or the closure procedure computes a close enough approximation to the strongest closure so that very few, if any, infeasible disjuncts are left in the specialized formula.

Property 4 (Completeness). *Let* $\mathtt{p}(\mathtt{v}^*)@L\#\emptyset * \hat{\sigma}|C$ *be a fully specialized formula that resulted from a specialization process that started with an annotated formula. Denote by* π_i*,* $1 \leq i \leq n$ *the pure parts of the disjuncts in the definition of* $\mathtt{p}(\mathtt{v}^*)$*, and assume that C is satisfiable. Then, for all* $i \in L$*,* $\pi_i \wedge C$ *is satisfiable.*

Proofs of the above properties, as well as a more detailed discussion of our calculus rules, can be found in [3].

5 Inferring Specializable Predicates

We present inference techniques that must be applied to each predicate definition so that they can support the specialization process. We refer to this process as *inference for specializable predicates*. A predicate is said to be *specializable* if it has multiple disjuncts and it has a non-empty set of pruning conditions. These two conditions would allow a predicate instance to be specializable from one form to another specialized form. Our predicates are processed in a bottoms-up order with the following key steps:

- Transform each predicate definition to its specialized form.
- Compute an *invariant* (in conjunctive form) for each predicate.
- Compute a *family of invariants* to support all specialized instances of the predicate.
- Compute a *set of pruning conditions* for the predicate.
- Specialize recursive invocations of the predicate, if possible.

As a running example for this inference process, let us consider the following predicate which could be used to denote a list segment of singly-linked nodes:

$\mathbf{data}\ \mathtt{snode}\ \{\ \mathtt{int\ val};\ \mathtt{snode\ next};\}$
$\mathtt{lseg(x,p,n)}\ \equiv\ \mathtt{x{=}p} \wedge \mathtt{n{=}0}\ \vee\ \exists \mathtt{q,m}\cdot \mathtt{x{\mapsto}snode(\_,q)} * \mathtt{lseg(q,p,m)} \wedge \mathtt{m{=}n{-}1}$

Our inference technique derives the following specializable predicate definition:

$$\begin{array}{l}\mathtt{lseg(x,p,n)} \equiv \mathtt{x{=}p} \wedge \mathtt{n{=}0} \mid \mathtt{x{=}p}{\wedge}\mathtt{n{=}0} \vee \\ \exists \mathtt{q,m}\cdot \mathtt{x{\mapsto}snode(\_,q)}{*}\mathtt{lseg(q,p,m)}{\wedge}\mathtt{m{=}n{-}1} \mid \mathtt{x{\neq}null}{\wedge}\mathtt{n{>}0}; \\ \mathcal{I} = \{\{1\}{\to}\mathtt{x{=}p}{\wedge}\mathtt{n{=}0},\ \{2\}{\to}\mathtt{x{\neq}null}{\wedge}\mathtt{n{>}0},\ \{1,2\}{\to}\mathtt{n{\geq}0}\}; \\ \mathcal{R} = \{\mathtt{x{=}p}{\leftarrow}\{1\},\ \mathtt{n{=}0}{\leftarrow}\{1\}, \mathtt{x{\neq}null}{\leftarrow}\{2\},\ \mathtt{n{>}0}{\leftarrow}\{2\}\}\end{array}$$

Note that we have a family of invariants, named $\mathcal{I}$, to cater to each of the specialized states. The most general invariant for the predicate is $\mathcal{I}\mathtt{nv}(\mathtt{lseg(x,p,n)},\{1,2\}) = \mathtt{n{\geq}0}$. This is computed by a fix-point analysis [4] on the body of the predicate. If we determine that a particular predicate instance can be specialized to $\mathtt{lseg(x,p,n)}@\{2\}$, we may use a stronger invariant $\mathcal{I}\mathtt{nv}(\mathtt{lseg(x,p,n)},\{2\}) = \mathtt{x{\neq}null} \wedge \mathtt{n{>}0}$ to propagate this constraint from the specialized instance. Such a family of invariants allows us to enrich the context of the predicate instances that are being progressively specialized.

Furthermore, we must process the predicate definitions in a bottom-up order, so that predicates lower in the definition hierarchy are inferred before predicates higher in the hierarchy. This is needed since we intend to specialize the body of each predicate definition with the help of specialized definitions that were inferred earlier. In the case of a set of mutually-recursive predicate definitions, we shall process this set of predicates simultaneously. Initially, we shall assume that the set of pruning conditions for

$$\frac{\kappa\wedge\pi \xrightarrow{\text{if}} \hat{\kappa}\wedge\pi \mid C_1 \qquad \hat{\kappa} \mid C_1 \xrightarrow{\text{sf}}{}^{*} \hat{\kappa}_1 \mid C_2}{\kappa\wedge\pi \xrightarrow{\text{msf}} \hat{\kappa}_1\wedge\pi \mid C_2}\ \underline{[\textsc{init}{-}[\textsc{mutli}{-}\textsc{spec}]]}$$

$$\frac{\begin{array}{c} \mathit{spred}_{\mathit{old}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \sigma_i)) \qquad \forall i{\in}\{1,..,n\} \cdot \sigma_i \xrightarrow{\text{msf}} \hat{\sigma}_i \mid C_i \\ \mathit{spred}_{\mathit{new}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i)) \end{array}}{\mathit{spred}_{\mathit{old}} \xrightarrow{\text{isp}} \mathit{spred}_{\mathit{new}}}\ \underline{[\textsc{isp}{-}[\textsc{spec}{-}\textsc{body}]]}$$

$$\frac{\begin{array}{c} \mathit{spred}_{\mathit{old}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i)) \qquad \rho = [\mathit{inv}_p(\mathtt{v}^*){\mapsto}\mathtt{fix}(\bigvee_{i=1}^{n} \exists u_i^* \cdot C_i)] \\ \mathcal{I} = \{(L{\rightarrow}\mathit{hull}(\bigvee_{i\in L} \exists u_i^* \cdot \rho C_i) \mid \emptyset{\subset}L{\subset}\{1..n\}\} \ \cup \ \{\{1..n\}{\rightarrow}\rho(\mathit{inv}_p(\mathtt{v}^*))\} \\ \mathit{spred}_{\mathit{new}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid \rho C_i);\ \mathcal{I}) \end{array}}{\mathit{spred}_{\mathit{old}} \xrightarrow{\text{isp}} \mathit{spred}_{\mathit{new}}}\ \underline{[\textsc{isp}{-}[\textsc{build}{-}\textsc{inv}{-}\textsc{family}]]}$$

$$\frac{\begin{array}{c} \mathit{spred}_{\mathit{old}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i);\ \mathcal{I}) \qquad G = \bigcup_{i=1}^{n} \mathit{closure}(\mathcal{I}(\{i\})) \\ \mathcal{R} = \bigcup_{\alpha\in G}\{\alpha{\leftarrow}\{i \mid 1{\le}i{\le}n \ \wedge \ \mathcal{I}(\{i\}) \Longrightarrow \alpha\}\} \quad \forall i{\in}\{1,..,n\} \cdot \hat{\sigma}_i \mid C_i \xrightarrow{\text{sf}}{}^{*} \hat{\sigma}_{i,2} \mid C_{i,2} \\ \mathit{spred}_{\mathit{new}} \ = \ (\mathtt{p}(\mathtt{v}^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_{i,2} \mid C_{i,2});\ \mathcal{I};\ \mathcal{R}) \end{array}}{\mathit{spred}_{\mathit{old}} \xrightarrow{\text{isp}} \mathit{spred}_{\mathit{new}}}\ \underline{[\textsc{isp}{-}[\textsc{build}{-}\textsc{prune}{-}\textsc{cond}]]}$$

where $\xrightarrow{\text{sf}}{}^{*}$ *is the transitive closure of* $\xrightarrow{\text{sf}}$; *and* $\mathcal{I}(\{i\}) = \pi_i$, *given* $(\{i\}{\rightarrow}\pi_i) \in \mathcal{I}$.

Fig. 5. Inference Rules for Specializable Predicates

each recursive predicate is empty, which makes its recursive instances unspecializable. However, once its set of pruning conditions has been determined, we can apply further specialization so that the recursive invocations of the predicate are specialized as well.

The formal rules for inferring each specializable predicate are given in Fig. 5. The rule $\underline{[\textsc{init}{-}[\textsc{multi}{-}\textsc{spec}]]}$ converts an unannotated formula into its corresponding specialized form. It achieves this by an initialization step via the $\xrightarrow{\text{if}}$ relation given in Fig. 6, followed by a multi-step specialization using $\xrightarrow{\text{sf}}{}^{*}$, without resorting to case specialization (that would otherwise result in an outer disjunctive formula). This essentially applies a transitive closure of $\xrightarrow{\text{sf}}$ until no further reduction is possible.

The rule $\underline{[\textsc{isp}{-}[\textsc{spec}{-}\textsc{body}]]}$ converts the body of each predicate definition into its specialized form. For each recursive invocation, it will initially assume a symbolic invariant, named $\mathit{inv}_{\mathtt{p}}(\mathtt{v}^*)$, without providing any pruning conditions. This immediately puts each recursive predicate instance in the fully-specialized form.

After the body of the predicate definition has been specialized, we can proceed to build a constraint abstraction for its predicate's invariant, denoted by $\mathit{inv}_{\mathtt{p}}(\mathtt{v}^*)$, in the $\underline{[\textsc{isp}{-}[\textsc{build}{-}\textsc{inv}{-}\textsc{family}]]}$ rule. For example, we may denote the invariant of predicate $\mathtt{lseg}(\mathtt{x},\mathtt{p},\mathtt{n})$ symbolically using $\mathit{inv}_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n})$, before building the following recursive constraint abstraction:

$$\mathit{inv}_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n}) \ \equiv \ \mathtt{x}{=}\mathtt{p}{\wedge}\mathtt{n}{=}0 \ \vee \ \exists \mathtt{q},\mathtt{m} \cdot \mathtt{x}{\neq}\mathtt{null}{\wedge}\mathtt{n}{=}\mathtt{m}{+}1{\wedge}\mathit{inv}_{\mathtt{lseg}}(\mathtt{q},\mathtt{p},\mathtt{m})$$

$$\left[\underline{\textbf{INIT}-[\textbf{EMP}]}\right] \qquad\qquad \left[\underline{\textbf{INIT}-[\textbf{CELL}]}\right]$$

$$\texttt{emp} \mathrel{-\!\text{ih}\!\!\rightarrow} \texttt{emp} \mid \texttt{true} \qquad x{\mapsto}p(v^*) \mathrel{-\!\text{ih}\!\!\rightarrow} x{\mapsto}p(v^*) \mid x{\neq}\texttt{null}$$

$$\left[\underline{\textbf{INIT}-[\textbf{PRED}]}\right]\;\dfrac{p(v^*) \equiv (\bigvee_{i=1}^{n}(\exists u_i^*{\cdot}\hat{\sigma}_i \mid C_i));\mathcal{I};\mathcal{R} \quad C = \mathcal{I}nv(p(v^*),\{1..n\})}{p(v^*) \mathrel{-\!\text{ih}\!\!\rightarrow} p(v^*)@\{1..n\}\#\mathcal{R} \mid C} \qquad \left[\underline{\textbf{INV}-\textbf{DEF}}\right]\;\dfrac{p(v^*) \equiv (\bigvee_{i=1}^{n}(\exists u_i^*{\cdot}\hat{\sigma}_i \mid C_i));\mathcal{I};\mathcal{R} \quad (L{\rightarrow}C) \in \mathcal{I}}{\mathcal{I}nv(p(v^*),L) = C}$$

$$\left[\underline{\textbf{INIT}-[\textbf{HEAP}]}\right]\;\dfrac{\forall i{\in}\{1,2\}{\cdot}\kappa_i \mathrel{-\!\text{ih}\!\!\rightarrow} \hat{\kappa}_i \mid C_i}{\kappa_1{*}\kappa_2 \mathrel{-\!\text{ih}\!\!\rightarrow} \hat{\kappa}_1{*}\hat{\kappa}_2 \mid C_1{\wedge}C_2} \qquad \left[\underline{\textbf{INIT}-[\textbf{FORMULA}]}\right]\;\dfrac{\kappa \mathrel{-\!\text{ih}\!\!\rightarrow} \hat{\kappa} \mid C}{\kappa{\wedge}\pi \mathrel{-\!\text{if}\!\!\rightarrow} \hat{\kappa}{\wedge}\pi \mid C{\wedge}\pi}$$

Fig. 6. Initialization for Specialization

If we apply a classical fix-point analysis to the above abstraction, we would obtain a closed-form formula as the invariant of the `lseg` predicate, that is $inv_{\texttt{lseg}}(\texttt{x},\texttt{p},\texttt{n}) = \texttt{n}{\geq}\texttt{0}$. With this predicate invariant, we can now build a family of invariants for each proper subset L of disjuncts, namely $\texttt{0}{\subset}L{\subset}\{\texttt{1..n}\}$. This is done with the help of the convex hull approximation. The size of this family of invariants is exponential to the number of disjuncts. While this is not a problem for predicates with a small number of disjuncts, it could pose a problem for unusual predicates with a large number of disjuncts. To circumvent this problem, we could employ either a lazy construction technique or a more aggressive approximation to cut down on the number of invariants generated. For simplicity, this aspect is not considered in the present paper.

Our last step is to build a set of pruning conditions for the disjunctive predicates using the $\left[\underline{\textbf{ISP}-[\textbf{BUILD}-\textbf{PRUNE}-\textbf{COND}]}\right]$ rule. This is currently achieved by applying a closure operation over the invariant $\mathcal{I}(\{\texttt{i}\})$ for each of the disjuncts. To obtain a more complete set of pruning conditions, we are expected to generate a set of strong atomic constraints for each of the closure operations. For example, if we currently have a formula $\texttt{a}{>}\texttt{b} \wedge \texttt{b}{>}\texttt{c}$, a strong closure operation over this formula may yield the following set of atomic constraints $\{\texttt{a}{>}\texttt{b},\ \texttt{b}{>}\texttt{c},\ \texttt{a}{>}\texttt{c}{+}\texttt{1}\}$ as pruning conditions and omit weaker atomic constraints, such as $\texttt{a}{>}\texttt{c}$.

Definition 10 (Sound invariant and sound pruning condition). *Given a predicate definition* $p(v^*){\equiv}\bigvee_{i=1}^{n}\texttt{D}_\texttt{i};\ \cdots$*:*

(1) an invariant $L{\rightarrow}\pi$ *is said to be sound w.r.t. the predicate* p *if (1.a)* $\emptyset \subset L \subseteq \{1,..,n\}$*, and (1.b)* $p(v^*)@L\#\_ \models \pi$*.*

(2) a family of invariants $\mathcal{I}$ *is sound if every invariant from* $\mathcal{I}$ *is sound and the domain of* $\mathcal{I}$ *is the set of all non-empty subsets of* $\{1,..,n\}$*;*

(3) a pruning condition $(\alpha{\leftarrow}L)$ *is sound w.r.t. the predicate* p *if (3.a)* $\emptyset \subset L \subseteq \{1,..,n\}$*, (3.b) vars*$(\alpha) \subseteq \{v^*\}$*, and (3.c)* $\forall i{\in}L{\cdot}\texttt{D}_\texttt{i} \models \alpha$*.*

(4) a set of pruning conditions R is sound if every pruning condition in R is sound w.r.t. the predicate p*.*

Property 5. *For each predicate* $\mathtt{p}(v^*)$*, the family of invariants and the set of pruning conditions derived for* $\mathtt{p}$ *by our inference process are sound, assuming the fixpoint analysis and the hulling operation used by the inference are sound.*

A proof of this property can be found in [3].

6 Experiments

We have built a prototype system for our specialization calculus inside an existing program verification system for separation logic, called HIP [14]. Our implementation benefits greatly from two optimizations: *memoization* and *incremental pruning*. The key here is to support the early elimination of infeasible states, by attempting a proof of each atomic constraint α being in contradiction with a given context C. In this way, the context C is allowed to evolve to a monotonically stronger context C_1, such that $C_1 \Longrightarrow C$. Hence, if indeed $C \Longrightarrow \neg\alpha$ is established, we can be assured that $C_1 \Longrightarrow \neg\alpha$ will also hold. This monotonic context change is the basis of the memoization optimization that leads to reuse of previous outcomes of implications and contradictions.

More specifically, we maintain a memoization set, I, for each context C. This denotes a set of atomic constraints that are implied by the context C; that is $\forall\alpha{\in}I\cdot C \Longrightarrow \alpha$. Contradictions of the form $(C{\wedge}\alpha) = \mathtt{false}$ are also memoized in the same way, since we can model it as an implication check $C \Longrightarrow \neg\alpha$. These memoization recalls are only sound approximations of the corresponding implication checks. In case both membership tests fail for a given pruning condition α, we could turn to automated provers (as a last resort) to help determine $C \Longrightarrow \neg\alpha$. Memoization would, in general, help minimize on the number of invocations to the more costly provers.

The early elimination of infeasible states has an additional advantage. We can easily *slice out* relevant constraints from a (satisfiable) context C that is needed to prove an atomic constraint α. This is possible because we detect infeasible branches by proving only one atomic pruning constraint at a time. For example, consider a context $\mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{>}0 \wedge \mathtt{S}{\neq}\{\}$. If we need to prove its contradiction with $\mathtt{n}{=}0$, a naive solution is to use $(\mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{>}0 \wedge \mathtt{S}{\neq}\{\}) \Longrightarrow \neg(\mathtt{n}{=}0)$. A better solution is to slice out just the constraint $\mathtt{n}{>}0$, and then proceed to prove the contradiction using $\mathtt{n}{>}0 \Longrightarrow \neg(\mathtt{n}{=}0)$, leading to an incremental pruning approach that uses smaller proof obligations. To implement this optimization, we partition each context into sets of connected constraints. Two atomic constraints in a context C are said to be *connected* if they satisfy the following relation.

$$\begin{array}{lll} \mathtt{connected}(\alpha_1,\alpha_2) & \text{:-} & (vars(\alpha_1){\cap}vars(\alpha_2)) \neq \{\} \\ \mathtt{connected}(\alpha_1,\alpha_2) & \text{:-} & \exists\alpha{\in}\mathtt{C}\cdot\mathtt{connected}(\alpha_1,\alpha){\wedge}\mathtt{connected}(\alpha_2,\alpha) \end{array}$$

Using this relation, we can easily slice out a set of constraints (from the context) that are connected to each pruning condition.

Fig 7 summarizes a suite of programs tested which included the 17 small programs (comprised of various methods on singly, doubly, sorted and circular linked lists, selection-sort, insertion-sort and methods for handling heaps, and perfect trees). Due to similar outcomes, we present the average of the performances for these 17 programs. We also experimented with a set of medium-sized programs that included complex

Programs (specified props)	LOC	HIP	HIP+Spec	HIP			HIP+Spec		
		Time(s)	Time(s)	Count	Disj	Size	Count	Disj	Size
17 small progs (size)	87	0.86	0.80	229	1.63	12.39	612	1.13	2.97
Bubble sort (size,sets)	80	2.20	2.23	296	2.13	18.18	876	1.09	2.79
Quick sort (size,sets)	115	2.43	2.13	255	3.29	17.97	771	1.27	3.08
Merge sort (size,sets)	128	3.10	2.15	286	2.04	16.74	1079	1.07	2.99
Complete (size,minheight)	137	5.01	2.94	463	3.52	43.75	2134	1.11	10.10
AVL (height, size,bal)	160	64.1	16.4	764	2.90	85.02	6451	1.07	9.66
Heap Trees (size, maxelem)	208	14.9	4.62	649	2.10	56.46	2392	1.02	8.68
AVL (height, size)	340	27.5	13.1	704	2.98	70.65	7078	1.09	10.74
AVL (height, size, sets)	500	657	60.7	1758	8.00	86.79	14662	1.91	10.11
Red Black (size, height)	630	25.2	15.6	2225	3.84	80.91	7697	1.01	3.79

Fig. 7. Verification Times and Proof Statistics (Proof Counts, Avg Disjuncts, Avg Size)

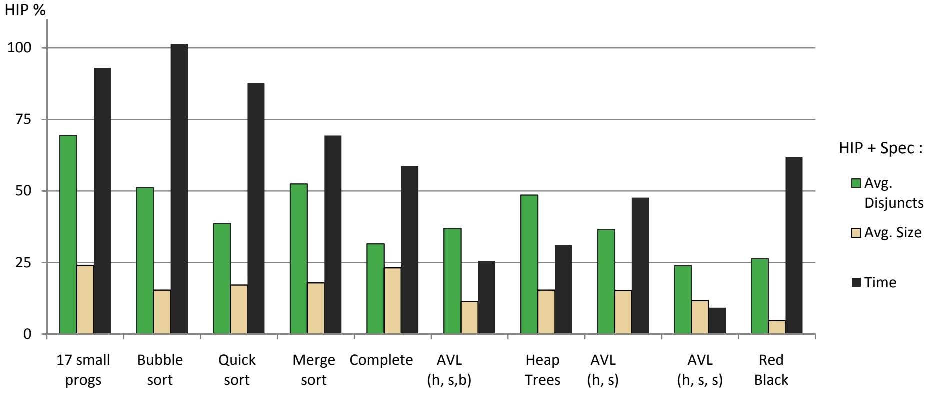


Fig. 8. Characteristic (disjunct, size, timing) of HIP+Spec compared to the Original HIP

shapes and invariants, to support full functional correctness. We measured the verification times taken for the original HIP system, and also the enhanced system, called HIP+Spec, with predicate specialization. For the suite of simple programs, the verifier with specializer runs about 7% faster. For programs with more complex properties (with the exception of bubble sort), predicate specialization manages to reduce verification times by between 12% and 90%. These improvements were largely due to the presence of smaller formulae with fewer disjuncts, as captured in Fig 8. This graph compares the characteristics (e.g. average disjuncts, sizes and timings) of formula encountered by HIP+Spec, as a percentage relative to the same properties of the original HIP system. For example, the average number of disjuncts per proof encountered went down from 3.2 to 1.1; while the size of each proof (based on as the number of atomic formulae) also decreased from an average of 48.0 to 6.5. This speed-up was achieved despite a six fold increase in the proof counts per program from 763 to 4375 used by the specialization and verification processes. We managed to achieve this improvement despite the overheads of a memoization mechanism and the time taken to infer annotations for specializable predicates. We believe this is due to smaller and simpler proof obligations generated with the help our specialization process.

We also investigated the effects of various optimizations on the specialization mechanism. Memoizing implications and contradictions saves 3.47%, while memoizing each context for state change saves 22.3%. For incremental pruning, we have utilized the slicing mechanism which saves 48% on average. We have not yet exploited the incremental proving capability based on strengthening of contexts since our current solvers, Omega and MONA, do not support such a feature. These optimizations were measured separately, with no attempt made to study their correlation. For an extended version of the present paper, including further experimental details, cf. [3].

7 Related Work and Conclusion

Traditionally, specialization techniques [8,17,11] have been used for code optimization by exploiting information about the context in which the program is executed. Classical examples are the partial evaluators based on binding-time analysers that divide a program into static parts to be specialized and dynamic parts to be residualized. However, our work focusses on a different usage domain, proposing a predicate specialization for program verification to prune infeasible disjuncts from abstract program states. In contrast, partial evaluators [8] use unfolding and specialised methods to propagate static information. More advanced partial evaluation techniques which integrate abstract interpretation have been proposed in the context of logic and constraint logic programming [18,17,11]. They can control the unfolding of predicates by enhancing the abstract domains with information obtained from other unfolding operations. Our work differs in its focus on minimizing the number of infeasible states, rather than on code optimization. This difference allows us to use techniques, such as memoization and incremental pruning, that were not previously exploited for specialization.

SAT solvers usually use a conflict analysis [22] that records the causes of conflicts so as to recognize and preempt the occurrences of similar conflicts later on in the search. Modern SMT solvers (e.g. [15,6]) use analogous analyses to reduce the number of calls to underlying theory solvers. Compared to our pruning approach, conflict analysis [22] is a backtracking search technique that discovers contexts leading to conflicts and uses them to prune the search space. These techniques are mostly complementary since they did not consider predicate specialization, which is important for expressive logics.

The primary goal of our work is to provide a more effective way to handle disjunctive predicates for separation logic [14,13]. The proper treatment of disjunction (to achieve a trade-off between precision and efficiency) is a key concern of existing shape analyses based on separation logic [5,10]. One research direction is to design parameterized heap materialization mechanisms (also known as focus operation) adapted to specific program statements and to specific verification tasks [21,12,20,1,16]. Another direction is to design partially disjunctive abstract domains with join operators that enable the analysis to abstract away information considered to be irrelevant for proving a certain property [7,23,2]. Techniques proposed in these directions are currently orthogonal to the contribution of our paper and it would be interesting to investigate if they could benefit from predicate specialization, and vice-versa.

Conclusion. We have proposed in this paper a specialization calculus for disjunctive predicates in a separation logic-based abstract domain. Our specialization calculus is proven sound and is terminating. It supports symbolic pruning of infeasible states within each predicate instance, under monotonic changes to the program context. We have designed inference techniques that can automatically derive all annotations required for each specializable predicate. Initial experiments have confirmed speed gains from the deployment of our specialization mechanism to handle separation logic specifications in program verification. Nevertheless, our calculus is more general, and is useful for program reasoning over any abstract domain that supports disjunctive predicates. This modular approach to verification is being enabled by predicate specialization.

Acknowledgement. We thank the reviewers for insightful feedback. This work is being supported principally by MoE research grant R-252-000-411-112.

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. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
3. Chin, W.N., Gherghina, C., Voicu, R., Le, Q.L., Craciun, F.: A specialization calculus for pruning disjunctive predicates to support verification. Technical report, School of Computing. National University of Singapore (2011), `http://loris-7.ddns.comp.nus.edu.sg/~project/hippruning`
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM POPL, pp. 238–252 (1977)
5. 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)
6. 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)
7. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: ACM PLDI, pp. 256–265 (2007)
8. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
9. Klarlund, N., Moller, A.: MONA Version 1.4 - User Manual. BRICS Notes Series (January 2001)
10. Laviron, V., Chang, B.-Y.E., Rival, X.: Separating shape graphs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 387–406. Springer, Heidelberg (2010)
11. Leuschel, M.: A framework for the integration of partial evaluation and abstract interpretation of logic programs. ACM Trans. Program. Lang. Syst. 26(3), 413–463 (2004)
12. Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 3–18. Springer, Heidelberg (2007)
13. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
14. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)

15. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
16. Podelski, A., Wies, T.: Counterexample-guided focus. In: ACM POPL, pp. 249–260 (2010)
17. Puebla, G., Hermenegildo, M.: Abstract specialization and its applications. In: ACM SIGPLAN PEPM, pp. 29–43 (2003)
18. Puebla, G., Hermenegildo, M., Gallagher, J.P.: An integration of partial evaluation in a generic abstract interpretation framework. In: ACM SIGPLAN PEPM, January 1999, pp. 75–84 (1999)
19. Pugh, W.: The Omega Test: A fast practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
20. Rinetzky, N., Poetzsch-Heffter, A., Ramalingam, G., Sagiv, M., Yahav, E.: Modular shape analysis for dynamically encapsulated programs. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 220–236. Springer, Heidelberg (2007)
21. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
22. Marques Silva, J.P., Sakallah, K.A.: GRASP—a new search algorithm for satisfiability. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, Springer, Heidelberg (1996)
23. 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)

KRATOS – A Software Model Checker for SystemC

Alessandro Cimatti, Alberto Griggio*, Andrea Micheli, Iman Narasamdya,
and Marco Roveri

Fondazione Bruno Kessler — Irst
{cimatti,griggio,amicheli,narasamdya,roveri}@fbk.eu

Abstract. The growing popularity of SystemC has attracted research aimed at the formal verification of SystemC designs. In this paper we present KRATOS, a software model checker for SystemC. KRATOS verifies safety properties, in the form of program assertions, by allowing users to explore two directions in the verification. First, by relying on the translation from SystemC designs to sequential C programs, KRATOS is capable of model checking the resulting C programs using the symbolic lazy predicate abstraction technique. Second, KRATOS implements a novel algorithm, called ESST, that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques to deal with the *Threads*. KRATOS is built on top of NUSMV and MATHSAT, and uses state-of-the-art SMT-based techniques for program abstractions and refinements.

1 Introduction

Formal verification of SystemC has recently gained significant interests [20,14,19,24,17,23,10]. Despite its importance, verification of SystemC designs is hard and challenging. A SystemC design is a complex entity comprising a multi-threaded program where scheduling is cooperative, according to a specific set of rules [22], and the execution of threads is mutually exclusive.

In this paper we present KRATOS, a new software model checker for SystemC. KRATOS provides two different analyses for verifying safety properties (in the form of program assertions) of SystemC designs. First, KRATOS implements a sequential analysis based on lazy predicate abstraction [16] for verifying sequential C programs. To verify SystemC designs using this analysis, we rely on the translation from SystemC to a sequential C program, such that the resulting C program contains both the mapping of SystemC threads in the form of C functions and the encoding of the SystemC scheduler. Second, KRATOS implements a novel concurrent analysis, called ESST [10], that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy predicate abstraction, to deal with the *Threads*.

In this paper we describe the verification flow of KRATOS, its architecture, and the novel functionalities that it features. Due to space limit, the results of an experimental evaluation that compares KRATOS with other model checkers on various benchmarks can be found in [9]. KRATOS is available for download at https://es.fbk.eu/tools/kratos

* Supported by the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 310–316, 2011.

2 Verification Flow

The flow of SystemC verification using Kratos consists of two directions, as shown in Figure 1. The first direction relies on the translation of a SystemC design into a sequential C program, such that the C program contains a function modeling each of the SystemC threads and the encoding of the SystemC scheduler.

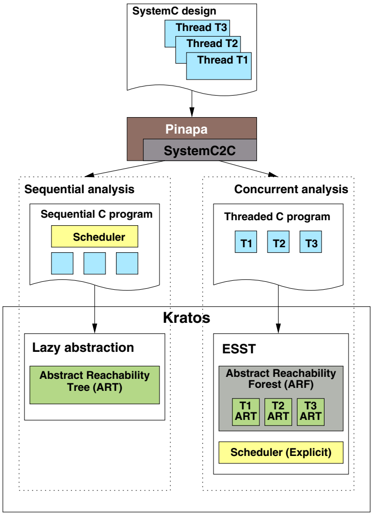


Fig. 1. The SystemC verification flow

Kratos implements a sequential analysis that model checks sequential C programs. This sequential analysis is essentially lazy predicate abstraction [16], which is based on the construction of an abstract reachability tree (ART) by unwinding the control-flow automaton (CFA) of the C program. The ART itself represents the reachable abstract state space. The sequential analysis that Kratos implements is not restricted to the results of SystemC translations, but it can also handle general sequential C programs.

The second direction uses the concurrent analysis, which is the ESST algorithm, to model check SystemC designs. Similar to the first direction, in the second direction the SystemC design is translated into a so-called threaded C program that contains a function for modeling each SystemC thread. But, unlike the sequential C program above, the encoding of the SystemC scheduler is no longer part of the threaded C program. The SystemC scheduler itself is now part of the ESST algorithm and its states are tracked explicitly. ESST is based on the construction of an abstract reachability forest (ARF) where each tree in the forest is an ART of the running thread. The ESST algorithm is described in detail in [10].

Translations from SystemC designs into sequential and threaded C programs are performed by SystemC2C, a new back-end of Pinapa [21].

3 Architecture

The architecture of Kratos is shown in Figure 2. It consists of a front-end that includes a parser for C programs, a type checker, a CFA encoder, and static data-flow analyses and optimization phases.

The parser translates a textual C program into its abstract syntax tree (AST) representation. The AST is then traversed by the type checker to build a symbol table. The CFA encoder builds a CFA or a set of CFAs from the AST. Currently, the CFA encoder provides three encodings: small-block encoding (SBE), basic-block encoding (BBE), and large-block encoding (LBE). In SBE each block consists only of at most one statement. In BBE each block is a sequence of statements that is always entered at the beginning and exited at the end. In LBE, as described in [1], each block is the largest directed

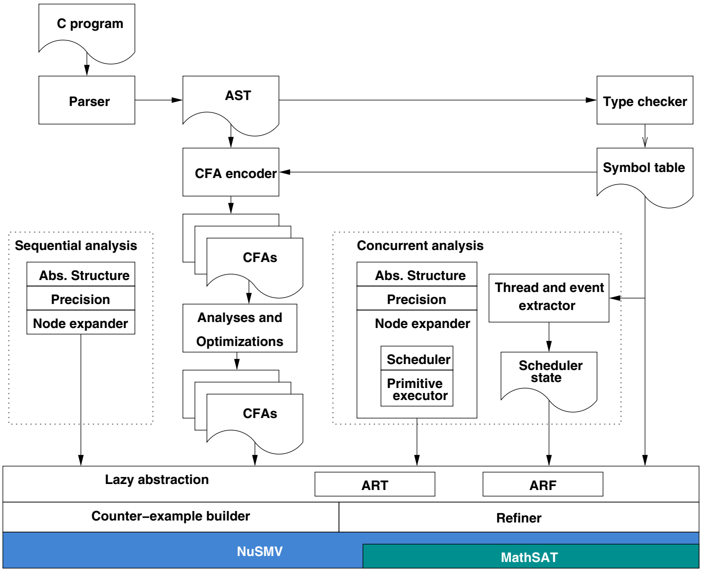


Fig. 2. The architecture of KRATOS

acyclic fragment of the CFA. LBE improves performances by reducing the number of abstract post image computation [1].

Static analyses and optimizations implemented by KRATOS include a simple cone-of-influence reduction that removes nodes of CFAs that do not lead to the error nodes, dead-code elimination, and constant propagation.

The sequential analysis is a reimplementation of the same analysis performed by existing software model checkers based on the lazy predicate abstraction, like BLAST [2] and CPACHECKER [3]. The analysis consists of an abstraction structure, a precision, and a node expander. The abstraction structure contains the representation of abstract states that label ART nodes. A state typically consists of a location (or node) in CFA, a formula denoting the abstract data state, and a stack that keeps track of the trace of function calls. The structure also implements the coverage criteria that stop the expansion of ART nodes. The precision encodes the mapping from locations in CFA to sets of predicates that have been discovered so far. These predicates are relevant predicates used to compute the abstract post images. The node expander expands an ART node by (1) unwinding each of the outgoing edges of the CFA node in the state labelling the ART node, and (2) computing the abstract post image of the state with respect to the statement labelling the outgoing edge. The node expander currently implements depth-first search (DFS), breadth-first search (BFS), and topological ordering strategies for expanding nodes.

For the concurrent analysis, we extract the threads and events from the input threaded C program to create the initial state of scheduler. In this analysis, the node expander is also equipped with a scheduler and a primitive executor. The scheduler explores all possible schedules given a scheduler state as an input. The primitive executor executes calls to functions that modify the state of scheduler. The executor only assumes that the actual arguments of the calls are known statically.

We remark that, the architecture of the concurrent analysis does not assume that the scheduler is a SystemC scheduler. In fact any implementation of a cooperative scheduler with one exclusively running thread in each schedule can be plugged into the analysis.

The sequential and concurrent analyses construct, respectively, the ART and the ARF by following the standard counterexample-guided abstraction refinement (CEGAR) loop [13]. When the analyses cannot reach any error location, then the analyzed program is safe (no assertion violation can occur). When the analyses reach an error location, then the counterexample builder builds a counterexample by constructing the path from the node labelled with the error location to the root of the ART, or to the root of the first ART in the ARF. If the counterexample is non-spurious, in the sense that the formula representing it is satisfiable, then the analyzed program is unsafe. If the counterexample is spurious, then it is passed to the refiner. The refiner tries to refine the precision by discovering new predicates that need to be kept track of by using the unsatisfiable core or interpolation based techniques as described in [15].

KRATOS is built on top of an extended version of NUSMV [7], which is tightly integrated with the MATHSAT SMT solver [5]. KRATOS relies on NUSMV and MATHSAT for abstraction computation, for representing the abstract state within each ART, for the coverage check, for checking the satisfiability of expressions representing counter examples, and for extracting the unsatisfiable core and for generating sequence of interpolants from counterexample paths.

4 Novel Functionalities

KRATOS offers the following novel functionalities.

ESST *algorithm.* The translation from SystemC designs to sequential C programs enables the verification of SystemC using the "off-the-shelf" software model checking techniques. However, such a verification is inefficient because the abstraction of SystemC scheduler is often too aggressive, and thus requires many refinements to reintroduce the abstracted details. The ESST algorithm attacks such an inefficiency by modeling the scheduler precisely, and, as shown in [10], outperforms the SystemC verification through sequentialization.

Partial-order reduction. Despite its relative effectiveness, ESST still has to explore a large number of thread interleavings, many of which are redundant. Such an exploration degrades the run time performance and yields high memory consumptions. Partial-order reduction (POR) is a well-known technique for tackling the state explosion problem by exploring only a representative subset of all possible interleavings. Recently a POR technique has been incorporated in the ESST algorithm [12]. KRATOS currently implements a POR technique based on persistent set, sleep set, and a combination of both.

Advanced abstraction techniques. KRATOS implements Cartesian and Boolean abstraction techniques that are implemented in BLAST and CPACHECKER. In addition, KRATOS also implements hybrid predicate abstraction that integrates BDDs and SMT solvers, as described in [11], and structural abstraction, as described in [8].

Translators. KRATOS is capable of translating the sequential and threaded C to the input languages of other verification engines. For example, KRATOS can translate sequential a C program into an SMV model. By such a translation, one can then use the model checking algorithms implemented by, for example, NUSMV [7] to verify the C program. In particular one can experiment with the bounded model checking (BMC) [4] technique of NUSMV that does not exist in KRATOS.

Under-approximation. KRATOS is also able to generate under-approximations for quick bug hunting. To this extent, KRATOS has recently featured a translation from threaded C programs into PROMELA models [6]. Such a translation enables the verification by under-approximations using the SPIN model checker [18].

Transition encoding. Each block in the CFA is translated into a transition expressed by a NUSMV expression. We have observed that different encodings for transitions can affect the performance of KRATOS. In particular, the encoding for the transitions affect the performance of MATHSAT in terms of abstraction computations and also lead MATHSAT to yielding different interpolants, and thus different discovered predicates. KRATOS provides several different encodings for transitions. They differs in the number of variables needed to encode the transition of each block of the CFA, from the fact that intermediate expressions are folded or not, or whether NUSMV if-then-else expressions are used to compactly represent intermediate expressions. Details about these encodings can be found in the user manual downloadable from the KRATOS' website. Depending on the nature of the problem, the availability of several encodings allows users to choose the most effective one for tackling the problem.

5 Conclusion and Future Work

We have presented KRATOS, a software model checker for SystemC. KRATOS provides two different analyses for verifying SystemC designs: sequential and concurrent analyses. The sequential analysis, based on the lazy predicate abstraction, verifies the C program resulting from the sequentialization of the SystemC design. The concurrent analysis, based on the novel ESST algorithm, combines explicit state techniques with lazy predicate abstraction to verify threaded C program that models a SystemC design. The results of an experimental evaluation, reported in [9], shows that ESST algorithm, for the verification of the considered SystemC benchmarks, outperforms all the other approaches based on sequential analysis. On the considered pure sequential benchmarks, the sequential analysis shows better performance than other state-of-the-art approaches for the majority of the benchmarks.

For future work, we will extend KRATOS to handle a larger subset of C constructs like data structures, arrays and pointers (which are currently treated as uninterpreted functions) and to be able to take into account the bit-precise semantics of operations. We will investigate how to extend the ESST approach to deal with symbolic primitive functions to generalize the scheduler exploration. We would also like to combine the over-approximation analysis, based on the lazy abstraction, with an under-approximation analysis, based on PROMELA translation or BMC. Finally, we will consider to extend the ESST techniques to the verification of concurrent C programs from other application domains (e.g. robotics, railways), where different scheduling policies have to be taken into account

References

1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE, Los Alamitos (2009)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD, pp. 189–197. IEEE, Los Alamitos (2010)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
6. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: An Analytic Evaluation of SystemC Encodings in Promela, submitted for publication
7. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
8. Cimatti, A., Dubrovin, J., Junttila, T., Roveri, M.: Structure-aware computation of predicate abstraction. In: FMCAD, pp. 9–16. IEEE, Los Alamitos (2009)
9. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. Tech. rep., FBK-irst (2011), `https://es.fbk.eu/tools/kratos`
10. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a Software Model Checking Approach. In: FMCAD, pp. 51–59. IEEE, Los Alamitos (2010)
11. Cimatti, A., Franzén, A., Griggio, A., Kalyanasundaram, K., Roveri, M.: Tighter integration of BDDs and SMT for Predicate Abstraction. In: Proc. of DATE, pp. 1707–1712. IEEE, Los Alamitos (2010)
12. Cimatti, A., Narasamdya, I., Roveri, M.: Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 341–356. Springer, Heidelberg (2011)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
14. Große, D., Drechsler, R.: CheckSyC: an efficient property checker for RTL SystemC designs. In: ISCAS, vol. 4, pp. 4167–4170. IEEE, Los Alamitos (2005)
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM, New York (2004)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM, New York (2002)
17. Herber, P., Fellmuth, J., Glesner, S.: Model checking SystemC designs using timed automata. In: CODES+ISSS, pp. 131–136. ACM, New York (2008)
18. Holzmann, G.J.: Software model checking with SPIN. Advances in Computers 65, 78–109 (2005)

19. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: MEMOCODE, pp. 101–110. IEEE, Los Alamitos (2005)
20. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: ACSD, pp. 26–35. IEEE, Los Alamitos (2005)
21. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In: EMSOFT, pp. 317–324. ACM, New York (2005)
22. Tabakov, D., Kamhi, G., Vardi, M.Y., Singerman, E.: A Temporal Language for SystemC. In: FMCAD, pp. 1–9. IEEE, Los Alamitos (2008)
23. Tabakov, D., Vardi, M.Y.: Monitoring Temporal SystemC Properties. In: MEMOCODE, pp. 123–132. IEEE, Los Alamitos (2010)
24. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM Semantics in PROMELA and Its Possible Applications. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 204–222. Springer, Heidelberg (2007)

Efficient Scenario Verification for Hybrid Automata*

Alessandro Cimatti, Sergio Mover, and Stefano Tonetta

Fondazione Bruno Kessler

Abstract. Hybrid Automata (HAs) are a clean modeling framework for systems with discrete and continuous dynamics. Many systems are structured into components, and can be modeled as networks of communicating HAs. Message Sequence Charts (MSCs) are a consolidated language to describe desired behaviors of a network of interacting components and have been extended in numerous ways. The construction of traces witnessing such behaviors for a given system is an important part of the validation. However, specialized tools to solve this problem are missing. The standard approach encodes the constraints in a temporal logic formula or in additional automata, and then use an off the shelf model checker to find witnesses. However, these approaches are too generic and often turn out to be inefficient.

In this paper, we propose a specialized algorithm to find the behaviors of a given network of HAs that satisfies a given scenario. The approach is based on SMT-based bounded model checking. On one side, we construct an encoding which exploits the events of the scenario and enables the incremental use of the SMT solver. On the other side we simplify the encoding with invariants discovered applying discrete model checking on an abstraction of the HAs. The experimental results demonstrate the potential of the approach.

1 Introduction

Complex embedded systems (e.g. control systems for railways, avionics, and space) are made of several interacting components, and feature both discrete and continuous variables. Networks of communicating hybrid automata [16] (HAs) are increasingly used as a formal framework to model and analyze the behavior of such systems: local activities of each component amount to transitions local to each HA; communications and other events that are shared between/visible for various components are modeled as synchronizing transitions of the automata in the network; time elapse is modeled as implicit shared timed transitions.

A fundamental step in the design of these networks is the validation of the models performed by checking if they accept some desired interactions among the components. The language of Message Sequence Charts (MSCs) and its extensions are often used to express scenarios of such interactions. MSCs are especially useful for the end users because of their clarity and graphical content.

The ability to construct traces of a network of HAs that satisfy a given MSC is an important feature to support user validation. However, there has been little research to

* Work supported by the E.U. project MISSA, contract no. ACP7-GA-2008-212088.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 317–332, 2011.

address the specific problem, and the typical approach reduces the problem to "off-the-shelf" model checking solutions: the MSC is translated into an equivalent automaton and the required trace is found by reachability in the cross-product of the automaton with the network. This can turn out to be ineffective, exacerbating the difficulty of the reachability problem, which is already very complex in the case of HAs.

In this paper, we tackle the problem of efficiently finding the traces of a network of HAs that satisfy an MSC. We work in the framework of Bounded Model Checking (BMC), which uses at its core an incremental encoding into Satisfiability-Modulo-Theory (SMT).

We first investigate different automata constructions, exploiting optimizations such as locality and partial-order reduction. Second, we propose a specialized, direct algorithm, where the search is structured around the events in the MSC, which are used as intermediate "islands". The idea is to pre-simplify fragments of the encoding based on the events attached to the islands and to incrementally increase the local paths between two consecutive islands. Further simplifications are achieved by means of invariants that are discovered by applying discrete model checking on an abstraction of the HAs. The generated invariants are either over-approximations of the states that are visited between two subsequent events, just before or just after an event of the scenario.

A key enabler for our work is the use of an alternative, "local time" semantics [6] for HAs, which exploits the fact that automata can be "shallowly synchronized" [7]. The intuition is that each automaton can proceed based on its individual "local time scale", unless they perform a synchronizing transition, in which case they must realign their absolute time. This results in a more concise semantics, where traces of the network are obtained by composing traces of local automata, each with local time elapse, by superimposing structure based on shared communication.

We implement the various approaches in the sub-case of linear hybrid automata, and we use an incremental SMT solver to check the satisfiability of the formulas encoding the reachability problem. We compare the proposed solutions over a wide set of networks and benchmark MSCs. The results show that the direct algorithm is able to construct witnesses for very wide networks, with very long traces, significantly outperforming the other approaches based on the automata construction, and that the use of invariants can be helpful in further reducing computation time.

The paper is structured as follows. In Section 2 we present some background on networks of HAs, and the SMT-based methods for their reachability analysis. In Section 3 we present the language for describing scenarios, and the methods based on automata construction. In 4 we discuss the proposed direct approach to MSC checking. In Section 5 we discuss related work. In Section 6 we experimentally evaluate our approach. In Section 7 we draw some conclusions.

2 Networks of Hybrid Automata

2.1 Networks of Transition Systems

We first define *Labelled Transition Systems* (LTSs), which are then used to define the semantics of Hybrid Automata. An LTS is a tuple $\langle Q, A, Q_0, R\rangle$ where:

- Q is the set of states,
- A is the set of actions/events (also called alphabet),
- $Q_0 \subseteq Q$ is the set of initial states,
- $R \subseteq Q \times A \times Q$ is the set of labeled transitions.

A *trace* is a sequence of events $w = a_1, \ldots, a_k \in A^*$. Given $A' \subseteq A$, the projection $w_{|A'}$ of w on A' is the sub-trace of w obtained by removing all events in w that are not in A'. A *path* π of S over the trace $w = a_1, \ldots, a_k \in A^*$ is a sequence $q_0 \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} \ldots \overset{a_k}{\rightarrow} q_k$ such that $q_0 \in Q_0$ and, $\langle q_{i-1}, a_i, q_i \rangle \in R$ for all i such that $1 \leq i \leq k$. We say that π accepts w. The *language* $L(S)$ of an LTS S is the set of traces accepted by some path of S. Given a state $q \in Q$, the language $L_q(S)$ of an LTS S is the set of traces accepted by some path $q_0 \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} \ldots \overset{a_k}{\rightarrow} q_k$ of S with $q_k = q$.

The *parallel composition* $S_1 || S_2$ of two LTSs $S_1 = \langle Q_1, A_1, Q_{01}, R_1 \rangle$ and $S_2 = \langle Q_2, A_2, Q_{02}, R_2 \rangle$ is the LTS $\langle Q, A, Q_0, R \rangle$ where:

- $Q = Q_1 \times Q_2$,
- $A = A_1 \cup A_2$,
- $Q_0 = Q_{01} \times Q_{02}$,
- $R := \{\langle \langle q_1, q_2 \rangle, a, \langle q_1', q_2' \rangle \rangle \mid \langle q_1, a, q_1' \rangle \in R_1, \langle q_2, a, q_2' \rangle \in R_2\}$
  $\cup \{\langle \langle q_1, q_2 \rangle, a, \langle q_1', q_2 \rangle \rangle \mid \langle q_1, a, q_1' \rangle \in R_1, a \notin A_2\}$
  $\cup \{\langle \langle q_1, q_2 \rangle, a, \langle q_1, q_2' \rangle \rangle \mid \langle q_2, a, q_2' \rangle \in R_2, a \notin A_1\}$.

The parallel composition of two or more LTSs $S_1 || \ldots || S_n$ is also called a *network*. If an event is shared by two or more components, we say that the event is a synchronization event; otherwise, we say that the event is local. We denote with τ_i the set of local events of the i-th component, i.e., $\tau_i = A_i \setminus \bigcup_{j \neq i} A_j$.

Given a network, the *language emptiness problem* is the problem of checking if the language of a network is empty. Given a network $\mathcal{N}$ and a predicate $q \in Q_1 \times \ldots \times Q_n$, the *reachability problem* is the problem of checking if the language $L_q(\mathcal{N})$ is empty.

2.2 Hybrid Automata

A *Hybrid Automaton* (HA) [16] is a tuple $\langle Q, A, Q_0, R, X, \mu, \iota, \xi, \theta \rangle$ where:

- Q is the set of states,
- A is the set of events,
- $Q_0 \subseteq Q$ is the set of initial states,
- $R \subseteq Q \times A \times Q$ is the set of discrete transitions,
- X is the set of continuous variables,
- $\mu : Q \rightarrow P(X, \dot{X})$ is the flow condition,
- $\iota : Q \rightarrow P(X)$ is the initial condition,
- $\xi : Q \rightarrow P(X)$ is the invariant condition,
- $\theta : R \rightarrow P(X, X')$ is the jump condition,

where P represents the set of predicates over the specified variables.

A Linear HA (LHA) is an HA where all the conditions are Boolean combinations of linear inequalities and the flow conditions contain variables in $\dot{X}$ only. We assume also that the invariant conditions of a LHA is a conjunction of inequalities.

A *network* $\mathcal{H}$ of HAs is the parallel composition of two or more HAs. We consider two semantics for networks of HAs: the global-time semantics, where all components synchronize on timed events, and the local-time (or time-stamps) semantics, where the timed events are local and components must synchronize the time on shared events.

In the following, we consider a network $\mathcal{H} = H_1||\dots||H_n$ of HAs with $H_i = \langle Q_i, A_i, Q_{0i}, R_i, X_i, \mu_i, \iota_i, \xi_i, \theta_i \rangle$ such that for all $1 \leq i < j \leq n$ $X_i \cap X_j = \emptyset$ (i.e. the set of continuous variables of the hybrid automata are disjoint).

The *global-time semantics* (or time-action semantics) [16] of $\mathcal{H}$ is the network of LTSs $\mathcal{N}_{\text{GLTIME}}(\mathcal{H}) = S_1||\dots||S_n$ with $S_i = \langle Q'_i, A'_i, Q'_{0i}, R'_i \rangle$ where

- $Q'_i = \{\langle q, \overline{x} \rangle \mid q \in Q_i, \overline{x} \in \mathbb{R}^{|X_i|}\}$,
- $A'_i = A_i \cup \{\langle \textsc{Time}, \delta \rangle \mid \delta \in \mathbb{R}_{\geq 0}\}$,
- $Q'_{0i} = \{\langle q, \overline{x} \rangle \mid q \in Q_{0i}, \overline{x} \in \iota_i(q)\}$,
- $R'_i = \{\langle \langle q, \overline{x} \rangle, a, \langle q', \overline{x}' \rangle \rangle \mid \langle q, a, q' \rangle \in R_i, \langle \overline{x}, \overline{x}' \rangle \in \theta_i(q, a, q'), \overline{x} \in \xi_i(q), \overline{x}' \in \xi_i(q')\} \cup \{\langle \langle q, \overline{x} \rangle, \langle \textsc{Time}, \delta \rangle, \langle q, \overline{x}' \rangle \rangle \mid$ there exists f satisfying $\mu_i(q)$ s.t. $f(0) = \overline{x}, f(\delta) = \overline{x}', f(\epsilon) \in \xi(q), \epsilon \in [0, \delta]\}$.

The *local-time semantics* (or time-stamps semantics) [6] of $\mathcal{H}$ is the network of LTSs $\mathcal{N}_{\text{LOCTIME}}(\mathcal{H}) = S_1||\dots||S_n$ with $S_i = \langle Q'_i, A'_i, Q'_{0i}, R'_i \rangle$ where

- $Q'_i = \{\langle q, \overline{x}, t \rangle \mid q \in Q_i, \overline{x} \in \mathbb{R}^{|X_i|}, t \in \mathbb{R}_{\geq 0}\}$,
- $A'_i = \{\langle a, t \rangle \mid a \in A_i, t \in \mathbb{R}_{\geq 0}\} \cup \{\textsc{Time}_i\}$,
- $Q'_{0i} = \{\langle q, \overline{x}, 0 \rangle \mid q \in Q_{0i}, \overline{x} \in \iota_i(q)\}$,
- $R'_i = \{\langle \langle q, \overline{x}, t \rangle, \langle a, t \rangle, \langle q', \overline{x}', t \rangle \rangle \mid \langle q, a, q' \rangle \in R_i, \langle \overline{x}, \overline{x}' \rangle \in \theta_i(q, a, q'), \overline{x} \in \xi_i(q), \overline{x}' \in \xi_i(q')\} \cup \{\langle \langle q, \overline{x}, t \rangle, \textsc{Time}_i, \langle q, \overline{x}', t' \rangle \rangle \mid$ there exists f satisfying $\mu_i(q)$ s.t. $f(t) = \overline{x}, f(t') = \overline{x}', f(\epsilon) \in \xi_i(q), \epsilon \in [t, t'], t \leq t'\}$.

The definition of the local-time semantics is such that the set of actions of each LTSs contains a local timed event $\textsc{Time}_i$ and couples containing a discrete action and a time stamp (i.e. the amount of time elapsed in the automaton). Thus, each automaton performs the time transition locally, changing its local time stamp. When two automata synchronize on $\langle a, t \rangle$ they agree on the action a and on the time stamp t. Instead, in the global-time semantics, all the automata are forced to synchronize on the time transition $\langle \textsc{Time}, \delta \rangle$, agreeing on the time elapsed during the transition (δ variable).

If $q = \langle \langle q_1, \overline{x_1}, t_1 \rangle, \dots, \langle q_n, \overline{x_n}, t_n \rangle \rangle$ is a state of $\mathcal{N}_{\text{LOCTIME}}$, we say that q is synchronized iff $t_i = t_j$ for $1 \leq i < j \leq n$, i.e., the local times are equal.

Theorem 1 ([6]). *$\langle \langle q_1, \overline{x_1} \rangle, \dots, \langle q_n, \overline{x_n} \rangle \rangle$, is reachable in $\mathcal{N}_{\text{GLTIME}}(\mathcal{H})$ iff there exists a synchronized state $\langle \langle q_1, \overline{x_1}, t \rangle, \dots, \langle q_n, \overline{x_n}, t \rangle \rangle$ reachable in $\mathcal{N}_{\text{LOCTIME}}(\mathcal{H})$.*

In an extended version of this paper, available at `http://es.fbk.eu/people/mover/hybrid_scenario/`, we prove a stronger version of the theorem, which shows that also a time-abstract version of the traces is preserved.

2.3 SMT Encoding of Hybrid Automata

As described in [16], LHAs can be analyzed with symbolic techniques. Let us consider a network $\mathcal{H} = H_1||\dots||H_n$ of LHAs whose semantics (either global or local time) is given by the network of LTSs $S_1||\dots||S_n$ where $S_i = \langle Q_i, A_i, Q_{i0}, R_i\rangle$. The states Q_i can be represented by a set V_i of symbolic variables such that there exists a mapping χ from the assignments of V_i to the states of the S_i. The events of A_i can be represented by a set of symbolic variables W_i such that χ maps every assignment of variables in W_i to an event in A_i. Sets of states are represented with formulas over V_i, while sets of transitions are represented with formulas over V_i, W_i, and V_i', which are the next values of V_i. In particular, it is possible to define a formula $I_i(V_i)$ such that $\mu(V_i) \models I_i$ iff $\chi(\mu) \in Q_{i0}$, and a formula T_i such that $\mu(V_i), \nu(W_i), \mu'(V_i') \models T_i$ iff $(\chi(\mu), \chi(\nu), \chi(\mu')) \in R_i$, where $\mu(V_i), \nu(W_i)$ and $\mu(V_i')$ are assignments to the set of variables V_i, W_i and V_i'.

The details of the encoding we use can be found in [7]. Here, we just notice that we use a scalar input variable ε to represent the events of H_i adding two distinguished values, namely T and S, to represent a timed transition and stuttering, respectively. When stuttering, the system does not change any variable. Moreover, when encoding the global-time semantics a further real input variable δ_i represents the time elapsed in a timed transition. Instead, when using the local-time semantics, the variable t_i represents the local time of H_i and is also used as time-stamp of the events (thus, to ensure that shared events happen at the same time).

As standard in Bounded Model Checking, given an integer k, we can build a formula whose models correspond to all paths of length k of the represented LTS S. The formula introduces $k + 1$ copies of every variable in the encoding of the automata. Given a formula ϕ, we denote with ϕ^i the result of substituting the current and next variables of ϕ with their i-th and $(i + 1)$-th copy, respectively. The paths of S of length k can be encoded into the formula $path(k) := I^0 \wedge \bigwedge_{0\leq i<k} T^i$.

Theorem 2. *There exists a mapping χ from the models of $path(k)$ to the paths of S of length k.*

Most of modern solvers, both for SAT and SMT, have an *incremental* interface such that, if a problem is fed to the solver incrementally, the solver can first tackle smaller parts of the problem and then pass to large parts managing to reuse the lemmas discovered during the previous searches. Suppose the problem is parametrized by a certain k, i.e. $PB = \exists k.PB(k)$. In order to exploit the incremental interface of the solver, the problem $PB(k)$ is formulated into two parts $PB(k) = \alpha(k) \wedge \beta(k)$, as in [9], such that $\alpha(0) = \gamma(0)$ and $\alpha(k+1) = \alpha(k) \wedge \gamma(k+1)$. This way, the solver faces sub-problems of incremental difficulties and can reuse previous results:
$\gamma(0) \wedge \beta(0)$
$\gamma(0) \wedge \gamma(1) \wedge \beta(1)$
$\gamma(0) \wedge \gamma(1) \wedge \gamma(2) \wedge \beta(2)$
...

If we want to solve the reachability problem, we look for a k for which the problem $PB(k) = path(k) \wedge target^k$ is satisfiable. The problem is usually presented to the solver in the following form: $\gamma(0) := I^0$, $\gamma(k) := T^{k-1}$, $\beta(k) := target^k$, for $k > 0$.

The non-monotonicity of the encoding is handled with a standard stack-based interface of the SMT solver (PUSH, ASSERT, SOLVE, POP primitives). This allows, after asserting $\gamma(k)$, to set a backtrack point (PUSH), assert $\beta(k)$ (ASSERT), check the satisfiability of the conjunction of the asserted formulas (SOLVE), and to restore the state of the solver (i.e. asserted formulas and learned clauses) at the backtrack point (POP). This way, the $k{+}1$-th problem is solved keeping all the learned clauses related to $\gamma(k)$.

3 Message Sequence Charts

3.1 MSCs for Networks of Hybrid Automata

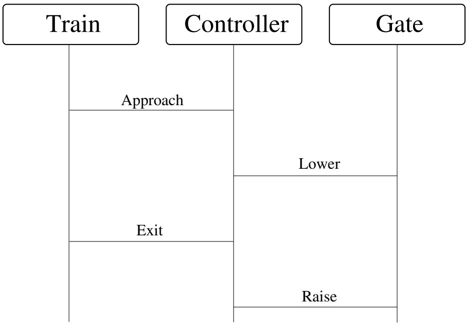


Fig. 1. An MSC for the Train-Gate-Controller model [16]

A Message Sequence Chart (MSC) [23] defines a possible interaction of components in a network $\mathcal{N}$. An MSC m is associated with a set of events $A_m \subseteq A_{\mathcal{N}}$, subset of the events of the network. The MSC defines a sequence of events for every component S of the network, called instance of S.

The typical implicit assumption is that the set A_m contains all the synchronization events of the network. However, if $\mathcal{N}$ is a network of hybrid automata, even if we consider the global-time semantics in which the timed event is shared, the timed event is not part of A_m and, thus, is not present in the sequence of events specified by the MSC. Therefore, we assume that independently from the semantics, if $\mathcal{N}$ is a network of the hybrid automata $H_1, \ldots, H_n$ with alphabet respectively $A_1, \ldots A_n$, then $A_m = \bigcup_{1 \leq i < j \leq n} A_i \cap A_j$ and thus the (global or local) timed event is not part of A_m[1].

An instance σ for the LTS S is a sequence $a_1; \ldots; a_h \in (A_m \cap A_S)^*$ of events of S. S accepts the instance ($S \models \sigma$) iff there exists a trace w accepted by S ($w \in L(S)$) such that the sub-sequence of events in A_m is equal to σ ($w_{|A_m} = \sigma$). In other words, S accepts the instance iff there exists a path π of S over a trace compatible with the instance σ. In such cases, we say that $\pi \models \sigma$.

An MSC is the parallel composition $\sigma_1 || \ldots || \sigma_n$ of $\sigma_1, \ldots, \sigma_n$ where σ_i is an instance of S_i. The network $\mathcal{N}$ of LTSs accepts the MSC m ($\mathcal{N} \models m$) iff there exists a trace w accepted by $\mathcal{N}$ ($w \in L(\mathcal{N})$) such that, for every S_i, the sub-sequence of events in $A_m \cap A_{S_i}$ is equal to σ ($w_{|(A_m \cap A_{S_i})} = \sigma$). In other words, $\mathcal{N}$ accepts the instance iff there exists a path of $\mathcal{N}$ over a trace compatible with every instance of the MSC. If $\mathcal{H}$ is a network of HAs, then we say that $\mathcal{H} \models m$ iff $\mathcal{N}_{\text{GLTIME}}(\mathcal{H}) \models m$.

The model checking problem for an MSC m is the problem of checking if a network satisfies an MSC. If we define the language $L(m)$ of an MSC m as the traces compatible with the instances of m, the model checking problem can be seen as the problem of checking if $L(\mathcal{N}) \cap L(m) \neq \emptyset$.

An MSC $\sigma_1 || \ldots || \sigma_n$ is consistent iff for every pair of instances σ_i and σ_j the projection on the common alphabet is the same, i.e., if $A = A_i \cap A_j$, $\sigma_{i|A} = \sigma_{j|A}$. In other words, the MSC m is consistent iff $L(m) \neq \emptyset$. Henceforth, we assume that the MSCs

[1] The techniques presented in this paper can be adapted to consider also the case where a synchronization is not in A_m.

are consistent. The check of consistency is trivial and can be done syntactically on the graphical representation of the MSC.

Example 1. Figure 1 shows an MSC for the railroad model from [16]. There is an instance for each automaton in the network, *Train*, *Controller* and *Gate*. The MSC represents a scenario where the *Train* communicates with the controller when approaching the *Gate* and the controller synchronizes with the *Gate* to close it. When the *Train* is far, it synchronizes with the *Controller*, which opens the *Gate*.

3.2 Standard Global Automata Construction

The classic construction of an automaton which monitors the satisfaction of an MSC m proceeds by building one state for combination of locations. Every instance has one location before each event and a final location after the last event. A cut through m is a set of locations, one for each instance (we do not require the cuts to be downward closed with regard to the events because we guarantee that the reachable cuts respect the events). It is called "cut" because it cuts the graphical representation of the MSC into two parts, the one already visited and the one to be monitored.

Formally, if the instance $\sigma_i = a^i_1; \ldots ; a^i_{h_i}$, we represent a location with indexes from 0 to h_i. A cut is therefore a tuple of indexes. Given a cut $c = \langle c_1, \ldots, c_n\rangle$, an event a is said enabled in c iff there exists a set of indexes $J \subseteq [1, n]$ such that for all $j \in J$, $a \in A_j$, the event after the location c_j is a (i.e., $a^i_{c_j+1} = a$), and, for all $j \notin J$, $a \notin A_j$. Note that for a given event a, the set J is unique and we denote it with J^c_a.

The LTS S_m corresponding to an MSC $m = \sigma_1|| \ldots ||\sigma_n$, with $\sigma_i = a^i_1; \ldots ; a^i_{h_i}$, is defined as follows:

- $Q = [0, h_1] \times \ldots \times [0, h_n]$,
- $A = A_m$,
- $Q_0 = \langle 0, \ldots, 0\rangle$,
- $R = \{(c, a, c') \mid a$ is enabled in c and, for all $j \in J^c_a$, $i'_j = i_j + 1$, and, for all $j \notin J^c_a$, $i'_j = i_j\}$.

Theorem 3. $L(m) = L(S_m)$.

3.3 Exploiting Independent Events

Reduced Global Automata. In the case of synchronizations on discrete events as for hybrid automata (thus without shared variables), two transitions on different components with different events are independent. This means that the order of the transitions does not affect the state after their application. If the order is irrelevant for the search problem, we can fix an arbitrary interleaving. As noted in [6], this reduction can be amplified if we adopt the local-time semantics, since timed transitions become local and independent from the timed transitions of other components.

In the case of model checking MSC, since we are interested in finding one trace compatible with the MSC, if we use the local-time semantics, we can fix an arbitrary interleaving of parallel events in an MSC, and produce an automaton which is linear in the number of events.

If J and J' are subsets of $[1, n]$, we say that $J < J'$ if J contains an integer lower than any integer in J'. If a_1 and a_2 are enabled in c, we say that $a_1 < a_2$ iff $J^c_{a_1} < J^c_{a_2}$. Clearly, given a cut c, there exists a minimum enabled event.

We can build a reduced LTS S_m corresponding to an MSC m as follows:

- $Q = [0, h_1] \times \ldots \times [0, h_n]$,
- $A = A_m \times \mathbb{R}_{\geq 0}$,
- $Q_0 = \langle 0, \ldots, 0 \rangle$,
- $R = \{\langle c, \langle a, t \rangle, c' \rangle \mid c \in Q, t \in \mathbb{R}_{\geq 0}$ and a is the minimum enabled event in c and, for all $j \in J^c_a$, $i'_j = i_j + 1$, and, for all $j \notin J^c_a$, $i'_j = i_j\}$.

Theorem 4. *If* $\mathcal{N}_{\text{LocTime}}(\mathcal{H}) = S_1 || \ldots || S_n$, $\mathcal{H} \models m$ *iff* $L(S_1 || \ldots || S_n || S_m) \neq \emptyset$.

Distributed Automata. Another way to exploit the independence of the parallel events in a MSC is to build a distributed version of the LTS S_m, with one component for every component of the network. We then apply the step-semantics encoding [14] to parallelize the encoding of independent transitions. Let us define the S^i_m as follows:

- $Q = [0, h_i]$,
- $A = A_m \times \mathbb{R}_{\geq 0}$,
- $Q_0 = 0$,
- $R = \{\langle u, \langle a, t \rangle, u' \rangle \mid u \in Q, a = a^i_{u+1}$ and $u' = u + 1\}$.

Theorem 5. *If* $\mathcal{N}_{\text{LocTime}}(\mathcal{H}) = S_1 || \ldots || S_n, \mathcal{H} \models m$ *iff* $L(S_1 || S^1_m || \ldots || S_n || S^1_m) \neq \emptyset$.

A similar result can be obtained for the global-time semantics, as proposed in [19].

4 Scenario-Driven Encoding

4.1 Encoding Tailored to MSCs

The drawbacks of the traditional SMT-based encoding is that it cannot exploit the sequence of messages prescribed by the MSC in order to simplify the search because of the uncertainty on the number of local steps between two events. We encode the path of each automaton independently, exploiting the local time semantics, and then we add constraints that force shared events to happen at the same time, as in *shallow synchronization* [7]. Moreover, we fix the steps corresponding to the shared events and we parametrize the encoding of the local steps with a maximum number of transitions.

Let us consider a network $\mathcal{H} = H_1 || \ldots || H_n$ of LHAs and the encoding $\langle V_i, W_i, I_i, T_i \rangle$ representing the LHA H_i, for $1 \leq i \leq n$, in the local-time semantics. We denote with $T_{i|\phi}$ the transition condition restricted to the condition ϕ, i.e., $T_{i|\phi} = T_i \wedge \phi$. We abbreviate $T_{i|\varepsilon = a}$ with $T_{i|a}$ and $T_{i|\varepsilon \in \tau_i \cup \{\text{S}\}}$ with $T_{i|\tau}$ (notice that τ_i, the set of local actions, contains also the timed event T).

Let us fix a maximum number k of local events between two shared events. We encode the path of the i-th component along the instance $\sigma_i = a_1; \ldots; a_{h_i}$ of an MSC into the following formula:

$$enc(\sigma_i,k) := I_i^0 \wedge \bigwedge_{1\le j\le k} T_{i|\tau}^{j-1} \wedge \bigwedge_{1\le u\le h_i} (T_{i|a_u}^{(u*k)+u-1} \wedge \bigwedge_{1\le j\le k} T_{i|\tau}^{(u*k)+u+j-1}) \quad (1)$$

Intuitively, $enc(\sigma_i,k)$ encodes the alternation of sequences of at most k local steps with the events in the instance σ_i. Note that the u-th event is encoded at the $((u*k)+u-1)$-th step because it is preceded by $u*k$ many local events and $u-1$ shared events.

The run of a network along a consistent MSC $m = \sigma_1||\ldots||\sigma_n$ can be encoded into:

$$enc(m,k) := \bigwedge_{1\le j\le n} enc(\sigma_j,k) \wedge \bigwedge_{1\le j<i\le n} sync(\sigma_j,\sigma_i) \wedge (t_j^{last_j} = t_i^{last_i}) \quad (2)$$

where $last_i = (k+1)*h_i + k$ is the index of the last state of the encoding and $sync(\sigma_j,\sigma_i)$ says that the h-th occurrence of a shared event must occur at the same time in σ_j and σ_i. More, specifically, if $A = A_i \cap A_j$ and $\sigma_{j|_A} = \sigma_{i|_A} = a_1;\ldots a_l$ and $f_i, f_j : \mathbb{N} \to \mathbb{N}$ are such that $a_z = \sigma_i(f_i(z)) = \sigma_j(f_j(z))$, for $1 \le z \le l$, then:

$$sync(\sigma_j,\sigma_i) := \bigwedge_{1\le z\le l} t_i^{(f_i(z)*k)+f_i(z)-1} = t_j^{(f_j(z)*k)+f_j(z)-1} \quad (3)$$

The function $f_i(z)$ maps the z-th event a_z in $\sigma_{i|_A}$ to the index of a_z in σ_i. Thus, the index $(f_i(z)*k)+f_i(z)-1$ is the same index used in the encoding of $enc(\sigma_i,k)$ to encode the transition labeled with the shared event a_z.

Note that the encoding allows to express very complex constraints on the MSC. In fact, it is possible to formulate constraints on the states of the network along events just referring to the symbolic variables that represent them.

Example 2. We show the encoding of the MSC of Example 1 for the *Train* automaton with 2 local steps and the synchronization constraints with the *Controller* automaton. I_{Train} and T_{Train} are the initial condition and the transition relation *Train*.

$$\begin{aligned} enc(\sigma_{train},2) := {} & I^0_{Train} \wedge T^0_{Train|\tau} \wedge T^1_{Train|\tau} \wedge \\ & T^2_{Train|Approach} \wedge T^3_{Train|\tau} \wedge T^4_{Train|\tau} \wedge \\ & T^5_{Train|Approach} \wedge T^6_{Train|\tau} \wedge T^7_{Train|\tau} \wedge \\ sync(\sigma_{Train},\sigma_{Controller}) := {} & t^2_{Train} = t^2_{Controller} \wedge t^5_{Train} = t^{11}_{Controller} \end{aligned}$$

Theorem 6.[2] *Given a consistent MSC m for the network $\mathcal{H}$, if $enc(m,k)$ is satisfiable then $\mathcal{H} \models m$. Vice versa, if $\mathcal{H} \models m$, then there exists an integer k such that $enc(m,k)$ is satisfiable.*

4.2 Incrementality

The encoding is conceived in order to maximize the incrementality of the solver, as described in Section 2.3, along the increase of k, the length of a sequence of local steps.

[2] The proof of the theorem is available at `http://es.fbk.eu/people/mover/hybrid_scenario/`.

The idea is that we keep encodings of the sequences of local transitions separated and we unroll them incrementally, while we add and remove accordingly the constraints which glue such sequences.

Let us fix a maximum K as a bound for the number k of local transitions between two shared events. We use K for distancing enough the (fixed) positions of events. With regard to the formulas introduced in Section 2.3, we define the partial encoding for an instance σ_i as follows:

$$\gamma_{enc(\sigma_i)}(0) := I_i^0 \wedge \bigwedge_{1 \leq u \leq h_i} T_{i|a_u}^{(u*K)+u-1}$$
$$\gamma_{enc(\sigma_i)}(k) := \bigwedge_{0 \leq u \leq h_i} T_{i|\tau}^{(u*K)+u+k-1}$$
$$\beta_{enc(\sigma_i)}(k) := \bigwedge_{0 \leq u \leq h-1} V^{(u*K)+u+k} = V^{(u*K)+u+K}$$

For each instance σ_i we encode the initial condition and all the h_i events in $\gamma_{enc(\sigma_i)}(0)$. We incrementally increase the length of the local step in $\gamma_{enc(\sigma_i)}(k)$ and in $\beta_{enc(\sigma_i)}(k)$, which glues the last state of a sequence of local steps with the first state that performs the next shared event.

The incremental encoding considering the whole MSC m is defined as follows:

$$\gamma(0) := \bigwedge_{1 \leq i \leq n} \gamma_{enc(\sigma_i)}(0) \wedge \bigwedge_{1 \leq i < j \leq n} sync(\sigma_i, \sigma_j)$$
$$\gamma(k) := \bigwedge_{1 \leq i \leq n} \gamma_{enc(\sigma_i)}(k)$$
$$\beta(k) := \bigwedge_{1 \leq i \leq n} \beta_{enc(\sigma_i)}(k) \wedge \bigwedge_{1 \leq j < n} t_j^{last_j} = t_{j+1}^{last_{j+1}}$$

Theorem 7. *There exists a renaming of variables ι such that $enc(m, k)$ and $\bigwedge_{0 \leq j \leq k} \gamma(j) \wedge \beta(k)$ are the syntactical equal modulo the renaming.*

4.3 Scenario-Driven Invariants Generation

In order to strengthen the scenario-driven encoding of Section 4.1, and thus speed up the search, we generate invariants from abstractions of the hybrid automata in the network.

Each instance σ of the MSC restricts the behavior of the automaton S. We abstract S to a finite state system $\hat{S}$ and we use standard techniques, in our case BDDs, to generate invariants which holds in different sections of σ. In particular, we find the reachable states of $\hat{S}$ between two events, just before an event, and just after an event. The invariants are then conjoined to the scenario encoding.

Consider the instance $\sigma = a_1; \ldots; a_h$ of an MSC m. If $S \models \sigma$, by definition, there exists a path π of S over trace w such that $w_{|A_m} = \sigma$. In order to satisfy σ, π alternates sequences of consecutive local events with shared events. More, specifically, if $\pi \models \sigma$, π must be in the form $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{j_1} \xrightarrow{a_1} q_{j_1+1} \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{j_h} \xrightarrow{a_h} q_{j_h+1} \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{j_{h+1}}$, where $q_i \in Q$ and τ are local events of S. We split the path π into a set Υ of sub-sequences such that $\Upsilon = \{\pi_{pre_i}, \pi_{post_i} \mid i \in 1, \ldots, h\} \cup \{\pi_{\alpha_i} | i \in 0, \ldots, h\}$, where:

- $\pi_{pre_i} = \{q_{j_i}\}$, it is the source state of the transition labeled with a_i in π.
- $\pi_{post_i} = \{q_{j_i+1}\}$, it is the destination state of the transition labeled with a_i in π.
- $\pi_{\alpha_i} = \{q_{j_i+1}, \dots, q_{j_{i+1}}\}$ where we denoted 0 with $j_0 + 1$.

Given an MSC instance $\sigma = a_1; \dots; a_h$ for the system S, we find the constraints $pre_i, post_i$, $1 \leq i \leq h$ and α_i, $0 \leq i \leq h$, such that, for every π such that $\pi \models \sigma$:

- for all u, $0 \leq u \leq h$, for all $q \in \pi_{\alpha_u}$, $q \models \alpha_u$;
- for all u, $1 \leq u \leq h$, for all $q \in \pi_{pre_u}$, $q \models pre_u$;
- for all u, $1 \leq u \leq h$, for all $q \in \pi_{post_u}$, $q \models post_u$.

Thus, the constraints are necessary conditions for the paths to satisfy the instance. We can safely strengthen the encoding of the scenario with such constraints in order to speed up the search.

We perform the invariant generation process for S and σ in three different steps: we compute the abstraction $\hat{S}$, we perform a forward reachability computing a first set of invariants and finally we refine the invariants with a backward reachability analysis. We compute the Boolean Abstraction $\hat{S}$ of S, replacing each predicate of S with a fresh Boolean variable, and we represent $\hat{S}$ with Binary Decision Diagrams (BDDs). Then, we perform a forward reachability analysis on $\hat{S}$ computing an over-approximation of $post_i$ and α_i. We start the reachability analysis from the initial states of $\hat{S}$, and we compute α_0 with a fixed-point of the image restricted to the local events τ. Then, starting from α_0, we compute $post_1$ with a single image computation restricted to a_1. We alternate these two steps for all a_i of σ. Finally, we perform a backward reachability analysis on $\hat{S}$ to compute pre_i and to refine $post_i$ and α_i. We start from $post_h$ and we compute the precise pre_h as the intersection of α_{h-1} and the pre-image of $post_h$ restricted to the event a_h. Then, we refine α_{h-1}, intersecting it with the fixed-point of the pre-image which starts from pre_h and is restricted to τ. At this point we refine $post_{h-i}$, intersecting it with α_{h-1}. We iterate these steps following σ in reverse order.

5 Related Work

There have been a lot of extensions to MSC: High-Level Message Sequence Charts [20], UML's sequence diagrams and Live Sequence Charts [10]. While several works aim to increase the expressiveness of these languages, MSCs are a basic building blocks, used to describe the parallel composition of sequences of shared events among components. In this paper, we consider the basic version of MSCs, which describe the parallel composition of sequences of events and, as in [18], we consider the semantics of MSCs in terms of traces, i.e., they constrain the sequence of observable/shared events.Our approach can be extended to manage more expressive languages, such as High-Level Message Sequence Chart, which adds the alternative, sequential and parallel composition of MSCs, using the scenario-based encoding as a building block. We can easily manage time constraints in the MSC as introduced in [2, 5].

MSC and its extensions have been used in [3, 22] to describe an entire system. In [3] they solve the model checking problem for a system expressed with MSCs translating then into automata, while in [22] the authors check the consistency of UML Message

Sequence Charts using linear programming. Both approaches differ from ours, since MSCs are used as a modeling language and not as a specification language.

Many works present different translations of MSCs into temporal logics or automata used for model checking a design. Most of these works focus on covering different aspects of the extensions proposed by LSCs. In [11], the authors study the expressive power of LSCs compared to CTL*. In [17], the authors consider charts with universal semantics concentrating on (discrete-time) timing constraints and synchronous events; they translate the charts into Timed Büchi Automata which are then translated into temporal logic to be checked with the model checker STATEMATE. The work is extended in [19] to handle more expressive timing constraints, translating an LSC either into a global timed automaton or into a network of timed automata; moreover, the translation is integrated with the UPPAAL model checker. In these works, the model checking techniques are used off-the-shelf, but the verification engine is not optimized to deal with the scenario specification. Surprisingly, there are not many works that optimize the algorithms in order to scale up the analysis exploiting the structure of the scenarios. The key difference with such works is in that here we focus on efficiency, rather than on covering all the features provided by MSC extensions. We note that our approach can be easily extended to deal with several features of extended MSCs.

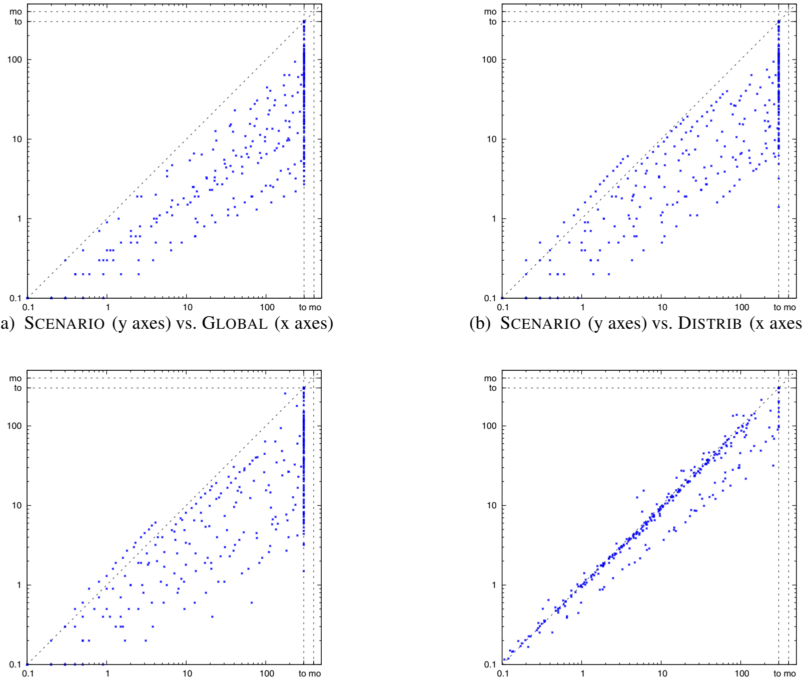
Bounded model checking for hybrid systems using SMT solvers has been investigated in [1, 4, 7, 12, 13]. These techniques can be used to verify a scenario by translating the scenario into an automaton. Still, such techniques are not tailored to the verification of a scenario, and they result in a loss of efficiency since the structure of the problem under analysis is not taken into account. Existing optimizations to the BMC encoding [1] are orthogonal to the scenario-based encoding and can be applied when encoding sequence of local events. Our specific encoding guided by the scenario is inspired by [7], where the authors present a different semantics of the BMC problem for hybrid systems, obtained by composing traces of the local automata, and superimposing compatibility constraints resulting from the synchronizations. In fact, in our approach, the encodings of the automata are local, and a synchronization between two or more automata can happen at different times in the encoding.

6 Experimental Evaluation

The techniques discussed in previous sections were implemented on top of the model checker NUSMV3, that features a bounded model checking approach to the reachability in networks of HAs., and uses at its core the MATHSAT SMT solver. We implemented the approach based on the automata constructions, for which we perform a reachability analysis using the incremental BMC search of NUSMV3 on the composition of the network and the scenario automaton. The reachability target is given by the final states of the automaton. The scenario-driven encodings were implemented in the same framework; for the invariant computation we use the BDD-based model checking of NUSMV3 applied on a Boolean abstraction of the HAs. Also the scenario-driven encoding search was implemented exploiting the incrementality of the SMT solver. In the following, we call SCENARIO the scenario-driven encoding and SCENARIOINVAR its variant simplified with invariants. As for the translations of the MSC into automata, GLOBAL is the reduced global automata which uses the local time semantics, DISTRIB is the distributed automata with global time semantics, and DISTRIBLOCAL is the distributed automata with local time semantics.

In the experimental evaluation, we used the following benchmarks taken from the literature and formalized using the HYDI language [8]. *Star-shape Fischer* is a hybrid version of the Fischer mutual exclusion protocol, that uses a shared variable to control the access to a critical session. *Ring-shape Fischer* is hybrid variant where processes are in a ring, and each process shares a lock variable with its neighbors. *Nuclear Reactor* [25] model the control of a nuclear reactor with n rods. *Distributed Controller* [15] models the interactions of n sensors with a preemptive scheduler and a controller. *Electronic Height Control System (EHC)* [21] is an industrial case study of a system which controls the height of a chassis by pneumatic suspension. The original non-linear model is linearized using *linear-phase portrait partitioning*, as proposed in [21]. For each benchmark, we defined meaningful MCSs that describe the interaction of all the automata in the benchmarks, possibly containing parallel event synchronizations. All the MSCs used in the experimental evaluation are satisfiable.

We evaluated the scalability of the proposed approaches with respect to the number of components in the network and with respect to the length of the MSCs. We increase the number of the components for all the benchmarks, except for the *EHC* which has a fixed number of processes. We increase the length of the MSCs repeating the sequence



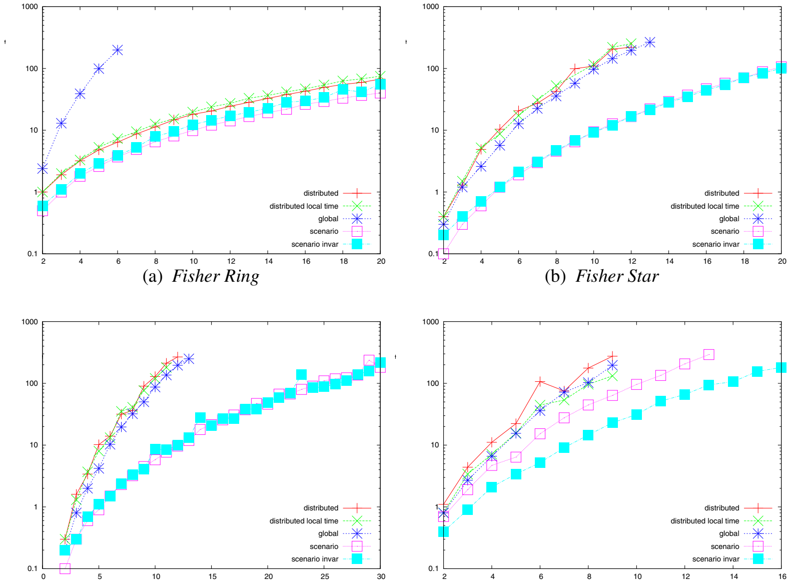
(a) SCENARIO (y axes) vs. GLOBAL (x axes)

(b) SCENARIO (y axes) vs. DISTRIB (x axes)

(c) SCENARIO (y axes) vs. DISTRIBLOCAL (x axes)

(d) SCENARIOINVAR (y axes) vs. SCENARIO (x axes)

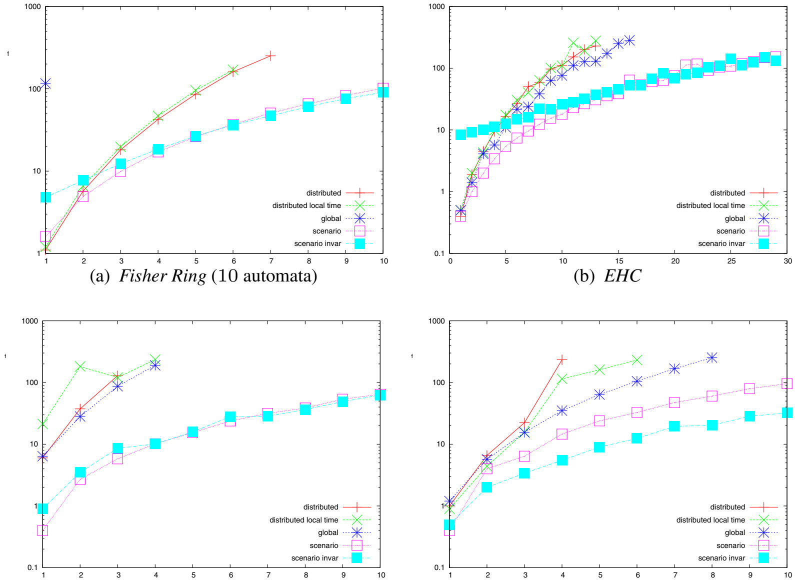
Fig. 2. (a)-(c) Scatter plots of run times (sec.). (d) the reduction due to invariants on search time.



(a) *Fisher Ring* (b) *Fisher Star*

(c) *Nuclear Reactor* (d) *Distributed Controller*

Fig. 3. Run times (sec. on the y axes) increasing the number of automata (x axes)



(a) *Fisher Ring* (10 automata) (b) *EHC*

(c) *Nuclear Reactor* (10 automata) (d) *Distributed Controller* (5 automata)

Fig. 4. Run times (sec. on the y axes) repeating the scenario n times (y axes)

of messages of the scenario for an arbitrary number of times. All the experiment were run on a Linux machine equipped with an Intel i7 CPU at 2.93 GHz, setting the timeout and the memory out for a single benchmark to 300 seconds and to 2 GB of RAM. All the results, the test cases and the executable used in the experimental evaluation are available at `http://es.fbk.eu/people/mover/tests/CAV11/`.

The main findings of the experimental evaluation regard the effectiveness of the scenario-based encoding, which outperforms the approaches based on the optimized automata construction. In Figure 2 we compare the run times (in logarithmic scale) for all the tested instances of benchmarks and scenarios of the scenario-driven encoding and the automata approaches. The scenario-driven encoding demonstrates its efficiency outperforming the automata approaches in nearly all the benchmarks, sometimes by orders of magnitude, terminating the execution on benchmarks where the automata approach reaches the time-out.

The plots in Figure 3 show the scalability with respect to the number of the automata in the network for all the benchmarks, except the *EHC*, for MSC of fixed structure. Each plot from Figure 3 shows the run time (in seconds) of the different methods on the y axes and the number of automata on the x axes. A point in one of these plots is the run time of a specific method for a specific number of automata. These plots show that the scenario-based encoding scales much better than the automata-based approaches.

Figure 4 (a)-(e) shows the effect of increasing the size of the scenario, fixing the number of automata in the network. In general, again, the scenario-driven encoding is more efficient and scales better than the automata-based approaches.

The simplification brought by the invariants computed from the abstraction is not always useful, although they rarely are detrimental (see Figure 3 and 4). When the system under analysis has a small discrete state space the invariants do not bring a speed up to the search. The impact of invariants is very strong when the complexity of the automata increases, as in the case of the *Distributed Controller* benchmark. The effect of the invariants on the search is highlighted in Figure 2(d), where we plot the *search* times, excluding the time taken to generate the invariants.

7 Conclusions

In this paper we have addressed the problem of finding traces satisfying MSCs. The problem is highly relevant to verification, in that MSCs are very useful to allow end users to validate both requirements and designs. We investigated the use of a specialized algorithm that uses the segments of the MSC to guide the search, based on the use of a local time semantics. The experiments show that the proposed method significantly outperforms optimized techniques based on automata construction.

In the future, we will extend the language support for MSCs and we apply techniques such as k-induction and abstraction [24] to prove that a network does not satisfy an MSC.

References

1. Ábrahám, E., Becker, B., Klaedtke, F., Steffen, M.: Optimizing bounded model checking for linear hybrid systems. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 396–412. Springer, Heidelberg (2005)

2. Akshay, S., Bollig, B., Gastin, P.: Automata and logics for timed message sequence charts. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 290–302. Springer, Heidelberg (2007)
3. Alur, R., Yannakakis, M.: Model checking of message sequence charts. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, p. 114. Springer, Heidelberg (1999)
4. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying Industrial Hybrid Systems with MathSAT. ENTCS 119(2), 17–32 (2005)
5. Ben-Abdallah, H., Leue, S.: Timing constraints in message sequence chart specifications. In: FORTE, pp. 91–106 (1997)
6. Bengtsson, J.E., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)
7. Bu, L., Cimatti, A., Li, X., Mover, S., Tonetta, S.: Model checking of hybrid systems using shallow synchronization. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 155–169. Springer, Heidelberg (2010)
8. Cimatti, A., Mover, S., Tonetta, S.: Hydi: a language for symbolic hybrid systems with discrete interaction. Technical report, Fondazione Bruno Kessler (2011)
9. Claessen, K., Sörensson, N.: New techniques that improve mace-style finite model finding. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, Springer, Heidelberg (2003)
10. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
11. Damm, W., Toben, T., Westphal, B.: On the expressive power of live sequence charts. Program Analysis and Compilation, 225–246 (2006)
12. Fränzle, M., Herde, C.: Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. ENTCS 133, 119–137 (2005)
13. Fränzle, M., Herde, C.: HySAT: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design 30(3), 179–198 (2007)
14. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3(4-5), 519–550 (2003)
15. Henzinger, T.A., Ho, P.: Hytech: The cornell hybrid technology tool. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 265–293. Springer, Heidelberg (1995)
16. Henzinger, T.A.: The Theory of Hybrid Automata. In: LICS, pp. 278–292. IEEE Computer Society Press, Los Alamitos (1996)
17. Klose, J., Klose, H.: An automata based interpretation of live sequence charts. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, p. 512. Springer, Heidelberg (2001)
18. Ladkin, P.B., Leue, S.: Ladkin and Stefan Leue. On the semantics of message sequence charts. In: FBT, pp. 88–104 (1992)
19. Li, S., Balaguer, S., David, A., Larsen, K.G., Nielsen, B., Pusinskas, S.: Scenario-based verification of real-time systems using uppaal. Formal Methods in System Design, 200–264 (2010)
20. Mauw, S., Reniers, M.A.: High-level message sequence charts. In: SDL Forum, pp. 291–306 (1997)
21. Müller, O., Stauner, T.: Modelling and verification using linear hybrid automata - a case study. Mathematical and Computer Modelling of Dynamical Systems 71 (2000)
22. Pan, M., Bu, L., Li, X.: TASS: Timing analyzer of scenario-based specifications. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 689–695. Springer, Heidelberg (2009)
23. Reniers, D.M.A.: Message Sequence Chart: Syntax and Semantics. PhD thesis (1999)
24. Tonetta, S.: Abstract model checking without computing the abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 89–105. Springer, Heidelberg (2009)
25. Wang, F.: Symbolic parametric safety analysis of linear hybrid systems with BDD-like data structures. IEEE Trans. Soft. Eng. 31(1), 38–51 (2005)

Temporal Property Verification as a Program Analysis Task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2,5,8,24,32]) together with techniques for discovering termination arguments (*e.g.* [3,6,17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential nonrecursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL). Existential reasoning would also be possible, but care is required to ensure that

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 333–348, 2011.

the underlying program analysis tools appropriately use universal abstractions ("may" transitions) as well as existential abstractions ("must" transitions). Finally, our method works best when properties do not involve deep and complex nesting of temporal operators. In order to better support these more complex properties our implementation would need to mix the construction of the program analysis problem with the analysis itself in the spirit of Impact [27], as invariants proved during a lazy unrolling could be used to prune away much of the work. As presented here, our approach instead creates a single encoding up front before performing program analysis.

2 From Temporal Logic to Program Analysis

In this section we introduce a reduction which, when given a transition system M and an ∀CTL temporal logic property φ, generates a program that encodes the search for the proof that φ holds of M. Existing program analysis tools can then be used to reason about the validity of the property. We begin with some definitions and terminology.

2.1 Preliminaries

Transition systems. A transition system $M = (S, R, I)$ is a set of states S, a transition relation $R \subseteq S \times S$, and a set of initial states $I \subseteq S$. A *trace* of a transition system is a sequence of states $(s_0, s_1, ...)$ such that $s_0 \in I$ and $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$. For convenience, we do not allow finite traces. The transition relation must be such that every state has at least one successor state: $\forall s \in S.\ \exists s'.\ R(s, s')$. This is without a loss of generality, as final states can be encoded as states that loop back to themselves.

Ranking functions. For a state space S, a ranking function f is a total map from S to a well ordered set with ordering relation $\prec$. A relation $R \subseteq S \times S$ is *well-founded* if and only if there exists a ranking function f such that $\forall (s, s') \in R.\ f(s') \prec f(s)$. We denote a finite set of ranking functions (or *measures*) as $\mathcal{M}$. Note that the existence of a finite set of ranking functions for a relation R is equivalent to containment of R within a finite union of well-founded relations [30]. That is to say that a set of ranking functions $\{f_1, ..., f_n\}$ can denote the disjunctively well-founded relation $\{(s, s') \mid f_1(s') \prec f_1(s) \vee ... \vee f_n(s') \prec f_n(s)\}$.

Temporal logic. We are concerned with verifying temporal properties that may be written either as trace-based properties in LTL or as state-based properties in the universal fragment of computation tree logic (∀CTL). The encoding we describe in this section is state-based in nature and, as such, is readily suitable to ∀CTL properties. To prove LTL properties we use a recently described iterative symbolic determinization technique [15] with the ∀CTL proving technique described here.

The syntax of a ∀CTL formula is $\varphi ::= \alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AG}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi]$. The standard semantics of ∀CTL are given in Fig. 1. α is an atomic proposition. ∀CTL's temporal operators are state-based in structure. The operator $\mathsf{AG}\varphi$

specifies that φ globally holds in all reachable future states. The operator $\mathsf{AF}\varphi$ specifies that, across all computation sequences from the current state, a state in which φ holds must be reached. Finally, the $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ operator specifies that φ_1 holds in every state where φ_2 does not yet hold.

We use $\mathsf{AF}, \mathsf{AG}, \mathsf{AW}$ as our base operators (as opposed to the more standard U and R), as each corresponds to a distinct form of proof: AF to termination, AG to safety, and AW to sequencing. We omit the next state operator AX. Formulae with U and R can be expressed in ∀CTL. We assume that formulae are written in negation normal form, in which negation only occurs next to atomic propositions (we also assume that the domain of atomic propositions is closed under negation). A formula that is not in negation normal form can be easily normalized. $\mathsf{sub}(\varphi)$ is defined to be the set of all subformulae of φ.

$$\frac{\alpha(s)}{R, s \vDash \alpha} \quad \frac{R, s \vDash \varphi_1 \quad R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \wedge \varphi_2} \quad \frac{R, s \vDash \varphi_1 \ \vee \ R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \vee \varphi_2}$$

$$\frac{\forall (s_0, s_1, ...).\ s_0 = s \Rightarrow \exists i \geq 0.\ R, s_i \vDash \varphi}{R, s \vDash \mathsf{AF}\varphi}$$

$$\frac{\forall (s_0, s_1, ...).\ s_0 = s \Rightarrow \forall i \geq 0.\ R, s_i \vDash \varphi}{R, s \vDash \mathsf{AG}\varphi}$$

$$\frac{\begin{array}{c}\forall (s_0, s_1, ...).\ s = s_0 \Rightarrow (\forall i \geq 0.\ R, s_i \vDash \varphi_1) \vee \\ (\exists j \geq 0.\ R, s_j \vDash \varphi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \varphi_1)\end{array}}{R, s \vDash \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]}$$

Fig. 1. Semantics of ∀CTL: $\vDash$

```
let rec $\mathcal{E}(\langle s, \psi \rangle, \mathcal{M}, R)$ : bool =
  match($\psi$) with
  | $\alpha$ → return $\alpha(s)$
  | $\psi' \wedge \psi''$ →
      if (*) return $\mathcal{E}(\langle s, \psi' \rangle, \mathcal{M}, R)$
      else return $\mathcal{E}(\langle s, \psi'' \rangle, \mathcal{M}, R)$;
  | $\psi' \vee \psi''$ →
      if ($\mathcal{E}(\langle s, \psi' \rangle, \mathcal{M}, R)$) return true;
      else return $\mathcal{E}(\langle s, \psi'' \rangle, \mathcal{M}, R)$;
  | AG$\psi'$ →
       while (true) {
          if ($\neg$ $\mathcal{E}(\langle s, \psi' \rangle, \mathcal{M}, R)$)
            return false;
          if (*) return true;
          $s$ := choose($\{s' \mid R(s, s')\}$);
       }
  | AF$\psi'$ → local dup := false; local $'s$ ;
      while (true) {
         if ($\mathcal{E}(\langle s, \psi' \rangle, \mathcal{M}, R)$) return true;
         if (dup $\wedge$ $\neg(\exists f \in \mathcal{M}.\ f(s) \prec f('s))$)
            return false;
         if ($\neg$ dup $\wedge$ *) { dup := true; $'s$ := $s$; }
         if (*) return true;
         $s$ := choose($\{s' \mid R(s, s')\}$);
      }
  | A[$\psi'$W$\psi''$] →
     while(true) {
        if ($\neg\mathcal{E}(\langle s, \psi' \rangle, \mathcal{M}, R)$)
          return $\mathcal{E}(\langle s, \psi'' \rangle, \mathcal{M}, R)$;
        if (*) return true;
        $s$ := choose($\{s' \mid R(s, s')\}$);
     }
```

Fig. 2. The encoding $\mathcal{E}$ which takes a state s, a property ψ, a finite set of ranking functions $\mathcal{M}$, and a transition relation R, and constructs a recursive program which can be used to prove ψ if $\mathcal{M}$ is sufficient. $\mathsf{choose}()$ nondeterministically selects an element from the set given by its argument. $* \equiv \mathsf{choose}(\{\mathsf{true}, \mathsf{false}\})$

2.2 Encoding

We now show that the problem of ∀CTL verification can be reduced to a program analysis task. Our encoding $\mathcal{E}$ is given in Fig. 2. When given a transition relation system $M = (S, R, I)$ and an ∀CTL property φ, the program $\mathcal{E}$ encodes the search for the proof that φ holds of M. The arguments $(\langle s, \psi\rangle, \mathcal{M}, R)$ passed to $\mathcal{E}$ are a pair consisting of the state s, a φ-subformula ψ of interest, a finite set of ranking functions $\mathcal{M}$ and the transition relation R. Executions of the procedure $\mathcal{E}$ explore the $S \times \mathsf{sub}(\varphi)$ state space from an initial state $s_0 \in I$ in a depth-first manner. At each recursive call, $\mathcal{E}$ is attempting to determine whether ψ holds of s. Rather than explicitly tracking this information, however, $\mathcal{E}$ returns false (recursively) whenever ψ does not hold of s. Consequently, if $\mathcal{E}$ can be proved to never return false, it must be the case that the overall property φ holds of the initial state s (we discuss the termination of $\mathcal{E}$ below). When a program analysis is applied to $\mathcal{E}$ it is implementing what is needed to prove branching-time behaviors of the original transition system (*e.g.* backtracking, eventuality checking, tree counterexamples, abstraction, abstraction-refinement, etc). Formally the relationship between $\mathcal{E}$ and $\vDash$ is: for a transition system $M = (S, R, I)$ and ∀CTL property φ,

$$[\exists \text{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ cannot return } \mathsf{false}] \ \Rightarrow\ \forall s \in I.\ R, s \vDash \varphi$$

where $\mathcal{M}$ is, as described earlier, a finite set of ranking functions. We formally define "cannot return false" by giving $\mathcal{E}$ as a guarded transition system in our technical report [16], but informally it means there is no execution of $\mathcal{E}$ where false is returned.

Completeness (*i.e.* $\Leftarrow$) holds when equality over S can be determined in finite time and the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs). These results can be found in Section 3.

What remains is to understand how $\mathcal{E}$ determines whether a subformula ψ holds of a state s. By passing the state on the stack, we can consider multiple branching scenarios. When a particular ψ is a $\wedge$ or AG subformula, then $\mathcal{E}$ ensures that all possibilities are considered by establishing feasible paths to all of them. When a particular ψ is a $\vee$ or AF subformula, $\mathcal{E}$ enables executions to consider all of the possible cases that might cause ψ to hold of s. As soon as one is found, true is returned. Otherwise, false will be returned if none are found. This is the intuition behind the first invariant maintained by $\mathcal{E}$:

$$INV_1 : \forall s, \psi, \mathcal{M}, R.\ \text{ if } R, s \nvDash \varphi \text{ then } \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ can return } \mathsf{false}$$

Consider this case from the definition of $\mathcal{E}$:

```
| $\psi \vee \psi' \rightarrow$ if ($\mathcal{E}(\langle s, \psi\rangle, \mathcal{M}, R)$) return true;
                 return $\mathcal{E}(\langle s, \psi'\rangle, \mathcal{M}, R)$;
```

Imagine that $\psi \equiv \mathsf{x} \neq 1$, and $\psi' \equiv \mathsf{AG}(\mathsf{x} = 0)$. In this case we want to know that one of the subformulae (*i.e.* $\mathsf{x} \neq 1$ or $\mathsf{AG}(\mathsf{x} = 0)$) holds. A recursive call is made with the current state s to explore whether $\mathsf{x} \neq 1$ as well as a separate recursive call with the same current state s to explore $\mathsf{AG}(\mathsf{x} = 0)$. During a symbolic execution of this program, all executions will be considered in a search for a way to cause the program to fail. If it is possible for both recursive calls to return false (*i.e.* they abide INV_1), then there will be an execution in which the current call can return false (also abiding INV_1). A standard program analysis tool (*e.g.* SLAM [2]) will find this case. By maintaining this invariant recursively, a proof that the outermost level of $\mathcal{E}$ cannot return false implies that the outermost property holds of the original system.

Because we want to consider every state that is reachable from a finite prefix of an infinite path, it must be possible for the recursive calls to return from every state. If it were possible for the checking of a subformula like $\mathsf{AG}(\mathsf{x} = 0)$ to diverge (thus never returning false) then the above code fragment would never return false, and thus the top-level call to $\mathcal{E}$ would never return false. To this end, $\mathcal{E}$ maintains a second invariant:

$$INV_2 : \forall s, \psi, \mathcal{M}, R.\ \ \mathcal{E}(\langle s, \psi\rangle, \mathcal{M}, R) \text{ can return } \mathsf{true}$$

It is this requirement that necessitates the additional nondeterministic "if (*) return true" commands found within each loop in $\mathcal{E}$. One can think of "if (*) return true" as a form of backtracking. In our encoding, a nondeterministic return of true is not declaring that the property holds (we must *always* return true to do that). Instead, a nondeterministic return of true in the encoding means that a program analysis can freely backtrack and switch to other possible scenarios during its search for a proof.

In the AF case, our encoding must allow a program analysis to demonstrate that all paths must eventually reach a state where the subformula holds. While exploring the reachable states in R the encoding may, at any point, nondeterministically decide to capture the current state (setting dup to true and saving s as $'s$). When each subsequent state s is considered, a check is performed that there is some rank function that witnesses the well-foundedness of this particular subset of the transitive closure of the transition system (we will precisely say which subset in Section 3). This is an adaptation of a known technique [17]. However, rather than using assert to check that one of the ranking functions in $\mathcal{M}$ holds, our encoding instead returns false, allowing other possibilities to be considered (if any exist) in outer disjunctive or AF formulae.

Partial evaluation. In practice, if the input transition system is implemented as a program, then we can perform a number of static optimizations from abstract interpretation and partial evaluation that facilitates the application of current program analysis tools. Our procedure PEval implements this mixture. For lack of space, we only briefly describe these transformations. Some additional details about PEval's optimizations are provided in our technical report [16].

PEval uses the following facts: (a) the input ∀CTL formula φ is always finite (b) the structure of $\mathcal{E}$ is unchanged and (c) the program and initial state are fixed. Thus we can partially evaluate $\mathcal{E}$ on φ and the input program and obtain a first-order program for which modern program analysis techniques can be effective. For example, consider the naïve implementation of AG given in Fig. 2 which, in essence, is interpreting the cross product of R together with the following program:

```
while true do
    if $(\neg\ \mathcal{E}((s, \psi'), \mathcal{M}, R))$ return false;
    if $(*)$ return true;
done
```

Since we are considering programs as our input systems, we can build an encoding where the following fragment is instrumented in each line of a procedure based on the original input program:

```
if $(\neg\ \mathcal{E}((s, \psi'), \mathcal{M}, R))$ return false;
if $(*)$ return true;
```

We will see an example of this in Section 4.

Because the program state is passed on the stack, recursive calls to $\mathcal{E}$ will not modify variables in the outer scope, and thus can be treated as `skip` statements when analyzing the iterations of R. Invariants within a given subprocedure can be vital to the pruning, simplification, and partial evaluation required to prepare the output of $\mathcal{E}$ for program analysis.

2.3 Looking for $\mathcal{M}$

Finally, recall that we must ultimately find a finite set of ranking functions $\mathcal{M}$ such that a program analysis can prove for every $s \in I$ that $\mathcal{E}(\langle s, \varphi \rangle, \mathcal{M}, R)$ does not return false. Our top-level procedure adapts a known method [17] in order to iteratively find a sufficient $\mathcal{M}$. See Fig. 3. This procedure first constructs an $\mathcal{E}_\varphi$, which is a version of $\mathcal{E}$ that has been specialized on P and φ. Then, in our implementation, new ranking functions are automatically synthesized by examining counterexamples. A counterexample in $\forall$CTL is tree-like as follows:

```
let prove$(P, \varphi)$ =
  let $\mathcal{E}_\varphi$ = $\mathrm{PE{\small VAL}}(\mathcal{E}, \varphi, P)$ in
  $\mathcal{M} := \emptyset$
  while $(\mathcal{E}_\varphi(\mathcal{M})$ can return false) do
      let $\chi$ be a counterexample in
      if $\exists$ lasso path fragment $\chi'$ from $\chi$ then
        if $\exists$ witness $f$ showing $\chi'$ w.f. then
          $\mathcal{M} := \mathcal{M} \cup \{f\}$
        else
          return $\chi$
      else
          return $\chi$
  done
  return Success
```

Fig. 3. Rank function refinement procedure where the input transition system P is assumed to be a program

$$\begin{aligned} \chi ::= {} & \mathsf{CEX}_\alpha \text{ of } s \mid \mathsf{CEX}_\wedge \text{ of } \chi \mid \mathsf{CEX}_\vee \text{ of } \chi \times \chi \\ & \mid \mathsf{CEX}_{\mathsf{AG}} \text{ of } \pi \times \chi \mid \mathsf{CEX}_{\mathsf{AF}} \text{ of } \pi \times \pi \times \chi \mid \mathsf{CEX}_{\mathsf{W}} \text{ of } \pi \times \chi \times \chi \end{aligned}$$

where π is a trace through the transformed program $\mathcal{E}$. Note that often tools will not report a concrete trace but rather a *path*, *i.e.* a sequence of program counter values corresponding to a class of traces (in rare instances paths may be reported that are spurious). The counterexample structure for an atomic proposition CEX_α is simply a state in which α does not hold. Counterexamples for conjunction and disjunction are as expected. A counterexample to an AG property is a path to a place where there is a counterexample to the sub-property. A counterexample to an AF property is a "lasso"—a stem path to a particular program location, then a cycle which returns to the same program location, and a sub-counterexample along that cycle in which the sub-property does not hold. Finally, an AW counterexample is a path to a place where there is a sub-counterexample to the first property as well as a sub-counterexample to the second property.

In our encoding we obtain these tree-shaped counterexamples effectively for free with program analysis tools like SLAM that report stack-based traces for assertion failures. Information about the stack depth available in the counterexamples allows us to re-construct the tree counterexamples. That is, by walking backward over the stack trace, we can determine the tree-shape of the counterexample. Consider, for example, the case of AF. The counterexample found by the underlying tool will visit commands through the encoding of $\mathcal{E}$, including points where dup is set to true. The commands from the input program can be used to populate an instance of χ.

When a counterexample is reported that contains an instance of $\mathsf{CEX}_{\mathsf{AF}}$ (*i.e.* a "lasso fragment") it is possible that the property still holds, but that we have simply not found a sufficient ranking function to witness the termination of the lasso. In this case our procedure finds the lasso fragments and attempts to enlarge the set of ranking functions $\mathcal{M}$. One source of incompleteness of our implementation comes from our reliance on lassos: some non-terminating programs have only well-founded lassos, meaning that in these cases our refinement algorithm will fail to find useful refinements. The same problem occurs in [17]. In industrial examples these programs rarely occur.

3 Correctness

Theorem 1 (Soundness and completeness). *For a transition system $M = (S, R, I)$ and $\forall$CTL property φ,*

$$[\exists \textit{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \textit{ cannot return } \mathsf{false}] \;\;\Rightarrow\;\; \forall s \in I.\ R, s \vDash \varphi$$

*where $\mathcal{M}$ is a finite set of ranking functions. Completeness (*i.e. *$\Leftarrow$) holds when equality over S can be determined in finite time and the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs).*

Using the Coq theorem prover we have proved the above theorem. Details can be found in the Coq proof script listed in our technical report [16]. In this section we discuss the structure of the proof and state some of the key lemmas.

$$\frac{I \subseteq \{s \mid \alpha(s)\}}{\langle R, I\rangle \vdash \alpha} \qquad \frac{\langle R, I\rangle \vdash \varphi_1 \quad \langle R, I\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \wedge \varphi_2}$$

$$\frac{\exists I_1, I_2.\ I = I_1 \cup I_2 \wedge \langle R, I_1\rangle \vdash \varphi_1 \wedge \langle R, I_2\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \vee \varphi_2}$$

$$\frac{\langle R, \{s' \mid \exists s \in I.\mathsf{reach}\ (s, s')\}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{AG}\varphi}$$

$$\frac{\exists \mathcal{F}.\mathsf{walk}_I^{\mathcal{F}} \text{ is w.f. } \wedge \langle R, \mathcal{F}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{AF}\varphi}$$

$$\frac{\exists \mathcal{F}.\forall (s, s') \in \mathsf{walk}_I^{\mathcal{F}}.\langle R, \{s\}\rangle \vdash \psi \wedge \langle R, \mathcal{F}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{A}[\psi \mathsf{W} \varphi]}$$

$$\frac{}{\mathsf{reach}_0\ (s, s)} \qquad \frac{\mathsf{reach}_n\ (s, s') \wedge R(s', s'')}{\mathsf{reach}_{n+1}\ (s, s'')}$$

$$\frac{R(s, s') \wedge s \notin \mathcal{F} \wedge s \in I}{\mathsf{walk}_I^{\mathcal{F}}\ (s, s')}$$

$$\frac{R(s', s'') \wedge s' \notin \mathcal{F} \wedge \mathsf{walk}_I^{\mathcal{F}}\ (s, s')}{\mathsf{walk}_I^{\mathcal{F}}\ (s', s'')}$$

We write $\mathsf{reach}\ (s, s')$
to mean $\exists n \geq 0.\mathsf{reach}_n\ (s, s')$.

Fig. 4. Relational formulation of ∀CTL: ⊢

For convenience, in the proof we introduce an alternative relational formulation of ∀CTL, ⊢. This formulation more closely matches our definition of $\mathcal{E}$ in that it is given over sets of states, AG is defined in terms of reachability, and AF is defined in terms of well-foundedness. In effect the encoding $\mathcal{E}$ is characterizing these sets with nondeterminism and by returning true or false. Our proof starts by showing that ⊢ is equivalent to ⊨ and then showing that

$$[\exists \text{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ cannot return } \mathsf{false}] \quad \Rightarrow \quad \langle R, I\rangle \vdash \varphi$$

from which point soundness directly follows.

Relational formulation of ∀CTL semantics. Our relational formulation of ∀CTL is displayed in Fig. 4. Unlike the standard formulation, ours is more amenable to reasoning about infinite-state systems because proof trees are based on *partitioning* the state space rather than *enumerating* the state space. We use the notation $\langle R, I\rangle \vdash \varphi$ to denote that a property φ is valid for a transition system. This entailment relation is then defined inductively.

An atomic proposition α involves a simple check to see if I is contained within the set of states in which α holds. The conjunction rule requires that both φ_1 and φ_2 hold of all states in I and the disjunction rule splits the states into two sets, one in which φ_1 holds and one in which φ_2 holds. The semantics of the property AGφ says that for every reachable state s', that s' entails φ.

Frontiers. The property AFφ depends on the existence of a set of states which we will call a *frontier* $\mathcal{F}$. Intuitively, the frontier $\mathcal{F}$ of a set of initial states I, is a set of states through which every trace originating at a state in I must pass.

We use frontiers in our formulation of AFφ to characterize the places where φ holds, requiring that all paths from I eventually reach a frontier. We formalize this idea by defining the inductive relation $\mathsf{walk}_I^{\mathcal{F}}$ given on the right in Fig. 4.

$\mathsf{walk}_I^{\mathcal{F}}$ is a subset of R that includes every possible transition along every trace from I up to $\mathcal{F}$. In our characterization of AF we require that $\mathsf{walk}_I^{\mathcal{F}}$ be well-founded. In this way, we recast the ∀CTL semantics of AF in terms of the well-foundedness of a relation, rather than the existence of an i-th state in every trace. This formulation allows us to more efficiently prove AF properties because we can discover well-founded relations that are over-approximations of $\mathsf{walk}_I^{\mathcal{F}}$ rather than searching for per-trace ranking functions. The final rule in the left of Fig. 4 is for the AW operator, which also uses a frontier and the relation $\mathsf{walk}_I^{\mathcal{F}}$ representing the arcs along the way to the frontier $\mathcal{F}$. To prove $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, all states along the path to the frontier must satisfy φ_1 and states at the frontier—*should one ever get there*—all must satisfy φ_2.

The following lemma shows that if a property holds in our relational semantics, then it also holds in the standard semantics of ∀CTL.

Lemma 1. *For every* $\varphi, I, R, \quad \langle R, I\rangle \vdash \varphi \quad \Longleftrightarrow \quad \forall s \in I.\ R, s \vDash \varphi.$

In our technical report [16] we formalize $\mathcal{E}$ as a guarded transition system. Since φ is finite, we can partially evaluate $\mathcal{E}$ with respect to φ, and represent $\mathcal{E}$ as a finite graph. The stack and return values are encoded in the configurations of the graph. Executions and the notion "cannot return false" are then defined in the natural way.

Lemma 2. *For a transition system* $M = (S, R, I)$ *and* ∀*CTL property* φ*,*

$$[\exists \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \textit{ cannot return } \mathsf{false}] \quad \Rightarrow \quad \langle R, I\rangle \vdash \varphi.$$

*Completeness (*i.e. $\Leftarrow$*) holds when equality over* S *can be determined in finite time and each of the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs).*

Proof. *By induction on* φ.

From these lemmas we can prove Theorem 1.

4 Example

Consider the example in Fig. 5. After applying $\mathcal{E}$ and PEval we obtain the program given in Fig. 6. The intermediate output without partial evaluation is given in the technical report [16]. The encoding has been partially evaluated with respect to φ, and with respect to the program counter. For every $\psi \in \mathsf{sub}(\varphi)$ and pc valuation, there is a corresponding method $\mathtt{E}_{\text{“}\psi\text{”}}\_\mathsf{pc}$. Since we are working with a linear arithmetic program where ranking functions can be given as linear inequalities, integer $<$ is a sufficient ordering for $\prec$. The `main` procedure in the encoding initializes the program state (*i.e.* `x,n`) and then asserts that $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}\_0$ cannot return false.

```
1  while(*) {
2     x := 1;
3     n := *;
4     while(n>0) {
5        n := n - 1;
6     }
7     x := 0;
8  }
9  while(1) {}
```

Fig. 5. Example where $\varphi = \mathsf{AG}[(\mathsf{x} = 1) \Rightarrow \mathsf{AF}(\mathsf{x} = 0)]$ and initially $\mathtt{x} = 0$

An execution of this program consists of a cascade of calls down the hierarchy of sub-procedures. Each procedure for a subformula maintains invariants INV_1 and INV_2. This encoding allows us to ask questions of the form "starting now (*i.e.* from this state) does there exist an execution that violates my property," and answer them using standard analysis tools.

For example, procedure $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$ corresponds to the property $\mathsf{AG}((x \neq 1) \vee \mathsf{AF}(x = 0))$ and returns false if there is a reachable state where $((x \neq 1) \vee \mathsf{AF}(x = 0))$ does not hold. It accomplishes this by calling $\mathtt{E}_{\text{“}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$ on each line and passing the current state.

If $((x \neq 1) \vee \mathsf{AF}(x = 0))$ does not hold from the current state, then there will be a way for $\mathtt{E}_{\text{“}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$ to return false, in which case $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$ immediately returns false (leading to an assertion failure in `main`). The procedures for disjunction ($\mathtt{E}_{\text{“}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$) and atomic propositions ($\mathtt{E}_{\text{“}x\neq 1\text{”}}$ and $\mathtt{E}_{\text{“}x=0\text{”}}$) are straight-forward following Fig. 2, and also maintain INV_1.

The procedure $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$ is, in some sense, the complement of AG. It is designed to return true whenever there is a path to a state where $x = 0$ holds, and will return false if there is an infinite execution that never reaches such a state. This is accomplished by checking at each state (*i.e.* on each line of the program) whether $\mathtt{E}_{\text{“}x=0\text{”}}$ returns true, and returning false if a location is reached multiple times and there is no ranking function in $\mathcal{M}$ that is decreasing.

With the transformation in hand, we can now apply the algorithm in Fig. 3. What remains is the task of finding a finite $\mathcal{M}$ such that in $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$ the check that $\exists f \in \mathcal{M}.\ f(\mathtt{x}_5, \mathtt{n}_5) > f(\mathtt{x},\mathtt{n})$ always holds. Initially we let $\mathcal{M} \equiv \emptyset$. Running a refinement-based safety prover will yield a counterexample pertaining to line $\mathsf{lab\_5}$ of $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$, where we denote a state as $\begin{bmatrix} \mathtt{x} \\ \mathtt{n} \\ \mathsf{pc} \end{bmatrix}$ and we denote transition relations as $\left[\left[\begin{matrix} \mathtt{'x=x} \\ \mathtt{'n=n} \\ {}'\mathsf{pc}{=}\mathsf{pc}. \end{matrix}\right]\right]$:

$$\begin{aligned}
&(\mathsf{CEX}_{\mathsf{AG}}\ \left(\begin{bmatrix} 0 \\ \mathtt{n} \\ 1 \end{bmatrix} :: \begin{bmatrix} 1 \\ \mathtt{n} \\ 2 \end{bmatrix} :: \begin{bmatrix} 1 \\ \mathtt{n} \\ 3 \end{bmatrix} :: \begin{bmatrix} 1 \\ \mathtt{n} \\ 4 \end{bmatrix} :: \begin{bmatrix} 1 \\ \mathtt{n} \\ 5 \end{bmatrix}\right), \\
&\qquad (\mathsf{CEX}_{\vee}\ \ (\mathsf{CEX}_{\alpha}\ \begin{bmatrix} 1 \\ \mathtt{n} \\ 5 \end{bmatrix}) \\
&\qquad\qquad (\mathsf{CEX}_{\mathsf{AF}}\ \begin{bmatrix} 1 \\ \mathtt{n} \\ 5 \end{bmatrix},\ \left[\left[\begin{matrix} \mathtt{x}_5\mathtt{=x} \\ \mathtt{n}_5\mathtt{=n+1} \\ \mathsf{pc}_5{=}\mathsf{pc} \end{matrix}\right]\right],\ (\mathsf{CEX}_{\alpha}\ \begin{bmatrix} 1 \\ \mathtt{n} \\ 5 \end{bmatrix}))))
\end{aligned}$$

In our implementation we then use a rank function synthesis tool on this counterexample (as described by Cook *et al.* [17]), find that ranking can be done on `n`, and obtain a new $\mathcal{M} \equiv \{\lambda s.\ s(\mathtt{n})\}$. With this new $\mathcal{M}$ in place, $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee \mathsf{AF}(x=0))\text{”}}$ *always* returns true, and consequently, by Theorem 1, φ holds of the original program.

```
void main {
   bool x; nat n;
   x := 0; n := *;
   assert(E“AG((x≠1)∨AF(x=0))”_0(x,n) ≠ false);
}

bool E“AG((x≠1)∨AF(x=0))”_0(bool x, nat n) {
  while(*) {
     x := 1;
     if (¬ E“(x≠1)∨AF(x=0)”_3(x,n))
     { return false; }
     if (*) return true;
     n := *;
     while(n>0) {
        if (*) return true;
        n--;
     }
     x := 0;
  }
  while(1) { if (*) return true; }
}

bool E“(x≠1)∨AF(x=0)”_3(bool x, nat n) {
  if (x ≠ 1) return true;
  return E“AF(x=0)”_3(x,n);
}
```

```
bool E“AF(x=0)”_3(bool x, nat n) {
  dup2 := dup5 := dup9 := false;
  goto lab_3;
  while(*) {
      if(x==0) return true;
      if(dup2 && ∄f ∈ M.f(x2,n2) > f(x,n))
      { return false; }
      if(¬dup2∧*){dup2:=1;x2:=x;n2:= n;}
      if(*) return true;
      x := 1;
    lab_3:
      if (x==0) return true;
      n := *;
      while(n>0) {
      lab_5:
        if(x==0) return true;
        if(dup5 && ∄f ∈ M.f(x5,n5) > f(x,n))
        { return false; }
        if(¬dup5∧*){dup5:=1;x5:=x;n5:= n;}
        if(*) return true;
        n--;
      }
      x := 0;
      if (x==0) return true;
  }
  while(1) {
     if(x==0) return true;
     if(dup9 && ∄f ∈ M.f(x9,n9) > f(x,n))
     { return false; }
     if(¬dup9∧*){dup9:=1;x9:=x;n9:= n;}
     if(*) return true;
  }
}
```

Fig. 6. The encoding $\mathcal{E}$ of property $\mathsf{AG}[(x = 1) \Rightarrow \mathsf{AF}(x = 0)]$ and the program given in Fig. 5 after PEval has been applied

5 Related work

There is a relationship between temporal logic verification and the problem of finding winning strategies in games or game-like structures such as alternating automata [4,25,34]. These previous results do not directly apply because they are geared toward finite state spaces. However, the technique presented in this paper can be viewed as a generalization of prior work to games over infinite state spaces. We explore this generalization in our technical report [16]. Specifically, we first show that the existence of solutions to infinite-state games (such as those used to represent the ∀CTL model checking problem) is equivalent to the existence of a solution to a mix of safety and liveness games, when those games have a certain structure. We then show that our encoding described here can be generalized to games that meet this constraint.

Other previous tools and techniques are known for proving temporal properties of finite-state systems (*e.g.* [7,11,25]) or classes of infinite-state systems with specific structure (*e.g.* pushdown systems [36,37] or parameterized systems [19]). Our proposal works for arbitrary transition systems, including programs.

A previous tool proves only trace-based (*i.e.* linear-time) properties of programs [14] using an adaptation of the traditional automata-theoretic approach [35]. In contrast, our reduction to program analysis promotes a state-based (*e.g.* branching-time) approach. Trace-based properties can be proved with our tool using a recently described iterative symbolic determinization technique [15]. In most cases our new approach is faster for LTL verification than [14] by several orders of magnitude.

When applying traditional bottom-up based methods for state-based logics (*e.g.* [12,18,20]) to infinite-state transition systems, one important challenge is to track reachability when considering relevant subformulae from the property. In contrast to the standard method of directly tracking the valuations of subformulae in the property with additional variables, we instead use recursion to encode the checking of subformulae as a program analysis problem. As an interprocedural analysis computes procedure summaries it is in effect symbolically tracking the valuations of these subformulae depending on the context of the encoded system's state. Thus, in contrast to bottom-up techniques, ours only considers reachable states (via the underlying program analysis).

Chaki *et al.* [9] attempt to address the same problem of subformulae and reachability for infinite-state transition systems by first computing a finite abstraction of the system *a priori* that is never refined again. Then standard finite-state techniques are applied. In our approach we reverse the order: rather than applying abstraction first, we let the underlying program analysis tools perform abstraction after we have encoded the search for a proof as a new program. The approach due to Schmidt and Steffen [33] is similar.

The tool YASM [23] takes an alternative approach: it implements a refinement mechanism that examines paths which represent abstractions of tree counterexamples (using multi-valued logic). This abstraction loses information that limits the properties that YASM can prove (*e.g.* the tool will usually fail to prove $\mathsf{AFAG}p$). With our encoding the underlying tools are performing abstraction-refinement over tree counterexamples. Moreover, YASM is primarily designed to work for unnested existential properties [22] (*e.g.* $\mathsf{EF}p$ or $\mathsf{EG}p$), whereas our focus is on precise support for arbitrary (possibly nested) universal properties.

Our encoding shares some similarities with the finite-state model checking procedure CEX from Figure 6 in Clarke *et al.* [13]. The difference is that a symbolic model checking tool is used as a sub-procedure within CEX, making CEX a recursively defined model checking procedure. The finiteness of the state-space is crucial to CEX, as in the infinite-state case it would be difficult to find a finite partitioning *a priori* from which to make a finite number of model checking calls when treating temporal operators such as AG and AF.

6 Experiments

In this section we report on experiments with a prototype tool that implements $\mathcal{E}$ from Fig. 2 as well as the refinement procedure from Fig. 3. In our tool we have implemented $\mathcal{E}$ as a source-to-source translation using the CIL compiler infrastructure. We use SLAM [2] as our implementation of the safety prover, and RANKFINDER [29] as the rank function synthesis tool.

We have drawn out a set of both ∀CTL and LTL liveness property challenge problems from industrial code bases. Examples were taken from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. In order to make these examples self-contained we have, by hand, abstracted away the unnecessary functions and struct definitions. We also include a few toy examples, as well as the example from Fig. 8 in [14]. Sources of examples can be found in our technical report [16]. Heap commands from the original sources have been abstracted away using the approach due to Magill *et al.* [26]. This abstraction introduces new arithmetic variables that track the sizes of recursive predicates found as a byproduct of a successful memory safety analysis using an abstract domain based on separation logic [28]. Support for variables that range over the natural numbers is crucial for this abstraction.

As previous mentioned in Section 5, there are several available tools for verifying state-based properties of general purpose (infinite-state) programs. Neither the authors of this paper, nor the developer of YASM [23] were able to apply YASM to the challenge problems in a meaningful way, due to bugs in the tool. Note that we expect YASM would have failed in many cases [22], as it is primarily designed to work for unnested existential properties (*e.g.* $\mathsf{EG}p$ or $\mathsf{EF}p$). We have also implemented the approach due to Chaki *et al.* [9]. The difficulty with applying this approach to the challenge problems is that the programs must first be abstracted to finite-state before branching-time proof methods are applied. Because the challenge problems focus on liveness, we have used transition predicate abstraction [31] as the abstraction method. However, because abstraction must happen first, predicates must be chosen ahead of time either by hand or using heuristics. In practice we found that our heuristics for choosing an abstraction *a priori* could not be easily tuned to lead to useful results.

Because the examples are infinite-state systems, popular CTL-proving tools such as Cadence SMV [1] or NUSMV [10] are not directly applicable. When applied to finite instantiations of the programs these tools run out of memory.

The tool described in Cook *et al.* [14] can be used to prove LTL properties if used in combination with an LTL to Büchi automata conversion tool (*e.g.* [21]). To compare our approach to this tool we have used two sets of experiments: Fig. 7 displays the results on challenge problems in ∀CTL verification; Fig. 8 contains results on LTL verification. Experiments were run using Windows Vista and an Intel 2.66GHz processor.

In both figures, the code example is given in the first column, and a note as to whether it contains a bug. We also give a count of the lines of code and the shape of the temporal property where p and q are atomic propositions specific to the program. For both the tools we report the total time (in seconds) and the result for each of the benchmarks. A ✓ indicates that a tool proved the property, and χ is used to denote cases where bugs were found (and a counterexample returned). In the case that a tool exceeded the timeout threshold of 4 hours, “>14400.00” is used to represent the time, and the result is listed as “**???**”.

			Prev. tool [14]		Our tool (Sec. 2)	
Program	**LOC**	**Property**	**Time**	**Result**	**Time**	**Result**
Acq/rel	14	$\mathsf{AG}(a \Rightarrow \mathsf{AF}b)$	103.48	✓	14.18	✓
Ex from Fig. 8 of [14]	34	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	209.64	✓	27.94	✓
Toy linear arith. 1	13	$p \Rightarrow \mathsf{AF}q$	126.86	✓	34.51	✓
Toy linear arith. 2	13	$p \Rightarrow \mathsf{AF}q$	>14400.00	???	6.74	✓
PostgreSQL smsrv	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	>14400.00	???	9.56	✓
PostgreSQL smsrv+bug	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	87.31	χ	47.16	χ
PostgreSQL pgarch	61	$\mathsf{AFAG}p$	31.50	✓	15.20	✓
Apache progress	314	$\mathsf{AG}(p \Rightarrow (\mathsf{AF} \vee \mathsf{AF}))$	685.34	✓	684.24	✓
Windows OS 1	180	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	901.81	✓	539.00	✓
Windows OS 4	327	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	???	1,114.18	✓
Windows OS 4	327	$(\mathsf{AF}a) \vee (\mathsf{AF}b)$	1,223.96	✓	100.68	✓
Windows OS 5	648	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	???	>14400.00	???
Windows OS 7	13	$\mathsf{AGAF}p$	>14400.00	???	55.77	✓

Fig. 7. Comparison between our tool and Cook *et al.* [14] on ∀CTL verification benchmarks. All of the above ∀CTL properties have equivalent corresponding LTL properties so they are suitable for direct comparison with the LTL tool [14].

			Prev. tool [14]		Our tool (Sec. 2)		
Program	**LOC**	**Property**	**Time**	**Result**	**Time**	**#**	**Result**
Ex. from [15]	5	$\mathsf{FG}p$	2.32	✓	1.98	2	✓
PostgreSQL dropbuf	152	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	53.99	✓	27.54	3	✓
Apache `accept` liveness	314	$\mathsf{G}p \Rightarrow \mathsf{GF}q$	>14400.00	???	197.41	3	✓
Windows OS 2	158	$\mathsf{FG}p$	16.47	✓	52.10	4	✓
Windows OS 2+bug	158	$\mathsf{FG}p$	26.15	χ	30.37	1	χ
Windows OS 3	14	$\mathsf{FG}p$	4.21	✓	15.75	2	✓
Windows OS 6	13	$\mathsf{FG}p$	149.41	✓	59.56	1	✓
Windows OS 6+bug	13	$\mathsf{FG}p$	6.06	χ	22.12	1	χ
Windows OS 8	181	$\mathsf{FG}p$	>14400.00	???	5.24	1	✓

Fig. 8. Comparison between our tool and Cook *et al.* [14] on LTL benchmarks. For our tool, we use a recently described iterative symbolic determinization strategy [15] to prove LTL properties by using Fig. 3 as the underlying ∀CTL proof technique. The number of iterations is reported in the # column.

When comparing approaches on ∀CTL properties (Fig. 7) we have chosen properties that are equivalent in ∀CTL and LTL and then directly compared our procedure from Fig. 3 to the tool in Cook *et al.* [14]. When comparing approaches on LTL verification problems (Fig. 8) we have used an iterative symbolic determinization strategy [15] which calls our procedure in Fig. 3 on successively refined ∀CTL verification problems. The number of such iterations is given as column "#." in Fig. 8. For example, in the case of benchmark Windows OS 3, our procedure was called twice while attempting to prove a property of the form FGp.

Our technique was able to prove or disprove all but one example, usually in a fraction of a minute. The competing tool fails on over 25% of the benchmarks.

7 Conclusions

We have introduced a novel temporal reasoning technique for (potentially infinite-state) transition systems, with an implementation designed for systems described as programs. Our approach shifts the task of temporal reasoning to a program analysis problem. When an analysis is performed on the output of our encoding, it is effectively reasoning about the temporal and possibly branching behaviors of the original system. Consequently, we can use the wide variety of efficient program analysis tools to prove properties of programs. We have demonstrated the practical viability of the approach using industrial code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Acknowledgments. We would like to thank Josh Berdine, Michael Greenberg, Daniel Kroening, Axel Legay, Rupak Majumdar, Peter O'Hearn, Joel Ouaknine, Nir Piterman, Andreas Podelski, Noam Rinetzky, and Hongseok Yang for valuable discussions regarding this work. We also thank the Gates Cambridge Trust for funding Eric Koskinen's Ph.D. degree program.

References

1. Cadence SMV, `http://www.kenmcmil.com/smv.html`
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, pp. 73–85 (2006)
3. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: POPL, pp. 211–224 (2007)
4. Bernholtz, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking (extended abstract). In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 142–155. Springer, Heidelberg (1994)
5. 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: PLDI, pp. 196–207 (2003)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. Automata, Languages and Programming, 1349–1361 (2005)
7. Burch, J., Clarke, E., et al.: Symbolic model checking: 10^{20} states and beyond. Information and computation 98(2), 142–170 (1992)
8. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
9. Chaki, S., Clarke, E. M., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., and Veith, H. State/event software verification for branching-time specifications. In *IFM* (2005), pp. 53–69.
10. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An openSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 359. Springer, Heidelberg (2002)
11. Clarke, E., Emerson, E.,, S.: Automatic verification of finite-state concurrent systems using temporal logic specifications. TOPLAS 8(2), 263 (1986)
12. Clarke, E., Grumberg, O., Peled, D.: Model checking (1999)
13. Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29 (2002)

14. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL, pp. 265–276 (2007)
15. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: POPL, pp. 399–410 (2011)
16. Cook, B., Koskinen, E., Vardi, M.: Branching-time reasoning for programs. Tech. Rep. UCAM-CL-TR-788, University of Cambridge, Computer Laboratory (January 2011), `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-788.html`
17. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
18. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
19. Emerson, E., Namjoshi, K.: Automatic verification of parameterized synchronous systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, Springer, Heidelberg (1996)
20. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Program specialization for verifying infinite state systems: An experimental evaluation. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 164–183. Springer, Heidelberg (2011)
21. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 53. Springer, Heidelberg (2001)
22. Gurfinkel, A.: Personal communication (2010)
23. 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)
24. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 526. Springer, Heidelberg (2002)
25. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM 47(2), 312–360 (2000)
26. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
27. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
28. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
29. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
30. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
31. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL (2005)
32. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61 (1995)
33. Schmidt, D.A., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
34. Stirling, C.: Games and modal mu-calculus. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 298–312. Springer, Heidelberg (1996)
35. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop, pp. 238–266 (1995)
36. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
37. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: FST TCS, pp. 127–138 (2000)

Time for Statistical Model Checking of Real-Time Systems*

Alexandre David[1], Kim G. Larsen[1], Axel Legay[2], Marius Mikučionis[1], and Zheng Wang[3]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France
[3] Shanghai Key Laboratory of Trustworthy Computing,
Software Engineering Institute, East China Normal University, China

Abstract. We propose the first tool for solving complex (some undecidable) problems of timed systems by using Statistical Model Checking (SMC). The tool monitors several runs of the system, and then relies on statistical algorithms to get an estimate of the correctness of the entire design. Contrary to other existing toolsets, ours relies on i) a natural stochastic semantics for networks of timed systems, ii) an engine capable to solve problems that are beyond the scope of classical model checkers, and iii) a friendly user interface.

1 Context

Timed model checking (TMC) is a technique used to prove the absence of bugs in systems whose behaviors depend on real or discrete time constraints. The approach has been implemented in several tools [4,2,5] capable of handling case studies of industrial size. Unfortunately, many applications are still out of scope of TMC. This is due to the complexity of the timed behaviors, which can even make the problem undecidable.

In a recent work [8], we presented Constant Slope Timed Automata (CSTA), that are timed systems in that clocks may have different rates (even potentially negative) in different locations. Such automata are as expressive as linear hybrid automata or priced timed automata, but the addition of features such as input and output modalities allows us to specify complex problems in an elegant manner. Unfortunately most of such problems are either undecidable or too complex to be solved with classical model checking approaches. In [8], we proposed to estimate undecidable problems by using Statistical Model Checking (SMC) [13,9]. SMC consists of monitoring some runs of the system and then uses a statistical algorithm to obtain an estimate for the system. Such simulation-based techniques were applied in other contexts where they outperformed classical model checking techniques with an order of magnitude [13,14,1].

* Work partially supported by VKR Centre of Excellence – MT-LAB, an "Action de Recherche Collaborative" ARC (TP)I, the Foundation for Abroad Studies from East China Normal University, and NFSC No. 90818024.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 349–355, 2011.

To apply SMC on CSTA, we had to define a stochastic semantics on their behaviors. This was done in a very natural manner by adding distributions on the delay before a transition is taken. Those distributions are uniform if the delays are bounded, and exponential otherwise. The semantics then establishes a race between the components and selects the smallest delay. One of its major advantages is that the composition of several timed systems remains a pure stochastic system, not a Markov Decision Process. The latter is needed to apply SMC[1].

In this paper, we report on an implementation of our work within the Uppaal toolset [2]. One of the major differences with classical Uppaal is the introduction of a new user interface that allows to specify CSTAs with respect to a stochastic semantics — such semantics is naturally needed to apply SMC. Another contribution is the implementation of several versions of the sequential hypothesis testing algorithm of Wald [12]. Contrary to other implementations of SMC [13,10,7], we also consider those tests that can compare two probabilities without computing them. Finally, contrary to other SMC-based tools, our tool comes with a wide range of functionalities that allow the user to visualize the results in the form of, e.g., probability distributions, evolution of the number of runs with time bounds, or computation of expected values.

Related work. Related work includes the very rich framework of stochastic timed systems of MoDeST [3]. Here, however, general hybrid variables are not considered and parallel composition do not yield fully stochastic models. For the notion of probabilistic hybrid systems considered in [11] the choice of time is resolved non-deterministically rather than stochastically as in our case and as required by SMC.

2 The Toolset

User Interface. Our extension supports the rich modeling constructs of Uppaal with additions specific to CTSA. We add a rational expression attached to locations to define the exponential rates for choosing (unbounded) delays stochastically. We add branching edges and associated weights for the probabilistic extension as shown in Fig. 1. We also generalize rates on clocks to be expressions that take value over integers (even negative) compared to just 0 and 1 for Uppaal.

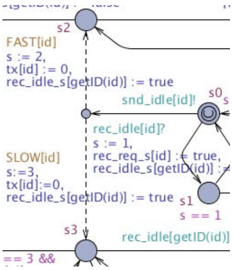


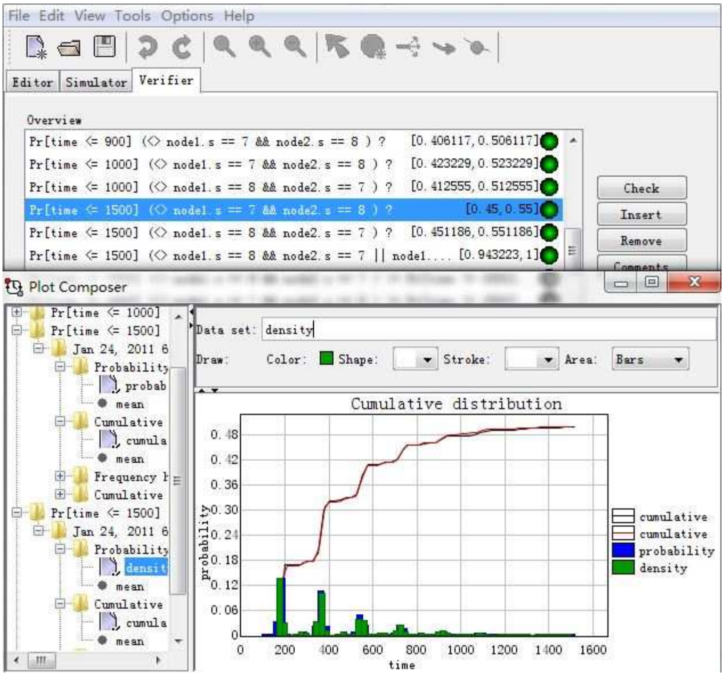
Fig. 1. Branching edges (from firewire case-study)

The verifier shows the estimated intervals of probabilities and provides a plot composer to visualize and compare different results. Figure 2 shows a screenshot with the verifier (above) and the plot composer (below). The verifier provides additional results in a form of plotted data which are accessible

[1] One could try to use classical heuristics for removing non determinism, but they are generally not easily applicable to timed systems.

via a popup-menu by right-clicking over the property. Any plot can be exported to a number of graphical formats and data saved in a textual format.

In addition, a custom plot can be created in plot composer accessible via Tools menu. On the left side of the plot composer, the data sets are grouped and displayed in a tree structure. Any data set can be selected for the composite plot by double clicking on its node and the same way de-selected. The details of the plot can be customized on the right side above the plot.



```
Pr[time <= 900] (<> node1.s == 7 && node2.s == 8 ) ?      [0.406117,0.506117]
Pr[time <= 1000] (<> node1.s == 7 && node2.s == 8 ) ?     [0.423229,0.523229]
Pr[time <= 1000] (<> node1.s == 8 && node2.s == 7 ) ?     [0.412555,0.512555]
Pr[time <= 1500] (<> node1.s == 7 && node2.s == 8 ) ?             [0.45,0.55]
Pr[time <= 1500] (<> node1.s == 8 && node2.s == 7 ) ?     [0.451186,0.551186]
Pr[time <= 1500] (<> node1.s == 8 && node2.s == 7 || node1....  [0.943223,1]
```

Data set: density

Draw: Color: 🟩 Shape: Stroke: Area: Bars

Fig. 2. The verifier shows the results of the different probabilistic queries and the plot composer shows probability distributions

Simulation-based Engine. Any SMC implementation is divided into three parts. First, one needs an algorithm capable of generating random runs, in our case according to our stochastic semantics. Second, those runs have to be monitored with respect to some property. Finally one needs a statistic algorithm to get a general confidence on the results. We implement a new engine to generate such runs that works on states with discrete clock valuations, which makes the computations cheaper compared to equivalent symbolic operations. The runs are bounded by either time, cost, or a number of discrete steps. The engine monitors the generated run with respect to a set of properties that are being checked. Currently the tool is capable of monitoring some properties written in cost-constrained temporal logic ("can I reach a from b with a cost less than
]]>

5?”), but in the future the monitoring procedure could be generalized. Runs are stopped when such properties hold. To do this, the algorithm is able to compute the relative upper bound of an invariant from a given state or compute the delay needed to satisfy a guard or more generally predicates in our properties.

Properties are evaluated on bounded runs by `time`. In fact `time` may be replaced by `steps` or by a clock, or by a cost constraint. The bound is a constant value. The expressions `expr` are state predicates. Our tool can answer of the three following questions:

- A qualititive check: `Pr[time<=bound](<> expr) >= p`.
- A quantitative check: `Pr[time<=bound](<> expr) ?`.
- A comparison check `Pr[time1<=bound1](<> expr1) >= Pr[time2<=bound2](<> expr2)`.

The first formula is to check if the probability of satisfying some property is at least `p`. The second one estimates this probability within an interval. The last one compares two probabilities without evaluating them and gives the result for all bounds up to `bound1` if both bounds are the same. All these checks rely on some statistic algorithm to estimate the correctness by observing runs of the system. The qualitative check is an extension of the sequential hypothesis testing of Wald, the quantitative checks estimates the probability with a Monte-Carlo based approach, finally the comparison check is an extension of sequential hypothesis testing that allows to compare probabilities without computing them. The algorithms are precise up to a certain value that can be chosen by the user.

3 Case Studies

We present two case-studies to highlight the features of our tool. More can be found in *http://www.cs.aau.dk/˜adavid/smc/*, where we handle jobshop scheduling problem and show that our tool performs better than Prism.

3.1 Firewire Protocol

We consider the IEEE 1394 High Performance Serial Bus (“FireWire” for short) that is used to transport video and audio signals on a network of multimedia devices. The protocol has two modes, one *fast* and one *slow* mode for the nodes. The model defines weights to enter these modes as shown in Fig. 1.

This is a leader election protocol that we model with two nodes. We compute the probability for node 1 to become the root (or leader) within different time bounds. We use variable s to denote the state of a node. At initialization, every node is in the *contention* state $s = 0$. After a sequence of steps, a node will enter

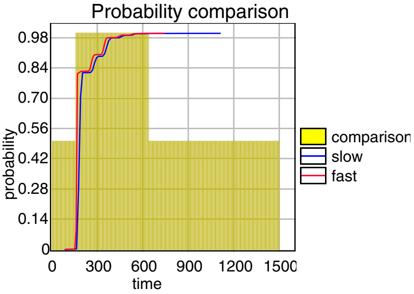


Fig. 3. Probability Comparison

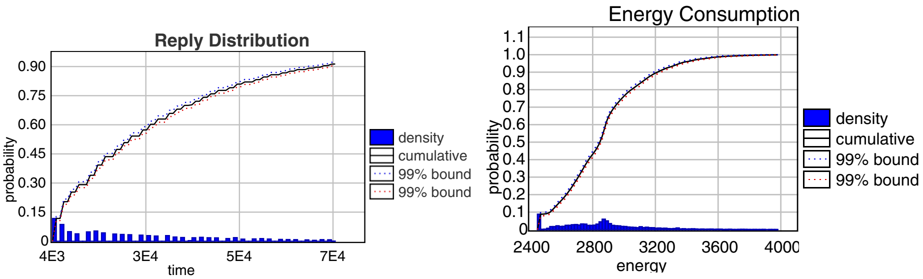
the *root* state $s = 7$ and the other node will enter the *child* state $s = 8$. The query formula is `Pr[time <= 1500] (<> node1.s==7 && node2.s==8) ?`. We are also interested in checking whether there is a difference between the probability that a *fast* node becomes the root or the child and the probability that a *slow* node does. We use a probability comparison property for this purpose.

The results in the plot composer of Fig. 2 show that the probability to elect node 1 as the root increases with the time bound. This probability density diagram shows that in most cases, node 1 may become the root after around 200ms. Uppaal confirms that the fastest possible to reach this state is 164ms. The result in Fig. 3 shows that at the beginning the probabilities are indistinguishable, then the *fast* node has higher probability to become a *root* or a *child*, and at the end the probabilities become very close again.

3.2 Bluetooth Protocol

Bluetooth is a wireless telecommunication protocol using frequency-hopping to cope with interference between the devices in the network. A device can be in a scan state where it replies to a request after two time slots (a slot is 0.3125 ms) and goes to a reply state where it waits for a random amount of time before coming back to the scan state. When a device stays in the scan state, it can also enter the sleeping state (2012 time slots) to save energy. We model energy consumption with a clock (called energy) for which we change the rate depending on these states.
We check the following properties:

- We evaluate the probability of replying within 70000 time units:
  `Pr[time<=70000](<> receiver1.Reply) ?`
  The result is between $[0.866977, 0.966977]$.
- We evaluate the probability of letting time pass 70000 time units with a limited energy budget:
  `Pr[energy<=4000] (<> time>=70000) ?`
  The result is between $[0.949153, 1]$.



(a) (b)

Fig. 4. Probability to reply and energy consumption with 99% confidence

In both cases the tool is able to compute a distribution of the probability over the bound given in argument. Fig. 4(a) shows the cumulative probability of successfully communicating with time. Fig. 4(b) shows that it costs at least $2,400$ energy units. The plot shows how bluetooth consumes energy.

4 Conclusion

We presented an extension of Uppaal for CSTA. Our tool handles problems that are out of scope of existing tools for SMC and timed stochastic systems. We are currently implementing a Bayesian extension of our work following the theory in [6]. We are also working on an extension that allows to handle nested probabilistic operators and unbounded cost constraints formulas.

References

1. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. Bohnenkamp, H., D'Argenio, P., Hermanns, H., Katoen, J.-P.: Modest: A compositional modeling formalism for real-time and stochastic systems. Technical Report CTIT 04-46, University of Twente (2004)
4. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
5. Henzinger, T.A., Horowitz, B., Majumdar, R., Howard, W.-T.: Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 130–144. Springer, Heidelberg (2000)
6. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
7. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: Proc. of 6th Int. Conference on the Quantitative Evaluation of Systems (QEST), pp. 167–176. IEEE Computer Society Press, Los Alamitos (2009)
8. Poulsen, D., David, A., Larsen, K.G., Legay, A., Mikucionis, M., Vliet, J.V., Zheng, W.: Efficient statistical model checking for constant slope timed i/o automata. Technical report, Aalborg University (2011) (submitted for publication)
9. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
10. Sen, K., Viswanathan, M., Agha, G.A.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: QEST, pp. 251–252. IEEE Computer Society Press, Los Alamitos (2005)

11. Teige, T., Eggers, A., Fränzle, M.: Constraint-based analysis of concurrent probabilistic hybrid systems: An application to networked automation systems. Nonlinear Analysis: Hybrid Systems (2010) (in Press, Corrected Proof)
12. Wald, R.: Sequential Analysis. Dove Publisher, New York (2004)
13. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon (2005)
14. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)

Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs*

Alastair Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl

Computer Science Department, Oxford University, United Kingdom

Abstract. *Predicate abstraction* is a key enabling technology for applying finite-state model checkers to programs written in mainstream languages. It has been used very successfully for debugging sequential system-level C code. Although model checking was originally designed for analyzing *concurrent* systems, there is little evidence of fruitful applications of predicate abstraction to shared-variable concurrent software. The goal of this paper is to close this gap. We have developed a *symmetry-aware* predicate abstraction strategy: it takes into account the replicated structure of C programs that consist of many threads executing the same procedure, and generates a Boolean program template whose multi-threaded execution soundly overapproximates the concurrent C program. State explosion during model checking parallel instantiations of this template can now be absorbed by exploiting symmetry. We have implemented our method in the SATABS predicate abstraction framework, and demonstrate its superior performance over alternative approaches on a large range of synchronization programs.

1 Introduction

Concurrent software model checking is one of the most challenging problems facing the verification community today. Not only does software generally suffer from *data state explosion*. Concurrent software in particular is susceptible to state explosion due to the need to track arbitrary thread interleavings, whose number grows exponentially with the number of executing threads.

Predicate abstraction [12] was introduced as a way of dealing with data state explosion: the program state is approximated via the values of a finite number of predicates over the program variables. Predicate abstraction turns C programs into finite-state *Boolean* programs [4], which can be model checked. Since insufficiently many predicates can cause spurious verification results, predicate abstraction is typically embedded into a counterexample-guided abstraction refinement (CEGAR) framework [9]. The feasibility of the overall approach was convincingly demonstrated for *sequential* software by the success of the SLAM project at Microsoft, which was able to discover numerous control-dominated errors in low-level operating system code [5].

The majority of concurrent software is written using mainstream APIs such as POSIX threads (pthreads) in C/C++, or using a combination of language and library support, such as the Thread class, Runnable interface and synchronized construct in Java. Typically, multiple threads are spawned — up front or dynamically, in response to

* This research is supported by EPSRC projects EP/G026254/1 and EP/G051100/1.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 356–371, 2011.

varying system load levels — to execute a given procedure in parallel, communicating via shared global variables. For such shared-variable concurrent programs, predicate abstraction success stories similar to that of SLAM are few and far between. The bottleneck is the exponential dependence of the generated state space on the number of running threads, which, if not addressed, permits exhaustive exploration of such programs only for trivial thread counts.

The key to obtaining scalability is to exploit the *symmetry* naturally exhibited by these programs, namely their invariance under permutations of the involved threads. Fortunately, much progress has recently been made on analyzing emphreplicated non-recursive Boolean programs executed concurrently by many threads [6]. In this paper, we present an approach to predicate-abstracting concurrent programs that leverages this recent progress. More precisely, our goal is a scheme that

- translates a non-recursive C program $\mathbb{P}$ with global-scope and procedure-scope variables into a Boolean program $\mathbb{B}$ such that the n-thread Boolean program, denoted $\mathbb{B}^n$, soundly overapproximates the n-thread C program, denoted $\mathbb{P}^n$. We call such an abstraction method *symmetry-aware*.
- permits predicates over arbitrary C program variables, local or global.

In the remainder of the Introduction, we illustrate why approaching this goal naïvely can render the abstraction **unsound**, creating the danger of missing bugs. In the main part of this paper, we present a sound abstraction method satisfying both of the above objectives. We go on to show how our approach can be implemented for C-like languages, complete with pointers and aliasing, and discuss the issues of spurious error detection and predicate refinement.

In the sequel, we present "programs" as code fragments that declare *shared* and *local* variables. Such code is to be understood as a procedure to be executed by any number of threads. The code can declare *shared* variables, assumed to be declared at the global scope of a (complete) program that contains this procedure. Code can also declare *local* variables, assumed to be declared locally within the procedure. We refer to such code fragments with shared and local variables as "programs". In program listings, we use == for the comparison operator, while = denotes assignment (as in C). Concurrent threads are assumed to interleave with statement-level granularity; see the discussion in the Conclusion on this subject.

1.1 Predicate Abstraction Using Mixed Predicates

The Boolean program $\mathbb{B}$ to be built from the C program $\mathbb{P}$ will consist of Boolean variables, one per predicate as usual. Since $\mathbb{B}$ is to be executed by parallel threads, its variables have to be partitioned into "shared" and "local". As these variables track the values of various predicates over C program variables, the "shared" and "local" attributes clearly depend on the attributes of the C variables a predicate is formulated over. We therefore classify predicates as follows.

Definition 1. *A* ***local*** *predicate refers solely to local* C *program variables. A* ***shared*** *predicate refers solely to shared variables. A* ***mixed*** *predicate is neither local nor shared.*

We reasonably assume that each predicate refers to at least one program variable. A mixed predicate thus refers to both local and shared variables.

Given this classification, consider a local predicate ϕ, which can change only as a result of a thread changing one of its local C variables; a change that is not visible to any other thread. This locality is inherited by the Boolean program if predicate ϕ is tracked by a local Boolean variable. Similarly, shared predicates are naturally tracked by shared Boolean variables.

For a mixed predicate, the decision whether it should be tracked in the shared or local space of the Boolean program is non-obvious. Consider first the following program $\mathbb{P}$ and the corresponding generated Boolean program $\mathbb{B}$, which tracks the mixed predicate s != l in a **local** Boolean variable b:

$\mathbb{P}$:

```
0: shared int s = 0;
   local  int l = 1;
1: assert s != l;
2: ++s;
```

$\mathbb{B}$:

```
0: local bool b = 1;

1: assert b;
2: b = b ? ⋆ : 1;
```

Consider the program $\mathbb{P}^2$, a two-thread instantiation of $\mathbb{P}$. It is easy to see that execution of $\mathbb{P}^2$ can lead to an assertion violation, while the corresponding concurrent Boolean program $\mathbb{B}^2$ is correct. (In fact, $\mathbb{B}^n$ is correct for any $n > 0$.) As a result, $\mathbb{B}^2$ is an **unsound** abstraction for $\mathbb{P}^2$. Consider now the following program $\mathbb{P}'$ and its abstraction $\mathbb{B}'$, which tracks the mixed predicate s == l in a **shared** Boolean variable b:

$\mathbb{P}'$:

```
0: shared int s = 0;
   shared bool t = 0;
   local  int l = 0;
1: if ⋆ then
2:     if t then
3:         assert s != l;
4: l = s + 1;
5: t = 1;
```

$\mathbb{B}'$:

```
0: shared bool b = 1;
   shared bool t = 0;

1: if ⋆ then
2:     if t then
3:         assert !b;
4: b = 0;
5: t = 1;
```

Execution of $(\mathbb{P}')^2$ leads to an assertion violation if the first thread passes the first conditional, the second thread does not and sets t to 1, then the first thread passes the guard t. At this point, s is still 0, as is the first thread's local variable l. On the other hand, $(\mathbb{B}')^2$ is correct. We conclude that $(\mathbb{B}')^2$ is **unsound** for $(\mathbb{P}')^2$. The unsoundness can be eliminated by making b local in $\mathbb{B}'$; an analogous reasoning removes the unsoundness in $\mathbb{B}$ as an abstraction for $\mathbb{P}$. It is clear from these examples, however, that in general a predicate of the form s == l that genuinely depends on s and l cannot be tracked by a shared or a local variable without further amendments to the abstraction process.

At this point it may be useful to consider whether, instead of designing solutions that deal with mixed predicates, we may not be better off by banning them, relying solely on shared and local predicates. Such restrictions on the choice of predicates render very simple bug-free programs **unverifiable** using predicate abstraction, including the following program $\mathbb{P}''$:

$\mathbb{P}''$:

```
0: shared int r = 0;
   shared int s = 0;
   local  int l = 0;
1: ++r;
2: if (r == 1) then
3:    f();
```

$f()$:

```
4: ++s, ++l;
5: assert s == l;
6: goto 4;
```

The assertion in $\mathbb{P}''$ cannot be violated, no matter how many threads execute $\mathbb{P}$, since no thread but the first will manage to execute f. It is easy to prove that, over a set of **non-mixed** predicates (i.e. no predicate refers to both l and one of $\{s, r\}$), no invariant is computable that is strong enough to prove $s == l$. We have included such a proof in the full version of this paper [11].

A technically simple solution to all these problems is to instantiate the template $\mathbb{P}$ n times, once for each thread, into programs $\{\mathbb{P}_1, \ldots, \mathbb{P}_n\}$, in which indices $1, \ldots, n$ are attached to the local variables of the template, indicating the variable's owner. Every predicate that refers to local variables is similarly instantiated n times. The new program has two features: (i) all its variables, having unambiguous names, can be declared at the global scope and are thus shared, including the original global program variables, and (ii) it is multi-threaded, but the threads no longer execute the same code. Feature (i) allows the new program to be predicate-abstracted in the conventional fashion: each predicate is stored in a shared Boolean variable. Feature (ii), however, entails that the new program is no longer symmetric. Model checking it will therefore have to bear the brunt of concurrency state explosion. Such an approach, which we refer to as *symmetry-oblivious*, will not scale beyond a very small number of threads.

To summarize our findings: Mixed predicates are necessary to prove properties for even very simple programs. They can, however, not be tracked using standard thread-local or shared variables. Disambiguating local variables avoids mixed predicates, but destroys symmetry. The goal of this paper is a solution without compromises.

2 Symmetry-Aware Predicate Abstraction

In order to illustrate our method, let $\mathbb{P}$ be a program defined over a set of variables V that is partitioned in the form $V = V_S \cup V_L$ into *shared* and *local* variables. The parallel execution of $\mathbb{P}$ by n threads is a program defined over the shared variables and n copies of the local variables, one copy for each thread. A thread is nondeterministically chosen to be *active*, i.e. to execute a statement of $\mathbb{P}$, potentially modifying the shared variables, and its own local variables, but nothing else. In this section, we ignore the specific syntax of statements, and we do not consider language features that introduce aliasing, such as pointers (these are the subject of Section 3). Therefore, an assignment to a variable v cannot modify a variable other than v, and an expression ϕ depends only on the variables occurring in it, which we refer to as $Loc(\phi) = \{v : v \text{ occurs in } \phi\}$.

2.1 Mixed Predicates and Notify-All Updates

Our goal is to translate the program $\mathbb{P}$ into a Boolean program $\mathbb{B}$ such that, for any n, a suitably defined parallel execution of $\mathbb{B}$ by n threads overapproximates the parallel execution of $\mathbb{P}$ by n threads. Let $E = \{\phi_1, \ldots, \phi_m\}$ be a set of predicates over $\mathbb{P}$, i.e. a set of Boolean expressions over variables in V. We say ϕ_i is

shared if $Loc(\phi_i) \subseteq V_S$,
local if $Loc(\phi_i) \subseteq V_L$, and
mixed otherwise, i.e. $Loc(\phi_i) \cap V_L \neq \emptyset$ and $Loc(\phi_i) \cap V_S \neq \emptyset$.

We declare, in $\mathbb{B}$, Boolean variables $\{b_1, \ldots, b_m\}$; the intention is that b_i tracks the value of ϕ_i during abstract execution of $\mathbb{P}$. We partition these Boolean variables into *shared* and *local* by stipulating that b_i is shared if ϕ_i is shared; otherwise b_i is local. In particular, **mixed predicates are tracked in local variables**. Intuitively, the value of a mixed predicate ϕ_i depends on the thread it is evaluated over. Declaring b_i shared would thus necessarily lose information. Declaring it local does not lose information, but, as the example in the Introduction has shown, is insufficient to guarantee a sound abstraction. We will see shortly how to solve this problem.

Each statement in $\mathbb{P}$ is now translated into a corresponding statement in $\mathbb{B}$. Statements related to flow of control are handled using techniques from standard predicate abstraction [4]; the distinction between shared, mixed and local predicates does not matter here. Consider an assignment to a variable v in $\mathbb{P}$ and a Boolean variable b of $\mathbb{B}$ with associated predicate ϕ. We first check whether variable v affects ϕ, written $\mathit{affects}(v, \phi)$. Given that in this section we assume no aliasing, this is the case exactly if $v \in Loc(\phi)$. If $\mathit{affects}(v, \phi)$ evaluates to *false*, b does not change. Otherwise, code needs to be generated to update b. This code needs to take into account the "flavors" of v and ϕ, which give rise to three different flavors of updates of b:

shared update: Suppose v and ϕ are both shared. An assignment to v is visible to all threads, so the truth of ϕ is modified for all threads. This is reflected in $\mathbb{B}$: by our stipulation above, the shared predicate ϕ is tracked by the *shared* variable b. Thus, we simply generate code to update b according to standard sequential predicate abstraction rules; the new value of b is shared among all threads.

local update: Suppose v is local and ϕ is local or mixed. An assignment to v is visible only by the active (executing) thread, so the truth of ϕ is modified only for the active thread. This also is reflected in $\mathbb{B}$: by our stipulation above, the local or mixed predicate ϕ is tracked by the *local* variable b. Again, sequential predicate abstraction rules suffice; the value of b changes only for the active thread.

notify-all update: Suppose v is shared and ϕ is mixed. An assignment to v is visible to all threads, so the truth of ϕ is modified for all threads. This is **not** reflected in $\mathbb{B}$: by our stipulation above, the mixed predicate ϕ is tracked by the *local* variable b, which will be updated only by the active thread. We solve this problem by (i) generating code to update b locally according to standard sequential predicate abstraction rules, and (ii) **notifying** all passive (non-active) threads of the modification of the shared variable v, so as to allow them to update their local copy of b.

We write $\mathit{must\_notify}(v, \phi)$ if the shared variable v affects the mixed predicate ϕ:

$$\mathit{must\_notify}(v, \phi) \quad = \quad \mathit{affects}(v, \phi) \ \wedge \ v \in V_S \ \wedge \ (Loc(\phi) \cap V_L \neq \emptyset) \, .$$

This formula evaluates to $true$ exactly when it is necessary to notify passive threads of an update to v. What remains to be discussed in the rest of this section is how notifications are implemented in $\mathbb{B}$.

2.2 Implementing Notify-All Updates

We pause to recap some terminology and notation from sequential predicate abstraction [4]. We present our approach in terms of the Cartesian abstraction as used in [4], but our method in general is independent of the abstraction used. Given a set $E = \{\phi_1, \ldots, \phi_m\}$ of predicates tracked by variables $\{b_1, \ldots, b_m\}$, an assignment statement st is translated into the following code, in parallel for each $i \in \{1, \ldots, m\}$:

$$\begin{array}{llcl} \textbf{if} & \mathcal{F}(\mathit{WP}(\ \ \phi_i, st))\ \textbf{then}\ b_i & = & 1 \\ \textbf{else if} & \mathcal{F}(\mathit{WP}(\neg\phi_i, st))\ \textbf{then}\ b_i & = & 0 \\ \textbf{else} & \qquad\qquad\qquad\qquad\quad b_i & = & \star\,. \end{array} \tag{1}$$

Here, $\star$ is the nondeterministic choice expression, WP the weakest precondition operator, and $\mathcal{F}$ the operator that strengthens an arbitrary predicate to a disjunction of cubes over the b_i. For example, with predicate $\phi :: (l\ \texttt{<}\ \texttt{10})$ tracked by variable b, $E = \{\phi\}$ and statement $st :: \texttt{++}l$, we obtain $\mathcal{F}(\mathit{WP}(\phi, st)) = \mathcal{F}(l\ \texttt{<}\ \texttt{9}) = \mathit{false}$ and $\mathcal{F}(\mathit{WP}(\neg\phi, st)) = \mathcal{F}(l\ \texttt{>=}\ \texttt{9}) = (l\ \texttt{>=}\ \texttt{10}) = \neg\phi$, so that (1) reduces to

$$b\ \texttt{=}\ \texttt{(}\ b\ \texttt{?}\ \star\ \texttt{:}\ \texttt{0}\ \texttt{)}\,.$$

In general, (1) is often abbreviated using the assignment

$$b_i\ \texttt{=}\ \mathit{choose}(\mathcal{F}(\mathit{WP}(\phi_i, st)), \mathcal{F}(\mathit{WP}(\neg\phi_i, st)))\,,$$

where $choose(x, y)$ returns $\texttt{1}$ if x evaluates to $true$, $\texttt{0}$ if $(\neg x) \wedge y$ evaluates to $true$, and $\star$ otherwise. Abstraction of control flow guards uses the $\mathcal{G}$ operator, which is dual to $\mathcal{F}$: $\mathcal{G}(\phi) = \neg\mathcal{F}(\neg(\phi))$.

Returning to symmetry-aware predicate abstraction, if $must\_notify(v, \phi)$ evaluates to $true$ for ϕ and v, predicate ϕ is mixed and thus tracked in $\mathbb{B}$ by some local Boolean variable, say b. Predicate-abstracting an assignment of the form $v\ \texttt{=}\ \chi$ requires updating the active thread's copy of b, as well as broadcasting an instruction to all passive threads to update their copy of b, in view of the new value of v. This is implemented using two assignments, which are executed in parallel. The first assignment is as follows:

$$b\texttt{=}\mathit{choose}(\mathcal{F}(\mathit{WP}(\phi, v\ \texttt{=}\ \chi)), \mathcal{F}(\mathit{WP}(\neg\phi, v\ \texttt{=}\ \chi)))\,. \tag{2}$$

This assignment has standard predicate abstraction semantics. Note that, since expression χ involves only local variables of the active thread and shared variables, only predicates over those variables are involved in the defining expression for b.

The second assignment looks similar, but introduces a new symbol:

$$[b]\texttt{=}\mathit{choose}(\mathcal{F}(\mathit{WP}([\phi], v\ \texttt{=}\ \chi)), \mathcal{F}(\mathit{WP}(\neg[\phi], v\ \texttt{=}\ \chi)))\,. \tag{3}$$

The notation $[b]$ stands for the copy of local variable b owned by some passive thread. Similarly, $[\phi]$ stands for the expression defining predicate ϕ, but with every local variable occurring in the expression replaced by the copy owned by the passive thread; this is the predicate ϕ in the context of the passive thread. Weakest precondition computation is with respect to $[\phi]$, while the assignment $v = \chi$, as an argument to WP, is unchanged: v is shared, and local variables appearing in the defining expression χ must be interpreted as local variables of the *active* thread. Assignment (3) has the effect of updating variable b in every passive thread. We refer to Boolean programs involving assignments of the form $[b]=\ldots$ as *Boolean broadcast programs*; a formal syntax and semantics for such programs is given in [11].

Let us illustrate the above technique using a canonical example: consider the assignment $s = l$, for shared and local variables s and l, and define the mixed predicate $\phi :: (s == l)$. The first part of the above parallel assignment simplifies to $b = true$. For the second part, we obtain:

$$[b] = choose(\mathcal{F}(WP(s{=}{=}[l], s{=}l)), \mathcal{F}(WP(\neg(s{=}{=}[l]), s{=}l)))\,.$$

Computing weakest preconditions, this reduces to:

$$[b] = choose(\mathcal{F}(l{=}{=}[l]), \mathcal{F}(\neg(l{=}{=}[l])))\,.$$

Precision of the Abstraction. To evaluate this expression further, we have to decide on the set of predicates available to the $\mathcal{F}$ operator to express the preconditions. If this set includes only predicates over the shared variables and the local variables of the passive thread that owns $[b]$, the predicate $l == [l]$ is not expressible and must be strengthened to $false$. The above assignment then simplifies to $[b] = choose(false, false)$, i.e. $[b] = \star$. The mixed predicates owned by passive threads are essentially *invalidated* when the active thread modifies a shared variable occurring in such predicates, resulting in a very imprecise abstraction.

We can exploit information stored in predicates local to other threads, to increase the precision of the abstraction. For maximum precision one could make *all* other threads' predicates available to the strengthening operator $\mathcal{F}$. This happens in the symmetry-oblivious approach sketched in the Introduction, where local and mixed predicates are physically replicated and declared at the global scope and can thus be made available to $\mathcal{F}$. Not surprisingly, in practice, replicating predicates in this way renders the abstraction prohibitively expensive. We analyze this experimentally in Section 5.

A compromise which we have found to work well in practice (again, demonstrated in Section 5) is to equip operator $\mathcal{F}$ with all shared predicates, all predicates of the passive thread owning $[b]$, **and also** predicates of the active thread. This arrangement is intuitive since the update of a passive thread's local variable $[b]$ is due to an assignment performed by some active thread. Applying this compromise to our canonical example: if both $s == [l]$ and $s == l$ evaluate to true before the assignment $s{=}l$, we can conclude that $[l] == l$ before the assignment, and hence $s == [l]$ after the assignment. Using $\oplus$ to denote exclusive-or, the assignment to $[b]$ becomes:

$$[b] = choose([b] \wedge b, [b] \oplus b)\,.$$

Algorithm 1. Predicate abstraction

```
Input:  Program template $\mathbb{P}$, set of predicates $\{\phi_1, \ldots, \phi_m\}$
Output: Boolean program $\mathbb{B}$ over variables $b_1, \ldots, b_m$
for each statement $d$: stmt of $\mathbb{P}$ do
    if stmt is goto $d_1, \ldots, d_m$ then
        output "$d$: goto $d_1, \ldots, d_m$"
    else if stmt is assume $\phi$ then
        output "$d$: assume $\mathcal{G}(\phi)$"
    else if stmt is $v = \chi$ then
        $\{i_1, \ldots, i_f\} \leftarrow \{i \mid 1 \leq i \leq m \wedge \mathit{affects}(v, \phi_i)\}$
        $\{j_1, \ldots, j_g\} \leftarrow \{j \mid 1 \leq j \leq m \wedge \mathit{must\_notify}(v, \phi_j)\}$
        output "$d$: $\begin{pmatrix} b_{i_1}, & & \mathit{choose}(\mathcal{F}(\mathit{WP}(\phi_{i_1}, v{=}\chi)), \mathcal{F}(\mathit{WP}(\neg\phi_{i_1}, v{=}\chi))), \\ \vdots & & \vdots \\ b_{i_f}, & = & \mathit{choose}(\mathcal{F}(\mathit{WP}(\phi_{i_f}, v{=}\chi)), \mathcal{F}(\mathit{WP}(\neg\phi_{i_f}, v{=}\chi))), \\ [b_{j_1}], & & \mathit{choose}(\mathcal{F}(\mathit{WP}([\phi_{j_1}], v{=}\chi)), \mathcal{F}(\mathit{WP}(\neg[\phi_{j_1}], v{=}\chi))), \\ \vdots & & \vdots \\ [b_{j_g}] & & \mathit{choose}(\mathcal{F}(\mathit{WP}([\phi_{j_g}], v{=}\chi)), \mathcal{F}(\mathit{WP}(\neg[\phi_{j_g}], v{=}\chi))) \end{pmatrix}$"
```

2.3 The Predicate Abstraction Algorithm

We now show how our technique for soundly handling mixed predicates is used in an algorithm for predicate abstracting C-like programs. To present the algorithm compactly, we assume a language with three types of statement: assignments, nondeterministic gotos, and assumptions. Control-flow can be modelled via a combination of gotos and assumes, in the standard way.

Algorithm 1 processes an input program template of this form and outputs a corresponding Boolean broadcast program template. Statements **goto** and **assume** are handled as in standard predicate abstraction: the former are left unchanged, while the latter are translated directly except that the guard of an **assume** statement is expressed over Boolean program variables using the $\mathcal{G}$ operator (see Section 2.2).

The interesting part of the algorithm for us is the translation of assignment statements. For each assignment, a corresponding parallel assignment to Boolean program variables is generated. The *affects* and *must_notify* predicates are used to decide for which Boolean variables regular and broadcast assignments are required, respectively.

3 Symmetry-Aware Predicate Abstraction with Aliasing

So far, we have ignored complications introduced by pointers and aliasing. We now explain how symmetry-aware predicate abstraction is realized in practice, for C programs that manipulate pointers. We impose one restriction: we do not consider programs where a shared pointer variable, or a pointer variable local to thread i, can point to a variable local to thread j (with $j \neq i$). This arises only when a thread copies the address of a stack or thread-local variable to the shared state. This unusual programming style allows thread i to directly modify the local state of thread j at the C program level, breaking the asynchronous model of computation assumed by our method.

For ease of presentation we consider the scenario where program variables either have a base type (e.g. **int** or **float**), or pointer type (e.g. **int**$\star$ or **float**$\star\star$). Our method can be extended to handle records, arrays and heap-allocated memory. As in [4], we assume that input programs have been processed so that l-values involve at most one pointer dereference.

Alias information is important in deciding, once and for all, whether predicates should be classed as local, mixed or shared. For example, let p be a local variable of type **int**$\star$, and consider predicate $\phi :: (\star p == 1)$. Clearly ϕ is not shared since it depends on local variable p. Whether ϕ should be regarded as a local or mixed predicate depends on whether p may point to the shared state: we regard ϕ as local if p can never point to a shared variable, otherwise ϕ is classed as mixed. Alias information also lets us determine whether a variable update may affect the truth of a given predicate, and whether it is necessary to notify other threads of this update. We now show how these intuitions can be formally integrated with our predicate abstraction technique. This involves suitably refining the notions of local, shared and mixed predicates, and the definitions of $\mathit{affects}$ and $\mathit{must\_notify}$ introduced in Section 2.

3.1 Aliasing, Locations of Expressions, and Targets of L-Values

We assume the existence of a sound pointer alias analysis for concurrent programs, e.g. [21], which we treat as a black box. This procedure conservatively tells us whether a shared variable with pointer type may point to a local variable. As discussed at the start of Section 3, we reject programs where this is the case.[1] Otherwise, for a program template $\mathbb{P}$ over variables V, alias analysis yields a relation $\mapsto_d \subseteq V \times V$ for each program location d. For $v, w \in V$, if $v \not\mapsto_d w$ then v provably does not point to w at d.

For an expression ϕ and program point d, we write $\mathit{loc}(\phi, d)$ for the set of variables that it may be necessary to access in order to evaluate ϕ at d, during an arbitrary program run. We have $\mathit{loc}(z, d) = \emptyset$ for a constant value z, $\mathit{loc}(v, d) = \{v\}$, and $\mathit{loc}(\&v, d) = \emptyset$ for $v \in V$: evaluating an "address-of" expression requires no variable access, as addresses of variables are fixed at compile time. Finally, for any $k > 0$:

$$\mathit{loc}(\underbrace{\star\ldots\star}_{k} v, d) = \{v\} \cup \bigcup_{w \in V} \{\mathit{loc}(\underbrace{\star\ldots\star}_{k-1} w, d) \mid v \mapsto_d w\}\,.$$

Evaluating a pointer dereference $\star v$ involves reading both v and the variable to which v points. For other compound expressions, $\mathit{loc}(\phi, d)$ is defined recursively in the obvious way. The precision of $\mathit{loc}(\phi, d)$ is directly related to the precision of alias analysis.

For an expression ϕ, we define $\mathit{Loc}(\phi) = \cup_{1 \le d \le k} \mathit{loc}(\phi, d)$ as the set of variables that may need to be accessed to evaluate ϕ at an *arbitrary* program point during an arbitrary program run. Note how this definition of Loc generalizes that used in Section 2.

We finally write $\mathit{targets}(x, d)$ for the set of variables that may be modified by writing to l-value x at program point d. Formally, we have $\mathit{targets}(v, d) = \{v\}$ and $\mathit{targets}(\star v, d) = \{w \in V \mid v \mapsto_d w\}$. Note that $\mathit{targets}(\star v, d) \neq \mathit{loc}(\star v, d)$: *Writing*

[1] This also eliminates the possibility of thread i pointing to variables in thread $j \neq i$: the address of a variable in thread j would have to be communicated to thread i via a shared variable.

through $\star v$ modifies only the variable to which v points, while *reading* the value of $\star v$ involves reading the value of v, to determine which variable w is pointed to by v, and then reading the value of w.

3.2 Shared, Local and Mixed Predicates in the Presence of Aliasing

In the presence of pointers, we define the notion of a predicate ϕ being shared, local, or mixed exactly as in Section 2.1, but with the generalization of *Loc* presented in Section 3.1. In Section 2.1, without pointers, we could classify ϕ purely syntactically, based on whether any shared variables appear in ϕ. In the presence of pointers, we must classify ϕ with respect to alias information; our definition of *Loc* takes care of this.

Recall from Section 2.1 that we defined $\mathit{affects}(v, \phi) = (v \in \mathit{Loc}(\phi))$ to indicate that updating variable v may affect the truth of predicate ϕ. In the presence of pointers, this definition no longer suffices. The truth of ϕ may be affected by assigning to l-value x if x may alias some variable on which ϕ depends. Whether this is the case depends on the program point at which the update occurs. Our definitions of loc and $targets$ allow us to express this:

$$\mathit{affects}(x, \phi, d) = (\mathit{targets}(x, d) \cap \mathit{loc}(\phi, d) \neq \emptyset).$$

We also need to determine whether an update affects the truth of a predicate only for the thread executing the update, or for all threads. The definition of $\mathit{must\_notify}$ presented in Section 2.1 needs to be adapted to take aliasing into account. At first sight, it seems that we must simply parameterise *affects* according to program location, and replace the conjunct $v \in V_S$ with the condition that x may target some shared variable:

$$\begin{aligned}\mathit{must\_notify}(x, \phi, d) = {} & \mathit{affects}(x, \phi, d) \;\wedge\; (\mathit{Loc}(\phi) \cap V_L \neq \emptyset) \\ & \wedge (\mathit{targets}(x, d) \cap V_S \neq \emptyset)\ .\end{aligned}$$

However, this is unnecessarily strict. We can refine the above definition to minimise the extent to which notifications are required, as follows:

$$\mathit{must\_notify}(x, \phi, d) = (\mathit{targets}(x, d) \cap \mathit{Loc}(\phi) \cap V_S \neq \emptyset) \wedge (\mathit{Loc}(\phi) \cap V_L \neq \emptyset)\ .$$

The refined definition avoids the need for thread notification in the following scenario. Suppose we have shared variables s and t, local variable l, local pointer variable p, and predicate $\phi :: (s\ >\ l)$. Consider an assignment to $\star p$ at program point d. Suppose that alias analysis tells us exactly $p \mapsto_d t$ and $p \mapsto_d l$. The only shared variable that can be modified by assigning through $\star p$ at program point d is t, and the truth of ϕ does not depend on t. Thus the assignment does *not* require a “notify-all” with respect to ϕ. Working through the definitions, we find that our refinement of $\mathit{must\_notify}$ correctly determines this, while the naïve extension of $\mathit{must\_notify}$ from Section 2.1 would lead to an unnecessary “notify-all”.

The predicate abstraction algorithm (Alg. 1) can now be adapted to handle pointers: parameter d is simply added to the uses of *affects* and $\mathit{must\_notify}$. Handling of pointers in weakest preconditions works as in standard predicate abstraction [4], using Morris’s general axiom of assignment [19].

4 Closing the CEGAR Loop

We have integrated our novel technique for predicate-abstracting symmetric concurrent programs with the SATABS CEGAR-based verifier [10], using the Cartesian abstraction method and the maximum cube length approximation [4]. We now sketch how we have adapted the other phases of the CEGAR loop: model checking, simulation and refinement, to accurately handle concurrency.

Model checking Boolean broadcast programs. Our predicate abstraction technique generates a concurrent Boolean broadcast program. The extended syntax and semantics for broadcasts mean that we cannot simply use existing concurrent Boolean program model checkers such as BOOM [6] for the model checking phase of the CEGAR loop. We have implemented a prototype extension of BOOM, which we call B-BOOM. B-BOOM extends the counter abstraction-based symmetry reduction capabilities of BOOM to support broadcast operations. Symbolic image computation for broadcast assignments is significantly more expensive than image computation for standard assignments. In the context of BOOM it involves 1) converting states from counter representation to a form where the individual local states of threads are stored using distinct BDD variables, 2) computing the intersection of $n-1$ successor states, one for each passive thread paired with the active thread, and 3) transforming the resulting state representation back to counter form using Shannon expansion. The expense of image computation for broadcasts motivates the careful analysis we have presented in Sections 2 and 3 for determining tight conditions under which broadcasts are required.

Simulation. To determine the authenticity of abstract error traces reported by B-BOOM we have extended the SATABS simulator. The existing simulator extracts the control flow from the trace. This is mapped back to the original C program and translated into a propositional formula (using standard techniques such as single static assignment conversion and bitvector interpretation of variables). The error is spurious exactly if this formula is unsatisfiable. In the concurrent case, the control flow information of an abstract trace includes which thread executes actively in each step. We have extended the simulator so that each local variable involved in a step is replaced by a fresh indexed version, indicating the executing thread that owns the variable. The result is a trace over the replicated C program $\mathbb{P}^n$, which can be directly checked using a SAT/SMT solver.

Refinement. Our implementation performs refinement by extracting new predicates from counterexamples via weakest precondition calculations. This standard method requires a small modification in our context: weakest precondition calculations generate predicates over shared variables, and local variables of *specific* threads. For example, if thread 1 branches according to a condition such as $l < s$, where l and s are local and shared, respectively, weakest precondition calculations generate the predicate $l_1 < s$, where l_1 is thread 1's copy of l. Because our predicate abstraction technique works at the template program level, we cannot add this predicate directly. Instead, we generalize such predicates by removing thread indices. Hence in the above example, we add the mixed predicate $l < s$, for *all* threads.

An alternative approach is to refine the abstract transition relation associated with the Cartesian abstraction based on infeasible steps in the abstract counterexample [3].

We do not currently perform such refinement, as correctly refining abstract transitions involving broadcast assignments is challenging and requires further research.

5 Experimental Results

We evaluate the SATABS-based implementation of our techniques using a set of 14 concurrent C programs. We consider benchmarks where threads synchronize via locks (lock-based), or in a lock-free manner via atomic *compare-and-swap* (cas) or *test-and-set* (tas) instructions. The benchmarks are as follows:[2]

Increment, Inc./Dec. (lock-based and cas-based): a counter, concurrently incremented, or incremented and decremented, by multiple threads [20]

Prng (lock-based and cas-based) concurrent pseudorandom number generator [20]

Stack (lock-based and cas-based) thread-safe stack implementation, supporting concurrent pushes and pops, adapted from an Open Source IBM implementation[3] of an algorithm described in [20]

Tas Lock, Ticket Lock (tas-based) concurrent lock implementations [18]

FindMax, FindMaxOpt (lock-based and cas-based) implementations of parallel reduction operation [2] to find maximum element in array. **FindMax** is a basic implementation, and **FindMaxOpt** an optimized version where threads reduce communication by computing a partial maximum value locally.

Mixed predicates were required for verification to succeed in all but two benchmarks: lock-based *Prng*, and lock-based *Stack*. For each benchmark, we consider verification of a safety property, specified via an assertion. We have also prepared a buggy version of each benchmark, where an error is injected into the source code to make it possible for this assertion to fail. We refer to correct and buggy versions of our benchmarks as *safe* and *unsafe*, respectively.

All experiments are performed on a 3GHz Intel Xeon machine with 40 GB RAM, running 64-bit Linux, with separate timeouts of 1h for the abstraction and model checking phases of the CEGAR loop. Predicate abstraction uses a maximum cube length of 3 for all examples, and MiniSat 2 (compiled with full optimizations) is used for predicate abstraction and counterexample simulation.

Symmetry-aware vs. symmetry-oblivious method. We evaluate the scalability of our symmetry-aware predicate abstraction technique (SAPA) by comparing it against the symmetry-oblivious predicate abstraction (SOPA) approach sketched near the end of Section 1, for verification of correct versions of our benchmarks. Recall that in SOPA, an n-thread symmetric concurrent program is expanded so that variables for all threads are explicitly duplicated, and n copies of all non-shared predicate are generated. The expanded program is then abstracted over the expanded set of predicates, using standard predicate abstraction. This yields a Boolean program for each thread; the parallel composition of these n Boolean programs is explored by a model checker. Because symmetry is not exploited, and no broadcasts are required, any Boolean program model

[2] All benchmarks and tools are available online: `http://www.cprover.org/SAPA`

[3] `http://amino-cbbs.sourceforge.net`

Table 1. Comparison of symmetry-aware and symmetry-oblivious predicate abstraction over our benchmarks. For each configuration, the fastest abstraction and model checking times are in bold.

		Pred.			SOPA		SAPA		
Benchmark	n	S	L	M	Abs	MC	#Its	Abs	MC
Increment	6	2	1	1	13	5	2	**1**	**<1**
(lock-based)	8				29	152			**1**
	9				40	789			**1**
	10				56	T.O.			**2**
	12								**7**
	14								**24**
	16								**100**
	18								**559**
	20								**2882**
Increment	4	2	4	2	50	12	3	**6**	**1**
(cas-based)	5				94	358			**13**
	6				159	T.O.			**116**
	7								**997**
Inc./Dec.	4	6	3	2	71	6	3	**11**	**2**
(lock-based)	5				132	656			**50**
	6				231	T.O.			**1422**
Inc./Dec.	2	6	10	4	125	<1	5	**78**	<1
(cas-based)	3				372	6			**3**
	4				872	4043			**252**
Tas Lock	3	4	2	2	3	2	3	**1**	**<1**
(tas-based)	4				9	114			**4**
	5				14	T.O.			**72**
	6								**725**
Ticket Lock	2	12	3	4	554	1	2	**251**	1
(tas-based)	3				1319	3			**1**
	4				T.O.	–			**2**
	6								**62**
	8								**2839**
Prng	1	1	5	0	<1	<1	2	<1	<1
(lock-based)	6	1	12	0	69	21	5	**26**	**<1**
	7				83	191			**1**
	8				96	T.O.			**2**
	16								**142**
	26								**3023**
Prng	1	1	5	0	<1	<1	2	<1	<1
(cas-based)	3	1	14	2	**29**	**<1**	5	48	1
	4				**40**	**12**		48	38
	5				57	**1049**		**48**	1832
FindMax	6	0	0	1	5	30	1	**<1**	**<1**
(lock-based)	7				9	244			**1**
	8				14	T.O.			**1**
	16								**125**
	25								**3005**
FindMax	3	0	5	1	4	7	3	**1**	**2**
(cas-based)	4				8	407			**368**
FindMaxOpt	4	0	1	1	3	40	1	**<1**	**3**
(lock-based)	5				6	1356			**33**
	6				11	T.O.			**269**
	7								**1773**
FindMaxOpt	3	0	6	1	9	11	3	**3**	**2**
(cas-based)	4				15	1097			**61**
	5				22	T.O.			**1240**
Stack	3	1	4	0	<1	14	2	<1	**8**
(lock-based)	4				<1	945			**374**
Stack	3	1	4	1	2	29	2	**<1**	**14**
(cas-based)	4				8	3408			**813**

checker can be used. We have tried both standard Boom [6] (without symmetry reduction) and Cadence SMV [17] to model check expanded Boolean programs. In all cases, we found Boom to be faster than SMV, thus we present results only for Boom.

Table 1 presents the results of the comparison. For each benchmark and each approach we show, for interesting thread counts (including the largest thread count that could be verified with each approach), the number of local, mixed, and shared predicates (L, M, S) over the template program that were needed to prove the program safe (which varies slightly with n), and the elapsed time for predicate abstraction and model checking. For each configuration, the fastest abstraction and model checking times are shown in bold. Model checking uses standard Boom, without symmetry reduction (SOPA) and B-Boom, our extension to Boom discussed in Section 4 (SAPA), respectively. T.O. indicates a timeout; succeeding cells are then marked '–'.

The results show that in the vast majority of cases our novel SAPA technique significantly outperforms SOPA, both in terms of abstraction and model checking time. The former can be attributed to the fact that, with SOPA, the number of predicates grows according to the number of threads considered, while with SAPA, this is thread count-independent. The latter is due to the exploitation of template-level symmetry by B-Boom. The exception to this is the cas-based *Prng* benchmark, for which SAPA yields slower verification. Profiling with respect to this benchmark shows that the inferior performance of the model checker with SAPA comes from the expense of performing broadcast operations. Note, however, that the ratio between model checking times for SOPA and SAPA on this benchmark decreases as the thread counts go up.

Table 2. Comparison of sound and unsound approaches; incorrect results in bold

	Symmetry-Aware				**Mixed as local**				**Mixed as shared**			
Benchmark	Safe	n	Unsafe	n	Safe	n	Unsafe	n	Safe	n	Unsafe	n
Increment (lock-based)	safe	>10	unsafe	2	safe	>10	**error**	2	safe	10	**error**	2
Incr. (cas-based)	safe	7	unsafe	2	safe	8	**safe**	5	**error**	2	**error**	2
Incr./Dec. (lock-based)	safe	6	unsafe	3	safe	>10	**safe**	>10	safe	>10	unsafe	3
Incr./Dec. (cas-based)	safe	4	unsafe	3	safe	6	**safe**	8	**error**	2	**error**	3
Tas Lock (tas-based)	safe	7	unsafe	2	safe	8	**error**	2	**error**	2	**error**	2
Ticket Lock (tas-based)	safe	8	unsafe	3	safe	>10	unsafe	3	safe	5	unsafe	3
Prng (lock-based)	safe	>10	unsafe	2	safe	>10	unsafe	2	safe	>10	unsafe	2
Prng (cas-based)	safe	5	unsafe	3	safe	7	unsafe	3	safe	6	unsafe	3
FindMax (lock-based)	safe	>10	unsafe	2	safe	>10	**safe**	>10	safe	2	**error**	2
FindMax (cas-based)	safe	4	unsafe	2	safe	5	**safe**	4	safe	2	**safe**	1
FindMaxOpt (lock-based)	safe	7	unsafe	2	safe	7	**safe**	6	**error**	2	**error**	2
FindMaxOpt (cas-based)	safe	5	unsafe	1	safe	5	unsafe	1	**error**	2	unsafe	1
Stack (lock-based)	safe	4	unsafe	4	safe	4	unsafe	4	safe	4	unsafe	4
Stack (cas)	safe	4	unsafe	2	safe	4	**safe**	6	safe	4	**error**	2

Comparison with Unsound Methods. In Section 1, we described two naïve solutions to the mixed predicate problem: uniformly using local or shared Boolean variables to represent mixed predicates, and then performing standard predicate abstraction. We denote these approaches *mixed as local* and *mixed as shared*, respectively. Although we demonstrated theoretically in Section 1 that both methods are unsound, it is interesting to see how they perform in practice. Table 2 shows the results of applying CEGAR-based model checking to safe and unsafe versions of our benchmarks, using our sound technique, and the unsound *mixed as local* and *mixed as shared* approaches. In all cases, B-BOOM is used for model checking. For the sound technique, we show the largest thread count for which we could prove correctness of each safe benchmark, and the smallest thread count for which a bug was revealed in each unsafe benchmark. The other columns illustrate how the unsound techniques differ from this, where "error" indicates a refinement failure: it was not possible to extract further predicates from spurious counterexamples. Bold entries indicate cases where the unsound approaches produce incorrect, or inconclusive results.[4] The number of cases where the unsound approaches produce false negatives, or lead to refinement failure, suggest that little confidence can be placed in these techniques, even for purposes of falsification. This justifies the more sophisticated and, crucially, sound techniques developed in this paper.

6 Related Work and Conclusion

There exists a large body of work on the different stages of CEGAR-based program analysis. We focus here on the abstraction stage, which is at the heart of this paper.

Predicate abstraction goes back to the foundational work by Graf and Saïdi [12]. It was first presented for *sequential* programs in a mainstream language (C) by Ball,

[4] We never expect the unsound techniques to report conclusively that a safe benchmark is unsafe: this would require demonstrating a concrete error trace in the original, safe, program.

Majumdar, Millstein, Rajamani [4] and implemented as part of the SLAM project. We have found many of the optimizations suggested by [4] to be useful in our implementation as well. Although SLAM has had great success in finding real bugs in system-level code, we are not aware of any extensions of it to concurrent programs (although this option is mentioned by the authors of [4]). We attribute this to a large part to the infeasibility, at the time, to handle realistic multi-threaded Boolean programs. We believe our own work on BOOM [6] has made progress in this direction that has made it attractive again to address concurrent predicate abstraction.

We are not aware of other work that presents precise solutions to the problem of "mixed predicates". Some approaches avoid it by syntactically disallowing such predicates, e.g. [22], whose authors don't discuss, however, the reasons for (or, indeed, the consequences of) doing so. Another approach *havocks* (assigns nondeterministically) global variables that may be affected by an operation [8], thus taking away the mixed flavor from certain predicates. In yet other work, "algorithmic circumstances" may make the treatment of such predicates unnecessary. The authors of [15], for example, use predicate abstraction to finitely represent the environment of a thread in multi-threaded programs. The "environment" consists of assumptions on how threads may manipulate the *shared* state of the program, irrespective of their local state. Our case of *replicated* threads, in which mixed predicates would constitute a problem, is only briefly mentioned in [15]. In [7], an approach is presented that handles *recursive* concurrent C programs. The abstract transition system of a thread (a pushdown system) is formed over predicates that are projected to the global or the local program variables and thus cannot compare "global against local" directly. As we have discussed, some reachability problems cannot be solved using such restricted predicates. We conjecture this problem is one of the potential causes of non-termination in the algorithm of [7].

Other model checkers with some support for concurrency include BLAST, which does not allow general assertion checking [14], and MAGIC [7], which does not support shared variable communication, making a comparison to our work little meaningful.

In conclusion, we mention that building a CEGAR-based verification strategy is a tremendous effort, and our work so far can only be the beginning of such effort. We have assumed a very strict (and unrealistic) memory model that guarantees atomicity at the statement level. One can work soundly with the former assumption by pre-processing input programs so that the shared state is accessed only via word-length reads and writes, ensuring that all computation is performed using local variables. Extending our approach to weaker memory models, building on existing work in this area [16,1], is future work. Our plans also include a more sophisticated refinement strategy, drawing upon recent results on abstraction refinement for concurrent programs [13], and a more detailed comparison with existing approaches that circumvent the mixed-predicates problem using other means.

Acknowledgments. We are grateful to Gérard Basler for assistance with BOOM, and Michael Tautschnig for his insightful comments on an earlier draft of this work.

References

1. Alglave, J.: A Shared Memory Poetics. PhD thesis, Université Paris 7 and INRIA (2010), http://moscova.inria.fr/~alglave/these

2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
3. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 388–403. Springer, Heidelberg (2004)
4. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Language Design and Implementation (PLDI), pp. 203–213 (2001)
5. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Principles of Programming Languages (POPL), pp. 1–3 (2002)
6. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Context-aware counter abstraction. In: Formal Methods in System Design (FMSD), vol. 36(3), pp. 223–245 (2010)
7. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
8. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a software model checking approach. In: Formal Methods in Computer-Aided Design, FMCAD (2010)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM, JACM (2003)
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. In: Formal Methods in System Design (FMSD), pp. 105–127 (2004)
11. Donaldson, A.F., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs (extended technical report). In: CoRR, pp. 1102–2330 (2011)
12. 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)
13. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344. ACM Press, New York (2011)
14. Henzinger, T., Jhala, R., Majumdar, R.: Race checking by context inference. In: Programming Language Design and Implementation (PLDI), pp. 1–13 (2004)
15. Henzinger, T., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
16. Lee, J., Padua, D.: Hiding relaxed memory consistency with a compiler. IEEE Transactions on Computers 50, 824–833 (2001)
17. McMillan, K.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, Boston (1993)
18. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. Transactions on Computer Systems (TOCS) 9(1), 21–65 (1991)
19. Morris, J.: A general axiom of assignment. In: Theoretical Foundations of Programming Methodology. Lecture Notes of an International Summer School, pp. 25–34. D. Reidel Publishing Company (1982)
20. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2005)
21. Rugina, R., Rinard, M.C.: Pointer analysis for multithreaded programs. In: PLDI, pp. 77–90 (1999)
22. Timm, N., Wehrheim, H.: On symmetries and spotlights – verifying parameterised systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 534–548. Springer, Heidelberg (2010)

Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, Czech Republic

Abstract. Predator is a new open source tool for verification of sequential C programs with dynamic linked data structures. The tool is based on separation logic with inductive predicates although it uses a graph description of heaps. Predator currently handles various forms of lists, including singly-linked as well as doubly-linked lists that may be circular, hierarchically nested and that may have various additional pointer links. Predator is implemented as a `gcc` plug-in and it is capable of handling lists in the form they appear in real system code, especially the Linux kernel, including a limited support of pointer arithmetic. Collaboration on further development of Predator is welcome.

1 Introduction

In this paper, we present a new tool called *Predator* for fully automatic verification of sequential C programs with dynamic linked data structures. In particular, Predator can currently handle various complex kinds of *singly-linked as well as doubly-linked lists* that may be circular, shared, hierarchically nested, and that can have various additional pointers (head/tail pointers, data pointers, etc.). Predator implicitly checks for absence of generic errors, such as null dereferences, double deletion, memory leakage, etc. It can also print out a symbolic representation of the shapes of the memory structures arising in a program. Finally, users can, of course, use Predator to check custom properties about the data structures being used in their code by writing (directly in C) tester programs exercising these structures.

Predator is based on *separation logic* with *higher-order inductive predicates*. It is inspired by the works [2,9,10] and the very influential tool called Space Invader[1] (or simply Invader). However, compared to Invader, the heap representation in Predator is not based on lists of separation logic formulae, but rather a *graph representation* of these formulae. The algorithms handling the symbolic heap representation (in particular, the abstraction and join operators based on detecting occurrences of heap structures that can be described by inductive predicates) have been newly designed.

Compared to Invader that contains a partial support of doubly-linked lists only, Predator supports them equally well as singly-linked lists. Predator also contains a special support for list segments of length 0 or 1 that are common in practice [9] and that may cause problems to Invader (as we illustrate further on).

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the BUT FIT project FIT-S-11-1. An extended version of the paper is available as the technical report [6].

[1] `http://www.eastlondonmassive.org/East_London_Massive/Invader_Home.html`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 372–378, 2011.

The long term goal of Predator is handling *real system code*, in particular, the Linux kernel. In such code, for efficiency reasons, special forms of lists are used. In order to be able to handle them, Predator comes with a limited support of *pointer arithmetic*, which, however, covers most practical needs. Therefore, in a heap representation, the points-to links are associated with an offset w.r.t. the object they point to. Despite such an extension is not mentioned in [2,10], the Invader tool seems to partially support it, but it fails in many practical cases that Predator can handle well.

Predator is written in C++. It is built as a `gcc` plug-in, hence its front end is the same compiler that is used in practice for compiling the code that Predator is intended to analyse. Predator is completely *open source*[2] in order to allow for an open collaboration on its further development, which is very welcome.

There of course exist many other works on verification of programs with dynamic linked data structures than those using separation logic, including works based on other logics [7], automata [3], upward-closed sets [1], etc. These approaches offer different degrees of generality, automation, or scalability. A proper discussion of such works is, however, beyond the scope of this tool paper. Throughout the paper, we instead concentrate on a comparison with Invader as the closest tool to Predator. A similar tool is also jStar [5] which, however, concentrates on Java-specific problems. In [4], a bi-abductive analysis based on separation logic was proposed and implemented in a version of Invader, called Abductor[3]. This analysis, which is more scalable but less precise than the classical analysis used in Invader and Predator, is not yet implemented in Predator (whose core can, however, be used to implement it in the future). Unlike Invader, Predator cannot currently handle entire modules of Linux (such as drivers[4]) due to a so far very weak support of non-pointer data, which is one of the planned future works on Predator (together with a support of tree data structures, bi-abduction, etc.).

Below, we first say a bit more on the Linux lists supported by Predator, then we briefly mention some implementation details of Predator, and we proceed to interesting cases studies that illustrate the power of our tool. For some of the case studies, we are not aware of any other fully automatic, freely available tool, capable of handling them.

2 Lists Used in The Linux Kernel

As there is no standard implementation of linked lists in the C language, the Linux kernel has to implement lists on its own. The list implementation in Linux is well-known for its efficiency, portability, readability, and scalability—for instance, it allows to create list nodes which are owned by many distinct lists at a time. The downside is that it operates at a low level, hence it is easy to misuse the routines, and cause a disaster within the kernel. Later on, we will mention some common mistakes in manipulating Linux lists, which Predator is able to detect.

[2] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`

[3] Abductor is publicly available, but we have not managed to make it run. There is also a commercial implementation (`www.monoidics.com`), which is, however, not freely available.

[4] Although Invader has already shown some interesting results on pre-processed source code of selected Linux drivers, it is not ready for analysing drivers using the native Linux lists.

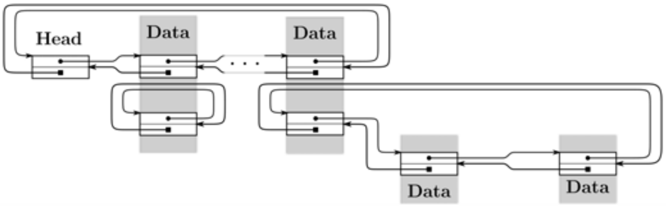


Fig. 1. A list of lists as implemented in the Linux kernel

The whole implementation of Linux lists is available in a single header file and consists of about 500 lines of code. It defines only one[5] type, which does all the job. The type contains a pair of pointers (`next` and `prev`), but no data. Such a structure is called a *head* and can be used in two ways—either as a starting point of a list (a *standalone head*), cf. the leftmost node in Fig. 1, or as part of list nodes (an *embedded head*).

Basic list operations (like addition, removal, reconnection of nodes) work only with heads and do not care about any associated data. In particular, the routines themselves do not distinguish between embedded and standalone heads.

The embedded head can be placed at an arbitrary offset in the surrounding structure. Moreover, it is possible to put many embedded heads into one structure such that one node is part of many lists. The standalone head can be placed on stack, but it can also be surrounded by another type. This way one can construct hierarchical list structures as shown in Fig. 1. Note that the beginning of the data nodes (depicted in gray in Fig. 1) needs not to be directly accessible by any pointer and can hence be mistakenly considered garbage if pointer arithmetic is not taken into account.

Linux lists are doubly-linked and *circular*, which significantly simplifies the design and boosts performance. That is, each routine for reconnection of a node (insertion, deletion, etc.) fits into a single basic block, which would not have been possible in case of regular NULL-terminated doubly-linked lists. It also implies that there is no need to have an explicit starting point (a standalone head) for each list. The Linux list library provides macros to define a standalone head and initialise it as an empty list. In case of Linux lists, an empty list means that `next` and `prev` fields point to the head itself.

The basic list traversal macro (`list_for_each`) provides a pointer to a head in each iteration and works without any type-awareness of the data nodes being traversed. The macro `list_entry` then allows to translate a pointer to a head into a pointer to the corresponding data node. As the offset of each embedded head in the structure is known at compile-time, the macro can easily use pointer arithmetic to compute the required address. There is also an extended macro for list traversal (`list_for_each_entry`) that efficiently wraps `list_entry` and this way gives us a pointer to the data node in each iteration, instead of the pointer to the embedded head. Note, however, that it may happen that some pointer variable points to the unallocated space around a standalone head and yet may be correctly used (by being subsequently moved forward by pointer arithmetic). On the other hand, dereferencing such a pointer is an error, which can, e.g., lead to stack smashing (and is detectable by Predator).

[5] Starting with Linux-2.5.64, there is also an optimised variant of lists for constructing hash tables. We are not yet able to analyse the code that uses these optimised lists.

A nice introduction into how Linux lists work can be found in [8]. We use the code from there as one of the case studies distributed as test-cases with Predator.

3 A Note on the Implementation of Predator

Predator implements a symbolic analysis based on separation logic with higher-order inductive predicates, inspired by the works [2,10] implemented in Space Invader. Predator uses a graph representation of separation logic formulae, a little bit similar to the graph representation introduced in [2] for description of the predicate discovery algorithm. Our representation is, however, more complex and used all the time.

In our graph-based symbolic representation of heaps, we use two kinds of nodes: *objects* (statically and automatically allocated program variables, dynamically allocated storage, etc.) and *values* of the objects (e.g., addresses of objects and the special undefined, deleted, and null values in the case of pointers and function pointers). Objects can be nested in order to represent the composition of C language structures. The appropriate nodes are linked by oriented graph edges *hasValue* (going from objects to values) and *pointsTo* (going from values to target objects). In order to allow for efficient equality testing, equal objects are simply linked to the same value node. To encode non-equality relations, value nodes may be linked by undirected *neq* edges. Further, when pointer arithmetic produces a value that does not point to a valid target, we use so-called *offset* edges between value nodes. Such values can later be used for a valid memory operation—they can either be translated by another use of pointer arithmetic to a valid address, or directly used for accessing an existing subobject of a non-existent surrounding object (which is common, e.g., when working with Linux lists).

We represent inductive predicates as special *abstract objects*. Currently, we support only singly- and doubly-linked list segments that may be shared, nested, and with various additional (head, tail, data, and the like) pointers. A doubly-linked list segment (DLS) has two endpoints, both of which may be pointed to. Therefore, since each object has exactly one address, we in fact represent DLS as pairs of abstract objects. To cope with pointer arithmetic, we equip abstract objects with *offsets* specifying the relative placement of the core linking pointers (`next`/`prev` for lists). Moreover, to cope with Linux lists, we also record the *head offset* that is a relative placement of the list pointer's target. For Linux lists, it corresponds to the offset of the embedded head, whereas for regular lists, it is simply zero. We do not treat the minimal segment length as an explicit property of a list segment as in [9]. Instead, our list segments are implicitly possibly empty (i.e., of length 0+). We use the generic mechanism of *neq* edges between nodes before and after a segment to construct non-empty segments (length 1+). For DLS, we use two *neq* edges for length 1+ (one edge for each direction) and three *neq* edges for length 2+ (the additional *neq* edge is in between the ends of the DLS). Such an approach leads to a simpler and more readable implementation. Apart from that, we then have special abstract objects for list segments of length 0 or 1.

Predator maintains a set of symbolic heaps for each basic block entry. The set is not yet implemented as an ordered or hashed container, but it utilises a join operator similar to the join operator introduced in [10], helping to significantly decrease the number of symbolic heaps to be maintained. Moreover, Predator uses a slightly modified version of the join algorithm to merge pairs of objects during a list segment abstraction, in particular to join nested predicates, shared (head, tail or other) pointers, and other data. The modified join algorithm operates on two parts of a single heap given by the pair of objects being merged, and constructs a joint description of both parts. The algorithm can also run in a read-only mode to decide whether the join operation is possible. The read-only mode can be safely used during predicate discovery. Thanks to this, the algorithms for abstraction and predicate discovery are implemented as a very thin purpose-specific layer on top of the generic join algorithm.

For inter-procedural analysis, Predator uses function summaries in a way similar to [10], including a support of indirect function calls and recursive calls of fixed depth.

Predator is tightly integrated with `gcc` (version 4.5.0 and newer) as a *plug-in*. Therefore, there is no need to manually pre-process the sources, neither to change the way they are built, whenever dealing with software natively compiled by `gcc`. Usage of Predator is as easy as adding a new compiler flag into `CFLAGS` while building a project. Code defects encountered during analysis are reported in the `gcc` format. Hence it is easy to reuse existing development tools, IDE, etc. In order to give users a clue about detected errors, Predator provides a backtrace for each error. Predator attempts to report as many errors and warnings as possible per run. For instance, if a memory leak is detected, a warning is issued, and Predator keeps searching for further errors (due to a garbage collector that gets the symbolic heap back to its consistent shape). Predator supports error recovery for most of the program errors which it is able to detect.

4 Experiments with Predator

Along with Predator, we distribute a comprehensive set of programs (over a hundred test cases) that can be handled by our tool, including various textbook implementations of lists (singly-linked, doubly-linked, circular, hierarchically nested, etc.) as well as examples using Linux lists[6]. These case studies are mid-size (up to 300 lines), however, they contain almost only pointer manipulations unlike larger programs whose big portions are often not relevant for pointer analyses like ours. Apart from basic list manipulation (creation of random lists, reversal, destruction, etc.), we provide also examples of various sorting algorithms: Merge-Sort, Insert-Sort, and Bubble-Sort[7]. The Merge-Sort case study operates on hierarchical singly-linked lists. The other two sorts use the native implementation of Linux lists. Predator is not proving that the resulting list is sorted, but it verifies memory safety of the code. Invader, as a freely available tool closest to Predator, is not able to analyse any of our sorting case studies.

Some of our test cases show common mistakes in using Linux lists such as mixing pointers to a head with pointers to data or treating a standalone head as if it was an embedded head. Only programmers know the purpose of each head, and if they use the

[6] `https://github.com/kdudka/predator/tree/61d5df3/sl/data`

[7] See [6] for file names under which the case studies can be found in the distribution of Predator.

head in a wrong way, it is likely to be noticed at run-time only (and often not immediately). For example, starting from a standalone head, the `list_for_each_entry` macro provides a valid pointer to data in each iteration. However, if one starts to traverse the list from the middle, it ends up by misinterpreting the standalone head's neighbourhood as list node's data. Predator is capable of detecting such mistakes. We, for instance, provide an example where a wrong head is used for a Linux list traversal. Despite even the dynamic analysis tool `valgrind`, often used by developers, claims there is an invalid write, Invader says the code is safe. On the contrary, Predator detects the flaw in 0.01s, which is even faster than `valgrind`.

Our test suite further contains various programs intended to stress test the discovery of inductive predicates. These case studies include, e.g., conversion of a singly-linked list into a doubly-linked and then back to singly-linked list, or construction of two independent lists starting from the same node, which other tools may inaccurately over-approximate as a hierarchically nested list or a binary tree.

Another case study considers a call of `free()` on an embedded head that appears in real code if the head is placed at zero offset within the data node. Tools that ignore address aliasing of fields placed at the same offset, like Invader, mistakenly report such an operation as an error in the analysed program. Since Predator uses the offset-based description of list segments, it can easily cope with address aliasing.

We also provide a few case studies of lists where each node *optionally* owns some nested objects. Those may be incorrectly abstracted as nested lists if only usual list segments are considered, and in case the program does not really treat such objects as lists, it leads to spurious memory leaks or even non-termination of the analysis. Predator covers these cases by special abstract objects of length 0 or 1, which allows a more precise analysis and solves the problems with spurious errors and non-termination.

Across all our case studies, Predator acts fully automatically. There is no need to tell Predator what kind of data structures to look for. Given a C program, it simply returns the corresponding list of errors and warnings. In all but one of the mentioned tests, the time consumption was under 1.0s on Intel Core i5 3.33GHz. Moreover, for a vast majority of the tests, it was under 0.1s. The only exception was the Merge-Sort example, which took 7.8s to analyse. We are, however, not aware of any comparable tool that is able to analyse the same example faster.

5 Conclusion

We have presented Predator, a new separation logic based tool for analysing programs with dynamic linked data structures. Despite the tool is only at the beginning of its development, we have argued that it already offers many interesting features. In the future, the tool should be, e.g., enriched with some (preferably light-weight) support of non-pointer data (integers, arrays), extended to handle further classes of dynamic data structures, extended to handle C++ code (which the `gcc`-based front-end can easily handle), and so on. Since Predator is open source, GPL-licensed, and written such that its code is readable, collaboration on its further development is very well possible.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic Abstraction for Programs with Dynamic Memory Heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
2. 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)
3. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-abduction. In: Proc. of POPL 2009. ACM Press, New York (2009)
5. Distefano, D., Parkinson, M.: jStar: Towards Practical Verification for Java. In: Proc. of OOPSLA 2008, ACM Press, New York (2008)
6. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Sep. Logic. Tech. rep. FIT-TR-2011-02, FIT BUT (2011)
7. Sagiv, S., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. TOPLAS 24(3) (2002)
8. Shanmugasundaram, K.: Linux Kernel Linked List Explained (2005), `http://isis.poly.edu/kulesh/stuff/src/klist`
9. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.W.: On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London (2007)
10. 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)

SpaceEx: Scalable Verification of Hybrid Systems

Goran Frehse[1], Colas Le Guernic[2], Alexandre Donzé[1], Scott Cotton[1], Rajarshi Ray[1], Olivier Lebeltel[1], Rodolfo Ripado[1], Antoine Girard[3], Thao Dang[1], and Oded Maler[1]

[1] Verimag, CNRS / Université Grenoble 1 Joseph Fourier, 38610 Gières, France
[2] New York University CIMS, New York, NY 10012, USA
[3] Laboratoire Jean Kuntzmann, Université de Grenoble
goran.frehse@imag.fr

Abstract. We present a scalable reachability algorithm for hybrid systems with piecewise affine, non-deterministic dynamics. It combines polyhedra and support function representations of continuous sets to compute an over-approximation of the reachable states. The algorithm improves over previous work by using variable time steps to guarantee a given local error bound. In addition, we propose an improved approximation model, which drastically improves the accuracy of the algorithm. The algorithm is implemented as part of SpaceEx, a new verification platform for hybrid systems, available at `spaceex.imag.fr`. Experimental results of full fixed-point computations with hybrid systems with more than 100 variables illustrate the scalability of the approach.

1 Introduction

Hybrid systems are a class of mathematical models of dynamical systems admitting both discrete-event (logical) and continuous (numerical) dynamics. They consist of a transition system, augmented with real-valued state variables that evolve according to a particular differential equation in every discrete state (mode). Conditions on the values of these variables may trigger discrete transitions (mode switching). Naturally, the verification of hybrid systems requires ingredients taken from the classical verification of transition systems, augmented with new special techniques for doing verification-like operations (successor computation) on the continuous dynamics. Early tools [11,1] focused on relatively simple continuous dynamics in each discrete state, where the derivative of the continuous variables does not depend on their values. For such "linear" hybrid automata, the computation of successors in the continuous domain can be realized by linear algebra. Nevertheless, it turned out that switching between such simple continuous modes one can easily construct undecidability gadgets and hence the *exact* verification of hybrid systems turned out to be a dead end.

The second wave of hybrid verification tools [6,3,12] indeed abandoned exact computations and focused more on computing *approximations* of the reachable states for systems admitting less trivial continuous dynamics. Such techniques

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 379–395, 2011.

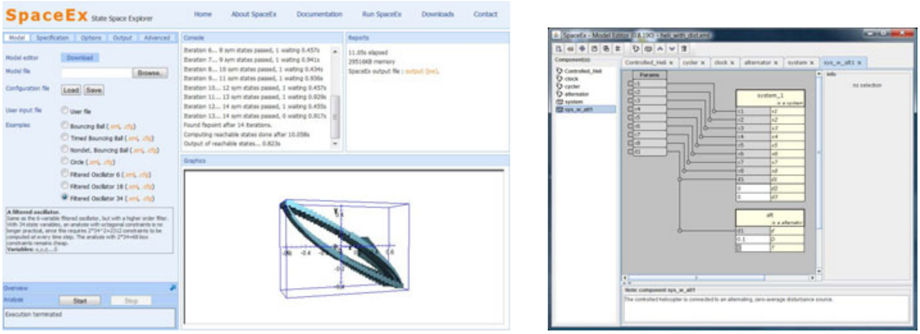
and tools could handle hybrid systems with continuous dynamics defined by linear differential equations with inputs. However, the size of systems that could be handled was modest, typically very few continuous state variables. Another thread in hybrid verification focuses on systems with a very large discrete state-space with very few continuous variables. Typically such models are aimed to verify the code of computerized control systems, with a very modest modeling of the external environment. The major preoccupation in these efforts is in combining the continuous part with techniques, such as BDD or SAT, for handling large discrete state-spaces [7,16].

In the last couple of years, an important breakthrough has been achieved in the reachability computation of the continuous part. A new algorithm based on a "lazy" representation of the reachable sets, first developed for the special case where reachable states are represented by Zonotopes [10] and then extended and crystalized using the more general representation via *support functions* [14,13,15], has dramatically increased the scope of linear systems that can be verified to several hundreds of state variables. Similar scale improvements have been reported recently on complementary approaches for handling linear systems [4]. Moreover, significant advances have been made on applying these linear techniques to nonlinear systems via "hybridization" (approximating a nonlinear system by a piecewise-affine one) [2], which have led to the verification of non-trivial nonlinear systems with about a dozen of state variables [8].

This paper is concerned with the transfer of such research achievements, typically obtained as theoretical results accompanied by a thesis-related prototype, into a robust and user-friendly toolset. It describes SpaceEx, a new extensible verification platform for hybrid systems, developed with systematic software engineering [9], implementing many of the above-mentioned developments, and featuring a web-based graphical user interface. As the reader will see, this transfer is not only about software engineering, modularity and user interface: it consists in improving and fine-tuning the algorithms to make them applicable to real-world problems. SpaceEx consists of three components:

- The *analysis core* is a command line program that takes a model file, a configuration file that specifies the initial states, the scenario and other options, and then analyzes the system and produces a series of output files.
- The *web interface*, shown in Fig. 1(a), is a graphical user interface with which one can comfortably specify initial states and other analysis parameters, run the analysis core, and visualize the output graphically. The web interface is browser-based, and accesses the analysis core via a web server, which may be running remotely or locally on a virtual machine.
- The *model editor*, shown in Fig. 1(b), is a graphical editor for creating models of complex hybrid systems out of nested components.

In this paper, we describe an efficient and scalable reachability algorithm, which builds on the one in [15], adapted specifically for maximum scalability. We present its extension to variable time steps, and propose an improved approximation model, which drastically improves the accuracy of the algorithm.



A filtered oscillator.
Same as the 6-variable filtered oscillator, but with a higher order filter. With 34 state variables, an analysis with octagonal constraints is no longer practical, since this requires 2*34^2=2312 constraints to be computed at every time step. The analysis with 2*34=68 box constraints remains cheap.
Variables: x,y,z,...,3

Execution terminated

```
Iteration 6... 8 sym states passed, 1 waiting 0.457s
Iteration 7... 9 sym states passed, 1 waiting 0.941s
Iteration 8... 10 sym states passed, 1 waiting 0.434s
Iteration 9... 11 sym states passed, 1 waiting 0.936s
Iteration 10... 12 sym states passed, 1 waiting 0.457s
Iteration 11... 13 sym states passed, 1 waiting 0.929s
Iteration 12... 14 sym states passed, 1 waiting 0.455s
Iteration 13... 14 sym states passed, 0 waiting 0.917s
Found fixpoint after 14 iterations.
Computing reachable states done after 10.058s
Output of reachable states... 0.823s
```

```
11.05s elapsed
29516KB memory
SpaceEx output file : output (txt).
```

Note: component sys_w_alt1
The controlled helicopter is connected to an alternating, zero-average disturbance source.

(a) Web interface

(b) Model editor

Fig. 1. Graphical user interfaces of the SpaceEx platform

Experimental results demonstrate the scalability of the algorithm and the performance of the tool. Unlike in classical verification there are no established benchmarks and reference tools for comparing high-dimensional hybrid reachability. We know of no tool that has been reported to treat systems of the dimensions in this paper. The examples used in this paper as well as the tool itself are available at `http://spaceex.imag.fr`.

The paper is structured as follows. Section 2 recalls hybrid automata, the basic reachability algorithm and data structures used in SpaceEx. Section 3 describes the variable time step algorithm and the new approximation model used to compute time elapse successor states. The computation of successor states of discrete transitions is presented in Sect. 4. Experimental results based on our implementation in SpaceEx are provided in Sect. 5, followed by some conclusions in Sect. 6. The proofs for this paper are available as an appendix at `http://www-verimag.imag.fr/~frehse/cav2011_appendix.pdf`.

2 Reachability of Hybrid Systems

2.1 Hybrid Automata

We model the interaction of discrete events and continuous, time-driven dynamics with a *hybrid automaton* [1]. A hybrid automaton $H = (\mathit{Loc}, \mathit{Var}, \mathit{Lab}, \mathit{Inv}, \mathit{Flow}, \mathit{Trans}, \mathit{Init})$ consists of a labeled graph that encodes the nondeterministic evolution of a finite set of continuous *variables* Var over time. In this paper, we associate each of the n variables with a dimension in $\mathbb{R}^n$. A vertex $l \in \mathit{Loc}$ of the graph is called a *location*. A *state* is a pair $(l, x) \in \mathit{Loc} \times \mathbb{R}^n$. In every state (l, x), the time-driven evolution of the continuous variables is given by the set of derivatives $\mathit{Flow}(l, x) \subseteq \mathbb{R}^n$. The edges of the graph are called *discrete transitions*. A transition $(l, \alpha, \mathit{Guard}, \mathit{Asgn}, l') \in \mathit{Trans}$, with label $\alpha \in \mathit{Lab}$ allows the system to jump from location l to location l', instantaneously modifying the values of the variables. A state (l, x) can jump to (l', x') according to the *guard* $Guard \subseteq \mathbb{R}^n$ and the *assignment* $Asgn(x) \subseteq \mathbb{R}^n$, i.e., if $x \in Guard$ and $x' \in Asgn(x)$. The system may only remain in a location l as long as the state is inside the *invariant* $Inv(l) \subseteq \mathbb{R}^n$. All behaviors originate from the *initial states* $Init \subseteq Loc \times \mathbb{R}^n$. In this paper, we consider $Flow(l)$ to be a continuous dynamics of the form

$$\dot{x}(t) = Ax(t) + u(t), \qquad u(t) \in \mathcal{U}, \tag{1}$$

where $x(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{U} \subseteq \mathbb{R}^n$ is a closed and bounded convex set. Transition assignments $Asgn$ are of the form

$$x' = Rx + w, \qquad w \in \mathcal{W}, \tag{2}$$

where R is a real-valued $n \times n$ matrix, and $\mathcal{W} \subseteq \mathbb{R}^n$ is a closed and bounded convex set of non-deterministic inputs.

An *execution* of the automaton is a sequence of discrete jumps and pieces of continuous trajectories according to its dynamics, and originates in one of the initial states. A state is *reachable* if an execution leads to it. We are concerned with computing the set of states that are reachable.

2.2 High-Level Reachability Algorithm

Our reachability algorithm is a classical fixed-point computation that operates on *symbolic states*. A symbolic state is a pair (l, Ω), where l is a location and Ω is a convex continuous set. For a set of symbolic states $\mathcal{R}$, let the *discrete post-operator* $\mathrm{post}_d(\mathcal{R})$ be the set of states reachable by a discrete transition from $\mathcal{R}$, and the *continuous post-operator* $\mathrm{post}_c(\mathcal{R})$ be the set of states reachable from $\mathcal{R}$ by letting an arbitrary amount of time elapse. The set of reachable states is the fixed-point of the sequence $\mathcal{R}_0 = \mathrm{post}_c(Init)$,

$$\mathcal{R}_{k+1} := \mathcal{R}_k \cup \mathrm{post}_c(\mathrm{post}_d(\mathcal{R}_k)). \tag{3}$$

In the following section we present how we represent convex continuous sets such that the post operators can be computed efficiently. The post operators themselves are presented in Sect. 3 and Sect. 4.

2.3 Support Functions and Template Polyhedra

Computing the image of the reachability post-operators for continuous sets is hard, in particular for the time elapse operator post_c. In [15], an efficient algorithm was proposed that uses support functions to represent convex continuous sets. The *support function* [5] of a closed and bounded continuous set $\mathcal{S} \subseteq \mathbb{R}^n$ assigns to any direction vector $\ell \in \mathbb{R}^n$ the value

$$\rho(\ell, \mathcal{S}) = \max_{x \in \mathcal{S}} \ell \cdot x.$$

The support function of a convex set $\mathcal{S}$ is an exact representation of the set, which is illustrated by the fact that $\mathcal{S} = \bigcap_{\ell \in \mathbb{R}^n} \{x \mid \ell \cdot x \leq \rho(\ell, \mathcal{S})\}$. Representing a convex set by its support function has the benefit that the majority of the set operations used in our algorithm can be implemented very efficiently:

- *linear map*: For a map $M \in \mathbb{R}^n \times \mathbb{R}^n$, $\rho(\ell, M\mathcal{S}) = \rho(M^\top \ell, \mathcal{S})$.
- *Minkowski sum*: For sets $\mathcal{S}_1$, $\mathcal{S}_2$, $\rho(\ell, \mathcal{S}_1 \oplus \mathcal{S}_2) = \rho(\ell, \mathcal{S}_1) + \rho(\ell, \mathcal{S}_2)$.
- *convex hull*: For sets $\mathcal{S}_1$, $\mathcal{S}_2$, $\rho(\ell, \mathrm{CH}(\mathcal{S}_1, \mathcal{S}_2)) = \max(\rho(\ell, \mathcal{S}_1), \rho(\ell, \mathcal{S}_2))$.

However, our algorithm requires two more operations, for which support functions are not efficient: intersection and deciding containment. For these operations, we use another set representation, *template polyhedra*, which are polyhedra with facets whose normal vectors are given a priori. Given a set $D = \{\ell_1, \ldots, \ell_m\}$ of vectors in $\mathbb{R}^n$ called *template directions*, a template polyhedron $\mathcal{P}_D \subseteq \mathbb{R}^n$ is a polyhedron for which there exist coefficients $b_1, \ldots, b_m \in \mathbb{R}$ such that

$$\mathcal{P}_D = \big\{x \in \mathbb{R}^n \mid \bigwedge_{\ell_i \in D} \ell_i \cdot x \leq b_i\big\}.$$

A template polyhedron can be represented by its coefficients b_i, which is particularly useful when working with sets of template polyhedra. In order to go from a support function representation of a set $\mathcal{S}$ to a template polyhedron, we use its *template hull* $\mathrm{TH}_D(\mathcal{S})$, which is the template polyhedron defined by coefficients $b_i = \rho(\ell_i, \mathcal{S})$. The support function of a polyhedron can be computed efficiently for a given direction ℓ using linear programming. In this paper, we consider:

- $2n$ *box* directions: $x_i = \pm 1$, $x_k = 0$ for $k \neq i$;
- $2n^2$ *octagonal* directions: $x_i = \pm 1$, $x_j = \pm 1$, $x_k = 0$ for $k \neq i$ and $k \neq j$;
- m *uniform* directions (as evenly as possible distributed over the unit sphere).

However, the algorithm supports a more general choice of directions, which remains to be investigated.

The use of both support functions and template hulls is justified by the fact that they are efficient for different operations, and both set representations are present in our implementation. Support functions are an exact and complete representation of convex sets – implemented as function objects, they can compute values for any direction. With template hulls, the directions D are fixed once and for all at the time of construction, and information for other directions is lost. By switching representations only when necessary (data-dependently) we remain as precise as possible.

3 Variable Time-Step Flowpipe Computation

We consider the affine continuous dynamics in (1), $\dot{x}(t) = Ax(t) + u(t)$, $u(t) \in \mathcal{U}$, where $x(0) \in \mathcal{X}_0$, and $\mathcal{U} \subseteq \mathbb{R}^n$ is a set of nondeterministic inputs. Let $\mathrm{Reach}_{t_1,t_2}(\mathcal{X})$ denote the reachable states starting from the set $\mathcal{X}$ with input set $\mathcal{U}$ in the time interval $[t_1, t_2]$,

$$\mathrm{Reach}_{t_1,t_2}(\mathcal{X}) = \{x(\tau) \mid t_1 \leq \tau \leq t_2, x(0) \in \mathcal{X}, x(t) \text{ satisfies } (1)\}. \qquad (4)$$

We compute a *flowpipe*, a sequence of continuous sets $\Omega_0, \ldots, \Omega_{N-1}$ that covers the reachable states up to time T (N depends on the chosen time steps).

Before we present the actual algorithm, we discuss how we take into account the invariant of the corresponding location of the hybrid automaton. In our implementation, we test at the k-th step whether Ω_k is entirely outside of the invariant, and stop the sequence once this is the case. Then we intersect the invariant with the computed Ω_k (see Sect. 4 for a discussion of the intersection operation). Note that this procedure may produce an over-approximation, as the invariant may block some of the trajectories starting in $\mathcal{X}_0$ without blocking them all. We include the invariant facet normals automatically in the template directions, so the result is usually of satisfactory precision. A variation of this algorithm with proper handling of invariants is presented in [14].

We now describe the variable time step algorithm. Given arbitrary time steps $\delta_0, \delta_1, \ldots$, we construct the sequence Ω_k that covers the set of reachable states. As we will show, each set Ω_k covers the reachable states in the time interval $[t_k, t_{k+1}]$, where $t_k = \sum_{i=0}^{k-1} \delta_i$. The algorithm is based on two functions $\Omega_{[0,\delta]}$ and Ψ_δ, called *approximation models*. They over-approximate the reachable states over a time interval $[0, \delta]$ as a function of $\delta, \mathcal{X}_0$, and $\mathcal{U}$:

$$\text{Reach}_{0,\delta}(\mathcal{X}_0) \subseteq \Omega_{[0,\delta]}(\mathcal{X}_0, \mathcal{U}), \qquad \text{Reach}_{\delta,\delta}(\{0\}) \subseteq \Psi_\delta(\mathcal{U}). \tag{5}$$

Each Ω_k is constructed by computing $\Omega_{[0,\delta_k]}$, which covers $\text{Reach}_{0,\delta_k}(\mathcal{X}_0)$, and then shifting this set forward in time so that it covers $\text{Reach}_{t_k,t_k+\delta_k}(\mathcal{X}_0)$. This is possible due to the following well-known property:

Lemma 1. $\text{Reach}_{t_k,t_k+\delta_k}(\mathcal{X}) = e^{At_k}\text{Reach}_{0,\delta_k}(\mathcal{X}) \oplus \text{Reach}_{t_k,t_k}(\{0\})$.

It therefore suffices to apply the linear map e^{At_k} (a constant matrix) to $\Omega_{[0,\delta_k]}$, and then to add $\text{Reach}_{t_k,t_k}(\{0\})$, which captures the influence of the inputs $\mathcal{U}$ up to time t_k. We over-approximate this summand with a sequence Ψ_k defined below:

$$\text{Reach}_{t_k,t_k}(\{0\}) \subseteq \Psi_k.$$

Given approximation models $\Omega_{[0,\delta]}$ and Ψ_δ, we compute the sequence Ω_k as follows, with $\Psi_0 = \{0\}$:

$$\begin{aligned} \Psi_{k+1} &= \Psi_k \oplus e^{At_k}\Psi_{\delta_k}(\mathcal{U}), \\ \Omega_k &= e^{At_k}\Omega_{[0,\delta_k]}(\mathcal{X}_0, \mathcal{U}) \oplus \Psi_k \end{aligned} \tag{6}$$

Now we formally state the main result of this section: Given the approximation models, which we will define in the next section, the sequence Ω_k covers the reachable set.

Proposition 1. *Given a sequence of time steps* $\delta_0, \ldots, \delta_{N-1}$ *with* $\sum_{i=0}^{N-1} \delta_i = T$, *the sequence* Ω_k *defined by (6) satisfies*

$$\text{Reach}_{0,T}(\mathcal{X}_0) \subseteq \bigcup_{k=0}^{N-1} \Omega_k. \tag{7}$$

Representing the sets Ψ_k and Ω_k by their support function, the operators used in (6) – linear map and Minkowski sum – can be computed efficiently as discussed in Sect. 2.3.

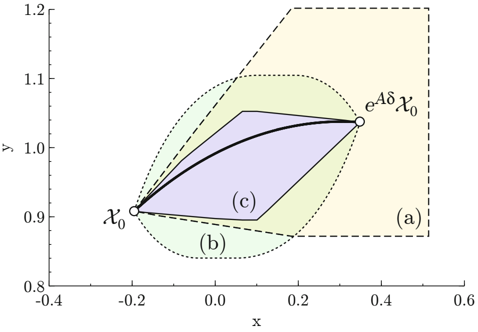


Fig. 2. Different approximation models for a segment of a circular trajectory, computed by SpaceEx: (a) using a first-order approximation of the ODE [15], (b) using a first-order approximation of the error of the linear interpolation between the states at time $t = 0$ and at $t = \delta$ [13], (c) the new model, which intersects a first-order approximation of the interpolation error going forward in time from $t = 0$ with one that goes backward from $t = \delta$.

In the next section, we present the new approximation model which we use to compute Ω_k as in (6). In Sect 3.2, we provide direction-wise error bounds on this computation, and discuss how to adapt the time steps in order to guarantee given error bounds.

3.1 Approximation Model

The approximation quality of the sequence Ω_k evidently hinges on the quality of the approximation model. In [15], an approximation model was proposed that uses the norms of A, $\mathcal{X}_0$ and $\mathcal{U}$ to bound the error of a first-order approximation of the solution of the state equations, see Fig. 2 for an illustration. If this allows one to establish an asymptotically optimal error over a given time interval, it is sometimes overly conservative in practice. In this section, we propose a new model, which is a strict improvement over the method in [15] if the norm used is the infinity norm. For other norms, it is possible to find cases for which the resulting sets are incomparable, but we have yet to find a pratical case where our approximation is worse. Similar to a model proposed in [13], it is based on a first-order approximation of the interpolation error between the reachable set at time 0 and at time δ. On top of that, it combines an approximation that goes forward in time with one that goes backward in time in order to further improve the accuracy.

Before presenting the model, we introduce the following notations. The *symmetric interval hull* of a set S, denoted $\boxdot(S)$, is $\boxdot(S) = [-\overline{|x_1|}; \overline{|x_1|}] \times \ldots \times [-\overline{|x_d|}; \overline{|x_d|}]$ where for all i, $\overline{|x_i|} = \sup\{|x_i| \mid x \in S\}$. Let $M = (m_{i,j})$ be a matrix, and $v = (v_i)$ a vector. We define as $|M|$ and $|v|$ the absolute values of M and v respectively, i.e., $|M| = (|m_{i,j}|)$ and $|v| = (|v_i|)$. These absolute values allow us to bound matrix-vector operations (component wise) without taking their norm.

The approximation model uses the following transformation matrices:

$$\Phi_1(A,\delta) = \sum_{i=0}^{\infty} \frac{\delta^{i+1}}{(i+1)!} A^i, \qquad \Phi_2(A,\delta) = \sum_{i=0}^{\infty} \frac{\delta^{i+2}}{(i+2)!} A^i. \tag{8}$$

If A is invertible, Φ_1 and Φ_2 can be computed as $\Phi_1(A,\delta) = A^{-1}\left(e^{\delta A} - I\right)$, $\Phi_2(A,\delta) = A^{-2}\left(e^{\delta A} - I - \delta A\right)$. Otherwise, they can be computed as submatrices of the block matrix

$$\begin{pmatrix} e^{A\delta} & \Phi_1(A,\delta) & \Phi_2(A,\delta) \\ 0 & I & I\delta \\ 0 & 0 & I \end{pmatrix} = \exp \begin{pmatrix} A\delta & I\delta & 0 \\ 0 & 0 & I\delta \\ 0 & 0 & 0 \end{pmatrix}.$$

We can now use these operators to obtain a precise over-approximation of $\mathrm{Reach}_{0,\delta}(\mathcal{X}_0)$ and $\mathrm{Reach}_{\delta,\delta}(\{0\})$. We start with a first-order approximation of the latter :

Lemma 2. *Let $\Psi_\delta(\mathcal{U})$ be the set defined by*

$$\Psi_\delta(\mathcal{U}) = \delta\mathcal{U} \oplus \mathcal{E}_\Psi(\mathcal{U},\delta), \tag{9}$$

$$\mathcal{E}_\Psi(\mathcal{U},\delta) = \boxdot\big(\Phi_2(|A|,\delta) \boxdot (A\mathcal{U})\big). \tag{10}$$

Then, $\mathrm{Reach}_{\delta,\delta}(0) \subseteq \Psi_\delta(\mathcal{U})$.

We now have discretized the differential equation, but we still have to over-approximate $\mathrm{Reach}_{0,\delta}(\mathcal{X}_0)$ with a function $\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})$. Our starting point is a linear interpolation between $\mathrm{Reach}_{0,0}$ and $\mathrm{Reach}_{\delta,\delta}$ using a parameter $\lambda = t/\delta$ representing normalized time. For each point in time t, $\mathrm{Reach}_{t,t} = \mathrm{Reach}_{\lambda\delta,\lambda\delta}$ is a convex set, for which we construct an over-approximation Ω_λ. Our over-approximation for the time interval $[0,\delta]$ is then the convex hull of all Ω_λ over $\lambda \in [0,1]$. Using a forward, respectively backward, interpolation leads to an error term proportional to λ, respectively $1-\lambda$. Intersecting both approximations gives the following result:

Lemma 3. *Let $\lambda \in [0,1]$, and let $\Omega_\lambda(\mathcal{X}_0,\mathcal{U},\delta)$ be the convex set defined by:*

$$\begin{aligned} \Omega_\lambda(\mathcal{X}_0,\mathcal{U},\delta) &= (1-\lambda)\mathcal{X}_0 \oplus \lambda e^{\delta A}\mathcal{X}_0 \oplus \lambda\delta\mathcal{U} \\ &\qquad \oplus \big(\lambda\mathcal{E}_\Omega^+(\mathcal{X}_0,\delta) \cap (1-\lambda)\mathcal{E}_\Omega^-(\mathcal{X}_0,\delta)\big) \oplus \lambda^2\mathcal{E}_\Psi(\mathcal{U},\delta), \textit{ with} \\ \mathcal{E}_\Omega^+(\mathcal{X}_0,\delta) &= \boxdot\left(\Phi_2(|A|,\delta) \boxdot \left(A^2\mathcal{X}_0\right)\right), \\ \mathcal{E}_\Omega^-(\mathcal{X}_0,\delta) &= \boxdot\left(\Phi_2(|A|,\delta) \boxdot \left(A^2 e^{\delta A}\mathcal{X}_0\right)\right), \\ \mathcal{E}_\Psi(\mathcal{U},\delta) &= \boxdot\left(\Phi_2(|A|,\delta) \boxdot (A\mathcal{U})\right). \end{aligned}$$

Then $\mathrm{Reach}_{\lambda\delta,\lambda\delta}(\mathcal{X}_0) \subseteq \Omega_\lambda(\mathcal{X}_0,\mathcal{U},\delta)$. *If we define $\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})$ as:*

$$\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U}) = \mathrm{CH}\Big(\bigcup_{\lambda\in[0,1]} \Omega_\lambda(\mathcal{X}_0,\mathcal{U},\delta)\Big), \tag{11}$$

then $\mathrm{Reach}_{0,\delta}(\mathcal{X}_0) \subseteq \Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})$.

$\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})$, as defined above, might seem hard to represent. In fact, its support function is not much harder to compute than the one of $\mathcal{X}_0$ and $\mathcal{U}$. Let

$$\begin{aligned}\omega(\mathcal{X}_0,\mathcal{U},\delta,\lambda) = (1-\lambda)\rho(\ell,\mathcal{X}_0) + \lambda\rho((e^{\delta A})^\top \ell,\mathcal{X}_0) + \lambda\delta\rho(\ell,\mathcal{U}) \\ + \rho\left(\ell,\lambda\mathcal{E}_\Omega^+ \cap (1-\lambda)\mathcal{E}_\Omega^-\right) + \lambda^2\rho(\ell,\mathcal{E}_\Psi). \quad (12)\end{aligned}$$

Since the signs of λ, $(1-\lambda)$, and λ^2 do not change on $[0,1]$, we have:

$$\rho(\ell,\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})) = \max_{\lambda\in[0,1]} \omega(\mathcal{X}_0,\mathcal{U},\delta,\lambda). \quad (13)$$

The support function of $\lambda\mathcal{E}_\Omega^+ \cap (1-\lambda)\mathcal{E}_\Omega^-$ can easily be expressed as a piecewise linear function of λ. For any λ, this set is a centrally symmetric box and its support function is:

$$\rho(\ell,\lambda\mathcal{E}_\Omega^+ \cap (1-\lambda)\mathcal{E}_\Omega^-) = \sum_{i=1}^{d} \min(\lambda e_i^+,(1-\lambda)e_i^-)|l_i|,$$

where e^+ and e^- are such that $\rho(\ell,\mathcal{E}_\Omega^+) = e^+ \cdot |l|$ and $\rho(\ell,\mathcal{E}_\Omega^-) = e^- \cdot |l|$. Thus we only have to maximize a piecewise quadratic function in one variable on $[0,1]$ after the evaluation of the support function of the sets involved.

3.2 Time Step Adaptation with Error Bounds

Time steps are generally hard to choose, and their value is rarely chosen in itself, but according to an expected precision. In order to efficiently choose our variable time step, we must be able to evaluate the error introduced by time discretization. In our error calculation, we do not bound the error introduced at each step, but the overall error introduced since the beginning of the current continuous evolution. We must take into account the accumulation of errors, not done carefully we might exhaust our error tolerance before the end of the computation and become unable to advance in time without exceeding it.

Our choice of error bound $\varepsilon(\ell)$ is the difference between the computed support function and the support function of the reachable set at time t. For each set Ω_k we define

$$\varepsilon_{\Omega_k}(\ell) = \rho\left(\ell,\Omega_k\right) - \rho\left(\ell,\mathrm{Reach}_{t_k,t_{k+1}}(\mathcal{X}_0)\right) \quad (14)$$

The total approximation error is then

$$\varepsilon(\ell) = \max_{k\in 0,\ldots,N-1} \varepsilon_{\Omega_k}(\ell).$$

Note that $\mathrm{Reach}_{t_k,t_{k+1}}(\mathcal{X}_0)$ is generally not a convex set, and by over-approximating it with the convex set Ω_k we incur an additional error that is not captured by $\varepsilon_{\Omega_k}(\ell)$. The bound $\varepsilon(\ell)$ allows us to decide whether or not the reachable set satisfies a linear constraint $\ell\cdot x \leq b$ (e.g., whether the states surpass a certain threshold) with an uncertainty margin of $\varepsilon(\ell)$.

To compute $\varepsilon_{\Omega_k}(\ell)$, we must take into account the error for $\Psi_\delta(\mathcal{U})$ defined as in Lemma 2 and $\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})$ defined as in Lemma 3. Let

$$\varepsilon_{\Psi_\delta(\mathcal{U})}(\ell) = \rho\left(\ell, \Psi_\delta(\mathcal{U})\right) - \rho\left(\ell, \mathrm{Reach}_{\delta,\delta}(\{0\})\right), \tag{15}$$

$$\varepsilon_{\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})}(\ell) = \rho(l, \Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})) - \rho(l, \mathrm{Reach}_{0,\delta}(\mathcal{X}_0)). \tag{16}$$

Lemma 4. *For any ℓ in $\mathbb{R}^n$:*

$$\varepsilon_{\Psi_\delta(\mathcal{U})}(\ell) \leq \rho(\ell, \mathcal{E}_\Psi(\mathcal{U},\delta)) + \rho\left(\ell, -A\Phi_2(A,\delta)\mathcal{U}\right) \tag{17}$$

$$\varepsilon_{\Omega_{[0,\delta]}(\mathcal{X}_0,\mathcal{U})}(\ell) \leq \max_{\lambda\in[0,1]} \Big\{\rho\Big(l, \left(\lambda\mathcal{E}^+_\Omega(\mathcal{X}_0,\delta) \cap (1-\lambda)\mathcal{E}^-_\Omega(\mathcal{X}_0,\delta)\right)\Big) + \lambda^2\rho(l, \mathcal{E}_\Psi(\mathcal{U},\delta)) + \lambda\rho\left(\ell, -A\Phi_2(A,\delta)\mathcal{U}\right)\Big\}. \tag{18}$$

Computing the support function of Ω_k as defined by (6) and applying the above lemmas as well as Lemma 1, we obtain $\varepsilon_{\Omega_k}(\ell)$ as follows.

$$\begin{aligned} \varepsilon_{\Psi_{k+1}}(\ell) &= \varepsilon_{\Psi_k}(\ell) + \varepsilon_{\Psi_{\delta_k}(\mathcal{U})}(e^{At_k\top}\ell), \\ \varepsilon_{\Omega_k}(\ell) &= \varepsilon_{\Omega_{[0,\delta_k]}(\mathcal{X}_0,\mathcal{U})}(e^{At_k\top}\ell) + \varepsilon_{\Psi_k}(\ell). \end{aligned} \tag{19}$$

Given the above error bounds, one can adapt the time steps during the computation of the sequence such that $\varepsilon(\ell)$ is kept arbitrarily small. The problem lies in the error $\varepsilon_{\Psi_k}(\ell)$, which accumulates with k, so that an a-posteriori refinement would require the whole squence to be recomputed. We therefore use the following simple heuristic to compute the sequence of $\rho(\ell,\Omega_k)$ for a given template direction ℓ such that $\varepsilon(\ell) \leq \hat{\varepsilon}$ for a given error bound $\hat{\varepsilon} \in \mathbb{R}^{>0}$. Instead of computing $\rho(\ell,\Omega_k)$ directly, we first compute the whole sequence $\rho(\ell,\Psi_k)$, then the sequence $\rho(e^{At_k\top}\ell, \Omega_{[0,\delta_k]}(\mathcal{X}_0,\mathcal{U}))$, and only then combine them to get the sequence $\rho(\ell,\Omega_k)$. This separation allows us to choose a separate time step for each sequence, adapting the error as necessary. Additionally, by computing the sequences one after another, the last one can pick up the slack in the error bound of the first sequence.

In the following we suppose that $\hat{\varepsilon}$ is distributed a-priori on both sequences, so that we have $\hat{\varepsilon}_\Omega$ and $\hat{\varepsilon}_\Psi$ in $\mathbb{R}^{>0}$ with $\hat{\varepsilon} = \hat{\varepsilon}_\Omega + \hat{\varepsilon}_\Psi$. This distribution can be established, e.g., by a prior coarse-grained run with a large error bound, or by a run with large fixed time steps. Because the error of Ψ_k accumulates with k, it is chosen (somewhat arbitrarily) to remain below a linearly increasing bound. The error of $e^{At_k}\Omega_{[0,\delta_k]}(\mathcal{X}_0,\mathcal{U})$ does not depend on previous computation steps, so it can be adapted on the fly to meet the required bound. We proceed as follows:

1. Compute $\rho(\ell,\Psi_k(\mathcal{U}))$ such that $\varepsilon_{\Psi_k}(\ell) \leq \frac{t_k^\Psi}{T}\hat{\varepsilon}_\Psi$. At each step k, we must find a δ_k^Ψ, ideally the biggest, such that:

$$\varepsilon_{\Psi_k}(\ell) + \varepsilon_{\Psi_{\delta_k^\Psi}(\mathcal{U})}(e^{At_k^\Psi\top}\ell) \leq \frac{t_k^\Psi + \delta_k^\Psi}{T}\hat{\varepsilon}_\Psi$$

Finding δ_k^{Ψ} is possible because $\varepsilon_{\Psi_\delta(\mathcal{U})}(e^{At^\top}\ell) = O(\delta^2)$. First, we fix an initial time step δ_{-1}^{Ψ}. Then, at each step k, we start a dichotomic search from δ_{k-1}^{Ψ} along the δ for which sets and matrices involved in the computation of $\Psi_\delta(\mathcal{U})$ have already been evaluated, trying new values only when necessary.

2. Compute $\rho(e^{At_k^{\Omega\top}}\ell, \Omega_{[0,\delta_k^{\Omega}]}(\mathcal{X}_0,\mathcal{U}))$ such that

$$\varepsilon_{\Omega_{[0,\delta_k^{\Omega}]}(\mathcal{X}_0,\mathcal{U})}(e^{At_k^{\Omega\top}}\ell) + \varepsilon_{\Psi_{i(k)}}(\ell) = \varepsilon_{\Omega_k}(\ell) \leq \hat{\varepsilon}$$

where $i(k)$ is such that $t_{i(k)-1}^{\Psi} < t_k^{\Omega} \leq t_{i(k)}^{\Psi}$. This can be done with a dichotomic search over the sequence of t_k^{Ψ} already computed for $\rho(\ell, \Psi_k(\mathcal{U}))$. If there is a k such that $\delta_{i(k)}^{\Psi}$ must be further refined, then for the newly introduced indices k_j, we have $i(k_j) = i(k)$.

To combine the above two sequences into $\rho(\ell, \Omega_k)$, we use the sequence of time steps δ_k^{Ω}. If this introduces new times t_k^{Ω} in the sequence of t_k^{Ψ}, we can compute the missing values for $\rho(\ell, \Psi_k(\mathcal{U}))$ by starting from $t_{i(k)-1}^{\Psi}$. This does not trigger the recomputation of $\rho(\ell, \Psi_k(\mathcal{U}))$ for other time points since the sequence $\varepsilon_{\Psi_k}(\ell)$ is increasing.

4 Computing Transition Successors

Each flow-pipe that is created by the time elapse step is passed to the computation of transition successors. States that take the transition must satisfy the guard, are then mapped according to the assignment and the result must satisfy the invariant of the target location. Consider a transition T with guard $\mathcal{G}$, assignment *Asgn*, and whose target location has the invariant $\mathcal{I}^+$. Let $Post_{Asgn}(\mathcal{X})$ be the set of states that result from mapping a continuous set $\mathcal{X}$ according to *Asgn*. Then for a set of states $\mathcal{X}$, the set of successor states is given by

$$\mathrm{post}_d(T, \mathcal{X}) = Post_{Asgn}(\mathcal{X} \cap \mathcal{G}) \cap \mathcal{I}^+.$$

We now discuss how the operations for this image computation, intersection and assignment, can be carried out efficiently. Then we present a method to decrease the number of convex sets produced by the successor computation so that an exponential blowup in the number of sets can be avoided.

Intersection. Since intersection with support functions is hard, we compute intersection on the template hull, say $\mathcal{P}_D = \mathrm{TH}_D(\mathcal{X})$. If $\mathcal{X}$ is a set of convex sets, the intersection is performed separately on each convex set. If $\mathcal{G}$ is a polyhedron in constraint form whose constraint normals are template directions, then this operation can be carried out very efficiently by taking the minimum of the template coefficients:

$$\mathcal{P}_D \cap \mathcal{G} = \big\{x \in \mathbb{R}^n \mid \bigwedge_{\ell_i \in D} \ell_i \cdot x \leq \min(b_i^{\mathcal{X}}, b_i^{\mathcal{G}})\big\}.$$

Assignment. Recall that according to (2) transition assignments are of the form $x' = Rx + w, w \in \mathcal{W}$, where $\mathcal{W} \subseteq \mathbb{R}^n$ is a convex set of non-deterministic inputs. In the general case, the assignment operator is therefore

$$Post_{Asgn}(\mathcal{X}) = R\mathcal{X} \oplus \mathcal{W},$$

and can be computed efficiently using support functions. If the assignment is invertible and deterministic, i.e., R is invertible and $\mathcal{W} = \{w_0\}$ for some constant vector w_0, the exact image can be computed efficiently on the polyhedron.

Clustering. Each flow-pipe consists of a possibly large number of convex sets that cover the actual trajectories. When we compute the successor states for a transition, each of these convex sets spawns its own flow-pipe in the next time elapse computation. This may multiply the number of sets with each iteration, leading to an explosion in the number of sets and slowing the analysis down to a stall. To avoid this effect and speed up the analysis, we apply what we call *clustering.* Given a hull operator, clustering reduces the number of sets by replacing groups of these sets with a single convex set, their hull. We use the following clustering algorithm for a given hull operator *HULL*. Let the *width* of $\mathcal{P}_1, \ldots, \mathcal{P}_z$ with respect to a direction $\ell \in D$ be

$$\delta_{\mathcal{P}_1,\ldots,\mathcal{P}_z}(\ell) = \max_{i=1..z} \rho(\ell, \mathcal{P}_i) - \min_{i=1..z} \rho(\ell, \mathcal{P}_i). \tag{20}$$

Given $\mathcal{P}_1, \ldots, \mathcal{P}_z$ and a *clustering factor* of $0 \leq c \leq 1$, the clustering algorithm produces a set of polyhedra $Q_1, \ldots, Q_r$, $r \leq z$, as follows:

1. Let $i = 1$, $r = 1$, $Q_r := \mathcal{P}_i$.
2. While $i \leq z$ and $\forall \ell \in D : \delta_{Q_r,\mathcal{P}_i}(\ell) \leq c\delta_{\mathcal{P}_1,\ldots,\mathcal{P}_z}(\ell)$,
   $Q_r := HULL(Q_r, \mathcal{P}_i)$, $i := i + 1$.
3. If $i \leq z$, let $r := r + 1$, $Q_r := \mathcal{P}_i$. Otherwise, stop.

We consider two hull operators: template hull, which is fast but very overapproximative, and convex hull, which is precise but slower. It can be advatageous to combine both, as illustrated by the following example:

Example 1. Consider the 8-variable filtered oscillator presented in Sect. 5.1. At the first discrete state change alone, 57 convex sets can take the transition. Without clustering, the computation is not feasible, as these sets would spawn 57 new flowpipes, and similarly for their successors. Template hull clustering with $c_{TH} = 0.3$ produces three sets and results in a total runtime of 11.5s. With $c_{TH} = 1$ it produces a single set and takes 3.36s, but with a large overapproximation. Convex hull clustering by itself with $c_{CH} = 1$ is very precise but takes 8.19s. Combining both with $c_{TH} = 0.3, c_{CH} = 1$ takes 3.64s without a noticeable loss in accuracy.

5 Experimental Results

To demonstrate the scalability of our algorithm and the performance of the tool SpaceEx, we present the following experiments. The first system is a simple

Table 1. Performance results for the filtered oscillator benchmark, varying the number of variables in the system. The time step is $\delta = 0.05$, applying template hull clustering with $c_{TH} = 0.3$ followed by convex hull clustering with $c_{CH} = 1$. Indicated are the runtime, memory and iterations required to compute a fixed-point, and the largest error for any of the directions in any of the time steps

Variables	Time [s]	Mem. [MB]	Iter.	Error
18	2,0	9,3	9	0,010
34	9,1	20,2	13	0,010
66	77,3	50,3	23	0,013
130	1185,6	194,3	39	0,030
198	7822,5	511,0	57	0,074

Box constraints

Variables	Time [s]	Mem. [MB]	Iter.	Error
2	0,7	11,8	6	0.009
4	1,4	11,8	6	0.025
6	4,7	13,3	6	0.025
10	33,0	23,0	7	0.025
18	538,4	67,9	10	0.025

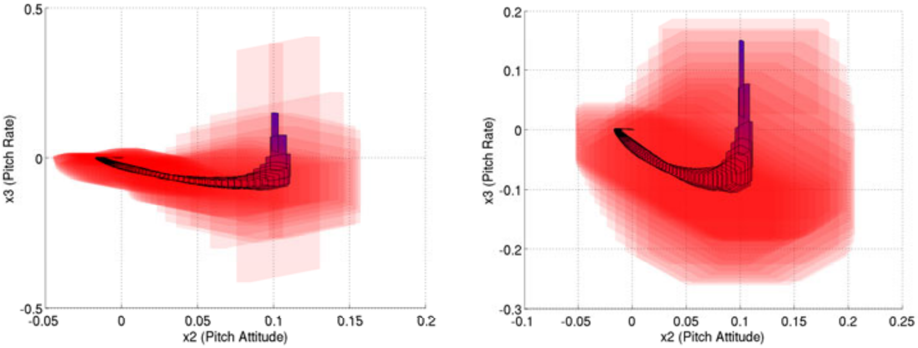
Octagonal constraints

parameterized system which we use to empirically measure the complexity of our algorithm. The second system is a multivariable continuous control system with complex, tightly coupled dynamics. It illustrates the faithfulness and accuracy of the continuous part of the algorithm. For lack of space we do not present the model equations, but instead refer the reader to the SpaceEx model files, which are available at `http://spaceex.imag.fr`.

5.1 Filtered Oscillator Benchmark

To measure the performance of our approach, we use a simple parameterized switched oscillator system whose complexity is increased by adding a series of first-order filters to the output x of the oscillator. The filters smooth x, producing a signal z whose amplitude diminishes as the number of filters increases. Note that the dynamics are rather simple, as the filter variables are only weakly coupled with one another. The oscillator is an affine system with variables x, y that switches between two equilibria in order to maintain a stable oscillation, which together with k filters yields a parameterized system with $k+2$ continuous variables and two locations. One location has the invariant $x \geq -1.4y$, the other $x \leq -1.4y$, and the guards consist of the boundaries of the invariants.

To empirically measure how the complexity depends on the n variables of the system and the m template directions, we run experiments varying just n, just m, and both. *Fixed n:* The average time for a successor computation (discrete followed by continuous) for the 6-variable system over m uniform directions, $m = 8, \dots, 256$, shows a root mean square (RMS) tendency of $O(m^{1.7})$. *Fixed m:* The average time for a successor computation with 200 uniform directions with $n = 6, \dots, 16$ shows an RMS tendency of $O(n)$. Table 1 shows the complete runtime of a fixed-point computation for box and uniform directions for the system with up to 198 variables. The RMS tendency is $O(n^{2.7})$ for box directions and $O(n^{4.7})$ for octagonal directions, which confirms the results for fixed n and fixed m.



(a) $\delta t = 0.05$ for both

(b) forw: $\delta t = 0.005$, interpfb: $\delta t = 0.05$

Fig. 3. The reachable states of the 28-variable controlled helicopter system in the plane (x_2, x_3) (corresponding to roll attitude and roll rate), computed with octagonal constraints. We compare the new error scheme (interfb), shown in dark blue, with that of [13] (a) and of [15] (b), shown in bright red.

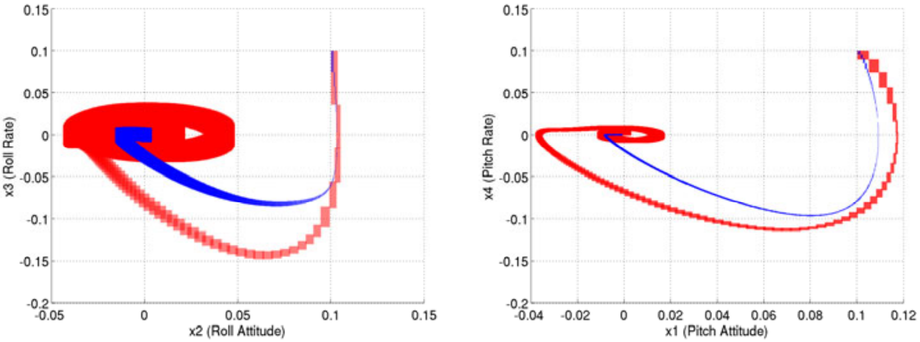
Table 2. Error models comparison with fixed step. The error model introduced in this paper (interpfb) clearly outperforms that proposed in [15] (forward) and the one proposed in [13] (interp). Memory is indicated in MBs.

	forward			interp			interpfb		
δt	Mem.	Time [s]	Error	Mem.	Time [s]	Error	Mem.	Time [s]	Error
0.05	9.44	1.48	9.67e+22	9.61	1.60	16.1	9.59	1.65	2.95
0.01	10.5	7.09	3.85e+5	10.5	7.60	0.191	10.5	8.16	0.178
5e-3	10.3	14.1	2.47e+3	10.2	15.2	4.37e-2	12.6	15.8	2.82e-2
1e-3	23.1	71.1	12.4	18.4	76.7	1.59e-3	18.5	78.7	1.07e-3
5e-4	27.9	142	2.56	27.9	155	3.89e-4	28.2	157	2.66e-4

5.2 Helicopter Controller

To measure the performance of our algorithm for complex dynamics, we analyze the helicopter controller from [17]. We analyze the controlled plant, a 28-dimensional continuous linear time-invariant (LTI) system. The plant is a small disturbance model of a helicopter, given as an 8-dimensional LTI system. The controller we examine is an $\mathcal{H}_\infty$ mixed-sensitivity design for rejecting atmospheric turbulence, given as a 20-dimensional LTI system. Figure 3 shows the increased accuracy of our new approximation model with respect to previous models. Tables 2 and 3 show the performance results with fixed time steps and with variable time steps, each for the different error models.

In [17], two different controllers are designed for the helicopter, one of which is specifically tuned for disturbance rejection. Letting the rotor collective be a nondeterministic input in the interval $[-1, 1]$, we compute the reachable states in 5s for one controller and in 14s for the other, as shown in Fig. 4.



(a) Comparison of roll stabilization

(b) Comparison of pitch stabilization

Fig. 4. Comparison of two disturbance rejection models. Reachable sets with nondeterministic inputs for the helicopter example for the two disturbance rejection models compared in [17] ($T = 20$, method interfb and error tolerance of 0.1 for both). This confirms that the better disturbance rejection model proposed (in blue) actually stabilizes the system faster. However, in [17], this analysis was based only on several simulations.

Table 3. Variable step performance. The variable step implementation outperforms a fixed step scheme even in the *ideal case*, i.e., with the best error model and assuming we know in advance the optimal time step δt to satisfy the error bound. This is always true for the number of steps taken and the slightly higher computational time for some case is explained by frequent changes in choice of the time step.

	Ideal fixed step (interpfb)			var step (interp)		var step (interpfb)	
Err. req.	nb steps	δt used	Time [s]	nb steps	Time [s]	nb steps	time
1	1500	0.02	11.68	1475	12.0	974	8.359
0.1	3418	8.78e-3	26.6	4334	33.9	2943	31.2
0.01	11539	2.6e-3	90.3	14070	108	9785	77.9
1e-3	44978	6.67e-4	351	39152	301	27855	234
1e-4	101352	2.96e-4	902	85953	688	64315	811

6 Conclusions

The reachability of hybrid systems is recognized as be a hard problem, and research efforts to find a scalable approach have been going on for more than two decades. In this paper, we presented a variable time-step extension of a scalable time-elapse algorithm and proposed an improved, highly accurate approximation model for it. Together with techniques to efficiently compute transition successors that avoid the well-known problem of an explosion in the number of sets that the algorithm needs to propagate, we have implemented the algorithm in a new tool called SpaceEx. In our experiments, SpaceEx can handle hybrid systems with affine dynamics and nondeterministic inputs with more than 100 variables. Further research is needed to automatically find suitable template directions to increase the accuracy of the approach. SpaceEx and the examples from this paper are available at `http://spaceex.imag.fr`.

Acknowledgments. This research was supported by the European Commission under grant INFSO-ICT-224249, the U.S. National Science Foundation under grant number CCF-0926181, and the French National Research Agency under grant for the Syne2Arti project.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Inf. 43(7), 451–476 (2007)
3. Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, p. 20. Springer, Heidelberg (2000)
4. Asarin, E., Dang, T., Maler, O., Testylier, R.: Using redundant constraints for refinement. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 37–51. Springer, Heidelberg (2010)
5. Bertsekas, D.P., Nedic, A., Ozdaglar, A.E.: Convex Analysis and Optimization. Athena Scientific, Belmont (2003)
6. Chutinan, A., Krogh, B.H.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 76–90. Springer, Heidelberg (1999)
7. Damm, W., Disch, S., Hungar, H., Jacobs, S., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact state set representations in the verification of linear hybrid systems with large discrete state space. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 425–440. Springer, Heidelberg (2007)
8. Dang, T., Le Guernic, C., Maler, O.: Computing reachable states for nonlinear biological models. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 126–141. Springer, Heidelberg (2009)
9. Frehse, G., Ray, R.: Design principles for an extendable verification tool for hybrid systems. In: ADHS (2009)
10. Girard, A., Le Guernic, C., Maler, O.: Efficient computation of reachable sets of linear time-invariant systems with inputs. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 257–271. Springer, Heidelberg (2006)
11. Henzinger, T., Ho, P.-H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer 1, 110–122 (1997)
12. Kurzhanski, A., Varaiya, P.: Reachability analysis for uncertain systems—the ellipsoidal technique. Dynamics of Continuous, Discrete and Impulsive Systems Series B: Applications and Algorithms 9(3b), 347–367 (2002)
13. Le Guernic, C.: Reachability analysis of hybrid systems with linear continuous dynamics. PhD thesis, Université Grenoble 1 - Joseph Fourier (2009)

14. Le Guernic, C., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 540–554. Springer, Heidelberg (2009)
15. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Analysis: Hybrid Systems 4(2), 250–262 (2010)
16. Scholl, C., Disch, S., Pigorsch, F., Kupferschmid, S.: Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 383–397. Springer, Heidelberg (2009)
17. Skogestad, S., Postlethwaite, I.: Multivariable Feedback Control: Analysis and Design. John Wiley & Sons, Chichester (2005)

From Cardiac Cells to Genetic Regulatory Networks

Radu Grosu[1], Gregory Batt[2], Flavio H. Fenton[3], James Glimm[4],
Colas Le Guernic[5], Scott A. Smolka[1], and Ezio Bartocci[1,4]

[1] Dept. of Comp. Sci., Stony Brook University, Stony Brook, NY, USA
[2] INRIA, Le Cesnay Cedex, France
[3] Dept. of Biomed. Sci., Cornell University, Ithaca, NY
[4] Dept. of Appl. Math. and Sta., Stony Brook University, Stony Brook, NY, USA
[5] Dept. of Comp. Sci., New York University, New York, NY

Abstract. A fundamental question in the treatment of cardiac disorders, such as tachycardia and fibrillation, is under what circumstances does such a disorder arise? To answer to this question, we develop a multiaffine hybrid automaton (MHA) cardiac-cell model, and restate the original question as one of identification of the parameter ranges under which the MHA model accurately reproduces the disorder. The MHA model is obtained from the minimal cardiac model of one of the authors (Fenton) by first bringing it into the form of a canonical, genetic regulatory network, and then linearizing its sigmoidal switches, in an optimal way. By leveraging the Rovergene tool for genetic regulatory networks, we are then able to successfully identify the parameter ranges of interest.

1 Introduction

A fundamental question in the treatment of cardiac abnormalities, such as ventricular tachycardia and fibrillation (see Fig. 1(a) and [7]), is under what conditions does such a disorder arise? To answer this question, experimentation performed in vitro or in vivo is nowadays complemented with the mathematical modeling, analysis and simulation of (networks of) cardiac cells [6]. Among the myriad of existing mathematical models, differential-equation models of reaction-diffusion type (DEMs) are arguably the most popular.

The past two decades have witnessed the development of increasingly sophisticated DEMs [9], which unravel in great detail the underlying cellular processes [17,22,24,14]. Such models are essential in the understanding of the intrinsic ionic mechanisms, and in the development of novel treatment strategies. However, they also have two significant drawbacks: 1) They often contain too many parameters to be reliably and robustly identified from experimental data. 2) They are often too complex to render their formal analysis or even simulation tractable. We refer to such models as *detailed ionic models* (DIMs).

Approximation is a well-established technique in science and engineering for dealing with complexity. In DEMs, where reaction is typically much faster than diffusion, one may use time-scale approximation techniques, to systematically

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 396–411, 2011.

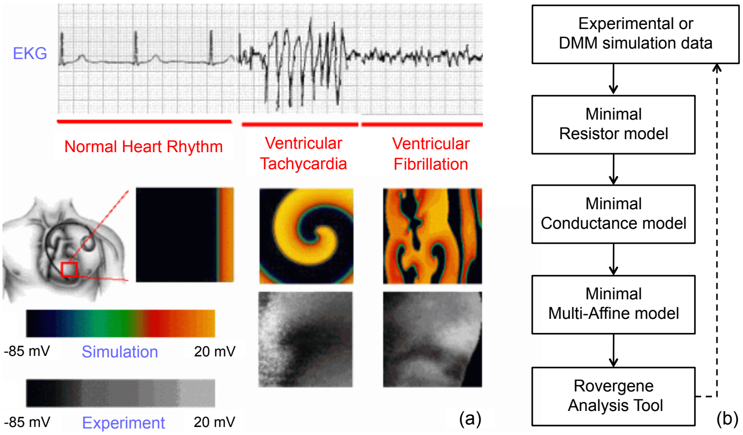


Fig. 1. (a) Emergent behavior in cardiac-cell networks. Top: Electrocardiogram. Middle and bottom: Simulation and experimental mappings of spiral waves of electrical activity occurring in the heart during tachycardia and fibrillation. (b) Overview of our approach.

eliminate fast transient regimes and compensate for their elimination [4]. For example, in enzymatic reactions, a substrate reacts very quickly with an enzyme to produce a compound, which subsequently, and much more slowly, breaks down into a product of the reaction and the enzyme itself. In this case, one may use the so-called *quasi-steady-state assumption* to eliminate the fast reaction and derive a sigmoidal dependence of the product concentration rate on the log of the substrate concentration, called the Michaelis-Menten equation [19,23].

Similar to the step (or Heaviside) switches used in digital-computer models, sigmoidal switches (dependencies) occur everywhere in biological models: from molecular to cellular models, and from organ to population models [25,15]. In most cases, they are the result of a time-scale approximation, applied to the associated DEMs. Unlike in digital-computer models, however, the switching speed of sigmoids plays an important role. Biology is more sophisticated!

DEMs with state variables whose rate of change is controlled with sigmoidal switches are still intractable from an analysis point of view. Research of qualitative properties in genetic regulatory networks overcomes this problem by approximating sigmoids with either steps or with ramps [12,18,3,8]. This leads to a piecewise-affine, or piecewise-multiaffine model, respectively. In such models, the dynamics within a hyper-rectangular region is completely determined by the dynamics of its corners, enabling model analysis through the use of powerful discrete abstraction techniques coupled with model-checking techniques [3].

In prior work, one of the authors (Fenton) co-developed an extremely versatile electrical model for cardiac cells involving just 4 state variables and 26 parameters [5]. For reasons to be made clear, we refer to this model as the *minimal resistor model* (MRM). After its parameters are identified from either experimental data or from DIM-based simulation results, the MRM is then able to accurately reproduce the desired behavior [5]. In fact, the MRM identified from experimental data reproduces the experimentally mesoescopic behavior with greater accuracy than any of the DIMs. Moreover, its simulation speed is orders of magnitude faster than that any of the DIM simulations [5].

The success of the MRM relies on a time-scale-like approximation: the large variety of currents traversing the cell membrane are lumped together into three currents: the fast input current, the slow input current, and the slow output current. These currents are regulated by three gate variables, which together with the voltage, define the MRM's state variables [5]. The lumping process is akin to removing and compensating for fast components [23,4].

In the MRM context, one may restate the cardiac-disorder question as follows: what are the parameter ranges for which the MRM accurately reproduces cardiac abnormalities? Once these ranges are identified, one may exploit the correspondence between the MRM and DIM models to infer the corresponding parameter ranges in DIMs, and the molecular relevance of the DIMs to target treatment strategies to the components responsible for the disorder.

Despite its simplicity compared to DIM, the MRM is still intractable from an analysis point of view. Its electrical formulation not only uses sigmoidal switches to control the gating variables, but also uses them to model gated resistors. As such, sigmoids occur both as numerators and denominators in the state equations. As part of our effort to simplify the MRM, we prove that *sigmoids are closed under the reciprocal operation*. This allows us to bring the MRM to a canonical form, which we call the *minimal conductance model* (MCM). Intriguingly, the MCM is of the form of a *genetic regulatory network model* (GRM). Hence, this transformation not only exposes the unity of DEMs, but also allows us to leverage tools developed for GRMs for the analysis of cardiac models.

In GRMs, as well as in MCMs, slow (or shallow) sigmoidal switches cannot be approximated with steps or ramps without considerably distorting the original behavior. We therefore approximate such sigmoids with a succession of ramps, the number of which depends on the desired accuracy. For analysis purposes, it is critical to minimize the number of ramps used and to avoid arbitrary choices. We therefore adapt and extend a dynamic programming technique [21], originally intended for the optimal approximation of digital curves, to find the optimal number of segments (typically of different length) that minimizes, for all sigmoids simultaneously, a sigmoidal-linearization error. This results in a *hybrid-automaton model with multiaffine behavior in each mode* (MHA).

By recasting the intractable parameter-range identification problem for MCMs in terms of MHAs, we now have a tractable problem. Moreover, certain MHA parameter-range identification problems can be seen as GRM parameter-range identification problems: find the parameter ranges that lead to a robust behavior satisfying a given temporal logic property [3]. Hence, for these disorders, we can leverage tools already developed for GRMs to address the MHA problem.

The particular cardiac-disorder question addressed in this paper is *under what circumstances may cardiac-cell excitability be lost?* A region of unexcitable cells can be responsible for ventricular tachycardia or fibrillation: the region becomes an obstacle to a propagating electrical wave, triggering a spiral rotation of the

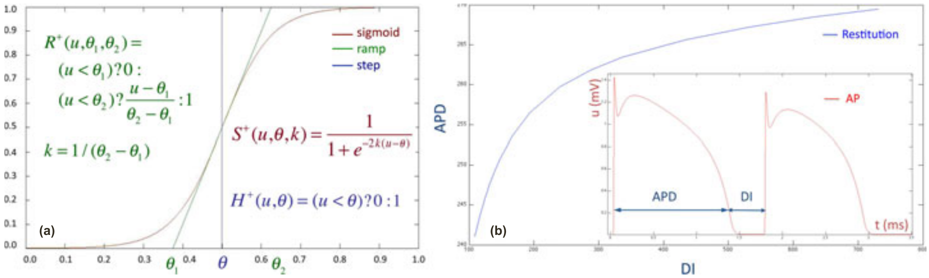


Fig. 2. (a) Threshold-based switching functions. (b) Action potential duration (APD), diastolic interval (DI), and restitution at 10% of the maximum value of the AP.

wave (tachycardia); the spiral may then break up into a disordered collection of spirals (fibrillation). Studying the parameter ranges for which cardiac cells loose excitability, and identifying the responsible ionic processes, is thus an important question in the treatment of cardiac disorders. We formulate loss of excitability as an LTL formula. The Rovergene tool [3], co-developed by co-author Batt, is then able to automatically infer nontrivial ranges for the MHA parameters, such that the MHA satisfies this formula. To the best of our knowledge, this is the first automated parameter-range estimation result for cardiac cells.

Our approach is summarized in Fig. 1(b), and the rest of the paper is organized accordingly. Section 2 introduces biological switches and their formal description. Section 3 reviews the MRM. In Section 4, we transform the MRM to an MCM, which is linearized in Section 5. Section 6 considers the parameter-range-identification problem. Section 7 concludes and discusses future work.

2 Biological Switching

As discussed in Section 1, biological switching is sigmoidal. We are interested in a particular class of on (+) and off (-) sigmoidal switches, namely the logistic functions. The sigmoidal on-switch is shown in Fig. 2(a). Equivalently, $S^{+}(u,k,\theta)=(1+tanh(k(u-\theta)))/2$. The off-switch is the quantitative complement of the on-switch, and is defined as $S^{-}(u,k,\theta)=1-S^{+}(u,k,\theta)$.

We typically scale S so that it varies between a minimum value a and maximum value b, both positive:

$$S^{+}(u,k,\theta,a,b) = a+(b-a)S^{+}(u,k,\theta), \quad S^{-}(u,k,\theta,a,b) = S^{+}(u,k,\theta,b,a)$$

If an on-sigmoid is very steep, then it can be approximated with a Heaviside (or step) switch, as shown in Fig. 2(a). The off-step is given by $H^{-}(u,\theta) = 1-H^{+}(u,\theta)$. As with sigmoids, step-switches can be scaled between a and b.

If an on-sigmoid is steep but not too steep, it can be approximated with a ramp, as shown in Fig. 2(a). The off-ramp is defined as $R^{-}(u,\theta_1,\theta_2) = 1-R^{+}(u, \theta_1, \theta_2)$. Ramps can also be scaled between a and b. If the sigmoid is shallow, then, as shown in Section 5, it can be approximated with a sequence of ramps.

3 The Minimal Resistor Model

Based on previously published data [20], Fenton co-developed a minimal (resistor) model (MRM) of the action potential produced by human ventricular myocytes [5]. An *action potential* (AP) is a change in the cell's transmembrane potential u, as a response to an external stimulus (current) e. If the stimulus is delivered from neighboring cells, then its value is $\nabla(D\nabla u)$, where D is the diffusion coefficient and ∇ is the gradient operator. The shape of an AP, its duration (APD), the diastolic interval (DI), and the AP restitution curve (dependence of the APD on the DI) are depicted in Fig. 2(b). Intuitively, the membrane acts like a capacitor, requiring time to recharge after it discharges. The more time it has to recharge, the greater (and longer) the AP. Note that the AP value u is scaled between 0 and 1.5 in the MRM model.

The detailed ionic models (DIMs) typically contain 40-80 state variables and 100-600 parameters, chosen to represent physiologically-relevant cellular entities, such as ion channels and currents and intracellular concentrations. Their associated ranges are bounded by experimental values.

The MRM instead, only considers the sum of these currents, partitioned into three main categories: fast inward J_{fi} (Na-like), slow inward J_{si} (Ca-like), and slow outward J_{so} (K-like). The flow of these total currents is controlled by a fast channel gate v and two slow gates w and s. Together, they retain enough structure such that, with parameters fitted from either experimental data or from DIM simulations, the MRM accurately reproduces the behavior in question.

Among fitted parameters are the voltage-controlled resistances τ_v, τ_w, and τ_s, and the equilibrium values v_∞ and w_∞. The differential equations for the state variables are as follows:

$$\begin{aligned}
\dot{u} &= e - (J_{fi}(u,v) + J_{si}(u,w,s) + J_{so}(u))\\
\dot{v} &= H^{-}(u,\theta_v)\,(v_\infty(u) - v)/\tau_v^{-}(u) \;\; - H^{+}(u,\theta_v)v/\tau_v^{+}\\
\dot{w} &= H^{-}(u,\theta_w)(w_\infty(u) - w)/\tau_w^{-}(u) - H^{+}(u,\theta_w)w/\tau_w^{+}\\
\dot{s} &= (S^{+}(u,k_s,u_s) - s)/\tau_s(u)
\end{aligned}$$

where the three currents are given by the following equations:

$$\begin{aligned}
J_{fi}(u,v) &= -H^{+}(u,\theta_v)v(u-\theta_v)(u_u - u)/\tau_{fi}\\
J_{si}(u,w,s) &= -H^{+}(u,\theta_w)ws/\tau_{si}\\
J_{so}(u) &= +H^{-}(u,\theta_w)u/\tau_o(u) + H^{+}(u,\theta_w)/\tau_{so}(u)
\end{aligned}$$

The voltage-controlled resistances are defined as follows:

$$\tau_v^{-}(u) = H^{+}(u,\theta_o,\tau_{v_1}^{-},\tau_{v_2}^{-}),\;\; \tau_o(u) = H^{-}(u,\theta_o,\tau_{o_2},\tau_{o_1}),\;\; \tau_s(u) = H^{+}(u,\theta_w,\tau_{s_1},\tau_{s_2})$$

$$\tau_w^{-}(u) = S^{-}(u,k_w^{-},u_w^{-},\tau_{w_2}^{-},\tau_{w_1}^{-}),\quad \tau_{so}(u) = S^{-}(u,k_{so},u_{so},\tau_{so_2},\tau_{so_1})$$

Finally, the steady state values for gates v and w are:

$$v_\infty(u) = H^{-}(u,\theta_o),\quad w_\infty(u) = H^{-}(u,\theta_o)(1 - u/\tau_{v\infty}) + H^{+}(u,\theta_v^{-})w_\infty^{*}$$

The values of the parameters for the epicardial (surface) myocytes, as fitted in [5], are given in Fig. 3(a).

(a)

MRM		MCM		MRM		MCM	
Par	Val	Par	Val	Par	Val	Par	Val
θ_o	0.006	θ_o	0.006	$\tau_{v_1}^-$	60	$g_{v_1}^-$	0.01666
u_w^-	0.03	$u_w^{'-}$	0.0406	$\tau_{v_2}^-$	1150	$g_{v_2}^-$	0.00086
θ_w	0.13	θ_w	0.13	$\tau_{w_1}^-$	60	$g_{w_1}^-$	0.01666
θ_v	0.3	θ_v	0.3	$\tau_{w_2}^-$	15	$g_{w_2}^-$	0.06666
u_{so}	0.65	u'_{so}	1.4824	τ_{o_1}	400	g_{o_1}	0.0025
u_s	0.9087	u_s	0.9087	τ_{o_2}	6	g_{o_2}	0.16666
u_u	1.55	u_u	1.55	τ_{so_1}	30.0181	g_{so_1}	0.03331
w_∞^*	0.94	w_∞^*	0.94	τ_{so_2}	0.9957	g_{so_2}	1.00431
k_w^-	65	k_w^-	65	τ_{s_1}	2.7342	g_{s_1}	0.36453
k_{so}	2.0458	k_{so}	2.0458	τ_{s_2}	16	g_{s_2}	0.0625
k_s	2.0994	k_s	2.0994	τ_{fi}	0.11	g_{fi}	9.0909
τ_v^+	1.4506	g_v^+	0.68936	τ_{si}	1.8875	g_{si}	0.5298
τ_w^+	200	g_w^+	0.005	$\tau_{w\infty}$	0.07	$g_{w\infty}$	142.8571

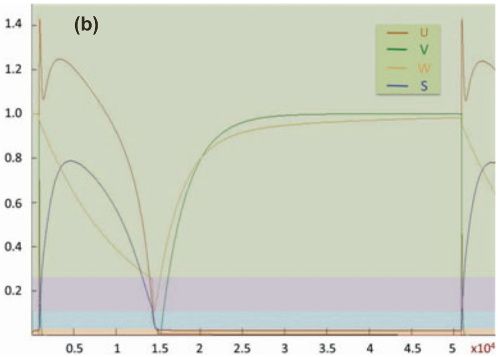


Fig. 3. (a) Parameter values for the MRM and MCM. (b) Evolution of the MCM HA state variables u, v, w and s in time and as a response to a superthreshold stimulus.

4 The Minimal Conductance Model

While much simpler than DIMs, the MRM model is still intractable from an analysis perspective. Its electrical formulation not only uses sigmoidal (and step) switches to control the state variables, but also uses them to control the value of the resistances. Consequently, sigmoids occur both as numerators and denominators in the state equations.

In order to simplify the MRM model, we prove that scaled sigmoids (or steps) are closed under division; that is, the reciprocal of a scaled sigmoid is also a sigmoid. This result allows us to bring the MRM model to a canonical form, which we call the *minimal conductance model* (MCM).

Theorem 1 (Sigmoid closure). *For $a, b > 0$, scaled sigmoids are closed under multiplicative inverses (division):*

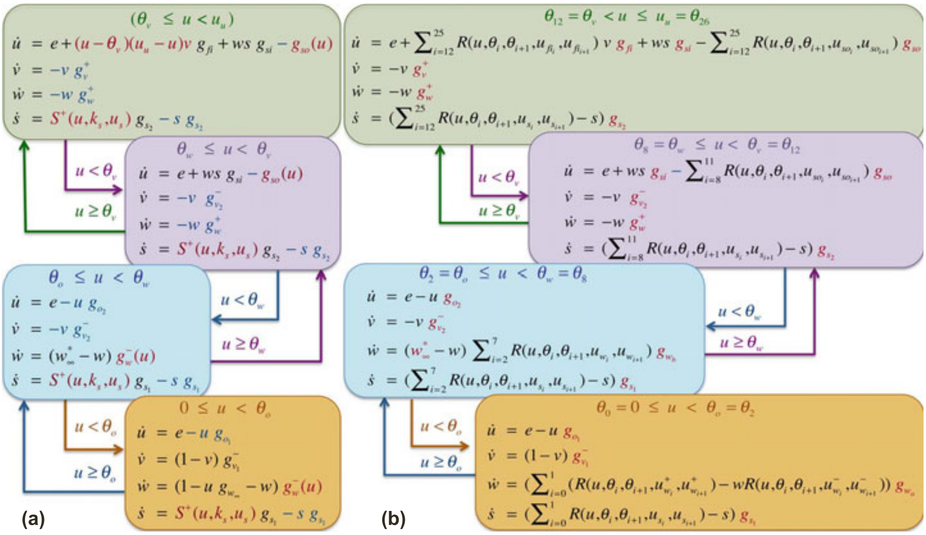
$$S^+(u, k, \theta, a, b)^{-1} = S^-(u, k, \theta + ln(a/b)/2k, b^{-1}, a^{-1})$$

Proof. The proof proceeds by successively transforming the inverse of a scaled sigmoid to a scaled sigmoid. $S^+(u,k,\theta,a,b)^{-1} =$

$$\frac{1}{a + \frac{b-a}{1+e^{-2k(u-\theta)}}} = \frac{1+e^{-2k(u-\theta)}}{b+ae^{-2k(u-\theta)}} = \frac{1}{a} \times \frac{a-b+b+ae^{-2k(u-\theta)}}{b+ae^{-2k(u-\theta)}} =$$

$$\frac{1}{a} - \frac{\frac{1}{a}-\frac{1}{b}}{1+\frac{a}{b}e^{-2k(u-\theta)}} = \frac{1}{a} - \frac{\frac{1}{a}-\frac{1}{b}}{1+e^{-2k(u-(\theta+\frac{ln\,a-ln\,b}{2k}))}} = S^-(u,k,\theta+\frac{ln\,\frac{a}{b}}{2k},\frac{1}{b},\frac{1}{a})$$

Obviously, $H^+(u, \theta, a, b)^{-1} = H^-(u, \theta, b^{-1}, a^{-1})$. Revising the MRM model by replacing each factor $1/\tau_i$ with g_i, and each MRM threshold u_i with the associated MCM threshold u'_i, results in the differential equations for the MCM model. Its parameters are given in Fig. 3(a). Note that sigmoids and steps appear only in the numerator. An interesting feature of the MCM is that it has the canonical form of a *genetic regulatory network* (GRN) model (GRM).



$(\theta_v \le u < u_u)$
$\dot{u} = e + (u - \theta_v)(u_u - u)v\, g_{fi} + ws\, g_{si} - g_{so}(u)$
$\dot{v} = -v\, g_v^+$
$\dot{w} = -w\, g_w^+$
$\dot{s} = S^+(u, k_s, u_s)\, g_{s_2} - s\, g_{s_2}$
$u < \theta_v$
$u \ge \theta_v$
$\theta_w \le u < \theta_v$
$\dot{u} = e + ws\, g_{si} - g_{so}(u)$
$\dot{v} = -v\, g_{v_2}^-$
$\dot{w} = -w\, g_w^+$
$\dot{s} = S^+(u, k_s, u_s)\, g_{s_2} - s\, g_{s_2}$
$u < \theta_w$
$u \ge \theta_w$
$\theta_o \le u < \theta_w$
$\dot{u} = e - u\, g_{o_2}$
$\dot{v} = -v\, g_{v_2}^-$
$\dot{w} = (w_\infty^* - w)\, g_w^-(u)$
$\dot{s} = S^+(u, k_s, u_s)\, g_{s_1} - s\, g_{s_1}$
$u < \theta_o$
$u \ge \theta_o$
$0 \le u < \theta_o$
$\dot{u} = e - u\, g_{o_1}$
$\dot{v} = (1 - v)\, g_{v_1}^-$
$\dot{w} = (1 - u\, g_{w_\infty} - w)\, g_w^-(u)$
$\dot{s} = S^+(u, k_s, u_s)\, g_{s_1} - s\, g_{s_1}$
(a)
$\theta_{12} = \theta_v < u \le u_u = \theta_{26}$
$\dot{u} = e + \sum_{i=12}^{25} R(u, \theta_i, \theta_{i+1}, u_{fi_i}, u_{fi_{i+1}})\, v\, g_{fi} + ws\, g_{si} - \sum_{i=12}^{25} R(u, \theta_i, \theta_{i+1}, u_{so_i}, u_{so_{i+1}})\, g_{so}$
$\dot{v} = -v\, g_v^+$
$\dot{w} = -w\, g_w^+$
$\dot{s} = (\sum_{i=12}^{25} R(u, \theta_i, \theta_{i+1}, u_{s_i}, u_{s_{i+1}}) - s)\, g_{s_2}$
$u < \theta_v$
$u \ge \theta_v$
$\theta_8 = \theta_w \le u < \theta_v = \theta_{12}$
$\dot{u} = e + ws\, g_{si} - \sum_{i=8}^{11} R(u, \theta_i, \theta_{i+1}, u_{so_i}, u_{so_{i+1}})\, g_{so}$
$\dot{v} = -v\, g_{v_2}^-$
$\dot{w} = -w\, g_w^+$
$\dot{s} = (\sum_{i=8}^{11} R(u, \theta_i, \theta_{i+1}, u_{s_i}, u_{s_{i+1}}) - s)\, g_{s_2}$
$u < \theta_w$
$u \ge \theta_w$
$\theta_2 = \theta_o \le u < \theta_w = \theta_8$
$\dot{u} = e - u\, g_{o_2}$
$\dot{v} = -v\, g_{v_2}^-$
$\dot{w} = (w_\infty^* - w) \sum_{i=2}^{7} R(u, \theta_i, \theta_{i+1}, u_{w_i}, u_{w_{i+1}})\, g_{w_b}$
$\dot{s} = (\sum_{i=2}^{7} R(u, \theta_i, \theta_{i+1}, u_{s_i}, u_{s_{i+1}}) - s)\, g_{s_1}$
$u < \theta_o$
$u \ge \theta_o$
$\theta_0 = 0 \le u < \theta_o = \theta_2$
$\dot{u} = e - u\, g_{o_1}$
$\dot{v} = (1 - v)\, g_{v_1}^-$
$\dot{w} = (\sum_{i=0}^{1} (R(u, \theta_i, \theta_{i+1}, u_{w_i}^+, u_{w_{i+1}}^+) - wR(u, \theta_i, \theta_{i+1}, u_{w_i}^-, u_{w_{i+1}}^-))\, g_{w_a}$
$\dot{s} = (\sum_{i=0}^{1} R(u, \theta_i, \theta_{i+1}, u_{s_i}, u_{s_{i+1}}) - s)\, g_{s_1}$
(b)

Fig. 4. (a) Hybrid automaton for the MCM model. (b) The multiaffine automaton.

Definition 1 (GRM). *The sigmoidal form of a GRM consists of a set of differential equations in which the i-th equation has the following form [25,15]:*

$$\dot{u}_i = \sum_{j=1}^{m_i} a_{ij} \prod_{k=1}^{n_j} S^{\pm}(u_k, k_k, \theta_k) - \sum_{j=1}^{m_i'} b_{ij} \prod_{k=1}^{n_j'} S^{\pm}(u_k, k_k, \theta_k)$$

where $S^{\pm}$ are either on- or off-sigmoidal switches, and a_{ij} and b_{ij} are the expression and inhibition coefficients, respectively.

The second summand is often a simple decay term. Approximating sigmoids with steps (or sequences of ramps) in the GRM and MCM results in a set of piecewise-affine [12,18] (or piecewise-multiaffine [3]) differential equations.

The steps in the differential equations of the MCM indicate that the MCM specifies a mixed discrete-continuous behavior. In fact, the MCM is equivalent to the MCM hybrid automaton (HA) shown in Fig. 4(a). Consider the partition of the u-axis by the thresholds occurring in step-switches. Each mode of the HA corresponds to the u-interval defined by two successive thresholds, and each transition corresponds to the discrete jump of one of the step-switches. Nonlinear terms are shown in red and exponential degradation terms in blue. The currents have been expanded and partitioned according to the modes.

The behavior of the MCM HA state variables in time and as a response to an super-threshold stimulus is shown in Fig. 3(b). Voltage intervals are highlighted with same color as the one used for corresponding modes in the HA.

Mode $[0, \theta_o)$ (orange) is a recovering resting mode. In this mode, gates v and w open to their maximum value, and gate s remains closed. Slow sigmoids $S^+(u, k_s, u_s)$ and $g_w^-(u)$ have essentially their minimum value. The only transmembrane current is the slow output current $J_{so}(u)$, whose overall behavior mimics the ionic (potassium) K-current. This current causes an exponential decay of u. Conductances $g_{v_1}^-$ and $g_{w_1}^-$ control the recovery speed of v and w. Hence, their values are important in properly reproducing AP restitution.

Mode $[\theta_v, u_u)$ (green) is a successful AP initiation mode to a superthreshold stimulus. Factor $(u - \theta_v)(u_u - u)$ in the fast-input current J_{fi} mimics the fast opening of the (sodium) Na channel. This leads to a dramatic membrane depolarization, during which u reaches its peak value u_u. With a slight delay, gate v, which mimics the closing of the Na channel, closes, thus blocking the J_{fi} current. The closing-time of v is solely controlled by the rate constant g_v^+ and the initial value of v. The slow-input (calcium) Ca-like current, J_{si}, is still flowing, which prolongs the duration of the AP. This gives the cardiac muscle time to contract. The value of J_{si} is essentially controlled by gate s, which mimics, through its slow sigmoid, the behavior of the Ca-channel opening-gates. Gate w, which mimics the Ca-channel closing-gate, eventually blocks J_{si}, at rate g_w^+. The slow-output, K-like current, J_{so}, reaches its peak value when the slow sigmoid g_{so} switches on towards its maximum value g_{so_2} $(u > u'_{so})$.

In mode $[\theta_o, \theta_w)$ (blue), gate v starts closing at rate $g_{v_2}^-$, while gate w is still opening. The closing/opening of these gates does not affect the value of u, as this still decays at rate g_{o_2}. It does, however, affect the initial values of v and w for the next AP, which in turn affects the length of this AP. It also affects the AP propagation speed, the so-called AP *conduction velocity* (CV).

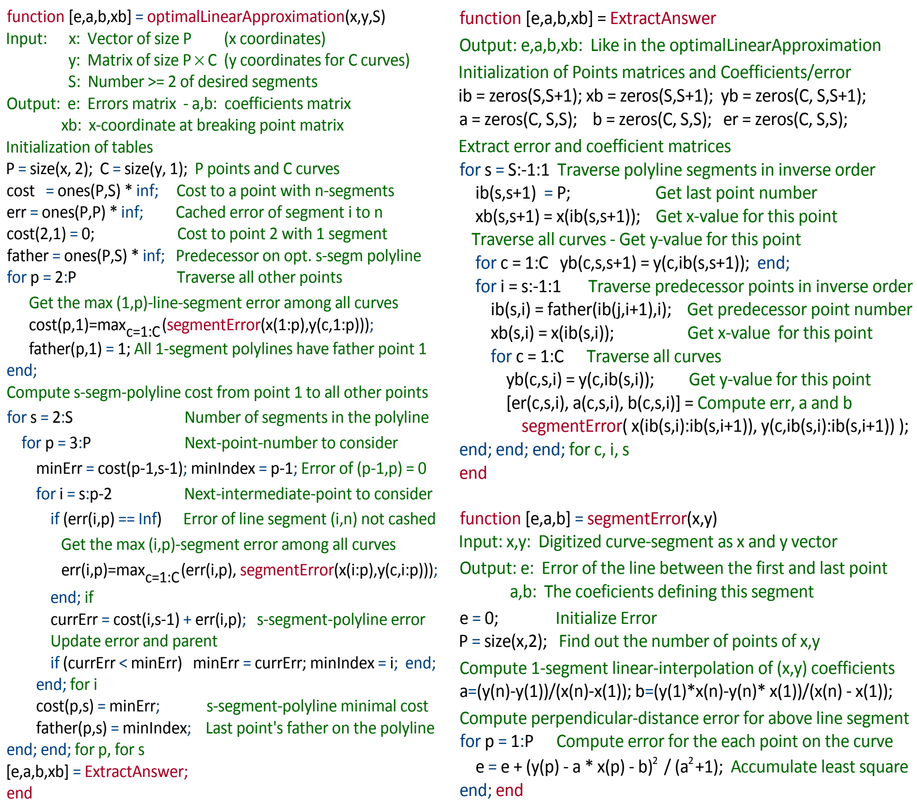
In mode $[\theta_w, \theta_v)$ (pink), gate v closes at the same rate as before, but now gate w is also closing, at rate g_w^+. Current J_{so} changes from an exponential decay to a sigmoid, and the slow-input current starts flowing proportional to ws. Gate s adjusts the "expression" coefficient of its slow sigmoid to g_{s_2}.

5 The Piecewise-Multiaffine Model

Although the MCM is simpler than the MRM and considerably simpler than the DIMs, its analysis is still intractable due to the presence of sigmoidal switches. Qualitative GRMs overcome this problem by assuming that every sigmoid is steep enough to be accurately approximated with one step or one ramp. This assumption is generally not appropriate for quantitative GRMs, and therefore not appropriate for the MCM as well: its sigmoids are too shallow to be approximated by either a single step or a single ramp without seriously distorting the original MCM behavior.

A shallow sigmoid can be accurately approximated with a sequence of ramps. This, however, raises a new question: how can one choose as few ramps as possible, while still maintaining a desired approximation error? Additionally, since each ramp introduces a new mode in the HA, how can the *ramp-thresholds* across sigmoids be chosen such that the number of modes is minimized?

In the following, we show that all of these goals are achievable; i.e., there is an optimal solution to the shallow-sigmoid approximation problem, which minimizes a given approximation error in a global way (i.e. simultaneously over a



```
function [e,a,b,xb] = optimalLinearApproximation(x,y,S)
Input:    x:  Vector of size P        (x coordinates)
          y:  Matrix of size P × C  (y coordinates for C curves)
          S:  Number >= 2 of desired segments
Output:  e:  Errors matrix  - a,b:  coefficients matrix
        xb:  x-coordinate at breaking point matrix
Initialization of tables
P = size(x, 2);  C = size(y, 1);  P points and C curves
cost   = ones(P,S) * inf;    Cost to a point with n-segments
err = ones(P,P) * inf;       Cached error of segment i to n
cost(2,1) = 0;               Cost to point 2 with 1 segment
father = ones(P,S) * inf;   Predecessor on opt. s-segm polyline
for p = 2:P                  Traverse all other points
    Get the max (1,p)-line-segment error among all curves
    cost(p,1)=max_{c=1:C}(segmentError(x(1:p),y(c,1:p)));
    father(p,1) = 1; All 1-segment polylines have father point 1
end;
Compute s-segm-polyline cost from point 1 to all other points
for s = 2:S                      Number of segments in the polyline
  for p = 3:P                    Next-point-number to consider
    minErr = cost(p-1,s-1); minIndex = p-1; Error of (p-1,p) = 0
    for i = s:p-2                Next-intermediate-point to consider
      if (err(i,p) == Inf)      Error of line segment (i,n) not cashed
        Get the max (i,p)-segment error among all curves
        err(i,p)=max_{c=1:C}(err(i,p), segmentError(x(i:p),y(c,i:p)));
      end; if
      currErr = cost(i,s-1) + err(i,p);   s-segment-polyline error
      Update error and parent
      if (currErr < minErr)   minErr = currErr; minIndex = i;  end;
    end; for i
    cost(p,s) = minErr;          s-segment-polyline minimal cost
    father(p,s) = minIndex;    Last point's father on the polyline
end; end; for p, for s
[e,a,b,xb] = ExtractAnswer;
end
```

```
function [e,a,b,xb] = ExtractAnswer
Output: e,a,b,xb:  Like in the optimalLinearApproximation
Initialization of Points matrices and Coefficients/error
ib = zeros(S,S+1); xb = zeros(S,S+1);  yb = zeros(C, S,S+1);
a = zeros(C, S,S);    b = zeros(C, S,S);   er = zeros(C, S,S);
Extract error and coefficient matrices
for s = S:-1:1  Traverse polyline segments in inverse order
   ib(s,s+1)  = P;                  Get last point number
   xb(s,s+1) = x(ib(s,s+1));    Get x-value for this point
  Traverse all curves - Get y-value for this point
   for c = 1:C   yb(c,s,s+1) = y(c,ib(s,s+1));  end;
   for i = s:-1:1      Traverse predecessor points in inverse order
     ib(s,i) = father(ib(j,i+1),i);    Get predecessor point number
     xb(s,i) = x(ib(s,i));               Get x-value  for this point
     for c = 1:C     Traverse all curves
       yb(c,s,i) = y(c,ib(s,i));       Get y-value for this point
       [er(c,s,i), a(c,s,i), b(c,s,i)] = Compute err, a and b
         segmentError( x(ib(s,i):ib(s,i+1)), y(c,ib(s,i):ib(s,i+1)) );
end; end; end; for c, i, s
end
```

```
function [e,a,b] = segmentError(x,y)
Input: x,y:  Digitized curve-segment as x and y vector
Output: e:  Error of the line between the first and last point
          a,b:  The coeficients defining this segment
e = 0;            Initialize Error
P = size(x,2);   Find out the number of points of x,y
Compute 1-segment linear-interpolation of (x,y) coefficients
a=(y(n)-y(1))/(x(n)-x(1)); b=(y(1)*x(n)-y(n)* x(1))/(x(n) - x(1));
Compute perpendicular-distance error for above line segment
for p = 1:P     Compute error for the each point on the curve
   e = e + (y(p) - a * x(p) - b)^2  / (a^2+1);  Accumulate least square
end; end
```

Fig. 5. The optimal linear approximation algorithm. Its main function optimalLinearApproximation uses dynamic programming to compute the optimal segmentation. For each segment, it calls function segmentError to compute the associated error. Function ExtractAnswer is called to extract the answer from the dynamic programming tables.

number of sigmoids). Our approach is based on and extends a dynamic programming algorithm developed in the computer graphics community for approximating digitized polygonal curves [21] with minimal error. The pseudo MATLAB-code for the main function, optimalLinearApproximation, is shown in Fig. 5, where, for readability, comments are displayed in green.[1]

The function's input is a set of curves (digitized with the same number of points), and a number S of segments to be used by the polylines optimally interpolating the curves. The curves are given as a vector x of P x-coordinates, and a matrix y of C rows, each consisting of P y-coordinates.

The function's output consists of matrices e, a, b and of vector xb. Each entry e(c,s,i), a(c,s,i) and b(c,s,i) gives the error, slope, and y-intercept, respectively,

[1] All MATLAB code and experimental results presented in this paper are available at `http://cmacs4heart.pbworks.com/w/page/35180610/CAV2011`

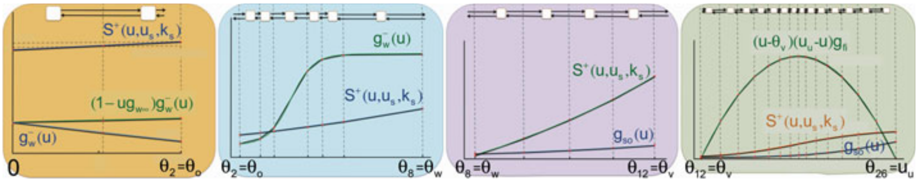


Fig. 6. Linearization of the MCM HA modes with the optimal approximation algorithm. Vertical dashed lines separate the global approximation segments. Each segment corresponds to a mode of the multiaffine hybrid automaton (MHA).

of the i-th segment, in the optimal interpolation polyline of curve c, using s segments. Each entry xb(s,i) is the x-coordinate of breaking point i of the optimal interpolation polylines, using s segments.

The function first determines the number of points P in each curve, and the number of curves C. It then initializes the dynamic programming storage tables cost(P,S) and father(P,S). Each entry cost(p,s) stores the cost from point 1 to point p of the optimal interpolation polyline consisting of s segments. Each entry father(p,s) stores the predecessor of point p on the optimal-cost polyline consisting of s segments. To speed up the search, we use an error matrix err(P,P), such that each entry err(p,q) caches the maximum error of the segment (p,q) with respect to all of the given curves. Then, in a classic dynamic programming fashion, optimalLinearApproximation fills its solution tables bottom up. First, for all points in the curve it computes the cost and father of the 1-segment polyline starting from point 1. Then, knowing the optimal cost of all s-segment polylines from point 1 to any point i that is less than or equal to p, it computes the optimal cost of an s+1-segment polyline from point 1 to point p+1, by choosing the s-segment polyline, whose cost is minimal when increased with err(i,p+1).

The value stored in err(p,q) is computed with the (nested) function segmentError. Its input consists of vectors x and y, defining a curve segment. Its output consists of error e, and coefficients a and b of the line $y(x) = \mathsf{a}x + \mathsf{b}$ passing through the first and the last points of the curve segment. Error e is computed by summing up, for each point p on the curve, the square of the perpendicular distance from (x(p),y(p)) to y. Once the solution tables are completely filled, optimalLinearApproximation calls nested function extractAnswer to traverse table father in reverse order, and produce matrices e, a, b, and xb. These matrices have the same format as the output of the caller function, optimalLinearApproximation.

Our implementation of segmentError also allows the use of linear regression instead of linear interpolation. This leads to an optimal approximation that, for the same error, has fewer segments. However, linear regression also introduces discontinuities at the breaking points of the optimal polylines, as the line segment resulting from regression does not typically start and end on the curve. When approximating a single curve, one can use instead the points where the polyline segments intersect. Unfortunately, it is not clear how to generalize this approach to a set of curves, without introducing unnecessary breaking points.

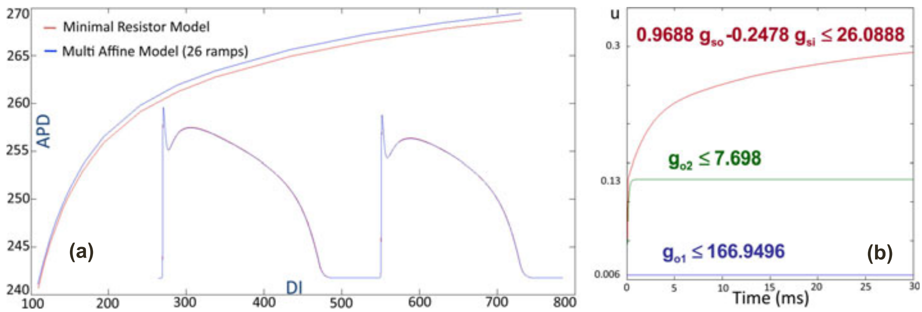


Fig. 7. (a) Comparison of the action potential (AP) shape (the inset figure) and the restitution (dependence of the AP duration (APD) on the diastolic interval (DI)), for the original MRM and the 26 segment MHA, in a cable. (b) Simulation results for the MEM with the values of the conductances g_{o_1}, g_{o_2}, g_{si} and g_{so} taken from the parameter ranges where the MEM robustly satisfies the LTL formula G $(u < \theta_v)$.

As in GRMs, we assume that the thresholds and slopes of switches (steps and sigmoids) are known and fixed. We can thus linearize the MCM HA one mode at a time. The result is a *multiaffine hybrid automaton* (MHA).

Fig. 6 presents our linearization of the MCM modes. Mode $[0, \theta_o)$ (orange) has three nonlinear functions: sigmoids $g_w^-(u)$ and $S^+(u, u_s, k_s)$, and product $(1 - u g_{w\infty}) g_w^-$. The last is treated separately, as the linearization of $g_w^-(u)$ multiplied by $(1 - u g_{w\infty})$ results in a u^2 term. A two-segment linearization (two modes in the MHA) results in a very small error.

Mode $[\theta_o\,, \theta_w)$ (blue) has two nonlinear functions: sigmoids $S^+(u, u_s, k_s)$ and $g_w^-(u)$. In this case, we needed a six-segment linearization (six modes in the MHA) to achieve a small approximation error. Note that the sensitivity of the MCM behavior to the linearization error is also very important.

Mode $[\theta_w\,, \theta_v)$ (pink) has two nonlinear functions: sigmoids $S^+(u, u_s, k_s)$ and $g_{so}(u)$. A four-segment linearization (four modes in the MHA) achieves a small enough approximation error and good overall behavior.

Finally, mode $[\theta_v, u_u)$ (green) has three nonlinear functions: sigmoids $g_{so}(u)$ and $S^+(u, u_s, k_s)$ and the parabolic term $(u - \theta_v)(u_u - u) g_{fi}$. This is the most sophisticated mode. Although gate v closes very rapidly, nullifying the parabolic term in J_{fi}, voltage u traverses in the meantime the entire interval $[\theta_v, u_u)$. Hence, one needs to linearize the parabolic term over this entire interval. This leads to a costly, but inevitable, linearization via 14 segments (modes in the MHA). In our experiments, fewer segments have lead to an unacceptable approximation.

To asses the accuracy of the MHA, we performed extensive 1D and 2D simulations in a cable of 100 cells and a grid of 800 × 800 cells, respectively. Although the 1D simulation was used to determine the behavior of a single cell only—for example, cell number 50—the use of a cable is necessary, as it is known that cells behave differently when interacting with neighboring cells. Many cardiac models, for example [17], accurately reproduce the AP when simulated in isolation, but fail to reproduce the desired behavior in a cable.

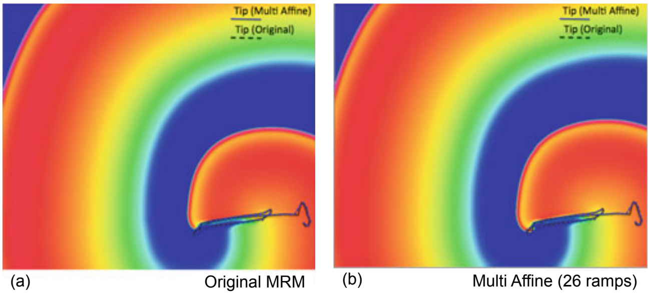


Fig. 8. Snapshot from the simulation of a spiral wave on an 800 × 800 grid, with isotropic diffusion. (a) The result of the original MRM simulation. (b) The result of the 26-segment MAM simulation. In each figure, we also show the movement over time of the tip of the spiral: dark line for the MAM, and dashed dark line for the MRM.

Fig. 7(a) shows the restitution curves of the MRM and MHA models. Each point APD(d) on these curves was obtained by first pacing the MRM and the MHA models at the largest DI value, and then abruptly changing the pacing to DI d. For each value of d, we also compared the AP shapes AP(d). Two such comparison are given as an inset in Fig. 7(a). In both cases (restitution and AP shape), the MHA approximated the MCM with sufficient accuracy.

In Fig. 8, we compare the behavior of the MRM and MHA models on a 2D grid of 800 × 800 cells. The comparison uses a well-established protocol for the initiation of a spiral wave in cardiac ventricular tissue. We have also tracked the movement of the tip of the spiral over time, which is shown as a dark-blue curve. The 2D simulation confirms the very good accuracy of the MHA model.

Simulations were implemented using CUDA, NVIDIA's parallel computing architecture for GPUs (graphics processing units), and were conducted on an Intel Core i7 workstation with 12 Gb RAM, and an NVIDIA Tesla C1060 processor with a 240 GPU processor core and 4GB GDDR3. In this setting, we observed a 1.43 speedup in MHA simulation time compared to MCM simulation time. Note that it is possible to table sigmoid and parabola values to speedup MCM simulation time [10]. This strategy, however, considerably increases the memory demand for the same accuracy and speed, and renders analysis intractable. Our linearization approach can be viewed as an optimal tabling.

6 Cardiac Disorder Parameter-Range Identification

We show how the MCM HA model can be put into a form suitable for analysis by the Rovergene tool for GRNs [3], thereby allowing us to automatically identify the parameter ranges for a significant cardiac disorder. The linearization algorithm presented in Section 5 returns, for each mode $[\theta_1, \theta_2)$, parameter sequences a_i and b_i, and threshold sequence x_i, where subscript i ranges over the segments chosen in order to fulfil a desired approximation error e. For each i, the returned values define a line segment $y(x) = \mathrm{a}x + \mathrm{b}$ within the interval $[x_i, x_{i+1})$.

For the first segment, $x_1 = \theta_1$, and for the last segment, $x_n = \theta_2$. Now consider segment $[x_i, x_{i+1})$. The minimum value of $y(x)$ is $y_i = \mathrm{a}_i x_i + \mathrm{b}_i$ and the maximum value of $y(x)$ is $y_{i+1} = \mathrm{a}_i x_{i+1} + \mathrm{b}_i$. Together with the threshold values, they define the scaled ramp $R^{\pm}(x, x_i, x_{i+1}, y_i, y_{i+1})$. This is an on (+) ramp, if $y_i \leq y_{i+1}$ and an off (-) ramp if $y_i \geq y_{i+1}$.

Since the ramps must be summed up, for each $i > 1$, we must adjust the y-coordinate by subtracting the maximum value of the previous ramp. Hence, these ramps become $R^{\pm}(x, x_i, x_{i+1}, y_i - y_{i-1}, y_{i+1} - y_{i-1})$.

Once the scaled ramps are computed and summed up, for each mode of the MCM HA, one obtains a multiaffine hybrid automaton (MHA), as shown in Fig. 4(b). The remaining parameters of the MHA are now highlighted in red. The MHA modes have become super-modes, each consisting of as many sub-modes as there are distinct indices in the sums.

The MHA is not a suitable Rovergene-GRN-analysis-tool input for two reasons: 1) Rovergene expressions must be scaled ramps; 2) Rovergene does not support steps. The first problem is overcome by replacing variables with ramps. For example, variable v occurring on the right-hand side of $\dot{u}$ in the green mode, is replaced with ramp $R^{+}(v, 0, 1)$. The second problem is overcome by replacing steps with very steep ramps. This amounts to introducing, for each threshold θ_i, separating modes $[\theta_{i-1}, \theta_i)$ and $[\theta_i, \theta_{i+1})$, a just-before θ_i threshold θ_i^-. The equations in mode $[\theta_{i-1}, \theta_i)$ are now multiplied with $R^{-}(u, \theta_i^-, \theta_i)$ and the ones in mode $[\theta_i, \theta_{i+1})$ are now multiplied with $R^{+}(u, \theta_i^-, \theta_i)$. The MHA becomes:

$$\begin{aligned}
\dot{u} = e &- R^{-}(u, \theta_o^-, \theta_o) R^{+}(u, 0, \theta_o^-, 0, \theta_o^-)\, g_{o_1} \\
&- R^{+}(u, \theta_o^-, \theta_o) R^{-}(u, \theta_w^-, \theta_w) R^{+}(u, \theta_o, \theta_w^-, \theta_o, \theta_w^-)\, g_{o_2} \\
&+ R^{+}(u, \theta_w^-, \theta_w) R^{+}(s, 0, 1) R^{+}(w, 0, 1)\, g_{si} \\
&- R^{+}(u, \theta_w^-, \theta_w) \textstyle\sum_{i=8}^{25} R(u, \theta_i, \theta_{i+1}, u_{so_i}, u_{so_{i+1}}) \\
&+ R^{+}(u, \theta_o, \theta_v) R^{+}(v, 0, 1) \textstyle\sum_{i=12}^{25} R(u, \theta_i, \theta_{i+1}, u_{fi_i}, u_{fi_{i+1}})\, g_{fi}
\end{aligned}$$

$$\begin{aligned}
\dot{v} = &\ R^{-}(u, \theta_o^-, \theta_o) R^{-}(v, 0, 1)\, g_{v_1}^- \\
&- R^{+}(u, \theta_o^-, \theta_o) R^{-}(u, \theta_o, \theta_v) R^{+}(v, 0, 1)\, g_{v_2}^- \\
&- R^{+}(u, \theta_o, \theta_v) \textstyle\sum_{i=12}^{25} R^{+}(v, 0, 1)\, g_v^+
\end{aligned}$$

$$\begin{aligned}
\dot{w} = &\ R^{-}(u, \theta_o^-, \theta_o) \textstyle\sum_{i=0}^{1} (R(u, \theta_i, \theta_{i+1}, u_{w_i}^+, u_{w_{i+1}}^+) - R(w, 0, 1) R(u, \theta_i, \theta_{i+1}, u_{w_i}^-, u_{w_{i+1}}^-)) g_{w_a} \\
&+ R^{+}(u, \theta_o^-, \theta_o) R^{-}(u, \theta_w^-, \theta_w)(w_\infty^* - R(w, 0, 1)) \textstyle\sum_{i=2}^{7} R(u, \theta_i, \theta_{i+1}, u_{w_i}^+, u_{w_{i+1}}^+)\, g_{w_b} \\
&- R^{+}(u, \theta_w^-, \theta_w) R(w, 0, 1)\, g_w^+
\end{aligned}$$

$$\dot{s} = (R^{-}(u, \theta_w^-, \theta_w) g_{s_1} + R^{+}(u, \theta_w^-, \theta_w) g_{s_2}) \textstyle\sum_{i=0}^{25} (R^{+}(u, \theta_i, \theta_{i+1}, u_{s_i}, u_{s_{i+1}}) - R^{+}(s, 0, 1))$$

where the thresholds (except for the new just-before thresholds θ_i^-), voltages, and conductances match the ones in the MHA. We call this system a *piecewise-multiaffine differential equations model* (MEM).

As discussed in Section 1, a biologically relevant question that we want to answer is, under what circumstances may a cell lose excitability? At the molecular level, this is due to an improper functioning of the cardiac-cell ionic channels.

To identify the ionic mechanisms responsible for this disorder, we first reformulate the question as one of parameter-range identification: What are the parameter ranges for which the MEM fails to generate an AP?

This property may be specified in linear temporal logic (LTL) as $\mathrm{G}\,(u < \theta_v)$, where G is the LTL globally (always) temporal operator. The property states that in all executions of the MEM and in all moments of time along a single execution, the voltage value is below θ_v. (Note that in an LTL formula, there is an implicit quantification over all executions.) We would like this property to hold for all stimulus durations. In terms of the MHA of Fig. 4(b), this property is true due to the interplay of the ranges of conductances g_{o_1}, g_{o_2}, g_{si} and g_{so}.

To identify these ranges in an automated fashion, we use Rovergene with input the above MEM and LTL formula, and with the following initial region:

$$u \in [0, \theta_1], \quad v \in [0.95, 1], \quad w \in [0.95, 1], \quad s \in [0, 0.01]$$

The u-thresholds and the initial region impose the following partition on the ranges of state variables (for u we have added the just-before thresholds):

$$u : [0, \ldots, \theta_{29}], \quad v, w : [0, 0.95, 1], \quad s : [0, 0.01, 1]$$

Parameter ranges with biological significance for the conductances were taken as below. They include known values for normal and abnormal cell behavior.

$$g_{o_1} \in [1, 180], \quad g_{o_2} \in [0, 10], \quad g_{si} \in [0.1, 100], \quad g_{so} \in [0.9, 50]$$

The behavior of the MEM in each hypercube of the state-space partition is completely determined by its corners, so the existence of transitions from one hypercube to its neighbors can be computed by evaluating the MEM in the corners. In each corner, the MEM becomes an affine system in the MEM parameters. Solving these systems, one obtains the separating hyperplanes of positive and negative sign of the derivatives in the MEM. Finally, taking into account the desired LTL property, one obtains the parameter ranges for which the property is satisfied. In our case the ranges returned are:

$$166.9494 \le g_{o_1} \le 180, \quad 7.6982 \le g_{o_2} \le 10$$

$$-0.24784\, g_{si} + 0.9688\, g_{so} \le 26.0888$$

They have the following meaning. If $g_{o_1} \ge 166.9494$, then, regardless of the duration of the magnitude-1 stimulus applied, the voltage u never leaves the orange interval (mode) $[0, \theta_o]$. If, on the other hand, $g_{o_1} < 166.9494$, then u reaches the blue interval (mode) $[\theta_o, \theta_w)$. Since we are considering stimuli of any width (time is abstracted away by Rovergene), once u enters the blue range, its behavior is completely determined by this mode. If $g_{o_2} \ge 7.6982$, then u can never leave the mode. If $g_{o_2} < 7.6982$, then u will enter the pink interval (mode) $[\theta_w, \theta_v)$. In this mode, the behavior of u is determined by the interplay between g_{so} and g_{si}. If the above linear combination is satisfied, one can never leave this mode.

The corresponding simulation, for a sample of values in the above parameter ranges, is shown in Fig. 7(b). To ensure that we run the same model as Rovergene, we also developed a Rovergene simulation tool that, given a Rovergene model as input, simulates its dynamic behavior in MATLAB. This tool proved to be an invaluable debugging tool during model encoding in Rovergene.

7 Conclusions and Future Work

Although formal techniques were used before to analyze cardiac-cell properties (see e.g. our work in [26,13]), this paper presents, to the best of our knowledge, the first approach for automatically identifying parameter ranges of a biologically-relevant cardiac model, guaranteeing that the model accurately reproduces a particular cardiac disorder.

Our approach takes the nonlinear cardiac model of [5], brings it first into a genetic regulatory network sigmoidal form, and then linearizes and transforms it into a piecewise-multiaffine set of differential equations. It then leverages the Rovergene tool, previously developed for automatic parameter-range identification in genetic regulatory networks [3], to automatically and robustly check a cardiac disorder expressed as a linear temporal logic (LTL) formula.

The particular property we considered in this paper is lack of cardiac-cell excitability. Our Rovergene-based predictions hold in the nonlinear cardiac model of [5], which matches and is actually based on real biological data. Confirming these results directly on experimental data could be done for a tissue that loses its excitability, as in the case of ischemia (low oxygen). This is work in progress, which we believe is out of the scope of the current paper.

Many abnormalities responsible for cardiac disorders are time- or rate-dependent properties that cannot be checked with the Rovergene tool, due to its underlying finite-automata abstraction. Action potential duration and spiral breakup (fibrillation) are examples of such properties. We therefore plan to investigate new parameter-range identification approaches that, in contrast, use abstractions based on timed-automata[2,16] or even hybrid linear-automata[1,11].

Acknowledgments. This work was supported by the NSF CNS-Expeditions-09-26190 award and by the AFOSR FA0550-09-1-0481 award.

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. In: Proc. of the 11th Int. Conf. on Analysis and optimization of systems: Discrete Event Systems. LNCIS, vol. 199, pp. 331–351. Springer, Heidelberg (1994)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 2(126), 183–235 (1994)
3. Batt, G., Belta, C., Weiss, R.: Temporal logic analysis of gene networks under parameter uncertainty. IEEE Trans. of Automatic Control 53, 215–229 (2008)
4. Bender, C.M., Orszag, S.A.: Advanced Mathematical Methods for Scientists and Engineers: Asymptotic Methods and Perturbation Theory. Springer, Heidelberg (1999)

5. Bueno-Orovio, A., Cherry, M.E., Fenton, F.H.: Minimal model for human ventricular action potentials in tissue. J. of Theor. Biology 253(3), 544–560 (2008)
6. Cherry, E.M., Fenton, F.H.: A tale of two dogs: Analyzing two models of canine ventricular electrophysiology. American Journal of Physiology - Heart and Circulatory Physiology 292, 43–55 (2007)
7. Cherry, E.M., Fenton, F.H.: Visualization of spiral and scroll waves in simulated and experimental cardiac tissue. New Journal of Physics 10, 125016 (2008)
8. de Jong, H.: Modeling and simulation of genetic regulatory systems: A literature review. Journal of Computational Biology 9(1), 69–105 (2002)
9. Fenton, F.H., Cherry, E.M.: Models of cardiac cell. Scholarpedia 3, 1868 (2008)
10. Fenton, F.H., Cherry, E.M., Hastings, H.M., Evans, S.J.: Real-time computer simulations of excitable media: Java as a scientific language and as a wrapper for C and Fortran programs. BioSystems 64(1-3), 73–96 (2002)
11. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. Springer Int. J. of Software Tools for Technology Transfer 10(3), 263–279 (2008)
12. Glass, L., Kauffman, S.A.: The logical analysis of continuous nonlinear biochemical control networks. Journal of Math. Biology 39(1), 103–129 (1973)
13. Grosu, R., Smolka, S.A., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. Communications of the ACM (CACM) 52(3), 1–10 (2009)
14. Iyer, V., Mazhari, R., Winslow, R.L.: A computational model of the human left-ventricular epicardial myocytes. Biophysical Journal 87(3), 1507–1525 (2004)
15. Keener, J., James Sneyd, J.: Mathematical Physiology. Springer, Heidelberg (2008)
16. Larsen, K., Pettersson, P., Yi, W.: Uppaal in a nutshell. Springer Int. Journal of Software Tools for Technology Transfer 1 (1997)
17. Luo, C.H., Rudy, Y.: A dynamic model of the cardiac ventricular action potential. I. Simulations of ionic currents and concentration changes. Circulation Research 74(6), 1071–1096 (1994)
18. Mestl, T., Plahte, E., Omholt, S.W.: A mathematical framework for describing and analysing gene regulatory networks. Journal of Theoretical Biology 176(2), 291–300 (1995)
19. Michaelis, L., Menten, M.L.: Die Kinetik der Invertinwirkung. Biochemische Zeitschrift 49, 333–369 (1913)
20. Morgan, J.M., Cunningham, D., Rowland, E.: Dispersion of monophasic action potential duration: Demonstrable in humans after premature ventricular extrastimulation but not in steady state. Journal of Am. Coll. Cardiol. 19, 1244–1253 (1992)
21. Perez, J.C., Vidal, E.: Optimum polygonal approximation of digitized curves. Pattern Recognition Letters 15, 743–750 (1994)
22. Priebe, L., Beuckelmann, D.J.: Simulation study of cellular electric properties in heart failure. Circulation Research 82, 1206–1223 (1998)
23. Segel, L.A., Slemrod, M.: The quasi-steady-state assumption: A case study in perturbation. SIAM Review 31(3), 446–477 (1989)
24. Ten Tusscher, K.H., Noble, D., Noble, P.J., Panfilov, A.V.: A model for human ventricular tissue. American Journal of Physiology 286, 1573–1589 (2004)
25. Yagil, G., Yagil, E.E.: On the relation between effector concentration and the rate of induced enzyme synthesis. Biophysical Journal 11(1), 11–27 (1971)
26. Ye, P., Grosu, R., Smolka, S.A., Entcheva, E.: Formal analysis of abnormal excitation in cardiac tissue. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 141–155. Springer, Heidelberg (2008)

Threader: A Constraint-Based Verifier for Multi-threaded Programs

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko

Technische Universität München

Abstract. We present a tool that implements Owicki-Gries and rely-guarantee methods for the compositional verification of multi-threaded programs. Our tool computes the requisite auxiliary assertions automatically using an abstraction and refinement procedure. Our procedure is based on a Horn clause encoding of refinement queries and facilitates the discovery of thread-modular proofs when such proofs exist. We present the tool and its evaluation on a collection of benchmarks, including a direct comparison of the effectiveness of the proof rules.

1 Introduction

Software running on our computers is becoming increasingly concurrent, i.e., it consists of several execution threads that process several tasks in parallel and interact with each other during the operation. Increasing concurrency is supported by the state-of-the-art in computer hardware, where modern CPUs have several computing cores and can execute several threads at the same time. However, it is extremely difficult to develop correct concurrent programs that are free of bugs, as evidenced by recent studies [5, 10].

In this paper we present Threader, a tool that automates verification of multi-threaded programs. The algorithms implemented in Threader are rooted in compositional proof rules [9, 13]. Following [6, 7], Threader uses abstraction and abstraction refinement to find adequate auxiliary assertions for verification. In this paper, we investigate the effectiveness of the compositional rules on a collection of benchmarks.

2 Threader Overview

Threader consists of three main modules that interact as shown in Figure 1. First, a C-frontend translates the input C program and its assert statements into a transition system that is represented using constraints over program variables. Next, the program safety is verified by iteratively applying abstract reachability computation and abstraction refinement steps. If no error state is unreachable then Threader reports that the program is safe. Otherwise, i.e., if an error state is discovered by the abstract reachability computation, Threader encodes the error state reachability using a set of recursion free Horn clauses and invokes a Horn solver. If the Horn clauses are not satisfiable then Threader returns a

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 412–417, 2011.

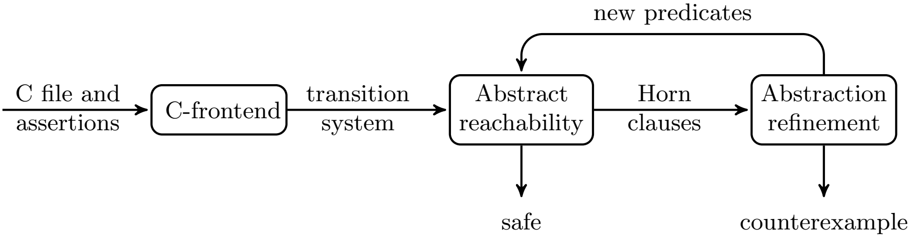


Fig. 1. The main modules of Threader. The abstract reachability module solves recursive equations (1), (2), or (3). The abstraction refinement module discovers (transition) predicates by solving Horn clauses.

counterexample. Otherwise, a solution of the Horn clauses yields predicate that are needed to refine the abstraction. We describe the modules of Threader below.

C-frontend. Threader's frontend is based on the CIL framework [12]. The frontend takes as input a C file containing N functions that represent N threads to be executed in parallel. We assume that the threads interact using shared variables. Threader does not support recursive functions and relies on inlining to deal with function calls. After inlining, each of the N functions is translated to a constraint-based representation. The frontend outputs a transition system $P = (V, \varphi_{init}, \varphi_{err}, \rho_1, \ldots, \rho_{\mathsf{N}})$ with variables V, initial states φ_{init}, error states φ_{err}, and thread transitions $\rho_1, \ldots, \rho_N$. The program variables $V = (V_G, V_1, \ldots, V_{\mathsf{N}})$ are partitioned into global variables shared by all threads, and local variables of each thread. The set of initial program states φ_{init} is obtained by initializing the global variables. The set of error states φ_{err} is derived from the assert statements in the input C program. Finally, each transition relation ρ_i can only change the values of global variables and local variables of thread i. We use $\rho_i^= = (V_i = V_i')$ to make this requirement explicit, i.e., for each $i \neq j \in 1..\mathsf{N}$ we require the validity of $\rho_i \rightarrow \rho_j^=$. We assume that each implication assertion in this paper is implicitly universally quantified over its free variables. The transition relation of the entire program is $\rho_1 \vee \cdots \vee \rho_{\mathsf{N}}$.

Abstract reachability (and environment transitions). Given an abstraction function, the abstract reachability module computes an over-approximation of the states reachable during any execution of a multi-threaded program and corresponding environment transitions, as described by the proof rules in Figure 2.

The rule (1) relies on a single, global auxiliary assertion R over program variables V. If R satisfies all three conditions of the proof rule then the program is safe. The first condition ensures that R over-approximates the initial states of the program φ_{init}. The second condition ensures that R is invariant under the

Find an assertion R over V such that:

$$\begin{array}{ll} \dot{\alpha}(\varphi_{init}) & \rightarrow R \\ \dot{\alpha}(post(\rho_1 \vee \dots \vee \rho_{\mathsf{N}}, R)) & \rightarrow R \\ R \wedge \varphi_{err} & \rightarrow \mathit{false} \end{array} \tag{1}$$

Find assertions $R_1, \dots, R_{\mathsf{N}}$ over V such that

$$\begin{array}{lll} \dot{\alpha}_i(\varphi_{init}) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_i, R_i)) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_j, R_i \wedge R_j)) & \rightarrow R_i & \text{for } i \neq j \in 1..\mathsf{N} \\ R_1 \wedge \dots \wedge R_{\mathsf{N}} \wedge \varphi_{err} & \rightarrow \mathit{false} & \end{array} \tag{2}$$

Find assertions $R_1, \dots, R_{\mathsf{N}}$ over V and $E_1, \dots, E_{\mathsf{N}}$ over V and V' such that

$$\begin{array}{lll} \dot{\alpha}_i(\varphi_{init}) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_i \vee (E_i \wedge \rho_i^{=}), R_i)) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \ddot{\alpha}_{j \triangleright i}(R_j \wedge \rho_j) & \rightarrow E_i & \text{for } i \neq j \in 1..\mathsf{N} \\ R_1 \wedge \dots \wedge R_{\mathsf{N}} \wedge \varphi_{err} & \rightarrow \mathit{false} & \end{array} \tag{3}$$

Fig. 2. Proof rules for safety of a program $(V, \varphi_{init}, \varphi_{err}, \rho_1, \dots, \rho_{\mathsf{N}})$. Given abstraction functions, Threader computes the strongest solution for the auxiliary assertions using either (1) "Monolithic", (2) "Owicki-Gries", or (3) "Rely-Guarantee" proof rule.

application of the thread transitions $\rho_1, \dots, \rho_{\mathsf{N}}$. The last condition requires that R does not intersect the error states φ_{err} .

The rule (2) is a formulation of the "Owicki-Gries" proof rule [13]. The reasoning about reachable states is localized by replacing a global auxiliary assertion R with N thread-reachability assertions $R_1, \dots, R_{\mathsf{N}}$. Each thread-reachability assertion R_i needs to over-approximate the initial states and needs to be invariant under the transition relation of thread i . In addition, R_i also needs to account for interference from the transition relation of thread j when it is applied to reachable states in R_j . The last condition requires that the intersection of the thread-reachability assertions and φ_{err} is empty.

The rule (3) reasons about the threads individually by relying on environment assertions. Each assertion E_i denotes a binary relation over V and V' that captures how all threads other than i can change the program states. As above, the thread-reachability assertion R_i is required to over-approximate the initial states and be invariant under the transition relation of thread i . Threader accounts for the interference from threads other than i by the environment transition $E_i \wedge \rho_i^{=}$, which does not modify the values of variables local to thread i .

Threader effectively computes the strongest candidate for the auxiliary assertions wrt. a given abstraction. See [7] for a corresponding algorithm for the proof rule (3) (other rules are similar).

In practice, it is crucial to maintain for each thread i an abstraction function $\dot{\alpha}_i$ that approximates the thread-reachability R_i . Environment transitions are approximated using different abstraction functions for different pairs of threads. For an abstraction function $\ddot{\alpha}_{i \triangleright j}$ the double dot indicates that the function $\ddot{\alpha}_{i \triangleright j}$ abstracts binary relations over states (not sets of states) and the index $i \triangleright j$ indicates that this function is used to abstract the effect of the thread i on the reachability of thread j .

THREADER uses predicate and transition predicate abstraction functions that are defined using sets of predicates $\dot{\mathcal{P}}_i$ and transition predicates $\ddot{\mathcal{P}}_{i \triangleright j}$ as follows.

$$\dot{\alpha}_i(S) = \bigwedge \{\dot{p} \in \dot{\mathcal{P}}_i \mid S \rightarrow \dot{p}\} \qquad \ddot{\alpha}_{i \triangleright j}(T) = \bigwedge \{\ddot{p} \in \ddot{\mathcal{P}}_{i \triangleright j} \mid T \rightarrow \ddot{p}\}$$

THREADER discovers the sets of (transition) predicates automatically. Initially, the empty sets are used to compute a coarse approximation of the reachable state space and environment transitions. If, for given abstraction functions, the reachability assertions intersect the set of error states, then the discovered error evidence needs to be checked for spuriousness. The reachability assertions computed so far are used to formulate a query to the abstraction refinement module.

Termination properties. THREADER can prove termination properties based on a “Rely-Guarantee” proof rule and automated construction of transition abstraction functions. More details are reported in [8].

3 Experiments

In this section, we present our experience with applying THREADER on 15 multi-threaded C programs. See Table 1. The name of the benchmark and the number of lines of C code are shown in the first two columns. Columns 3, 4 and 5 present verification results obtained from our implementation of the proof rules (1), (2), and (3), respectively. The source code for the examples together with additional data can be found at http://www.model.in.tum.de/˜popeea/research/threader.html.

The programs shown in Table 1 are small but intricate. We are not aware of any automatic tool that can deal with these examples.

The first example from the table illustrates a program used as running example in the paper introducing thread-modular model checking [4]. The program safety can be proven using each of the proof rules. Due to the low complexity of this safety proof, this program illustrates that monolithic verification may conclude faster than localized reasoning.

The second part of the table reports on various algorithms to establish mutual exclusion between a number of threads. For all these examples, we instrumented a safety assertion to check mutual exclusion. Dekker, Peterson, and Szymanski are classical algorithms. Readers-writer-lock and Time-varying-mutex are tests for the Calvin model checker [3]. QRCU [11] is a variant of the read-copy-update algorithm that is used in the Linux kernel, and is the most complex of

Table 1. Applying different proof rules implemented in THREADER. All programs are safe. Time is measured in seconds. "T/O" stands for time out after 15 minutes.

Benchmark programs	LOC	"Monolithic"	"Owicki-Gries"	"Rely-Guarantee"
Spin2003	18	0.02s	0.02s	0.1s
Dekker	39	T/O	0.3s	1.1s
Peterson	26	T/O	0.7s	2.3s
Szymanski	43	T/O	1.8s	12.2s
Readers-writer-lock	22	0.03s	0.03s	0.1s
Time-varying mutex	29	0.3s	0.73s	7.5s
NaïveBakery	22	T/O	0.3s	2.4s
Bakery	37	T/O	1.4s	97s
Lamport	62	T/O	11.4s	57s
QRCU-2processes	120	T/O	1.8s	11.3s
QRCU-3processes	148	T/O	89s	T/O
QRCU-4processes	182	T/O	T/O	T/O
Mozilla-fixed-vulnerab	168	T/O	0.8s	0.8s
See-Saw	98	T/O	T/O	7.8s
Scull	451	T/O	T/O	41.2s

the presented mutual exclusion algorithms. We test its simple variant with two processes (one reader and one updater), as well as variants with two and three readers. For all benchmarks, we observed that the monolithic verification cannot cope with the transition relation of the entire program. Furthermore, the "Owicki-Gries" proof rule captures concisely the interference between threads, as the state space is overly constrained by values of the variables establishing the mutual exclusion invariant. Finally, the "Rely-Guarantee" proof rule requires representing intricate environment transitions and therefore it needs to discover supporting transition predicates. We include the example QRCU-4processes that times-out for both "Owicki-Gries" and "Rely-Guarantee" abstraction refinement methods. We note that for the moment THREADER does not implement algorithms for symmetry reduction that would be beneficial for the efficiency of a compositional verification approach, as demonstrated in [1].

In the third part of the table, Mozilla-fixed-vulnerab is a fix from the Mozilla CVS repository for a vulnerability described in a study of concurrency bugs [10]. See-Saw is a multi-threaded version of the program reported in [14] where we instrumented the invariants obtained by the StInG prover as assertions in the C program. Scull [2] is a Linux character driver that implements access to a global memory area. Different invocations of the open, read, write, and release functions implemented by the device driver access common variables and these accesses should be performed in a critical section. For these programs, we observed that "Rely-Guarantee" reasoning allows a natural encoding of environment transitions as binary relations. In contrast, the "Owicki-Gries" proof rule is not able to capture the thread interference since the thread-reachability assertions are in this case expressed over sets of states.

To conclude, we note some of the significant advantages of the algorithms implemented in THREADER.

- They are applicable to arbitrary (or ad-hoc) synchronization patterns, not only nested locking patterns or datarace free code.
- THREADER does not restrict the analysis to a bounded number of context-switches, but instead analyzes (implicitly) an unbounded number of context switches.
- The proofs constructed by THREADER are not restricted to thread-modular proofs. In addition, the search for new (transition) predicates can be restricted to thread-modular solutions that favor compositional reasoning, as described in [7].
- THREADER allows an experimental comparison of the two state-of-the-art proof rules for compositional verification of multi-threaded programs in a uniform setting.

References

1. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: FMSD, vol. 34(2), pp. 104–125 (2009)
2. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers, 3rd edn. O'Reilly Media. Sebastopol (2005)
3. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
4. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
5. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: DSN, pp. 221–230 (2010)
6. Gupta, A., Popeea, C., Rybalchenko, A.: Non-monotonic refinement of control abstraction for concurrent programs. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 188–202. Springer, Heidelberg (2010)
7. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344 (2011)
8. Gupta, A., Popeea, C., Rybalchenko, A.: Transition invariants and environment transitions for proving termination of multi-threaded programs (2011) (under submission)
9. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
10. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339 (2008)
11. McKenney, P.: Using Promela and Spin to verify parallel algorithms. LWN.net weekly edition (2007)
12. 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, p. 213. Springer, Heidelberg (2002)
13. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs. I. Acta Inf. 6, 319–340 (1976)
14. 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)

Interactive Synthesis of Code Snippets

Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac*

Swiss Federal Institute of Technology Lausanne (EPFL)
`firstname.lastname@epfl.ch`

Abstract. We describe a tool that applies theorem proving technology to synthesize code fragments that use given library functions. To determine candidate code fragments, our approach takes into account polymorphic type constraints as well as test cases. Our tool interactively displays a ranked list of suggested code fragments that are appropriate for the current program point. We have found our system to be useful for synthesizing code fragments for common programming tasks, and we believe it is a good platform for exploring software synthesis techniques.

1 Introduction

Algorithmic software synthesis from specifications is a difficult problem. Yet software developers perform a form of synthesis on a daily basis, by transforming their intentions into concrete programming language expressions. The goal of our tool, InSynth, is to explore the relationship and synergy between algorithmic synthesis and developers' activities by deploying synthesis for code fragments in interactive settings. To make the problem more tractable, InSynth aims to synthesize small fragments, as opposed to entire algorithms. InSynth builds code fragments containing functions drawn from large and complex libraries. It aims to save the developers the effort of searching for appropriate methods and their compositions. InSynth is deployed within an integrated development environment. When invoked, it suggests multiple meaningful expressions at a given program point, using *type information* and *test cases.*

InSynth primarily relies on type information to perform its synthesis task. When the developer needs a piece of code that computes a value of a given type, they declare the type of this value, using the usual syntax of the Scala programming language [OSV08]. They then invoke InSynth to find suggested code fragments of this type. To find the building blocks for the code fragments, InSynth examines the scope of the given declaration. It uses Ensime [Can11], an incremental Scala compiler integrated into the editor, to gather the available values, fields, and functions. The use of type information is inspired by Prospector [MXBK05], but InSynth has an important additional dimension: it handles generic (parametric) types [DM82]. Generic types are a mainstream mechanism to write safe and reusable code in, e.g., Java, ML, and Scala. They are particularly frequent in libraries.

* Alphabetical order of authors.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 418–423, 2011.

The support for generic types is a fundamental generalization compared to previous tools, which handled only ground types. With generic types, a finite set of declarations will generate an infinite set of possible values, and the synthesis of a value of a given type becomes undecidable. InSynth therefore encodes the synthesis problem in first-order logic. This encoding has the property that a value of the desired type can be built from functions of given types iff there exists a proof for the corresponding theorem in first-order logic. It is therefore related to known connections between proof theory and type theory. In type-theoretic terms, InSynth attempts to check whether there exists a term of a given type in a given polymorphic type environment. If such terms exist, the goal of InSynth is to produce a finite subset of them, ranked according to some criterion.

InSynth implements a custom resolution-based algorithm to find multiple proofs representing candidate code fragments. The use of resolution is related to the traditional deductive program synthesis [MW80], but our approach attempts to derive code fragments by using type information instead of the code itself. As a post-processing step, InSynth filters out the candidate code fragments that crash the program, or that violate assertions or postconditions. This functionality incorporates input/output behavior [JGST10], but uses it mostly to improve the precision of the primary mechanism, the type-driven synthesis.

We believe that an important aspect of the software development process is that an accurate specification is often not available. A synthesis tool should be equipped to deal with under-specified problems, and be prepared to generate multiple alternative solutions when asked to do so. Our algorithm fulfills this requirement: it generates multiple solutions and ranks them using a system of weights. The current weight computation takes into account the proximity of values to the point in which the values are used, as well as user-specified hints, if any. A database of code samples, if available, could be used to derive weights, providing effects similar to some of the previous systems [SC06, MXBK05]. Given a weight function, InSynth directs its search using a technique related to ordered resolution [BG01].

Contributions. InSynth is an interactively deployed synthesis tool based on parameterized types, test cases, and weights indicating preferences. Its algorithmic base is a variation of ordered resolution. We have found InSynth to be fast enough for interactive use and helpful in synthesizing meaningful code fragments.

2 Examples

Consider the problem of retrieving data stored in a file. Suppose that we have the following definitions:

```
def fopen(name:String):File = { ... }
def fread(f:File, p:Int):Data = { ... }
var currentPos : Int = 0
var fname : String = ""
def getData():Data = □
```

The developer is about to define, at the position marked by □, the body of the function getData that computes a value of type Data. When the developer invokes InSynth, the result is a list of valid expressions (snippets) for the given program point, composed from the values in the scope. Assuming that among the definitions we have functions fopen and fread, with the types shown above, InSynth will return as one of the suggestions fread(fopen(fname), currentPos), which is a simple way to retrieve data from the file given the available operations. In our experience, InSynth often returns snippets in a matter of milliseconds. Such snippets may be difficult to find manually for complex and unknown APIs, so InSynth can also be thought as a sophisticated extension of a search and code completion functionality.

Parametric Polymorphism. We next illustrate the support of parametric polymorphism in InSynth. Consider the standard higher-order function map that applies a given function to each element of the list. Assume that the map function is in the scope. Further assume that we wish to define a method that takes as arguments a function from integers to strings and a list of strings, and returns a list of strings.

```
  def map[A,B](f:A ⇒ B, l:List[A]):List[B] = { ... }
  def stringConcat(lst:List[String]):String = { ... }
  def printInts(intList:List[Int], prn:Int ⇒ String):String = □
```

InSynth returns stringConcat(map[Int, String](prn, intList)) as a result, instantiating polymorphic definition of map and composing it with stringConcat. InSynth efficiently handles polymorphic types through resolution and unification.

Using code behavior. The next example shows how InSynth applies testing to discard those snippets that would make code inconsistent. Define the class FileManager containing methods for opening files either for reading or for writing.

```
class Mode(mode:String)
class File(name:String, val state:Mode)
object FileManager {
  private final val WRITE:Mode = new Mode("write")
  private final val READ:Mode = new Mode("read")
  def openForReading(name:String):File = □
    ensuring { result => result.state == READ}
}
object Tests { FileManager.openForReading("book.txt") }
```

If it were based only on types, InSynth would return both **new** File(name, WRITE) and **new** File(name, READ). However, InSynth also checks run-time method contracts (pre- and post-conditions) and verifies whether each of the returned snippets passes the test cases with them. Because of postconditions requiring that the file is open for reading, InSynth discards the snippet **new** File(name, WRITE) and returns only **new** File(name, READ).

Applying User Preferences. The last example demonstrates one way in which a developer can influence the ranking of the returned solutions. We consider the following functionality for managing calendar events.

```
private val events:List[Event] = List.empty[Event]
def reserve(user:User, date:Date):Event = { ... }
def getEvent(user:User, date:Date):Event = { ... }
def remove(user:User, date:Date):Event = □
```

Assume that a user wishes to obtain a code snippet for remove. In general, InSynth ranks the results based on the weight function. We have found that the default computation of the weight is often adequate. Running the above example returns reserve(user, date) and getEvent(user, date), in this order. If this order is not the preferred one, the developer can modify it using elements of text search. To do so, the developer supplies a list of suggested strings indicating the names of some of the methods expected to appear in the code snippet. For example, if the developer invokes InSynth with "getEvent" as a suggestion, the ranking of returned snippets changes, and getEvent(user, date) appears first in the list.

3 Foundations and Algorithm

Our main algorithm is based on first-order resolution. We therefore formalize type constrains in first-order logic. We introduce a predicate $\texttt{hasType}(v, T)$ to indicate that a value v is of a type T. We use a function symbol `arrow` to indicate the function type constructor ($\rightarrow$). First-order logic makes it possible to encode polymorphism using universally quantified variables.

The ranking of the snippets and the entire algorithm strongly rely on a system of weights. The system considers snippets of a smaller weight as preferable to those of a larger weight. The weights of terms extend to the weights of clauses, as in the multiset ordering of clauses in first-order resolution [BG01].

To begin with, we define an ordering on the symbols and assign a weight to each symbol. The user-preferred symbols have the smallest weight (highest preference). They are followed by the local symbols occurring in the current method. The remaining symbols of the corresponding class have a larger weight than the local symbols. Finally, the symbols outside the current class have the largest weight. This includes symbols from the imported libraries and APIs.

Once the ordering and the weights of the symbols are fixed, we compute the weight of a term similarly as in the Knuth-Bendix ordering. The only difference is that we additionally recalculate the weight of every term containing a user-preferred symbol. We do this so that they do not "vanish" when combined with symbols of a larger weight.

Snippet Synthesis Algorithm. Figure 1 describes the basic version of our algorithm. It takes as an input a partial Scala program and a program point where the user asks for a code snippet. Additionally, it also takes as an argument a resource bound, the maximum number of resolution steps.

The first step of the algorithm is to traverse the program syntax tree, create the clauses, and assign the weights to the symbols and clauses. We pick a minimal-weight clause and resolve it with all other clauses of a larger weight.

If we derive a contradiction (empty clause), we extract its proof tree. To generate multiple solutions, we use the proof tree to derive a blocking clause that prevents the same derivation of the empty clause in the future. We add this blocking clause to the clause set. We repeat this procedure until either the clause set becomes saturated, or the given threshold on the resolution steps is exceeded. We then reconstruct terms from the collected proof trees, and create the code snippets. We test the generated snippets by invoking any user-defined test cases and discarding the snippet for which the code crashes.

Backward Reasoning. The implementation of InSynth combines the algorithm described in Figure 1 with backward reasoning. With $?\ T$ we denote the query asking for a value of the type T. The main rule we use is

$$\frac{\texttt{hasType}(x, \texttt{Arrow}(T_1, T_2)) \qquad ?\ T_2}{?\ T_1}$$

By applying backward reasoning we were able to accelerate search for solutions compared to using purely forward reasoning.

INPUT: partial Scala program, program point, maximal number of steps
OUTPUT: list of code snippets

```
def basicSynthesizeSnippet(p : PartialScalaProgram, maxSteps : Int) : List[Snippet] = {
  var weightedClauses = extractClauses(p)
  var saturated = false
  var solutions = emptySet
  var step = 0
  while (step < maxSteps && !saturated) {
    val c : Clause = pickMinWeight(weightedClauses)
    saturated = true
    for (c' ← weightedClauses if weight(c) ≤ weight(c') && c != c')) {
      val newC = resolution(c,c')
      if !(newC in weightedClauses) {
        saturated = false
        if (newC.isEmptyClause) {
          val s = extractSolution(newC)
          solutions = solutions ∪ { s }
          val cBlock = createClausePreventingThisProof(s)
          weightedClauses = weightedClauses union { cBlock }
        }
      }
    }
    step++
  }
  return (solutions.map(proofToSnippet)).filter(passesTest(p))
}
```

Fig. 1. Basic algorithm for synthesizing code snippets

Program	# Loaded Declarations	# Methods in Synthesized Snippets	Time [s]
FileReader	6	4	< 0.001
Map	4	4	< 0.001
FileManager	3	3	< 0.001
Calendar	7	3	< 0.001
FileWriter	320	6	0.093
SwingBorder	161	2	0.016
TcpService	89	2	< 0.001

Fig. 2. Basic algorithm for synthesizing code snippets

4 InSynth Implementation and Evaluation

InSynth is implemented in Scala and built on top of the Ensime plugin [Can11]. It can therefore directly use program information computed by the Scala compiler, including abstract syntax trees and the inferred types. Furthermore, it can generate an appropriate pop-up window with suggested synthesized snippets and allow the user to interactively select the desired fragment.

Figure 2 gives an idea of the performance of the system. We ran all examples on Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM. The running times to find the first solution are usually bellow two milliseconds. Our experience suggests that the algorithm scales well. As an illustration, we were able to synthesize a snippet containing six methods in 0.093 seconds from the set of 320 declarations. Times to encode declarations into FOL formulas range from 0.015 (Calendar) to 0.046 (FileWriter) seconds. If the synthesized snippets need to use more methods from imported libraries, the synthesis typically takes longer, but is typically fast enough to be useful. The above examples and the system InSynth are available on the following web site: `http://lara.epfl.ch/w/insynth`.

References

[BG01] Bachmair, L., Ganzinger, H.: Resolution theorem proving. Handbook of Automated Reasoning, 19–99 (2001)

[Can11] Aemon Cannon. Ensime. Retrieved 20 April (2011), `https://github.com/aemoncannon/ensime/`

[DM82] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL (1982)

[JGST10] Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE, vol. 1 (2010)

[MW80] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. 2(1), 90–121 (1980)

[MXBK05] Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: PLDI (2005)

[OSV08] Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)

[SC06] Sahavechaphan, N., Claypool, K.: Xsnippet: mining for sample code. In: OOPSLA (2006)

Forest Automata for Verification of Heap Manipulation*

Peter Habermehl[1], Lukáš Holík[2,4], Adam Rogalewicz[2],
Jiří Šimáček[2,3], and Tomáš Vojnar[2]

[1] LIAFA, Université Paris Diderot—Paris 7/CNRS, France
[2] FIT, Brno University of Technology, Czech Republic
[3] VERIMAG, UJF/CNRS/INPG, Gières, France
[4] Uppsala University, Sweden

Abstract. We consider verification of programs manipulating dynamic linked data structures such as various forms of singly and doubly-linked lists or trees. We consider important properties for this kind of systems like no null-pointer dereferences, absence of garbage, shape properties, etc. We develop a verification method based on a novel use of tree automata to represent heap configurations. A heap is split into several "separated" parts such that each of them can be represented by a tree automaton. The automata can refer to each other allowing the different parts of the heaps to mutually refer to their boundaries. Moreover, we allow for a hierarchical representation of heaps by allowing alphabets of the tree automata to contain other, nested tree automata. Program instructions can be easily encoded as operations on our representation structure. This allows verification of programs based on a symbolic state-space exploration together with refinable abstraction within the so-called abstract regular tree model checking. A motivation for the approach is to combine advantages of automata-based approaches (higher generality and flexibility of the abstraction) with some advantages of separation-logic-based approaches (efficiency). We have implemented our approach and tested it successfully on multiple non-trivial case studies.

1 Introduction

We address verification of sequential programs with complex *dynamic linked data structures* such as various forms of singly- and doubly-linked lists (SLL/DLL), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and the like) pointers, as well as various forms of trees. We in particular consider C pointer manipulation, but our approach can easily be applied to any other similar language. We concentrate on *safety properties* of the considered programs which includes generic properties like absence of null dereferences, double free operations, dealing with dangling pointers, or memory leakage. Furthermore, to check various shape properties of the involved data structures one can use testers, i.e., parts of code which, in case some desired property is broken, lead the control flow to a designated error location.

* This work was supported by the Czech Science Foundation (projects P103/10/0306, P201/09/P531, and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the Czech-French Barrande project 021023, the BUT FIT project FIT-S-11-1, and the French ANR-09-SEGI project Veridyc .

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 424–440, 2011.

For the above purpose, we propose a novel approach of representing sets of heaps via *tree automata* (TA). In our representation, a heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [15]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by the so called *forest automata* (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we allow alphabets of FA to contain *nested FA*, leading to a *hierarchical encoding of heaps*, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described. Finally, since FA are not closed under union, we work with sets of forest automata, which are an analogy of disjunctive separation logic formulae.

As a nice theoretical feature of our representation, we show that *inclusion* of sets of heaps represented by finite sets of non-nested FA (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLL with head/tail pointers. Moreover, we show how inclusion can be safely approximated for the case of nested FA. Further, C program statements manipulating pointers can be easily encoded as operations modifying FA. Consequently, the symbolic verification framework of *abstract regular tree model checking* [6,7], which comes with automatically refinable abstractions, can be applied.

The proposed approach brings the principle of *local heap manipulation* (i.e., dealing with separated parts of heaps) from separation logic into the world of automata. The motivation is to combine some advantages of using automata and separation logic. Automata provide higher generality and flexibility of the abstraction (see also below) and allow us to leverage the recent advances of efficient use of non-deterministic automata [2,3]. As further discussed below, the use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed in [7].

We have implemented our approach in a prototype tool called *Forester* as a `gcc` plug-in. In our current implementation, if nested FA are used, they are provided manually (similar to the use of pre-defined inductive predicates common in works on separation logic). However, we show that Forester can already successfully handle multiple interesting case studies, proving the proposed approach to be very promising.

Related work. The area of verifying programs with dynamic linked data structures has been a subject of intense research for quite some time. Many different approaches based on logics, e.g., [13,16,15,4,10,14,19,18,8,12], automata [7,5,9], upward closed sets [1], and other formalisms have been proposed. These approaches differ in their generality, efficiency, and degree of automation. Due to space restrictions, we cannot discuss all of them here. Therefore, we concentrate on a comparison with the two closest lines of work, namely, the use of automata as described in [7] and the use of separation logic in the works [4,18] linked with the Space Invader tool. In fact, as is clear from the above, the approach we propose combines some features from these two lines of research.

Compared to [4,18], our approach is more general in that it allows one to deal with tree-like structures, too. We note that there are other works on separation logic, e.g., [14], that consider tree manipulation, but these are usually semi-automated only. An exception is [10] which automatically handles even tree structures, but its mechanism of synthesising inductive predicates seems quite dependent on the fact that the dynamic linked data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning, e.g., that lists are built by adding elements at the end only[1]).

Further, compared to [4,18], our approach comes with a more flexible abstraction. We are not building on just using some inductive predicates, but we combine a use of our nested FA with an automatically refinable abstraction on the TA that appear in our representation. Thus our analysis can more easily adjust to various cases arising in the programs being verified. An example is dealing with lists of lists where the sublists are of length 0 or 1, which is a quite practical situation [17]. In such cases, the abstraction used in [4,18] can fail, leading to an infinite computation (e.g., when, by chance, a list of regularly interleaved lists of length 0 or 1 appears) or generate false alarms (when modified to abstract even pointer links of length 1 to a list segment). For us, such a situation is easy to handle without any need to fine-tune the abstraction manually.

On the other hand, compared with the approach of [7], our newly proposed approach is a bit less general (we cannot, e.g., handle structures such as trees with linked leaves[2]), but on the other hand more scalable. The latter comes from the fact that the representation in [7] is monolithic, i.e., the whole heap is represented by one tree-like structure whereas our new representation is not monolithic anymore. Therefore, the different operations on the heap, e.g., corresponding to a symbolic execution of the verified program, influence only small parts of the encoding (unlike in [7], where the transducers used for this purpose are always operating on the entire automata). Also, the monolithic encoding of [7], based on a fixed tree skeleton over which additional pointer links were expressed using the so-called routing expressions, had problems with deletion of elements inside data structures and with detection of memory leakage (which was in theory possible, but it was so complex that it was never implemented).

2 From Heaps to Forests

In this section, we outline how sets of heaps can be represented by hierarchical forest automata. These automata are tuples of tree automata which accept trees that may refer to each other through the alphabet symbols. Furthermore their alphabet can contain strictly hierarchically nested forest automata. For the purpose of the explanation, *heaps* may be viewed as oriented graphs whose nodes correspond to allocated memory cells and edges to pointer links between these cells. The nodes may be labelled by non-pointer data stored in them (assumed to be from a finite data domain) and by program variables pointing to the nodes. Edges may be labelled by the corresponding selectors.

In what follows, we are representing sets of *garbage free* heaps only, i.e., all memory cells are reachable from pointer variables by following pointer links. However,

[1] We did not find an available implementation of [10], and so we could not try it out ourselves.

[2] Unless a generalisation to FA nested not just strictly hierarchically, but in an arbitrary, possibly cyclic way is considered, which is an interesting subject for future research.

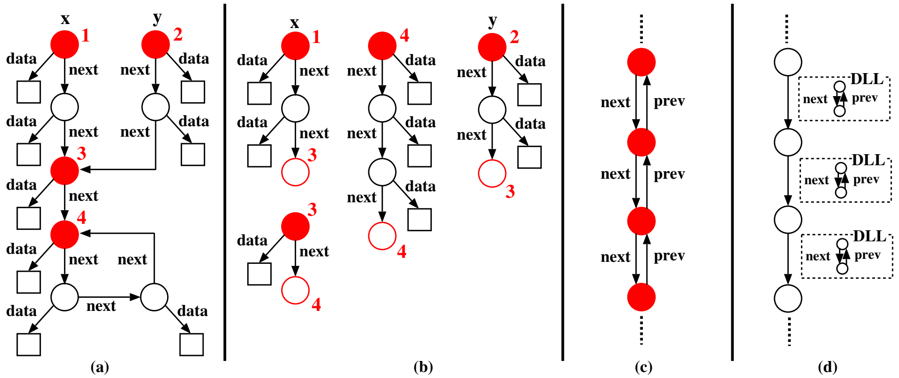


Fig. 1. (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with x ordered before y, (c) a part of a DLL, (d) a hierarchical encoding of the DLL

practically this is not a restriction since the emergence of garbage can be checked for each program statement to be fired and if garbage arises, an error message can be issued and the computation stopped or the garbage removed and the computation continued.

Now, note that each heap graph may be *canonically decomposed* into a tuple of trees as follows. We first identify the *cut-points*, i.e. nodes that are either pointed to by a program variable or that have several incoming edges. Then, we totally order program variables and selectors. Next, cut-points are canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables, taking them in accordance with their order, and exploring the graph according to the order of selectors. Finally, we split the heap graph into tree components rooted at particular cut-points. These components contain all the nodes reachable from their root while not passing through any cut-point, plus a copy of each reachable cut-point, labelled by its number. The tree components are then canonically ordered according to the numbers of their root cut-points. For an illustration of the decomposition, see Figure 1 (a) and (b).

Now, tuples of tree automata (TA), called *forest automata* (FA), accepting tuples of trees whose leaves may refer to the root of any tree out of a given tuple, may be viewed as representing a set of heaps as follows. We simply take a tree from the language of each of the TA and obtain a heap by gluing the tree roots corresponding to cut-points with the leaves referring to them.

Further, we consider in particular *canonicity respecting forest automata* (CFA). CFA encode sets of heaps decomposed in a canonical way, i.e., such that if we take any tuple of trees accepted by the given CFA, construct a heap from them, and then canonically decompose it, we get the tuple of trees we started with. This means that in the chosen tuple there is no tree with a root that does not correspond to a cut-point and that the trees are ordered according to the depth-first traversal as described above. The canonicity respecting form allows us to test inclusion on the sets of heaps represented by CFA by component-wise testing inclusion on the languages of the TA constituting the given CFA.

Note, however, that FA are not closed under union. Clearly, even if we consider FA having the same number of components, uniting the TA component-wise may yield an FA overapproximating the union of the sets of heaps represented by the original FA (cf.

Section 3). Hence, we will have to represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of separation logic formulae. However, as we shall see, inclusion on the sets of heaps represented by SFA is still easily decidable.

The described encoding allows one to represent sets of heaps with a bounded number of cut-points. However, to handle many common dynamic data structures one needs to represent sets of heaps with an *unbounded number of cut-points*. Indeed, in doubly-linked lists (DLLs) for instance, every node is a cut-point. We solve this problem by representing heaps in a *hierarchical way*. In particular, we collect sets of repeated subgraphs (called *components*) in the so-called *boxes*. Every occurrence of such components can then be replaced by a single hyperedge labelled by the appropriate box[3]. In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap hypergraphs* with a bounded number of cut-points at each level of the hierarchy. Figures 1 (c) and (d) illustrate how this approach can reduce DLLs into singly-linked lists (with a DLL segment used as a kind of meta-selector). Sets of heap hypergraphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into (hyper)tree components and represented using FA whose alphabet can contain nested FA.[4] Intuitively, FA that appear in the alphabet of some superior FA play a role similar (but not equal) to that of inductive predicates in separation logic.[5]

The question of deciding inclusion on sets of heaps represented by hierarchical FA remains open. However, we propose a *canonical decomposition of hierarchical hypergraphs* allowing inclusion to be decided for sets of heap hypergraphs represented by FA in the case when the nested FA labelling hyperedges are taken as atomic alphabet symbols. Note that this decomposition is by far not the same as for non-hierarchical heap graphs due to a need to deal with nodes that are not reachable on the top level, but are reachable through edges hidden in some boxes. This result allows one to safely approximate inclusion checking on hierarchically represented heaps, which appears to work quite well in practice.

3 Hypergraphs and Their Representation

We now formalise the notion of hypergraphs and forest automata.

3.1 Hypergraphs

Given a set A and $n \in \mathbb{N}$, let A^n denote the n^{th}-Cartesian power of A and let $A^{\leq n} = \bigcup_{0 \leq i \leq n} A^i$. For an n-tuple $\overline{a} = (a_1, \ldots, a_n) \in A^n$, $n \geq 1$, we let $\overline{a}.i = a_i$ for any $1 \leq i \leq n$.

[3] We may obtain hyperedges here since we allow components to have a single designated input node, but possibly several output nodes.

[4] Since graphs are a special case of hypergraphs, in the following, we will work with hypergraphs only. Moreover, to simplify the definitions, we will work with hyperedge-labelled hypergraphs only. Node labels mentioned above will be put at specially introduced nullary hyperedges leaving from the nodes whose label is to be represented.

[5] For instance, we use a nested FA encoding a DLL segment of length 1, not of length 1 or more as in separation logic: the repetition of the segment is encoded in the structure of the top-level FA.

We call a set A *ranked* if there is a function $\# : A \to \mathbb{N}$. The value $\#(a)$ is called the *rank* of $a \in A$. We call $\#(A) = max(\{\#(a) \mid a \in A\})$ the maximum rank of an element in the given set. For any $n \geq 0$, we denote by A_n the set of all elements of rank n from A.

Given a finite ranked set Γ called a hyperedge alphabet, a Γ-labelled oriented *hypergraph* with designated input and output ports—denoted simply as a hypergraph if no confusion may arise—is a tuple $G = (V, E, I, O)$ where V is a finite set of vertices, $E \subseteq V \times \Gamma \times V^{\leq \#(\Gamma)}$ is a set of hyperedges such that $\forall (v, a, \overline{v}) \in E : \ \overline{v} \in V^{\#(a)}$, and $I, O \subseteq V$ are sets of input and output ports, respectively[6]. We assume that there is a total ordering $\preceq_p \subseteq P \times P$ on the set $P = I \cup O$ of all ports of G. The sets I, O of input/output ports may be empty in which case we may drop them from the hypergraph. For symbols $a \in \Gamma$ with $\#(a) = 0$, we write $(v, a) \in E$ to denote that $(v, a, ()) \in E$.

Given a hyperedge $e = (v, a, (v_1, \ldots, v_n)) \in E$ of a hypergraph $G = (V, E, I, O)$, v is the *source* of e and $v_1, \ldots, v_n$ are *a-successors* of v in G. An (oriented) *path* in G is a sequence $\langle v_0, a_1, v_1, \ldots, a_n, v_n \rangle$, $n \geq 0$, where for all $1 \leq i \leq n$, v_i is an a_i-successor of v_{i-1} in G. G is called *deterministic* iff $\forall (v, a, \overline{v}), (v, a', \overline{v}') \in E$: $a = a' \implies \overline{v} = \overline{v}'$. A hypergraph G is *well-connected* if each node $v \in V$ is reachable through some path from some input port of G. Figure 1 (a) shows a (hyper)graph with two input ports corresponding to the two variables. Edges are labelled by selectors `data` and `next`.

3.2 A Forest Representation of Hypergraphs

A Γ-labelled hypergraph $T = (V, E)$ without input and output ports is an unordered, oriented Γ-labelled *tree* (denoted simply as a tree below) iff (1) it has a single node with no incoming hyperedge (called the *root* of T, denoted $root(T)$), (2) all other nodes of T are reachable from $root(T)$ via some path, and (3) each node has at most one incoming hyperedge. Nodes with no successors are called *leaves*.

Given a finite ranked hyperedge alphabet Γ such that $\Gamma \cap \mathbb{N} = \emptyset$, we call a tuple $F = (T_1, \ldots, T_n, I, O)$, $n \geq 1$, an ordered Γ-labelled *forest* with designated input and output ports (or just a forest) iff (1) for every $i \in \{1, \ldots, n\}$, $T_i = (V_i, E_i)$ is a $\Gamma \cup \{1, \ldots, n\}$-labelled tree where $\forall i \in \{1, \ldots, n\}$, $\#(i) = 0$ and a vertex v with $(v, i) \in E$ is not a source of any other edge (hence it is a leaf), (2) $\forall 1 \leq i_1 < i_2 \leq n : \ V_{i_1} \cap V_{i_2} = \emptyset$, and (3) $I, O \subseteq \{1, \ldots, n\}$ denote the input and output ports, respectively.

We call the sources of edges labelled by $\{1, \ldots, n\}$ *root references* and denote by $rr(T_i)$ the set of all root references in T_i, i.e., $rr(T_i) = \{v \in V_i \mid (v, k) \in E_i, k \in \{1, \ldots, n\}\}$ for each $i \in \{1, \ldots, n\}$. A forest $F = (T_1, \ldots, T_n, I_F, O_F)$, $n \geq 1$, *represents* the hypergraph $\otimes F$ that is obtained by first uniting the trees $T_1, \ldots, T_n$ and then removing every root reference $v \in V_i$, $1 \leq i \leq n$, and redirecting the hyperedges leading to v to the root of T_k where $(v, k) \in E_i$. Formally, $\otimes F = (V, E, I, O)$ where:

- $V = \bigcup_{i=1}^{n} V_i \setminus rr(T_i)$, $E = \bigcup_{i=1}^{n} \{(v, a, \overline{v}') \mid a \in \Gamma \wedge \exists (v, a, \overline{v}) \in E_i \ \forall 1 \leq j \leq \#(a) :$ if $\exists (\overline{v}.j, k) \in E_i$ with $k \in \{1, \ldots, n\}$, then $\overline{v}'.j = root(T_k)$, else $\overline{v}'.j = \overline{v}.j\}$,
- $I = \{root(T_i) \mid i \in I_F\}$, $O = \{root(T_i) \mid i \in O_F\}$,
- the ordering of the set of ports $P = I \cup O$ is defined by : $\forall i, j \in (I_F \cup O_F) : \ root(T_i) \preceq_p root(T_j) \iff i \leq j$.

[6] Intuitively, in hypergraphs representing heaps, input ports correspond to nodes pointed to by program variables or to input nodes of components, and output ports correspond to output nodes of components.

Figure 1 (b) shows a forest decomposition of the graph of Figure 1 (a). It is decomposed into four trees which have designated roots which are referred to in the trees. The decomposition respects the ordering of the two ports corresponding to the variables.

3.3 Minimal and Canonical Forests

We call a forest $F = (T_1, \ldots, T_n, I_F, O_F)$ representing the well-connected hypergraph $G = (V, E, I, O) = \otimes F$ *minimal* iff the roots of the trees $T_1, \ldots, T_n$ correspond to the *cut-points* of G which are those nodes that are either ports or that have more than one incoming hyperedge in G. A minimal forest representation of a hypergraph is unique up to permutations of $T_1, \ldots, T_n$. In order to get a canonical forest representation of a well-connected *deterministic* hypergraph $G = (V, E, I, O)$, we need to canonically order the trees in its minimal forest representation. We do this as follows: First, we assume the set of hyperedge labels Γ to be totally ordered via some ordering $\preceq_\Gamma$. Then, a depth-first traversal (DFT) on G is performed starting with the DFT stack containing the set $I \cup O$ in the given order $\preceq_p$, the smallest node being on top of the stack. We now call a forest representation $F = (T_1, \ldots, T_n, I_F, O_F)$ of G *canonical* iff it is minimal and the trees $T_1, \ldots, T_n$ appear in F in the following order: First, the trees whose roots correspond to ports appear in the order given by $\preceq_p$, and then the rest of the trees appears in the same order in which their roots are visited in the described DFT of G. A canonical representation is obtained this way since we consider G to be deterministic. Clearly the forest of Figure 1 (b) is a canonical representation of the graph of Figure 1 (a).

3.4 Forest Automata

We now define forest automata as tuples of tree automata encoding sets of forests and hence sets of hypergraphs. To be able to use classical tree automata, we will need to work with trees that are ordered, node-labelled, with the node labels being ranked.

Ordered Trees. Let ε denote the empty sequence. An *ordered tree t* over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ satisfying the following conditions: (1) $dom(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$, and (2) for each $p \in dom(t)$, if $\#(t(p)) = n \geq 0$, then $\{i \mid pi \in dom(t)\} = \{1, \ldots, n\}$. Each sequence $p \in dom(t)$ is called a *node* of t. For a node p, the i^{th} *child* of p is the node pi, and the i^{th} *subtree* of p is the tree t' such that $t'(p') = t(pip')$ for all $p' \in \mathbb{N}^*$. A *leaf* of t is a node p with no children, i.e., there is no $i \in \mathbb{N}$ with $pi \in dom(t)$. Let $\mathbb{T}(\Sigma)$ be the set of all ordered trees over Σ.

For an $\preceq_\Gamma$-ordered hyperedge alphabet Γ, it is easy to convert Γ-labelled trees into node-labelled ordered trees and back (up to isomorphism). We label a node of an ordered tree by the set of labels of the hyperedges leading from the corresponding node in the original tree, and we order the successors of the node w.r.t. the hyperedge labels through which they are reachable (while always keeping tuples of nodes reachable via the same hyperedge together). The rank of the new node label is then given by the sum of the original hyperedge labels embedded into it. Below, we use the notion Σ_Γ to denote the ranked node alphabet obtained from Γ as described above (w.r.t. a total ordering $\preceq_\Gamma$ that we will from now on assume to be always associated with Γ) and $ot(T)$ to denote the ordered tree obtained from a Γ-labelled tree T. For a formal description, see [11].

Tree Automata. A (finite, non-deterministic, bottom-up) *tree automaton* (abbreviated as TA in the following) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1, \ldots, q_n), f, q)$ where $n \geq 0$, $q_1, \ldots, q_n, q \in Q$, $f \in \Sigma$, and $\#(f) = n$. We use $(q_1, \ldots, q_n) \xrightarrow{f} q$ to denote that $((q_1, \ldots, q_n), f, q) \in \Delta$. In the special case where $n = 0$, we speak about the so called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{f} q$.

A *run* of $\mathcal{A}$ over a tree $t \in \mathbb{T}(\Sigma)$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $p \in dom(t)$ where $q = \pi(p)$, if $q_i = \pi(pi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1, \ldots, q_n) \xrightarrow{t(p)} q$. We write $t \overset{\pi}{\Longrightarrow} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \overset{\pi}{\Longrightarrow} q$ for some run π. The *language* of a state q is defined by $\mathcal{L}(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} \mathcal{L}(q)$.

Forest Automata. Let Γ be a ranked hyperedge alphabet ordered by $\preceq_\Gamma$. We call an n-tuple $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$, $n \geq 1$, a *forest automaton* with designated input/output ports (called also FA) over Γ iff for all $1 \leq i \leq n$, $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ is a TA with $\Sigma = \Sigma_{\Gamma \cup \{1, \ldots, n\}}$ where $\forall 1 \leq i \leq n : \#(i) = 0$. The sets $I, O \subseteq \{1, \ldots, n\}$ are sets of input/output ports, respectively. $\mathcal{F}$ defines the *forest language* $\mathcal{L}_F(\mathcal{F}) = \{(T_1, \ldots, T_n, I, O) \mid (\forall 1 \leq i \leq n : ot(T_i) \in \mathcal{L}(\mathcal{A}_i)) \wedge (\forall 1 \leq i < j \leq n : T_i = (V_i, E_i) \wedge T_j = (V_j, E_j) \implies V_i \cap V_j = \emptyset)\}$. The *hypergraph language* of $\mathcal{F}$ is then the set $\mathcal{L}(\mathcal{F}) = \{\otimes F \mid F \in \mathcal{L}_F(\mathcal{F})\}$. An FA $\mathcal{F}$ *respects canonicity* iff each forest $F \in \mathcal{L}_F(\mathcal{F})$ is a canonical representation of some well-connected hypergraph, namely, the hypergraph $G = \otimes F$. We abbreviate canonicity respecting FA as CFA. It is easy to see that comparing sets of hypergraphs represented by CFA can be done *component-wise* as described in the below lemma.

Lemma 1. *Let $\mathcal{F}_1 = (\mathcal{A}_1^1, \ldots, \mathcal{A}_{n_1}^1, I_1, O_1)$ and $\mathcal{F}_2 = (\mathcal{A}_1^2, \ldots, \mathcal{A}_{n_2}^2, I_2, O_2)$ be two CFA. Then, $\mathcal{L}(\mathcal{F}_1) \subseteq \mathcal{L}(\mathcal{F}_2)$ iff (1) $n_1 = n_2$, (2) $I_1 = I_2$, (3) $O_1 = O_2$, and (4) $\forall 1 \leq i \leq n : \mathcal{L}(\mathcal{A}_i^1) \subseteq \mathcal{L}(\mathcal{A}_i^2)$.*

Sets of Forest Automata. The class of languages of forest automata is not closed under union. The reason is that a forest language of an FA is the Cartesian product of the languages of all its components and that not every union of Cartesian products may be expressed as a single Cartesian product. For instance, consider two CFA $\mathcal{F} = (\mathcal{A}, \mathcal{B}, I, O)$ and $\mathcal{F}' = (\mathcal{A}', \mathcal{B}', I, O)$ such that $\mathcal{L}_F(\mathcal{F}) = \{(a, b, I, O)\}$ and $\mathcal{L}_F(\mathcal{F}') = \{(c, d, I, O)\}$ where a, b, c, d are distinct trees. The forest language of the FA $(\mathcal{A} \cup \mathcal{A}', \mathcal{B} \cup \mathcal{B}', I, O)$ is $\{(x, y, I, O) \mid (x, y) \in \{a, c\} \times \{b, d\}\}$) and thus there is no CFA with the hypergraph language equal to $\mathcal{L}(\mathcal{F}) \cup \mathcal{L}(\mathcal{F}')$. Therefore, we will work with *finite sets of (canonicity-respecting) forest automata*, S(C)FA for short, where the language $\mathcal{L}(S)$ of a finite set S of FA is defined as the union of the languages of its elements.

Note that any FA can be transformed (split) into an SCFA whose CFA represent hypergraphs having a different interconnection of the cut-points (see [11] for details).

Testing Inclusion on SFA. The problem of checking inclusion on SFA, this is, checking whether $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ where S, S' are SFA, can be reduced to a problem of checking inclusion on tree automata. We may w.l.o.g. assume that S and S' are SCFA.

For an FA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$ where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for each $1 \leq i \leq n$, we define the TA $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{\curlywedge_n^{I,O}\}, Q, \Delta, \{q^{top}\})$ where $\curlywedge_n^{I,O} \notin \Sigma$ is a symbol with $\#(\curlywedge_n^{I,O}) = n$, $q^{top} \notin \bigcup_{i=1}^{n} Q_i$, $Q = \bigcup_{i=1}^{n} Q_i \cup \{q^{top}\}$, and $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \Delta^{top}$. The set Δ^{top} contains the rule $\curlywedge_n^{I,O}(q_1, \ldots, q_n) \to q^{top}$ for each $(q_1, \ldots, q_n) \in F_1 \times \cdots \times F_n$. Intuitively, $\mathcal{A}^{\mathcal{F}}$ accepts the trees where n-tuples of ordered trees representing hypergraphs from $\mathcal{L}(\mathcal{A})$ are topped by a designated root node labelled by $\curlywedge_n^{I,O}$. It is now easy to see that the following lemma holds (in the lemma, "$\cup$" stands for the usual tree automata union).

Lemma 2. *For two SCFA S and S', $\mathcal{L}(S) \subseteq \mathcal{L}(S') \iff \mathcal{L}(\bigcup_{\mathcal{F} \in S} \mathcal{A}^{\mathcal{F}}) \subseteq \mathcal{L}(\bigcup_{\mathcal{F}' \in S'} \mathcal{A}^{\mathcal{F}'})$.*

4 Hierarchical Hypergraphs

We inductively define hierarchical hypergraphs as hypergraphs with hyperedges possibly labelled by hierarchical hypergraphs of a lower level. Let Γ be a ranked alphabet.

4.1 Hierarchical Hypergraphs, Components, and Boxes

A Γ-labelled (hierarchical) *hypergraph of level* 0 is any Γ-labelled hypergraph. For $j \in \mathbb{N}$, a *hypergraph of level* $j+1$ is defined as a hypergraph over the alphabet $\Gamma \cup \mathbb{B}_j$.

To define the set $\mathbb{B}_j$, we first define a Γ-labelled *component of level* j as a hypergraph $C = (V, E, I, O)$ of level j which satisfies the requirement that $|I| = 1$ and $I \cap O = \emptyset$.

Then, $\mathbb{B}_j$ is the set of Γ-labelled *boxes* of level j where each box $B \in \mathbb{B}_j$ is a set of Γ-labelled components of level j which all have the same number of output ports. We call this number the *rank* of B, require that $\Gamma \cap \mathbb{B}_j = \emptyset$ and call boxes over Γ that appear as labels of hyperedges of a hierarchical hypergraph H over Γ *nested boxes* of H.

Semantics of hierarchical hypergraphs and boxes. We are going to define the semantics of a hierarchical hypergraph H as a set of hypergraphs $[\![H]\!]$. If H is of level 0, then $[\![H]\!] = \{H\}$. The semantics of a box B, denoted $[\![B]\!]$, is the union of semantics of its elements (i.e., it is a set of components of level 0). In the semantics of a hypergraph $H = (V, E, I, O)$ of level $j > 0$, each hyperedge labelled by a box $B \in \mathbb{B}_{j-1}$ is substituted in all possible ways by components from the semantics of B (as in ordinary hyperedge replacement used in graph grammars). To define this formally, we use an auxiliary operation *plug*. Let $e = (v, a, \overline{v}) \in E$ be a hyperedge with $\#(a) = k$ and let $C = (V', E', I', O')$ be a component of level $j-1$ to be plugged into H instead of e. Let $(o_1, \ldots, o_k)$ be the set O' ordered according to $\preceq_P$. W.l.o.g., assume $V \cap V' = \emptyset$. For any $w \in V'$, we define an auxiliary port matching function $\rho(w)$ such that (1) if $w \in I'$, $\rho(w) = v$, (2) if $w = o_i, 1 \leq i \leq k$, $\rho(w) = \overline{v}.i$, and (3) $\rho(w) = w$ otherwise. We define $plug(H, e, C) = (V'', E'', I, O)$ by setting $V'' = V \cup (V' \setminus (I' \cup O'))$ and $E'' = (E \setminus \{e\}) \cup \{(v'', a', \overline{v}'') \mid \exists (v', a', \overline{v}') \in E' : \ \rho(v') = v'' \ \wedge \ \forall 1 \leq i \leq k : \ \rho(\overline{v}'.i) = \overline{v}''.i\}$. Now, the semantics of a hypergraph $H = (V, E, I, O)$ of level j is defined recursively as follows: Let $Plug(H) = \{plug(H, e, C) \mid e = (v, B, \overline{v}) \in E \ \wedge \ B \in \mathbb{B}_{j-1} \ \wedge \ C \in [\![B]\!]\}$. If $Plug(H) = \emptyset$, $[\![H]\!] = \{H\}$, otherwise $[\![H]\!] = \bigcup_{H' \in Plug(H)} [\![H']\!]$. Figure 1 (d) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Figure 1 (c) obtained using *Plug*. The only box used represents a DLL segment.

4.2 Hierarchical Forest Automata

To represent sets of deterministic hierarchical hypergraphs, we propose to use (hierarchical) FA whose alphabet contains SFA representing the needed nested boxes. For a hierarchical FA $\mathcal{F}$, we will denote by $\mathcal{L}_H(\mathcal{F})$ the set of hierarchical hypergraphs represented by it. Likewise, for a hierarchical SFA S, we let $\mathcal{L}_H(S) = \bigcup_{\mathcal{F} \in S} \mathcal{L}_H(\mathcal{F})$.

Let Γ be a *finite* ranked alphabet. Formally, an FA $\mathcal{F}$ over Γ of level 0 is an ordinary FA over Γ, and we let $\mathcal{L}_H(\mathcal{F}) = \mathcal{L}(\mathcal{F})$. For $j \in \mathbb{N}$, $\mathcal{F}$ is an FA over Γ of level $j+1$ iff $\mathcal{F}$ is an ordinary FA over an alphabet $\Gamma \cup X$ where X is a finite set of SFA of level j (called *nested SFA* of $\mathcal{F}$) such that for every $S \in X$, $\mathcal{L}_H(S)$ is a box over Γ of level j. The rank $\#(S)$ of S equals the rank of the box $\mathcal{L}_H(S)$.

For FA of level $j+1$, $\mathcal{L}_H(\mathcal{F})$ is defined as the set of hierarchical hypergraphs that arise from the hypergraphs in $\mathcal{L}(\mathcal{F})$ by replacing SFA on their edges by the boxes they represent. Formally, $\mathcal{L}_H(\mathcal{F})$ is the set of hypergraphs of level $j+1$ such that $(V,E,I,O) \in \mathcal{L}_H(\mathcal{F})$ iff there is a hypergraph $(V,E',I,O) \in \mathcal{L}(\mathcal{F})$ where $E = \{(v,a,\bar{v}) \mid (v,a,\bar{v}) \in E' \wedge a \in \Gamma\} \cup \{(v,\mathcal{L}_H(S),\bar{v}) \mid (v,S,\bar{v}) \in E' \wedge S \in X\}$.

Notice that a hierarchical SFA of any level has finitely many nested SFA of a lower level only, and the number of levels if finite. Therefore, a hierarchical SFA is a finitely representable object. Notice also that even though the maximum number of cut-points of hypergraphs from $\mathcal{L}_H(S)$ is fixed (SFA always accept hypergraphs with a fixed maximum number of cut-points), the number of cut-points of hypergraphs in $[\![\mathcal{L}_H(S)]\!]$ may be unbounded. The reason is that hypergraphs from $\mathcal{L}_H(S)$ may contain an unbounded number of hyperedges labelled by boxes B such that hypergraphs from $[\![B]\!]$ contain cut-points too. These cut-points then appear in hypergraphs from $[\![\mathcal{L}_H(S)]\!]$, but they are not visible at the level of hypergraphs from $\mathcal{L}_H(S)$.

Hierarchical SFA are therefore finite representations of sets of hypergraphs with possibly unbounded numbers of cut-points.

4.3 Inclusion and Well-Connectedness on Hierarchical SFA

In this section, we aim at checking well-connectedness and inclusion of sets of hypergraphs represented by hierarchical FA. Since considering the full class of hierarchical hypergraphs would unnecessarily complicate our task, we introduce restrictions of hierarchical automata that rule out some rather artificial scenarios and that allow us to handle the automata hierarchically (i.e., using some pre-computed information for nested FA rather than having to unfold the entire hierarchy all the time). In particular, we enforce that for a hierarchical hypergraph H, well-connectedness of hypergraphs in $[\![H]\!]$ is equivalent to the so-called box-connectedness of H introduced below, and, further, determinism of graphs from $[\![H]\!]$ is equivalent to determinism of H.[7]

Proper boxes and well-formed hypergraphs. Given a component C of level 0 over Γ, we define its *backward reachability* set $br(C)$ as the set of indices i for which there is

[7] Notice that for a general hierarchical hypergraph H, well-connectedness of H is nor implied neither implies well-connectedness of hypergraphs from $[\![H]\!]$. This holds also for determinism. The reason is that a component C in a nested box of H may interconnect its ports in an arbitrary way. It may contain paths from output ports to both input and output ports, but it may be missing paths from the input port to some of the output ports.

a path from the i-th output port of C *back* to the input port of C. Given a box B over Γ, we inductively define B to be *proper* iff all its nested boxes are proper, $br(C_1) = br(C_2)$ for any $C_1, C_2 \in [\![B]\!]$ (we use $br(B)$ to denote $br(C)$ for $C \in [\![B]\!]$), and the following holds for all components $C \in [\![B]\!]$: (1) C is well-connected. (2) If there is a path from the i-th to the j-th output port of C, $i \neq j$, then $i \in br(C)$.[8] A hierarchical hypergraph H is called *well-formed* if all its nested boxes are proper. In that case, the conditions above imply that either all or no graphs from $[\![H]\!]$ are well-connected and that well-connectedness of graphs in $[\![H]\!]$ may be judged based only on the knowledge of $br(B)$ for each nested box B of H, without a need to reason about the semantics of B (in particular, Condition 2 guarantees that we do not have to take into account paths that interconnect output ports of B). This is formalised below.

Box-connectedness. Let $H = (V, E, I, O)$ be a well-formed hierarchical hypergraph over Γ with a set X of nested boxes. We define the *backward reachability graph* of H as the hypergraph $H^{br} = (V, E \cup E^{br}, I, O)$ over $\Gamma \cup X \cup X^{br}$ where $X^{br} = \{(B,i) \mid B \in X \wedge i \in br(B)\}$ and $E^{br} = \{(v_i, (B,i), (v)) \mid B \in X \wedge (v, B, (v_1, \ldots, v_n)) \in E \wedge i \in br(B)\}$. Then we say that H is *box-connected* iff H^{br} is well-connected. The below lemma clearly holds.

Lemma 3. *If H is a well-formed hierarchical hypergraph, then the hypergraphs from $[\![H]\!]$ are well-connected iff H is box-connected. Moreover, if hypergraphs from $[\![H]\!]$ are deterministic, then both H and H^{br} are deterministic hypergraphs.*

We straightforwardly extend the above notions to hypergraphs with hyperedges labelled by hierarchical SFA, treating these SFA-labels as if they were the boxes they represent. Particularly, we call a hierarchical SFA S proper iff it represents a proper box, we let $br(S) = br([\![\mathcal{L}_H(S)]\!])$, and for a hypergraph H over $\Gamma \cup Y$ where Y is a set of proper SFA, its backward reachability hypergraph H^{br} is defined based on br in the same way as backward reachability hypergraph of a hierarchical hypergraph above (just instead of boxes, we deal with their SFA representations). We also say that H is box-connected iff H^{br} is well-connected.

Given an FA $\mathcal{F}$ over Γ with proper nested SFA, we can check well-connectedness of graphs from $[\![\mathcal{L}_H(\mathcal{F})]\!]$ as follows: (1) for each nested SFA S of $\mathcal{F}$, we compute (and cache for further use) the value $br(S)$, and (2) using this value, we check box-connectedness of graphs in $\mathcal{L}(\mathcal{F})$ without a need of reasoning about the inner structure of the nested SFA. In [11], we describe how this computation may be done by inspecting rules of the component TA of $\mathcal{F}$. Properness of nested SFA may be checked on the level of TA too as also described [11].

Checking inclusion on hierarchical automata over Γ with nested boxes from X, i.e., given two hierarchical FA $\mathcal{F}$ and $\mathcal{F}'$, checking whether $[\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$, is a hard problem, even under the assumption that nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ are proper. We have not even answered the question of its decidability yet. In this paper, we choose a pragmatic approach and give only a semialgorithm that is efficient and works well in practical cases. The idea is simple. Since the implications $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}') \implies \mathcal{L}_H(\mathcal{F}) \subseteq \mathcal{L}_H(\mathcal{F}') \implies [\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$ obviously hold, we may safely approximate the

[8] Notice that this definition is correct since boxes of level 0 have no nested boxes, and the recursion stops at them.

solution of the inclusion problem by deciding whether $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ (i.e., we abstract away the semantics of nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ and treat them as ordinary labels).

From now on, assume that our hierarchical FA represent only deterministic well-connected hypergraphs, i.e., that $[\![\mathcal{L}_H(\mathcal{F})]\!]$ and $[\![\mathcal{L}_H(\mathcal{F}')]\!]$ contain only well-connected deterministic hypergraphs. Note that this assumption is in particular fulfilled for hierarchical FA representing garbage-free heaps.

We cannot directly use the results on inclusion checking of Section 3.4, based on a canonical forest representation and canonicity respecting FA, since they rely on well-connectedness of hypergraphs from $\mathcal{L}(\mathcal{F})$ and $\mathcal{L}(\mathcal{F}')$, which is now *not* necessarily the case. However, by Lemma 3, every graph H from $\mathcal{L}(\mathcal{F})$ or $\mathcal{L}(\mathcal{F}')$ is box-connected and both H and H^{br} are deterministic. As we show below, these properties are still sufficient to define a canonical forest representation of H, which in turn yields a canonicity respecting form of hierarchical FA.

Canonicity respecting hierarchical FA. Let Y be a set of proper SFA over Γ. We aim at a canonical forest representation $F = (T_1, \ldots, T_n, I, O)$ of a $\Gamma \cup Y$-labelled hypergraph $H = \oplus F$ which is box-connected and such that both H and H^{br} are deterministic. By extending the approach used in Section 3.4, this will be achieved via an unambiguous definition of the *root-points* of H, i.e., the nodes of H that correspond to the roots of the trees $T_1, \ldots, T_n$, and their ordering.

The root-points of H are defined as follows. First, every cut-point (port or a node with more than one incoming edge) is a *root-point of Type 1*. Then, every node with no incoming edge is a *root-point of Type 2*. Root-points of Type 2 are entry points of parts of H that are not reachable from root-points of Type 1 (they are only backward reachable). However, not every such part of H has a unique entry point which is a root-point of Type 2. Instead, there might be a simple loop such that there are no edges leading into the loop from outside. To cover a part of H that is reachable from such a loop, we have to choose exactly one node of the loop to be a root-point. To choose one of them unambiguously, we define a total ordering $\preceq_H$ on nodes of H and choose the smallest node wrt. this ordering to be a *root-point of Type 3*. After unambiguously determining all root-points of H, we may order them according to $\preceq_H$ and we are done.

A suitable total ordering $\preceq_H$ on V can be defined taking an advantage of the fact that H^{br} is well-connected and deterministic. Therefore, it is obviously possible to define $\preceq_H$ as the order in which the nodes are visited by a deterministic depth-first traversal that starts at input ports. The details on how this may be algorithmically done on the structure of forest automata may be found in [11].

We say that a hierarchical FA $\mathcal{F}$ over Γ with proper nested SFA and such that hypergraphs from $[\![\mathcal{L}_H(\mathcal{F})]\!]$ are deterministic and well-connected *respects canonicity* iff each forest $F \in \mathcal{L}_F(\mathcal{F})$ is a canonical representation of the hypergraph $\otimes F$. We abbreviate canonicity respecting hierarchical FA as hierarchical CFA. Analogically as for ordinary CFA, respecting canonicity allows us to compare languages of hierarchical CFA component-wise as described in the below lemma.

Lemma 4. *Let $\mathcal{F}_1 = (\mathcal{A}^1_1, \ldots, \mathcal{A}^1_{n_1}, I_1, O_1)$ and $\mathcal{F}_2 = (\mathcal{A}^2_1, \ldots, \mathcal{A}^2_{n_2}, I_2, O_2)$ be two hierarchical CFA. Then, $\mathcal{L}(\mathcal{F}_1) \subseteq \mathcal{L}(\mathcal{F}_2)$ iff (1) $n_1 = n_2$, (2) $I_1 = I_2$, (3) $O_1 = O_2$, and (4) $\forall 1 \leq i \leq n:\ \mathcal{L}(\mathcal{A}^1_i) \subseteq \mathcal{L}(\mathcal{A}^2_i)$.*

Lemma 4 allows us to safely approximate inclusion of the sets of hypergraphs encoded by hierarchical FA (i.e., to safely approximate the test $[\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$ for hierarchical FA $\mathcal{F}$, $\mathcal{F}'$). This turns out to be sufficient for all our case studies (cf. Section 6). Moreover, the described inclusion checking is precise at least in some cases as discussed in [11]. A generalization of the result to sets of hierarchical CFA can be obtained as for ordinary SFA. Hierarchical FA that do not respect canonicity may be algorithmically split into several hierarchical CFA, similarly as ordinary CFA (see [11]).

5 The Verification Procedure Based on Forest Automata

We now briefly describe our verification procedure. As already said, we consider sequential, non-recursive C programs manipulating dynamic linked data structures via program statements given below[9]. Each allocated cell may have several next pointer selectors and contain data from some finite domain[10] (below, *Sel* denotes the set of all selectors and *Data* denotes the data domain). The cells may be pointed by program variables (whose set is denoted as *Var* below).

Heap Representation. As discussed in Section 2, we encode a single heap configuration as a deterministic $(Sel \cup Data \cup Var)$-labelled hypergraph with the ranking function being such that $\#(x) = 1 \Leftrightarrow x \in Sel$ and $\#(x) = 0 \Leftrightarrow x \in Data \cup Var$, in which nodes represent allocated memory cells, unary hyperedges (labelled by symbols from *Sel*) represent selectors, and the nullary hyperedges (labelled by symbols from $Data \cup Var$) represent data values and program variables[11]. Input ports of the hypergraphs are nodes pointed to by program variables. Null and undefined values are modelled as two special nodes `null` and `undef`. We represent sets of heap configurations as hierarchical $(Sel \cup Data \cup Var)$-labelled SCFA.

Symbolic Execution. The symbolic computation of reachable heap configurations is done over a control flow graph (CFG) obtained from the source program. A control flow action a applied to a hypergraph H (i.e., to a single configuration) returns a hypergraph $a(H)$ that is obtained from H as follows. Nondestructive actions $\mathtt{x} = \mathtt{y}$, $\mathtt{x} = \mathtt{y\text{->}s}$, or $\mathtt{x} = \mathtt{null}$ remove the x-label from its current position and label with it the node pointed by y, the s-successor of that node, or the `null` node, respectively. The destructive action $\mathtt{x\text{->}s} = \mathtt{y}$ replaces the edge $(v_{\mathtt{x}}, \mathtt{s}, v)$ by the edge $(v_{\mathtt{x}}, \mathtt{s}, v_{\mathtt{y}})$ where $v_{\mathtt{x}}$ and $v_{\mathtt{y}}$ are the nodes pointed to by x and y, respectively. Further, `malloc(x)` moves the x-label to a newly created node, `free(x)` removes the node pointed to by x (and links x and all aliased variables with `undef`), and $\mathtt{x\text{->}data} = \mathtt{d}_{new}$ replaces the edge $(v_{\mathtt{x}}, \mathtt{d}_{old})$ by the edge $(v_{\mathtt{x}}, \mathtt{d}_{new})$. Evaluating a guard g applied on H amounts to a simple test of equality of nodes or equality of data fields of nodes. Dereferences of `null` and `undef`

[9] Most C statements for pointer manipulation can be translated to these statements, including most type casts and restricted pointer arithmetic.

[10] No abstraction for such data is considered.

[11] Below, to simplify the informal description, we say that a node is labelled by a variable instead of saying that the variable labels a nullary hyperedge leaving from that node.

are of course detected (as an attempt to follow a non-existing hyperedge) and an error is announced. Emergence of garbage is detected iff $a(H)$ is not well-connected.[12]

We, however, compute not on single hypergraphs representing particular heaps but on sets of them represented by hierarchical SCFA. For now, we assume the nested SCFA used to be provided by the user. For a given control flow action (or guard) x and a hierarchical SCFA S, we need to symbolically compute an SCFA $x(S)$ s.t. $[\![\mathcal{L}_H(x(S))]\!]$ equals $\{x(H) \mid H \in [\![\mathcal{L}_H(S)]\!]\}$ if x is an action and $\{H \in [\![\mathcal{L}_H(S)]\!] \mid x(H)\}$ if x is a guard.

Derivation of the SCFA $x(S)$ from S involves several steps. The first phase is materialisation, where we unfold nested SFA representing boxes that hide data values or pointers referred to by x. We note that we are unfolding only SFA in the closest neighbourhood of the involved pointer variables; thus, on the level of TA, we touch only nested SFA adjacent to root-points. In the next phase, we introduce additional root-points for every node referred to by x to the forest representation. Third, we perform the actual update, which due to the previous step amounts to manipulation with root-points only (see [11] for details). Last, we repeatedly fold (apply) boxes and normalise (transform the obtained SFA into a canonicity respecting form) until no further box can be applied, so that we end up with an SCFA. We note that like unfolding, folding is also done only in the closest neighbourhood of root-points.

Unfolding is, loosely speaking, done by replacing a TA rule labelled by a nested SFA by the nested SFA itself (plus the proper binding of states of the top-level SFA to ports of the nested SFA). Folding is currently based on detecting isomorphism of a part of the top-level SFA and a nested SFA. The part of the top-level SFA is then replaced by a single rule labelled by the nested SFA. We note that this may be further improved by using language inclusion instead of isomorphism of automata.

The Fixpoint Computation. The verification procedure performs a classical (forward) control-flow fixpoint computation over the CFG, where flow values are hierarchical SCFA that represent sets of possible heap configurations at particular program locations. We start from the input location with the SCFA representing an empty heap with all variables undefined. The join operator is the union of SCFA. With every edge from a source location l labelled by x (an action or a guard), we associate the flow transfer function f_x. Function f_x takes the flow value (SCFA) S at l as its input and (1) computes the SCFA $x(S)$, (2) applies abstraction to $x(S)$, and returns the result.

Abstraction may be done by applying the general techniques described in [6] to the individual TA inside FA. Particularly, the abstraction collapses states with similar languages (based on their languages up-to certain tree depth or using predicate languages).

To detect spurious counterexamples and to refine abstraction, we use a *backward run* similarly as in [6]. This is possible since the steps of the symbolic execution may be reversed, and it is also possible to compute almost precise intersections of hierarchical SFA. More precisely, given SCFA S_1 and S_2, we can compute an SCFA S such that $[\![\mathcal{L}_H(S)]\!] \subseteq [\![\mathcal{L}_H(S_1)]\!] \cap [\![\mathcal{L}_H(S_2)]\!]$. This underapproximation is safe since it can lead

[12] Further, we note that we also handle a restricted pointer arithmetic. This is basically done by indexing elements of *Sel* by integers to express that the target of a pointer is an address of a memory cell plus or minus a certain offset. The formalism described in the paper may be easily adapted to support this feature.

Table 1. Experimental results

Example	Forester	Invader	ARTMC	Example	Forester	Invader	ARTMC
SLL (delete)	0.04	0.1	0.5	SLL (reverse)	0.04	0.03	
SLL (bubblesort)	0.12	Err		SLL (insertsort)	0.09	0.1	
SLL (mergesort)	0.12	Err		SLL of CSLLs	0.11	T	
SLL+head	0.04	0.06		SLL of 0/1 SLLs	0.13	T	
SLL$_{\text{Linux}}$	0.05	T		DLL (insert)	0.07	0.08	0.4
DLL (reverse)	0.05	0.09	1.4	DLL (insertsort1)	0.35	0.18	1.4
DLL (insertsort2)	0.16	Err		CDLL	0.04	0.09	
DLL of CDLLs	0.32	T		SLL of 2CDLLs$_{\text{Linux}}$	0.11	T	
tree	0.11		3	tree+stack	0.10		
tree+parents	0.18			tree (DSW)	0.41		o.o.m.

neither to false positives nor to false negatives (it could only cause the computation not to terminate). Moreover, for the SCFA that appear in the case studies in this paper, the intersection we compute actually is precise. More details can be found in [11].

6 Implementation and Experimental Results

We have implemented the proposed approach in a prototype tool called *Forester*, having the form of a gcc plug-in. The core of the tool is our own library of TA that uses the recent technology for handling nondeterministic automata (particularly, methods for reducing the size of TA and for testing language inclusion on them [2,3]). The fixpoint computation is accelerated by the so-called finite height abstraction that is based on collapsing states of TA that have the same languages up to certain depth [6].

Although our implementation is an early prototype, the results are encouraging with regard to the generality of structures the tool can handle, precision of the generated invariants as well as the running times. We tested the tool on sample programs with various types of lists (singly, doubly linked, cyclic, nested), trees, and their combinations. Basic memory safety properties—in particular, absence of null and undefined pointer dereferences, double free operations, and absence of garbage—were checked.

We have compared performance of our tool with the tool Space Invader [4] based on separation logic and also with the tool ARTMC [7] based on abstract regular tree model checking. The comparison with Space Invader was done against examples with lists only since Invader does not handle trees. A higher flexibility of our automata abstraction manifests itself on several examples where Invader does not terminate. This is particularly well visible at the test case with a list of sublists of lengths 0 or 1 (discussed already in the introduction). Our technique handles this example smoothly (without any need to add some special inductive predicates that could decrease the performance or generate false alarms). The ARTMC tool can, in principle, handle more general structures than we can currently handle (such as trees with linked leaves). However, the used representation of heap-configurations is much heavier which causes ARTMC not to scale that well. (Since it is difficult to encode the input for ARTMC, we have tried only some interesting cases.)

Table 1 summarises running times (in seconds) of the three tools on our case studies. The value T means that the running time exceeded 30 minutes, o.o.m. means that the tool ran out of memory, and the value Err stands for a failure of symbolic execution. The names of experiments in the table contain the name of the data structure handled by the program, which ranges over "SLL" for singly-linked lists, "DLL" for doubly linked lists (the prefix "C" means cyclic), "tree" for binary trees, "tree+parents" for trees with parent pointers. Nested variants of SLL are named as "SLL of" and the type of the nested list. In particular, "SLL of 0/1 SLLs" stands for SLL of nested SLL of length 0 or 1. "SLL+head" stands for a list where each element points to the head of the list, "SLL of 2CDLLs" stands for SLL whose each node is a source of two CDLLs. The flag "Linux" denotes the implementation of lists used in the Linux kernel that uses a restricted pointer arithmetic which we can also handle. All experiments start with a random creation and end with a disposal of the specified structure. An indicated procedure (if any) is performed in between the creation and disposal phase. In the experiment "tree+stack", a randomly created tree is disposed using a stack in a top-down manner such that we always dispose a root of a subtree and save its subtrees into the stack. "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). We have run our tests on a machine with Intel T9600 (2.8GHz) CPU and 4GiB of RAM.

7 Conclusion

We have proposed hierarchically nested forest automata as a new means of encoding sets of heap configurations when verifying programs with dynamic linked data structures. The proposal brings the principle of separation from separation logic into automata, allowing us to combine some advantages of automata (generality, less rigid abstraction) with a better scalability stemming from local heap manipulation. We have shown some interesting properties of our representation from the point of view of inclusion checking. We have implemented the approach and tested it on multiple non-trivial cases studies, demonstrating the approach to be really promising.

In the future, we would like to first improve the implementation of our tool Forester, including support for predicate language abstraction within abstract regular tree model checking [6] as well as implementation of automatic learning of nested FA. From a more theoretical perspective, it is interesting to show whether inclusion checking is or is not decidable for the full class of nested FA. Another interesting direction is then a possibility of allowing truly recursive nesting of FA, which would allow us to handle very general structures such as trees with linked leaves.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic Abstraction for Programs with Dynamic Memory Heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
2. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing Simulations over Tree Automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 93–108. Springer, Heidelberg (2008)

3. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
4. 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)
5. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking. In: ENTCS, vol. 149(1), Elsevier, Amsterdam (2006)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
8. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-abduction. In: Proc. of POPL 2009. ACM Press, New York (2009)
9. Deshmukh, J.V., Emerson, E.A., Gupta, P.: Automatic Verification of Parameterized Data Structures. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 27–41. Springer, Heidelberg (2006)
10. Guo, B., Vachharajani, N., August, D.I.: Shape Analysis with Inductive Recursion Synthesis. In: Proc. of PLDI 2007. ACM Press, New York (2007)
11. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. Technical Report FIT-TR-2011-01, FIT BUT, Czech Republic (2011), http://www.fit.vutbr.cz/~isimacek/pub/FIT-TR-2011-01.pdf
12. Madhusudan, P., Parlato, G., Qiu, X.: Decidable Logics Combining Heap Structures and Data. In: Proc. of POPL 2011. ACM Press, New York (2011)
13. Møller, A., Schwartzbach, M.: The Pointer Assertion Logic Engine. In: Proc. of PLDI 2001. ACM Press, New York (2001)
14. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated Verification of Shape and Size Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
15. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE Computer Society Press, Los Alamitos (2002)
16. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. TOPLAS 24(3) (2002)
17. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.W.: On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London (2007)
18. 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)
19. Zee, K., Kuncak, V., Rinard, M.: Full Functional Verification of Linked Data Structures. In: Proc. of PLDI 2008. ACM Press, New York (2008)

Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints

Christine Hang, Panagiotis Manolios, and Vasilis Papavasileiou

Northeastern University
christine.d.hang@gmail.com,
{pete,vpap}@ccs.neu.edu

Abstract. We present techniques that enable designers to algorithmically synthesize cyber-physical architectural models with real-time constraints. We do this by providing a *meta*-architectural specification language that allows designers to specify *what* properties their architectural models should have, not *how* to achieve them. This provides designers with a qualitatively new level of abstraction that enables the exploration of design spaces at the earliest stages of design, when doing so provides the most benefit. Our key technical contribution is the development of an Integer linear programming Modulo Theories (IMT) solver along with a scheduling theory solver. Our solver was used to automatically synthesize cyber-physical architectural models with hard real-time constraints from a large-scale industrial design.

1 Introduction

The complexity of cyber-physical systems, such as ground, air, space, and sea vehicles, continues to increase at an exponential rate, independent of any reasonable metric used. These systems tend to be distributed and consist of numerous interconnected components that share resources, interact in complex, safety-critical ways, and have real-time constraints. Real-time constraints are particularly important because cyber-physical systems interface with the physical world and have to respond to physical events in real-time; if they do not, collision and loss of life are possible outcomes. The design of such systems is a major challenge, *e.g.*, the verification and validation of critical avionics software is estimated to cost seven times as much as its software development costs.

There is wide consensus that the design of complex systems requires raising the level of discourse by utilizing high-level modeling. The highest-level models commonly used are *architectural* models: they describe the structural properties of components and the connections between components. Many *architecture description languages* (ADLs), such as AADL [8], have been proposed to describe and reason about this structure [12]. Even such high-level modeling languages require users to specify what components are to be used and how they are to be connected. The effort required to do this can be significant, *e.g.*, the current approach used by our industrial partner required several engineering teams closely

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 441–456, 2011.

working together over a considerable amount of time to develop an architectural model for the case study we consider.

In this paper, we revisit synthesis with the goal of enabling designers to work at a higher level of abstraction and to algorithmically synthesize cyber-physical architectural models that are correct by construction. Designers should only specify *what* they want, not *how* to achieve it. The emphasis on *what* not *how* is a departure from current high-level design methods. Such a shift will enable designers to rapidly explore the design space during the earliest stages of design. This is when the benefits of design exploration are greatest, as it is well known that errors tend to become exponentially more expensive to correct the further along the life-cycle they are discovered.

To solve the synthesis problem, in Section 4, we introduce the idea of *Integer linear programming Modulo Theories (IMT)* solvers. An IMT solver resembles an SMT (Satisfiability Modulo Theories) solver, except that instead of using a SAT solver at the core, we use an ILP (Integer Linear Programming) solver. To our knowledge, this is the first time that the combination of ILP with background theories appears in the literature. We believe that the IMT approach has the potential to be widely applicable, as many practical problems from Operations Research and Engineering are routinely expressed using mathematical programming tools such as CPLEX. If one wants to consider the combination of such problems with other theories, then the IMT approach has the potential to enable the kinds of advances enabled in verification by SMT.

For cyber-physical systems, dealing with real-time constraints is of paramount importance. We consider static cyclic scheduling, an industrially-relevant and particularly demanding type of real-time scheduling problem in Section 3. We show how to solve multi-processor static-cyclic scheduling constraints using a theory solver that can be used in our IMT approach and which is parameterized by a decision procedure for the uniprocessor static-cyclic problem. Not surprisingly, care is needed in the generation of theory lemmas, as discussed in section 5. With synthesis problems, the expectation is that constraints are satisfiable. In Section 6 we show how to take advantage of this by using a general-purpose resource limit mechanism that tends to bias the solver towards parts of the state space that are easier to reason about, often leading to dramatic performance improvements.

We successfully implemented and used our approach on an industrial case study from a very complex state-of-the-art aerospace design provided to us by our industrial partners. Section 2 presents a high-level overview of the class of non-scheduling constraints appearing in our case study and how they are modeled in CoBaSA, our modeling language. CoBaSA and how it solves architectural synthesis problems that do not include scheduling constraints was introduced in previous work [11,10]. In section 8, we experimentally evaluate our work using the above mentioned case study. We were able to quickly and fully automatically synthesize cyber-physical architectural models with real time constraints for very complex aerospace designs.

2 Models and Constraints

In this section, we describe the *system assembly* constraints, the class of non-scheduling constraints found in our case study, and how they are modeled in CoBaSA [11], our modeling language. The case study is based on a real, production design provided by an industrial partner in the aerospace domain. We use terms consistent with the ARINC standards 651-1 and 664-7 [1]. A more detailed description of the constraints is also available [10].

We considered a number of models during the course of several years. The basic components of the models included: anywhere from 8 to 22 *cabinets* (cabinets provide various resources such as processors and battery backup units), anywhere from 177 to 257 *applications* (we also refer to applications as *hosted functions* or *jobs*), and anywhere from 70 to 288 *global memory spaces* (GMSs) (GMSs allow applications to share memory). Other components include *messages* (for communicating between applications, sensor, and other components) and *virtual links* (virtual links are part of a publish-subscribe network and are used to aggregate and multicast messages). The models had between 1,000 and 2,000 virtual links and between 10,000 and 20,000 messages.

Resource Utilization: Cabinets provide various resources, including CPU time, RAM memory, ROM memory, non-volatile memory, buffers, and send and receive bandwidth for virtual links. Cabinets also have limits on how many virtual links they can receive and transmit. Hosted functions, global memory spaces, virtual links, and messages consume these resources. Our meta models include constraints stating that the sum of any resource used does not exceed the amount of the resource that we have available.

Hosted Function Allocation: Hosted functions have to be mapped to cabinets subject to the resource utilization constraints above, but we also have to satisfy constraints of the following types. (a) *Fixed cabinet constraints* specify that a particular hosted function has to be mapped to a particular cabinet. (b) *Separation constraints* state that no pair of hosted functions in a given set can reside on the same cabinet. (c) *Co-location constraints* state that given a non-empty sequence of non-empty sets of hosted functions, we have to create m groups, where m is the maximum of the cardinalities of the sets in the sequence. Furthermore, each hosted function in a set has to be assigned to one of the m groups, no two hosted functions in the same set can be assigned to the same group, and all hosted functions in a group have to be assigned to the same cabinet.

Spare Cabinets and Hosted Functions: Spare cabinets allow us to operate safely in the presence of a small number of cabinet failures. To that end, *spare* cabinets are only allowed to run *spare* hosted functions. We are given a non-empty set of hosted functions and are allowed to map at most one hosted function from the set to a spare cabinet. Hosted functions that do not appear in such constraints cannot be mapped to spare cabinets. If a non-spare cabinet fails, the idea is to migrate its jobs to a spare cabinet.

Global Memory Spaces and Hosted Functions: Constraints between GMSs and hosted functions include. (a) *Fixed cabinet constraints* specify that a particular global memory space has to be mapped to a particular cabinet. (b) *Co-location constraints* specify that a particular global memory space and all hosted functions in a given non-empty set have to be mapped to the same cabinet. (c) *Read-only constraints* specify that a particular global memory space has to be allocated to all of the cabinets that a given set of hosted functions map to. Note that read-only GMSs can be arbitrarily replicated.

Virtual links: Hosted functions publish and subscribe to virtual links. Virtual links have exactly one publisher, but can have multiple subscribers. The sum of the bandwidth required for the virtual links that the hosted functions located on a cabinet publish or subscribe to cannot exceed the available outgoing or incoming bandwidth, respectively. For the incoming bandwidth constraints, if multiple hosted functions located on the same cabinet subscribe to the same virtual link, then the cost of the virtual link is only counted once.

Message buffering: A virtual link is comprised of a non-empty set of messages. Hosted functions are only really interested in messages, not virtual links. Therefore, they only read the messages they care about from the virtual links they subscribe to. Hosted functions buffer a given number of bytes for each message; this can differ among subscribers of the same message. Each cabinet provides a single buffer that is used for both message transmission and reception. The sum of the buffering requirements for the messages that the hosted functions located in the cabinet publish or subscribe to cannot exceed the capacity of the buffer. When multiple hosted functions on the same cabinet subscribe to the same message with different buffer requirements, they share space, so only incur the cost of the maximum number of bytes buffered. Finally, each hosted function that subscribes to a message uses either a queue buffer, or a sampling buffer for it and hosted functions that subscribe to the same message but use different buffer types cannot reside on the same cabinet.

Objective Functions: Our framework also allows for objective functions, which we have used for load balancing, minimizing the maximum bandwidth used per cabinet, etc.

Figure 1 shows simplified snippets of CoBaSA code for modeling a small subset of the constraints. CoBaSA includes an object-oriented modeling language, *e.g.*, the cabinet *entity* above can be thought of a *class* with three fields, where the last two correspond to resources and have default values. `C_1` is an instance of the cabinet entity, and `cabs` is an array of cabinets. Similarly, we define an array of jobs, `jobs` (though the entity definition is not shown).

CoBaSA also includes a declarative language for describing constraints. For example the line starting with `map` defines a map, `jobs-to-cabs` from `jobs` to `cabs`. The next five lines constrain `jobs-to-cabs` by requiring that cabinets have enough CPU and RAM resources to satisfy all jobs mapped to them.

The constraint starting with `for_all` is a separation constraint: it states that jobs `J_1` and `J_2` have to reside on different cabinets. The example demonstrates

```
entity cab {
  ; id STRING
  ; cpu-time-avail 1000000
  ; ram-memory-avail 4294967296
}

var
  ; cab C_1 = { ; "C_1"; ; }
  ; cab[12] cabs =
      [C_1, C_2, ..., C_12]
  ; job[200] jobs =
      [J_1, J_2, ..., J_200]
```

```
// mapping jobs to cabinets
map jobs-to-cabs jobs cabs
constraint jobs-to-cabs
  ((cpu-time-req,
    ram-memory-req))
  ((cpu-time-avail,
    ram-memory-avail))

// jobs J_1 and J_2 separated
for_all c in cabs
  { jobs-to-cabs(J_1, c) implies
    (not jobs-to-cabs(J_2, c)) }
```

Fig. 1. CoBaSA Modeling Language Examples

the high-level, declarative way in which we describe the system. We specify only *what* properties our model should have, not *how* to connect jobs and cabinets to achieve these properties. Figure 2 visualizes one of the solutions we synthesized.

We end this section by noting that many other cyber-physical systems (*e.g.*, consider the automotive industry) will have similar types of constraints, thus, we expect that our approach will be applicable to these systems as well.

3 Static Cyclic Scheduling

While our approach is independent of the types of real-time scheduling constraints used, in this paper, we consider static cyclic scheduling constraints. Static cyclic scheduling is non-preemptive and periodic; it is easy to describe, difficult to satisfy, and is used in industry.

We first describe uniprocessor static cyclic scheduling (USCS). Time is divided into infinitely many slots. A job is defined as a triple (p, c, i). The period, p, is the number of slots between successive executions of the job. The cost, c, is the number of slots each execution takes. The identifier, i, is a natural number that uniquely identifies the job. Given a set of jobs, a schedule is simply a starting time t for each job such that $t < p$ and no two jobs occupy the same slot. The slots occupied by a job are slots of the form $t+k\cdot p+m$, for $k \in \mathbb{N}$ and $0 \leq m < c$. That is, if a job is scheduled to start during slot t, then it occupies c consecutive slots starting with slot t and this process repeats at slot $t+p$, $t+2\cdot p$, $\ldots$.

Given a set of jobs, the USCS problem is to determine whether there exists a schedule.

Theorem 1. *The USCS problem is NP-complete.*

Proof. We reduce the NP-complete *set partition problem* to the USCS problem. Given a multiset $S = \{n_1, n_2, \ldots, n_k\}$ of natural numbers, we construct a set of jobs

$$J = \{(s, n_i, i) \mid 1 \leq i \leq k\} \cup \{(s/2, 1, k+1)\}$$

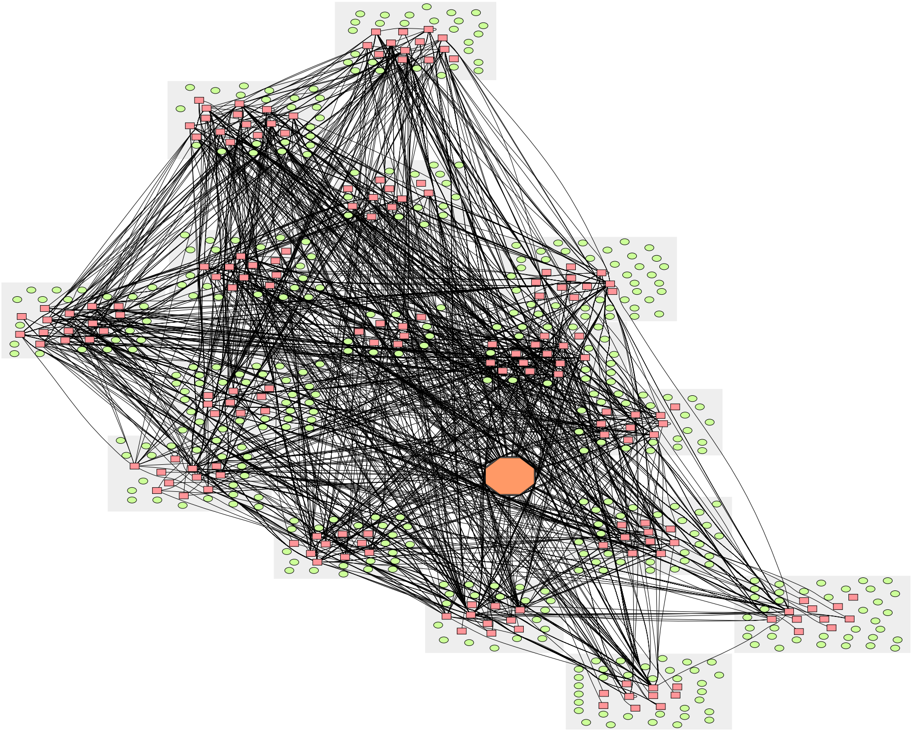


Fig. 2. One of the architectural models we synthesized, visualized as a graph. Pink rectangles correspond to hosted functions, and green ovals to memory spaces. The grey containers are cabinets. Edges between hosted functions visualize communication through virtual links. The orange octagon is the external network.

where $s = 2 + \sum S$. It is not hard to see that S can be partitioned into S_1 and S_2 such that $\sum S_1 = \sum S_2 = (\sum S)/2$ iff J is schedulable. To see that USCS is in NP, notice that we can verify a schedule by checking than no pair of jobs leads to a collision. With a little bit of analysis, this can be done by checking for all pairs of jobs (p_1, c_1, i_1) and (p_2, c_2, i_2) starting at t_1 and t_2 respectively that if $i_1 \neq i_2$ then $t_1 \neq t_2$, so without loss of generality assume $t_1 < t_2$, and we have $c_1 \leq t_2 - t_1 \leq \gcd(p_1, p_2) - c_2$.

In Multiprocessor Static Cyclic Scheduling (MSCS), we have multiple processors that run in different speeds. The number of slots per cycle (*e.g.*, per second) is processor-dependent; a processor is defined as a pair (s, i) where $s \in \mathbb{N}$ is the number of slots that the processor provides per cycle and $i \in \mathbb{N}$ is an identifier unique among processors. In MSCS, we denote a job by a triple (r, c, i), where r is the rate of the job, c is the cost, and i is the identifier. The period of a job depends on the processor: a job (r, c, i) has period s/r on a processor that provides s slots per cycle. We assume that for each job and each processor, the rate of the job r divides the number of slots per cycle s that the processor provides.

Given a set of jobs, J, and a set of processors, P, the multiprocessor static cyclic scheduling (MSCS) problem is to determine whether there exists a mapping from J to P such that for each processor, the set of jobs mapped to that processor are schedulable. As USCS is a special case of MSCS, the following corollary follows trivially:

Corollary 1. *The MSCS problem is NP-complete.*

We conclude this section by noting that given a static cyclic schedule, we can statically determine the exact slots allocated to any job. On the one hand, this makes static cyclic scheduling hard.[1] On the other hand, this makes the schedule very predictable, which in turn dramatically simplifies the analysis of the system as a whole. This advantage is why static-cyclic scheduling is used for complex cyber-physical systems.

4 ILP Modulo Theories

Our models include both system assembly constraints (as described in section 2) and static cyclic scheduling constraints (as described in section 3). We cannot simply run the ILP solver to assemble the components, and then run the scheduler, as the solution provided by the ILP solver might not be schedulable even if a solution to the system assembly and scheduling constraints exists. To solve this problem, we develop an *ILP Modulo Theories (IMT)* solver that combines a background decision procedure with an ILP solver in much the same way that SMT allows for the integration of theory solvers with SAT solvers [2,4,14].

We explored SAT as an alternative to ILP for our class of problems. We tried different encodings of the system assembly constraints into SAT and different SAT solvers. Performance was always at least three orders of magnitude worse than using an ILP solver. We conjecture that the reason for this is that our resource constraints are heavily arithmetic. Therefore, in this paper, we assume an ILP solver as our core solver, but note that our work can also be used in an SMT framework or with a pseudo-Boolean core solver.

4.1 Formal Preliminaries

Let J be a set of jobs and P be a set of processors. Let $V_{J,P} = \{V_{(j,p)} \mid j \in J, p \in P\}$ be a set of propositional variables. Note that we can represent any mapping, M, from J to P as an assignment to $V_{J,P}$. The variable $V_{(j,p)}$ is *true* iff M maps j to p. The variables in $V_{J,P}$ are called map variables.

[1] For example, for many scheduling problems, if the processor utilization is less than some constant, then there exists a polynomial-time algorithm that is guaranteed to find a schedule; alas this is not true for static cyclic scheduling.

Definition 1. *Let J be a set of jobs, P a set of processors, $V_{J,P}$ a set of map variables, and A an assignment to the variables in $V_{J,P}$. We say that the assignment A is consistent with respect to a scheduling theory iff*

$$\langle \forall p \in P :: \mathbf{sched}(\{j \in J \mid A(V_{(j,p)}) = \mathit{true}\}, p) \rangle$$

where $\mathbf{sched}(J, p)$ is a predicate that evaluates to true iff J is schedulable on p.

Let F be an ILP formula (a conjunction of linear equalities and inequalities) over a set of integer and Boolean variables V_F, such that $V_F \supseteq V_{J,P}$, where $V_{J,P}$ is a set of map variables for a set of jobs J and processors P, as described above. We say that an assignment A T-entails $G = F \wedge \mathit{Schedulable}(J, P, V_{J,P})$ (written as $A \models_T G$) iff A satisfies the linear constraints of F, and A is consistent with respect to the scheduling theory. We say G is T-satisfiable iff there is an assignment A such that $A \models_T G$. The problem of determining whether G is T-satisfiable is called the ILP Modulo (Scheduling) Theory problem.

Note that an IMT solver allows us to tackle the MSCS problem as described in section 3. In this case, the formula F has constraints over $V_{J,P}$ that force each job to be mapped to exactly one processor.

4.2 Lazy IMT Approach

Our IMT solver (algorithm 1) involves an ILP solver and a specialized decision procedure for scheduling. The ILP solver suggests assignments which are checked by the scheduler. Each failed attempt to produce a schedule from an assignment results in the learning of some theory lemmas. In analogy to [14], we call this approach the lazy IMT approach, because it learns lemmas only when necessary.

We did try to express the scheduling and system assembly constraints as a monolithic problem, thus following a more eager approach. The time required by the solver was unreasonable. We believe that the clean separation between the real-time and the architectural constraints was one of the reasons why we were successful: a specialized solver works best for the former, an ILP encoding for the latter, and IMT allows for efficient coordination between the two.

5 Theory Lemmas

To prevent the ILP solver from wasting its time on assignments that will not lead to valid schedules, we generate theory lemmas that rule out as many assignments as possible. Observe that when a set of jobs J is unschedulable on some processor p, it is possible that some strict subset J' of J is also unschedulable on p. By identifying such subsets, we can generate "good" theory lemmas that preclude the allocation of any set of jobs that are "at least as hard" as J' on processors that are "at most as powerful" as p.

5.1 Unschedulable Cores

Given an unschedulable set of jobs J, we identify the unschedulable subsets of J that do not contain any unschedulable proper subset themselves. We call such subsets unschedulable cores.

Algorithm 1. The IMT algorithm

```
procedure IMT-TOP-LEVEL($J, P, F$)
    $Lemmas \leftarrow true$
    while $true$ do
        $Ans \leftarrow$ ILPSolver($F \wedge L$)
        if $Ans = UNSAT$ then
            return $UNSAT$
        else
            $A \leftarrow$ Assignment from $Ans$
            $Schedulable \leftarrow true$
            for all $p \in P$ do
                $J_p \leftarrow \{j \in J | A(V_{(j,p)}) = true\}$
                if $\neg$sched($J_p, p$) then
                    $Lemmas \leftarrow Lemmas \wedge$ LEARN($J_p$)
                    $Schedulable \leftarrow false$
            if $Schedulable$ then
                return $Ans$
```

Definition 2. *C is an unschedulable core of a set of jobs J, if (a) $C \subseteq J$, (b) C is unschedulable, and (c) every proper subset of C is schedulable.*

Property 1. Let J be an unschedulable set of jobs. $\forall j \in J$, if $J \setminus \{j\}$ is schedulable, then j must be in every unschedulable core of J.

Given a set of jobs, J, that has been found to be unschedulable on processor p, the recursive function ALL-CORES(J, p) shown in algorithm 2 returns the set of all unschedulable cores. To acquire all cores, we consider all possible ways of removing jobs from J. Some jobs will be in any unschedulable core of J by property 1, so we only try to remove the remaining jobs (set D). If none of the jobs in J can be removed, then J is an unschedulable core. Otherwise we recurse on all sets $J \setminus \{d\}$ for $d \in D$.

The algorithm can be thought of as performing search on a tree: if a node corresponds to a set of jobs J, the children of the node are the $|J|$ subsets of J with cardinality $|J| - 1$ (one of the jobs removed). Note that instead of finding all possible cores we can terminate the search after finding some given number of cores.

We can make algorithm 2 more efficient by using a DAG instead of a tree, since the tree may have distinct nodes annotated with the same set. In addition, jobs with the same rate and cost can be aggregated and handled separately. Space constraints prohibit a full accounting.

5.2 Subsumption

Given an unschedulable core, we want to introduce a lemma that precludes the co-location of sets of jobs "harder" than the core. To this end, we need to formalize the notion of "hard" with respect to jobs and sets thereof, and the notion of "powerful" with respect to processors.

Algorithm 2. Finding all unschedulable cores

```
procedure ALL-CORES($J, p$)
    $D \leftarrow \{j \in J \mid \neg\mathbf{sched}(J \setminus \{j\}, p)\}$
    if $D = \emptyset$ then
        return $\{J\}$
    else
        $A \leftarrow \emptyset$
        for all $d \in D$ do
            $A \leftarrow A \cup$ ALL-CORES($J \setminus \{d\}, p$)
        return $A$
```

Definition 3 (Measurement of Difficulty for Jobs). $\preceq_J$ *is a partial order on jobs:* $(r_1, c_1, i_1) \preceq_J (r_2, c_2, i_2)$ *iff* r_1 *divides* r_2 *and* $c_1 \leq c_2$.

Definition 4 (Measurement of Difficulty for Sets). $\preceq_S$ *is a partial order on sets of jobs:* $S \preceq_S T$ *iff* $|S| \leq |T|$ *and there exists an injective mapping* F *from* S *to* T *such that* $\forall j \in S, j \preceq_J F(j)$. *We say that* T *is* subsumed *by* S.

Intuitively, given an unschedulable set of jobs J, if we replace each job j in J with a job j' such that $j \preceq_J j'$ (j' is "at least as hard" as j) then the resulting set of jobs $(J \setminus \{j\}) \cup \{j'\}$ will also be unschedulable. Similarly, given a set of jobs J which is schedulable, if we replace each job j in J with a job j' such that $j' \preceq_J j$, then the resulting set of jobs will be schedulable as well. Thus, the following property holds:

Property 2. For sets of jobs S, T such that $S \preceq_S T$, (a) if S is unschedulable, then T is unschedulable, and (b) if T is schedulable then S is schedulable.

Definition 5 (Measurement of Power for Processors). $\preceq_P$ *is a partial order on processors:* $(s_1, i_1) \preceq_P (s_2, i_2)$ *iff* s_1 *divides* s_2.

5.3 Lemma Generation

Given a set of jobs, J, and a set of processors, P, $\forall j \in J, \forall p \in P$, let the map variable $V_{(j,p)}$ denote that job j is allocated to processor p. Let $C = \{h_1, ..., h_m\}$ be an unschedulable core on some processor $q \in P$. We generate theory lemmas for each processor $p \in P$ where $p \preceq_P q$.

Non-Exhaustive Lemmas

For each processor p where $p \preceq_P q$, and for each job $h_i \in C$ we construct a *bucket* $B_{(i,p)}$ that contains jobs that are "at least as hard" as h_i: $B_{(i,p)} \subseteq \{j \in J \mid h_i \preceq_J j\}$. The buckets do not overlap ($B_{(i,p)} \cap B_{(j,p)} = \emptyset$ for $i \neq j$). Each job j that can be mapped to processor p[2] and is harder than one of the jobs h_i

[2] If we can deduce that j cannot be mapped to p, we record this fact and use it to determine which jobs can be mapped to which processors.

$(h_i \preceq_J j)$ has to be included in one of the buckets. If we replace any $h_i \in C$ with a job in the corresponding bucket, we get an unschedulable set of jobs. We generate the lemma $\neg \bigwedge_{1 \leq i \leq m} \left(\bigvee_{j \in B_{(i,p)}} V_{(j,p)} \right)$, which states that if we allocate at least one job from each bucket to processor p, p will be unschedulable.

We attempt to construct buckets in a way that gives rise to useful lemmas. Whenever a job j can go to multiple buckets, we choose (1) the bucket corresponding to the job itself, if the job is in the core; otherwise (2) randomly among the buckets corresponding to jobs that have the same rate and cost as j; if there are no such jobs, (3) among the buckets corresponding to jobs that have the same rate as j; if there are no such buckets, (4) among the remaining buckets. The rationale behind these choices is to ensure that we rule out the allocation of the exact unschedulable set of jobs (and sets very similar to it) to any processor. This guarantees that we make progress towards a solution and ensures termination.

Notice that the lemmas we generate are *non-exhaustive*. It is possible that an assignment A maps a set of jobs S such that $C \preceq_S S$ to a processor p such that $p \preceq_P q$, and A is consistent with our lemmas. This is because some of the jobs of S could have gone to multiple buckets, but our choices when building the lemma resulted in a bucket not being "inhabited" by any $j \in S$. We use non-exhaustive lemmas because their encoding is small.

Exhaustive Lemmas

We can also construct buckets that include the map variables for *all* jobs that are "at least as hard" as a job $h_i \in C$ and can be mapped to processor p: $B'_{(i,p)} = \{j \in J \mid h_i \preceq_J j\}$. Now a job can be in more than one bucket. For each set of jobs $\{j_1 \in B'_{(1,p)}, j_2 \in B'_{(2,p)}, \ldots, j_m \in B'_{(m,p)}\}$ such that $\forall i, k, 1 \leq i < k \leq m : j_i \neq j_k$ we can introduce the clause $\bigvee_{1 \leq i \leq m} \neg V_{(j_i,p)}$. If all j_i were mapped to a processor p such that $p \preceq_P q$, we would have allocated a set harder than the core C, which is therefore unschedulable.

With this encoding, we need $\prod_{1 \leq i \leq m} |B'_{(i,p)}|$ clauses in the worst case. The lemma allows us to rule out *all* sets of jobs S such that $C \preceq_S S$. We use exhaustive lemmas for cores below a certain size, because smaller cores are more frequently applicable and the product above remains manageable.

The recursive function EXHAUSTIVE-LEMMA (algorithm 3) generates all clauses for a list of buckets B' and a processor p. The argument c is a partial clause (initially empty) that corresponds to finding a job j_k for each bucket $B'_{(k,p)}$, where $1 \leq k \leq |c|$; c it is of the form $\bigvee_{1 \leq k \leq |c|} \neg V_{(j_k,p)}$. We use suffixes that the clauses share to keep the encoding more compact. Assume that we have constructed a partial clause c of length l, and that the jobs $j, j' \in B'_{(l+1,p)}$ do not appear in any subsequent bucket. The clauses with prefixes $c \vee \neg V_{(j,p)}$ and $c \vee \neg V_{(j',p)}$ share the suffixes corresponding to the buckets $B'_{(i,p)}$, where $i > l+1$. We use an auxiliary variable for the disjunction $V_{(j,p)} \vee V_{(j',p)}$, instead of generating a separate set of clauses for each of j, j'.

Algorithm 3. Exhaustive lemma generation

```
procedure EXHAUSTIVE-LEMMA($p$, $c$, $B'$)
    if $B' = \emptyset$ then
        output clause $c$
    else
        $L \leftarrow \{j \in \text{first}(B') \mid \forall b \in \text{rest}(B'), j \notin b\}$
        if $L \neq \emptyset$ then
            $v \leftarrow \bigvee_{j \in L} V_{(j,p)}$
            EXHAUSTIVE-LEMMA($p$, $c \vee \neg v$, $\text{rest}(B')$)
        for all $j \in (\text{first}(B') \setminus L)$ do
            $B'_{\text{new}} \leftarrow \{b \setminus \{j\} \mid b \in \text{rest}(B')\}$
            EXHAUSTIVE-LEMMA($p$, $c \vee \neg V_{(j,p)}$, $B'_{\text{new}}$)
```

5.4 Memoization

The purpose of memoization is to avoid making expensive calls to the scheduler when the (un)schedulability of a set of jobs can be inferred from the result of a previous call. For sets S and T such that $S \preceq_S T$, if we have memoized that S is unschedulable, T is also unschedulable by property 2 and we don't have to call the scheduler. Similarly, if we know that T is schedulable we can immediately infer that S is schedulable. The result of each call to the scheduler is memoized in a list as a pair containing the set of jobs on which it was called and the corresponding (un)schedulability. Note that we can decide whether $S \preceq_S T$ in polynomial time by reducing the problem to matching in a bipartite graph.

When the (un)schedulability of a set of jobs is questioned, we access the memoization list sequentially from its head until we find an element from which we can infer (un)schedulability. To speed this operation up we eliminate redundant elements: if the (un)schedulability of some set S in the memoization list becomes inferable from the result of a new call to the scheduler for some set T, then we memoize the (un)schedulability of T, and also remove S from the memoization list. In addition, we keep sets ordered by their success rate (number of successful inferences), so that the most frequently used sets are towards the beginning of the list. If the list becomes too long, we can forget the least useful sets.

The memoization list is complementary to lemmas. For example, the memoization list allows us to quickly infer that a set of jobs is *schedulable*, whereas lemmas only rule out certain assignments. Even for unschedulable assignments the memoization list can complement lemmas. For example, (1) we can infer unschedulability before a lemma is introduced, say during core generation (algorithm 2) and (2) we catch instances subsumed by a core but not caught by the corresponding non-exhaustive lemma.

6 Resource Limits

It is possible that a query to the theory solver takes a long time to complete or uses too much of some other resource (like memory). Since we expect that our synthesis problems have solutions, it make sense to avoid such queries and to instead bias our solver towards solutions that are easier to justify. To that end, we introduce *resource-limit* lemmas that allow us to quickly rule out difficult-to-solve (but potentially satisfiable) scheduling instances in the hope that the ILP solver will find instances that require fewer resources to justify. In order to maintain completeness, we have to sometimes undo resource-limit lemmas to prevent the ILP solver from becoming over-constrained. When resource-limit lemmas are undone, we allow the exploration of previously blocked parts of the search space with increased resource limits. Due to space limitations, we informally describe how resource-limit lemmas work.

We set resource limits for both the scheduler and the ILP solver. In our class of problems, it makes the most sense to restrict time, since that is the bottleneck. If the scheduler times out on some instance, we consider that instance to be unschedulable and generate lemmas to prevent the ILP solver from generating similar scheduling problems. These lemmas are called resource-limit lemmas. They are used in the same way as regular lemmas during ILP solving and will be kept as long as the resulting ILP problem is satisfiable. However, when the ILP problem becomes unsatisfiable, resource-limit lemmas will be removed and the resource limit for the scheduler will be increased. The rationale behind this is that the previous resource-limit lemmas might have over-constrained the search space of the ILP solver and might have led to unsatisfiability. Therefore, removing the lemmas and increasing the resource limit will give the ILP solver a chance to explore a potentially bigger search space. When the ILP solver times out, not only will we remove resource-limit lemmas and increase the resource limit for the scheduler, but we will also increase the resource limit for the ILP solver.

The idea of resource-limit lemmas can also be used in the context of SMT as a technique to manage the balance of resources used by the SAT solver and theory solvers. The technique is likely to be useful in applications like synthesis, where we expect a satisfying assignment to exist. In this case, steering the core solver towards an area of the search space where we can find solutions and justify them easily makes sense. In contexts where we expect the problem to be unsatisfiable, the technique may be counterproductive since resource limits will eventually have to be made large enough to fully explore the search space. On the other hand, purging resource-limit lemmas is somewhat similar to a restart and may exhibit similar benefits.

Note that we can use resource-limit lemmas selectively. For example, if the last round of scheduling attempts led to the discovery of regular lemmas, we can decide to not use any resource-limit lemmas. In this way we make progress (new regular lemmas), but do not needlessly constrain the ILP (or SAT) problem with resource-limit lemmas. Other options for selectively using resource-limit lemmas include filtering the lemmas or using some small number of them per round.

7 Related Work

Architecture Description Languages (ADLs), such as AADL, can be used to model and reason about safety-critical systems [12,8]. There has been work on

ADLs that takes scheduling into account. However, these approaches check that a particular architectural model satisfies scheduling and other constraints [6]. In contrast, we *synthesize* the architectural models, and they are correct by construction.

Different kinds of real-time constraints have been studied. Liu and Layland [9] laid out the basis for the real-time scheduling theory by studying the Rate-Monotonic (RM) and Earliest Deadline First (EDF) algorithms. Sha et al. [17] provide a historical overview of the topic. In contrast to these kinds of scheduling, static cyclic is non-preemptive. In addition, Liu and Layland proved that if utilization is below a specific bound then a rate-monotonic schedule exists. Thus, there is a simple schedulability test for RM. This is not the case for static cyclic scheduling.

There is recent work on allocating jobs to processors with using multi-dimensional bin-packing algorithms, in the presence of scheduling, and other constraints [5,7]. The scheduling policy in these cases is rate-monotonic. Under restrictions on CPU utilization, rate-monotonic schedulability is ensured. Therefore, such scheduling problems can be turned into bin-packing problems. Unfortunately, this approach is not complete, *e.g.*, there are schedulable problems for which this approach will not find solutions. In addition, the approach does not work for static-cyclic scheduling. Finally the approach is too restrictive to express the constraints we need and has not been shown to scale to the complexity of designs our work handles.

There are many task allocation algorithms for distributed real-time systems that have been studied. This includes, but is not limited to, branch and bound algorithms [15], SAT solving [13], and linear programming [3]. However, the scheduling constraints studied in these cases are not as demanding as static cyclic, and the class of constraints that can be handled is limited.

8 Experimental Evaluation

We summarize our experiences in synthesizing architectural models from the constraints provided by our industrial partner. We only focus on the most complex problem we considered.

Our framework can be parameterized in many ways. We used a 0.1-second CPU resource-limit for the scheduler and, in the presence of resource-limit lemmas, a 1 minute CPU resource-limit for the core solver. If either the scheduler or the core solver time out, then we increase the CPU limit by 40%. We specified a minimum load of 80% per processor, and we limited the search for cores to one per unschedulable processor. If the scheduler timed out while trying to determine the schedulability of a particular processor, we only generated resource-limit lemmas when we failed to extract regular cores from any processor.

We ran our framework with both CPLEX and bsolo [16] as the core ILP solvers. The experiments were run on an eight-core, 3.2 GHz Intel Xeon EM64T server with 96GB of memory (we never needed more than 1GB of memory for any experiment). Our decision procedure for scheduling and the IMT solver were implemented in OCaml. With the set of parameters described above, CPLEX provided a solution after 103 seconds and required 13 iterations. bsolo required 1834 seconds and 23 iterations.

To evaluate the importance of resource-limits, we also ran the above experiment with resource-limits disabled. Both CPLEX and bsolo fail to provide an answer after two hours. CPLEX goes through 5 iterations and bsolo goes through 4 iterations. Analysis shows that the reason for this failure is that we spend lots of time trying to determine the schedulability of processors. The resource-limit mechanism works because it steers CoBaSA towards parts of the search space with easier scheduling problems. We get the same failures (for both CPLEX and bsolo) if we only disable scheduler resource-limits. On the other hand, if we only disable solver resource-limits, then this has no effect on CPLEX (since it does not timeout), but bsolo goes through 22 iterations and times out.

CoBaSA interacts with a collection of tools that our group has developed and with off-the-shelf solvers. In order to increase our confidence in the validity of the solutions CoBaSA generates, we implemented an independent checker that validates solutions. The independent checker helped us identify several bugs in our handling of constraints. Also, our industrial partner checks our solutions in several independent ways. This has also been useful because there were cases where we received incorrect specifications and constraints, or there was miscommunication between different groups.

The experiments show that our approach is capable of synthesizing industrial-scale cyber-physical architectural models with real-time constraints. According to our industrial partner, the current process by which architectural models are created requires significant iteration between multiple engineering teams. Our experiment evaluation clearly shows that our IMT approach leads to significant performance and cost improvements.

9 Conclusions and Future Work

We showed how to algorithmically synthesize cyber-physical architectural models by using a *meta*-architectural specification language that allows designers to specify *what* constraints their architectural models must satisfy, not *how* to achieve them. We did this by developing an ILP Modulo Theories (IMT) solver with a resource-limit capability and a theory solver for static cyclic scheduling. We successfully implemented and used our approach on an industrial case study from a very complex state-of-the-art aerospace design. We believe that the IMT approach has the potential to be widely applicable, as many practical problems are routinely handled using ILP solvers, and the IMT approach allows one to combine the power of ILP with specialized solvers for background theories. For future work, we plan to further develop and explore the IMT approach.

Acknowledgments

We are indebted to John Chilenski of Boeing Commercial Airplanes for providing extensive guidance, support, feedback, and collaboration. This research was funded in part by NASA Cooperative Agreement NNX08AE37A.

References

1. ARINC. ARINC Specifications and Reports, `https://www.arinc.com/`
2. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 236. Springer, Heidelberg (2002)
3. Cambazard, H., Hladik, P.-E., Déplanche, A.-M., Jussien, N., Trinquet, Y.: Decomposition and learning for a hard real time task allocation problem. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 153–167. Springer, Heidelberg (2004)
4. de Moura, L., Ruess, H.: Lemmas on Demand for Satisfiability Solvers. In: SAT (2002)
5. de Niz, D., Feiler, P.H.: On Resource Allocation in Architectural Models. In: ISORC (2008)
6. Delange, J., Pautet, L., Plantec, A., Kerboeuf, M., Singhoff, F., Kordon, F.: Validate, simulate, and implement ARINC653 systems using the AADL. In: SIGAda (2009)
7. Dougherty, B., White, J., Balasubramanian, J., Thompson, C., Schmidt, D.C.: Deployment Automation with BLITZ. In: ICSE (2009)
8. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction (2006)
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. JACM 20(1), 46–61 (1973)
10. Manolios, P., Papavasileiou, V.: Virtual Integration of Cyber-Physical Systems by Verification. In: AVICPS (2010)
11. Manolios, P., Vroon, D., Subramanian, G.: Automating component-based system assembly. In: ISSTA (2007)
12. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
13. Metzner, A., Herde, C.: RTSAT – An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures. In: RTSS (2006)
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). JACM 53(6), 937–977 (2006)
15. Peng, D.-T., Shin, K., Abdelzaher, T.: Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems. IEEE Transactions on Software Engineering 23, 745–758 (1997)
16. Santos, J., Manquinho, V.M.: Learning Techniques for Pseudo-Boolean Solving. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, Springer, Heidelberg (2008)
17. Sha, L., Abdelzaher, T., Arzen, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real Time Scheduling Theory: A Historical Perspective. In: RTSS (2004)

μZ– An Efficient Engine for Fixed Points with Constraints

Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura

Manchester University and Microsoft Research

Abstract. The μZ tool is a scalable, efficient engine for fixed points with constraints. It supports high-level declarative fixed point constraints over a combination of built-in and plugin domains. The built-in domains include formulas presented to the SMT solver Z3 and domains known from abstract interpretation. We present the interface to μZ, a number of the domains, and a set of examples illustrating the use of μZ.

1 Introduction

Classical first-order predicate and propositional logic are a useful foundation for many program analysis and verification tools. Efficient SAT and SMT solvers and first-order theorem provers have enabled a broad range of applications based on this premise. However, fixed points are ubiquitous in software analysis. Model-checkers compute a set of reachable states as a least fixed point, or dually a set of states satisfying an inductive invariant as a greatest fixed point. Abstract interpreters compute fixed points over an infinite lattice using approximations. An additional layer is required when using first-order engines in these contexts.

The μZ tool is a scalable, efficient engine for fixed points with constraints. At the core is a bottom-up Datalog engine. Such engines have found several applications for static program analysis. A distinguishing feature of μZ is a pluggable and composable API for adding alternative finite table implementations and abstract relations by supplying implementations of relational algebra operations join, projection, union, selection and renaming. Lattice join and widening can be supplied to use μZ in an abstract interpretation context. The μZ tool is part of Z3 [3] and is available from Microsoft Research since version 2.18[1].

2 Architecture

A sample program is in Fig. 1 and the main components of μZ are depicted on Fig. 2. As input μZ receives a set of relations, rules (Horn clauses) and ground facts (unit clauses). The last rule uses the

$$\begin{array}{lll}
\ell_0 & : & [\mathit{Int}] \text{ using pentagon} \\
\ell_1 & : & [\mathit{Int}] \text{ using pentagon} \\
\ell_0(0). & & \\
\ell_0(x) & \leftarrow & \ell_0(x_0), x = x_0 + 1, x_0 < n. \\
\ell_1(x) & \leftarrow & \ell_0(x), n \leq x.
\end{array}$$

Fig. 1. Sample μZ input

[1] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 457–462, 2011.

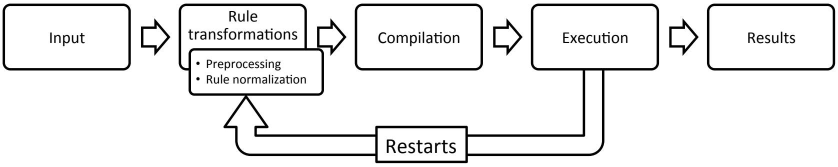


Fig. 2. μZ architecture

head predicate ℓ_1 and constraint $n \leq x$. The parameter n is symbolic, even during evaluation. A relation is specified by its domain and by its representation. The example from Fig. 1 uses the *pentagon* abstract domain. Representations may support approximation through widening that ensures convergence over infinite domains.

2.1 Rule Transformations

To allow for optimizations and/or additional features, we may perform various transformations on the input rules.

Free Variable Elimination. This transformation seeks to avoid large intermediary tables. It replaces rules of the form $p(\boldsymbol{x}, y) \leftarrow B[\boldsymbol{x}]$ by $p'(\boldsymbol{x}) \leftarrow B[\boldsymbol{x}]$. ($B[\boldsymbol{x}]$ stands for the body of the rule, and $\boldsymbol{x}$ denotes all the variables appearing in it.) Furthermore, each occurrence of p in a body of some rule requires a version with p'. E.g., for a rule $q(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y), r(\boldsymbol{x}, \boldsymbol{z}), p(\boldsymbol{z}, y)$ we would need to introduce three rules $q(\boldsymbol{x}, y) \leftarrow p'(\boldsymbol{x}), r(\boldsymbol{x}, \boldsymbol{z}), p(\boldsymbol{z}, y)$, $q(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y), r(\boldsymbol{x}, \boldsymbol{z}), p'(\boldsymbol{z})$, $q(\boldsymbol{x}, y) \leftarrow p'(\boldsymbol{x}), r(\boldsymbol{x}, \boldsymbol{z}), p'(\boldsymbol{z})$. The last rule introduces another free head variable, which can be eliminated using the same procedure. The transformation may increase the source program by an exponential factor, so μZ uses a limit on the number of such transformations.

Magic Sets. The classical magic sets transformation [1] is an option in μZ that specializes a set of rules with respect to a query.

Coalescing Rules. This transformation moves constants into a new relation and a group of rules with a single rule:

$$p(\boldsymbol{x}, \boldsymbol{c_1}) \leftarrow B[\boldsymbol{x}, \boldsymbol{c_1}] \;..\; p(\boldsymbol{x}, \boldsymbol{c_n}) \leftarrow B[\boldsymbol{x}, \boldsymbol{c_n}] \qquad \mapsto \qquad p(\boldsymbol{x}, \boldsymbol{y}) \leftarrow B[\boldsymbol{x}, \boldsymbol{y}], r(\boldsymbol{y})$$

where r is a fresh relation containing tuples $\boldsymbol{c_1}, \ldots, \boldsymbol{c_n}$. The transformation trades n updates to p using unions by one update to p and a join between B and the new relation r. In the context of μZ the transformation is particularly useful after a Magic set transformation. We have not found it useful directly for the user input.

Join Planning. The join planner splits long rules so that each rule contains at most two positive relation predicates in its body. Number of negative relation predicates and non-relation predicates in rules is not limited, because these do not lead to introduction of intermediate relations. The planner uses information on the expected size of relations in order to make the intermediate relations as small as possible. To this end the solver periodically restarts and reruns the planner to make use of better size estimates. The join planner also attempts to identify shared parts of rules in order to avoid their repeated evaluation.

2.2 Compilation to an Abstract Machine

The compiler transforms the bodies of rules into relational algebra operations. These operations are atomic instructions in an abstract machine, which also contains control and data-flow instructions to handle applications of the rules until a fixed point is reached.

We use a binary relation $r(x, y)$ to illustrate the compilation of rule bodies into relation algebra. Renaming is used to reorder the arguments to correspond to $r(x, y)$. Selection restricts r by fixing a column to a constant, equating columns, or constraining r with respect to an arbitrary predicate φ. Multiple relations are combined using joins, and projection removes variables that are not used in the head.

$r(y, x)$	$\rho_{\#1\mapsto\#2,\#2\mapsto\#1}(r)$
$r(a, y)$	$\sigma_{\#1=a}(r)$
$r(x, x)$	$\sigma_{\#1=\#2}(r)$
$r(x, y), \varphi[x, y]$	$\sigma_{\varphi[\#1,\#2]}(r)$
$r(x, y), q(z, x)$	$r \bowtie_{\#1=\#2} q$
$T[x, \boldsymbol{y}]$	$\pi_{\#1}(T[x, \boldsymbol{y}])$

The abstract machine furthermore contains instructions for conditional jumps, swap, copy, load, complementation (for stratified Datalog programs), and creating empty relations.

The effect of applying a rule is to update the relation in the head by taking a union with the relation computed in the body. The corresponding union operation comes in two flavors: if the head relation is used in a non-recursive context, the corresponding operation destructively updates the head relation. If the head relation is used in a recursive context, the union operation furthermore computes a *differenec* relation Δ that is used to detect termination and to minimize the number of records that need to be examined in subsequent joins. This contrasts using a containment relation to check for termination.

```
                                                      content
$R_0 := \{0\}$                                          $\ell_0$
$R_2 := R_0$                                            $\Delta_{\ell_0}$
while($R_2 \neq \bot$) {
  $R_3 := \top$
  $R_4 := R_3 \bowtie R_2$                              $x, x_0$
  $R_4 := \sigma_{\#1=\#2+1}(R_4)$
  $R_4 := \sigma_{\#2<n}(R_4)$
  $R_5 := \pi_{\#2}(R_4)$                               $x$
  $R_2, R_0 := R_5 \setminus R_0, R_0 \cup R_5$
}
$R_1 := \sigma_{\neg(\#1<n)}(R_0)$                      $\ell_1$
$\ell_0 := R_0$
$\ell_1 := R_1$
```

Fig. 3. Compiled version of Fig. 1

The requirement on Δ is the following, where p is the head relation and q is the relation from the body: $q \subseteq p \to \Delta = \bot$, and $q \not\subseteq p \to q \setminus p \subseteq \Delta \subseteq p \cup q$. Loops are terminated when Δ is empty. The natural value for Δ would be $q \setminus p$, but for some representations this might be expensive to evaluate.

We use the relation dependency graph to obtain loops of smaller size and evaluation order which leads to faster propagation of newly derived facts. We identify strongly connected components of recursive head predicates and saturate each of these components separately. We attempt to find an acyclic induced subgraph of each component, and use this subgraph to obtain the evaluation order inside the loop. The benefit of such order is that we do not have to carry the "differences" of relations which are in the acyclic subgraph across loop iterations.

The engine supports two compilation modes: In standard Datalog compilation mode, rules are compiled into instructions that perform a bottom-up saturation. In the presence of stratified negation, it performs the saturation per stratum. The compiler generates two phases in abstract interpretation mode: union on recursive predicates is replaced by widening, and the recursive predicates are reset (to the empty relations) between phases. Other compilation modes are possible in future versions of μZ. In particular a mode for bounded-model checking where fixed points are unrolled to a fixed depth, is of interest in applications.

2.3 Execution

Execution of the compiled code is performed by a register machine interpreter. Registers store relation objects that implement relational algebra methods. Compile time transformations suffice to avoid using counters or other data types in registers.

2.4 Tables and Relations

The abstract machine works at the level of relational algebra while the representation of relations is delegated to implementations.

Finite Collections. A basic representation of relations is as a finite collection of records. Finite sets admit *iterators* that can enumerate elements from the collection. Our default representation of finite collections is by hash tables with *on-demand indexing.* Thus, one hash table may be indexed by multiple columns at a time depending on which columns are used in different joins. An index is created or updated when the correponding column is used in a join or a selection by a constant. We found that hash tables offered a more efficient representation for our benchmarks when compared with BDDs, though it is possible to create examples where BDDs are significantly more compact [5]. We have plugged in BDDs over the external relation API using the BuDDy package[2].

Abstract Relations. The real utility of the relational algebra core is achieved by also admitting relation representations that are truly abstract.

A precise, but abstract representation is achieved by mapping relational algebra operations back to first-order formulas. We call this the *SMT relation* as it uses the SMT solver Z3 for quantifier-elimination during projection and checking for convergence. Translation from relational algebra into first-order logic is a simple transliteration. For example $\lceil \bot \rceil = \mathit{false}$, $\lceil \pi_x R \rceil = \exists x . \lceil R \rceil$

[2] http://buddy.wiki.sourceforge.net

and $\lceil R \bowtie_{\mathcal{E}} S \rceil = \lceil R \rceil \wedge \lceil S \rceil \wedge \mathcal{E}$. The set of satisfying assignments to the free variables in the formula correspond to records that are members of the relation. Computing Δ requires a satisfiability check of $\lceil R \rceil \wedge \neg \lceil S \rceil$, which introduces a formula with quantifier alternation. μZ relies on Z3's support for quantifier elimination for bit-vectors, Presburger arithmetic and algebraic data types to compute Δ.

μZ also contains two built-in abstract relations for conjunctions of integer intervals and bounds (relations of the form $x < y$). These domains are well-known from abstract interpretation [2]. They are closed under join, projection and selection, but they are not closed under union. Union is instead approximated by a convex hull operation. The domains also support widening operations.

Compositionality. Explanations can be tracked by adding a column to each relation and track rules by accumulating a term for the rules that are applied:

$$rl : p(\boldsymbol{x}) \leftarrow q(\boldsymbol{x}, y), r(\boldsymbol{x}, y). \quad \mapsto \quad rl' : p'(\boldsymbol{x}, rl(u, v, y)) \leftarrow q'(\boldsymbol{x}, y, u), r'(\boldsymbol{x}, y, v).$$

There can be an unbounded number of explanations for a derived fact, but it suffices to consider just one representative. We can encode this in a special *relational algebra of explanations*, where unions of two sets of explanations selects a suitable (in the case of μZ, oldest) representative. The remaining columns of p' do not belong to the algebra of explanations, but may be stored in a finite table or an abstract relation. To support such joint representations, μZ allows composing arbitrary tables and relations. The composition of a finite table with another finite table or relation is obtained by adding an additional column to the finite table to point to a table (relation) that contains values corresponding to a row. The usual relational algebra operations are extended directly for this representation. For example, the joint relation $r(x, y, z) : \{(1, 0, a), (1, 0, b), (1, 1, c)\}$ is represented as the map $[(1, 0) \mapsto \{a, b\}, (1, 1) \mapsto \{c\}]$, and projecting the second column produces $\pi_y r(x, y, z) : [1 \mapsto \{a, b, c\}]$. The product and intersection of two abstract relations is also available, but in this case projection and union are no longer precise because the normalized representation is as a vector of relations. Some precision is retained by supporting *reduced products* that lets domains communicate constraints. For example, we obtain the *Pentagon* domain by taking the product of the Interval i and Bound b domain subject to having restriction $(b \cup \{x < y\}) \wedge (z = x - y)$ contribute the interval constraint $(z \in [-\infty, -1])$; and extending unions on bounds to also accept intervals: $b \cup i := \{x < y \in b \mid sup_i(x) < inf_i(y)\}$.

3 Usage

A diagram showing the integration of options for μZ is in Fig. 4.

Interface. User can interact with μZ either by the means of the Z3 API (managed or C) or pass the problem specification in a file from the command line.

Input files can be in one of the following formats: SMT2 format extended by commands *rule* and *query* to add rules and start the fix-point search, in the *Bddbddb* [5] format, or in the *tuple* format which allows fast reading of large amounts of facts.

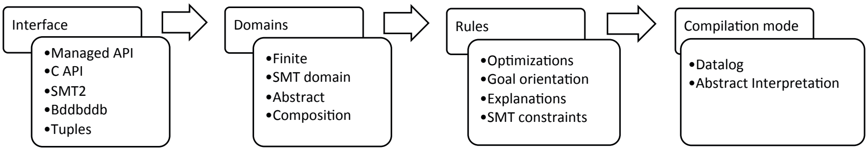


Fig. 4. Possible configurations of μZ

Applications. Although μZ is a new tool, it has been already used in several contexts[3]. We have run μZ on moderate size (2- 25K lines of pure Datalog code) benchmarks extracted from the Javascript security analyzer Gatekeeper [4]. It suffices to configure μZ using hash tables for storing relations. It spends in the order of 100ms on these benchmarks due to transformations such as the *free variable elimination* that eliminated many rules in favor of ground facts. The Bddbddb tool [5], in contrast relies on the existence of good variable orderings to avoid running out of physical memory. Using finite domains, we have also loaded a representation of the Windows base kernel, and ran various queries on it. The resulting data-base contained in the order of 10^6 facts. The efficient tuples front-end loads the 2 GB data-base within 20 seconds, but stand-alone saturation is infeasible. Queries can still be answered within a second on a standard dekstop PC after the Magic sets transformation. A demonstration of μZ for solving Traffic Jam puzzles is available. It illustrates the use of explanations.

4 Conclusion

μZ is a new efficient engine for fixed points with logical constraints. It integrates and is available with Z3. This tool paper explained the main architecture of μZ and provided background on pluggable and composable relations. We hope this tool will enable several future applications that rely on efficient fixed point core with special needs on domain representations.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: PODS, pp. 1–15. ACM, New York (1986)
2. 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)
3. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Guarnieri, S., Livshits, V.B.: Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In: USENIX, pp. 151–168 (2009)
5. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144 (2004)

[3] `http://research.microsoft.com/projects/z3/fixedpoints-index.html`

BAP: A Binary Analysis Platform

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz

Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA, USA

Abstract. BAP is a publicly available infrastructure for performing program verification and analysis tasks on binary (i.e., executable) code. In this paper, we describe BAP as well as lessons learned from previous incarnations of binary analysis platforms. BAP explicitly represents all side effects of instructions in an intermediate language (IL), making syntax-directed analysis possible. We have used BAP to routinely generate and solve verification conditions that are hundreds of megabytes in size and encompass 100,000's of assembly instructions.

1 Introduction

Program analysis of binary (i.e., executable) code has become an important and recurring goal in software analysis research and practice. Binary code analysis is attractive because it offers high fidelity reasoning of the code that will actually execute, and because not requiring source code makes such techniques more widely applicable.

BAP, the Binary Analysis Platform, is the third incarnation of our infrastructure for performing analysis on binary code. Like other platforms such as CodeSurfer/x86 [3], McVeto [15], Phoenix [11], and Jakstab [9], BAP first disassembles binary code into assembly instructions, lifts the instructions to an intermediate language (IL), and then performs analysis at the IL level. BAP provides the following salient features:

- BAP makes all side effects of assembly instructions explicit in the IL. This enables all subsequent analyses to be written in a syntax-directed fashion. For example, the core code of our symbolic executor for assembly is only 250 lines long due to the simplicity of the IL. The operational semantics of the IL are formally defined and available in the BAP manual [4].
- Common code representations such as CFGs, static single assignment/three-address code form, program dependence graphs, a dataflow framework with constant folding, dead code elimination, value set analysis [3], and strongly connected component (SCC) based value numbering.
- Verification capabilities via Dijkstra and Flanagan-Saxe style weakest preconditions and interfaces with several SMT solvers. The verification can be performed on dynamically executed traces (e.g., via an interface with Intel's Pin Framework), as well as on static code sequences.
- BAP is publicly available with source code at `http://bap.ece.cmu.edu/`. BAP currently supports x86 and ARM.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 463–469, 2011.

We have leveraged BAP and its predecessors in dozens of security research applications ranging from automatically generating exploits for buffer overflows to inferring types on assembly. A recurring task in our research is to generate logical verification conditions (VCs) from code, usually so that satisfying answers are inputs that drive execution down particular code paths. Generating VCs that are actually solvable in practice is important; we routinely solve VCs hundreds of megabytes in size that capture the semantics of 100,000s of assembly instructions using BAP.

In the rest of this paper we discuss these features, how they evolved, compare them to other platforms where possible, and provide examples of how we have used them in various projects.

2 BAP Goals and Related Work

Fully representing the semantics of assembly is more challenging than it would seem. In order to appreciate the difficulty, consider the three line assembly program below. Suppose we want to create a verification condition (VC) that is satisfied only by inputs that take the conditional jump (e.g., to find inputs that take the jump). The challenge is that arithmetic operations set up to 6 status flags, and control flow in assembly depends upon the values of those flags. Simply lifting line 1 to something like `ebx = eax + ebx` does not expose those side effects.

```
1  add %eax, %ebx   # ebx=eax+ebx (sets OF, SF, ZF, AF, CF, PF)
2  shl %cl, %ebx    # ebx=ebx<<cl (sets OF, SF, ZF, AF, CF, PF)
3  jc target        # jump to target if carry flag is set
```

The first generation of our binary analysis tools, asm2c, attempted to directly decompile x86 assembly to C, and then perform all software analysis on the resulting C code. asm2c left instruction side effects implicit, which made it difficult to analyze control flow. Other binary tools such as instrumentors, disassemblers, and editors (e.g., DynInst [13], Valgrind [12], and Microsoft Phoenix [11]) also did not represent these side effects explicitly.

Our next incarnation, Vine, was designed to address the problem by explicitly encoding side-effects in the IL. The result is that subsequent analyses and verification could rely upon the IL syntax alone. Vine is significantly more successful than asm2c, and has been used in dozens of research projects (see [5]).[1] Vine used VEX [12] to provide a rough IL for each instruction, which was then augmented by Vine to expose all otherwise-implicit side effects. An important implementation decision was to implement the Vine back-end in OCaml (asm2c was in C++). We found OCaml's language features to be a much better match for program analysis and verification. However, the Vine IL grew over time, lacked a formal semantics for the IL itself, and did not handle bi-endian architectures such as ARM correctly.

BAP is a complete re-design of Vine that encompasses lessons learned from our previous work on binary analysis. The main goals of BAP are: 1) explicitly

[1] Vine is still actively developed at Berkeley under the BitBlaze project [5].

represent all assembly side-effects to allow for syntax-directed analysis; 2) use a simple IL with formally defined semantics; 3) include useful analyses and verification techniques appropriate for binary code (either by design or by adaptation); and 4) allow user-defined analyses. The semantics of the BAP IL is formally defined, which weeded out several bugs from Vine and allowed us to better argue about the correctness of implemented analyses and algorithms. The IL also adds primitives to handle instruction issues discovered in Vine such as bi-endian memory operations, and is simpler overall. In addition to modeling the semantics of instructions explicitly, BAP also exposes the low-level semantics of memory where loads and stores are byte-addressable and thus can result in “overlapping operations”.

An example of the IL produced for Example 1 is (after deadcode elimination):

```
addr 0x0 @asm ”add %eax,%ebx”
t:u32 = R_EBX:u32
R_EBX:u32 = R_EBX:u32 + R_EAX:u32
R_CF:bool = R_EBX:u32 < t:u32
addr 0x2 @asm ”shl %cl,%ebx”
t1:u32 = R_EBX:u32 >> 0x20:u32 - (R_ECX:u32 & 0x1f:u32)
R_CF:bool =
    ((R_ECX:u32 & 0x1f:u32) = 0:u32) & R_CF:bool |
   ~((R_ECX:u32 & 0x1f:u32) = 0:u32) & low:bool(t1:u32)
addr 0x4 @asm ”jc 0x000000000000000a”
cjmp R_CF:bool, 0xa:u32, ”nocjmp0” # branch to 0xa if R_CF = true
label nocjmp0
```

3 BAP Architectural Overview

BAP is divided into front-end and back-end components that are connected by the BAP intermediate language (IL), as shown in Figure 1. The front end is responsible for lifting binary code for the supported architectures to the IL. The back-end implements our program analyses and verifications for low-level code.

The front end reads binary code from an execution trace or a region of a binary executable. When lifting instructions from a binary, BAP uses a linear sweep disassembly algorithm. The user or an analysis is responsible for directing BAP to properly aligned instructions. The result of lifting is an IL program.

An abbreviated definition of the IL syntax is shown in Table 1; the full IL syntax and semantics are provided at [4]. The `special` statement indicates a system call or other unmodeled behavior. Other statements have their obvious meaning. All expressions in BAP are side-effect free. The `unknown` expression indicates an unknown value; for instance, we use this to model the contents of

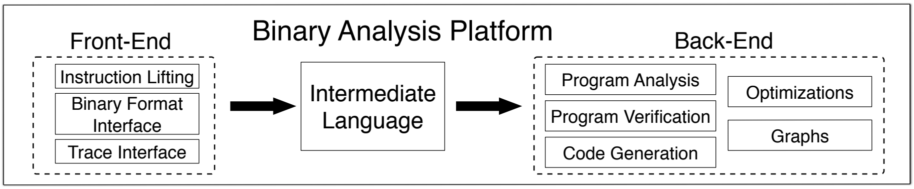


Fig. 1. The BAP binary analysis architecture and components

Table 1. An abbreviated syntax of the BAP IL

$$\begin{aligned}
\mathit{program} &::= \mathit{stmt}^* \\
\mathit{stmt} &::= \mathit{var} := \mathit{exp} \mid \mathtt{jmp}\ \mathit{exp} \mid \mathtt{cjmp}\ \mathit{exp},\mathit{exp},\mathit{exp} \mid \mathtt{assert}\ \mathit{exp} \\
&\quad\ \mid \mathtt{label}\ \mathit{label\_kind} \mid \mathtt{addr}\ \mathit{address} \mid \mathtt{special}\ \mathit{string} \\
\mathit{exp} &::= \mathtt{load}(\mathit{exp},\ \mathit{exp},\ \mathit{exp},\ \tau_{reg}) \mid \mathtt{store}(\mathit{exp},\ \mathit{exp},\ \mathit{exp},\ \mathit{exp},\ \tau_{\mathrm{reg}}) \mid \mathit{exp}\ \Diamond_b\ \mathit{exp} \\
&\quad\ \mid \Diamond_u\ \mathit{exp} \mid \mathit{var} \mid \mathtt{lab}(\mathit{string}) \mid \mathit{integer} \mid \mathtt{cast}(\mathit{cast\_kind},\ \tau_{\mathrm{reg}},\ \mathit{exp}) \\
&\quad\ \mid \mathtt{let}\ \mathit{var}\ \mathtt{=}\ \mathit{exp}\ \mathtt{in}\ \mathit{exp} \mid \mathtt{unknown}(\mathit{string},\ \tau)
\end{aligned}$$

registers having an undefined state after a specific instruction (e.g. the AF flag after shl). The semantics of $\mathtt{load}(e_1, e_2, e_3, \tau_{\mathrm{reg}})$ is to load from the memory specified by e_1 at address e_2. e_3 tells us the endianness to use when loading bytes from memory, which can vary at runtime on ARM. τ_{reg} tells us how many bytes to load. store is similar, but takes an additional parameter to specify the stored value.

The BAP IL can be transformed into other useful representations. One example is static single assignment (SSA) [1] form. SSA form makes use-def and def-use chains explicit in syntax, and enforces the use of three-address code. These changes often make it significantly easier to implement new analyses and optimizations.

Once a binary is lifted to the BAP IL, it can be analyzed by the BAP back-end. The BAP back-end consists of program analyses and transformations. We discuss these in more detail in Section 4.

Usage. Users are expected to use BAP's front-end to lift binary code to IL form, and then to interact with the analyses and transformations in the back-end. Users can use BAP command line utilities out of the box to perform standard operations. For instance, users can use the iltrans tool to create a pipeline of actions that 1) converts an IL program to SSA form; 2) applies all BAP optimizations; 3) converts back to IL form; 4) outputs a verification condition (VC) for the optimized program.

BAP can also be extended programatically. New analyses can build on existing analyses and transformations, allowing for modularity and reuse of implemented analyses similar to a source-level compiler architecture.

4 BAP Capabilities

Analyses and Optimizations. Analyses can either be accessed programatically, or via the command line iltrans utility. Built in analyses include the ability to:

- Compute slices for a source or a chop for a source/sink pair so that subsequent analysis only considers relevant parts of a program. For example, if we are interested in whether integer overflow occurs for a particular variable we can reason about the slice of statements affecting (backwards slicing) or affected by (forward slicing) that variable.

- Optimize the IL. Optimizations are an important part of the BAP framework for several reasons. First, the IL makes all side-effects explicit by default, many of which may not matter for a particular analysis. Deadcode elimination will remove these. For instance, deadcode elimination will remove OF, SF, ZF, AF, and PF in Example 2 because they are not relevant.

  In our coreutils experiments [8], we found that the use of optimizations resulted in an overall speedup of 4.5x in the time it took to generate and solve formulas, and enabled us to solve 81% of the VCs that could not be solved without optimizations.
- Evaluate the IL. Our evaluator allows us to run a BAP program and examine any dynamic properties. For instance, the evaluator can be used to record control flow, perform randomized testing of a software property, or verify that the IL semantics are consistent with the real program's.

Verification Conditions. BAP can create verification conditions using several methods. A verification condition (VC) is a syntactically generated boolean predicate over a program's input variables that is true if and only if some program property holds over the program's execution on that input. Naturally, a VC is valid if and only if the respective program property holds for all inputs. BAP generates VCs with respect to a postcondition, such that if the formula is true then the program terminates and the postcondition holds.

Built-in methods for generating VCs include:

- Dijkstra's weakest preconditions (WP). The process involves converting the BAP IL, which represents unstructured code, to Dijkstra's guarded command language. The resulting VC is $O(2^n)$ in size where n is the number of IL statements. Other methods produce smaller VCs.
- Efficient weakest preconditions. We implement two algorithms. First, we have implemented Flanagan and Saxe's algorithm, which guarantees the generated VC will be only $O(n^2)$ in size where n is the number of IL statements. Second, we have developed and proved correctness of a variant of Flanagan and Saxe that can be run in the forward direction [8].
- Forward symbolic execution [14]. Symbolic execution is built into BAP's evaluator.
- Direct (API) and filesystem bindings to STP [7], as well as the ability to interact via the filesystem with SMTLIB1 compliant decision procedures.

5 Applications

We have used the BAP toolchain for a number of binary analysis and verification tasks. Due to space, please refer to [6] for a full list. Example applications are:

- We designed and performed type reconstruction on compiled C programs in a system called TIE [10]. TIE analyzes each memory access in x86 to find variable locations (similar to VSA [2]), creates a system of type constraints based upon variable usage, and solves for a typing on all variables.

- We evaluated the performance of VC generation algorithms by checking VCs for leaf functions in GNU coreutils [8]. For instance, we tested each function to see if the overflow flag could be set, or if the return address could be overwritten[2]. For each condition, we generated a VC and checked its validity with standard SMT solvers (CVC3 and Yices).
- Perform binary-only symbolic execution. We are able to lift TEMU [5] instruction traces to our IL, add constraints on the input, e.g., to find inputs where a safety property breaks, generate an input that takes a specific branch in the trace, and so on. We have used this to perform automatic patch-based exploit generation, malware analysis, and other security-related tasks [5,6].

6 Limitations

BAP currently supports subsets of the x86 and ARM ISAs. Some features, like floating point and privileged instructions are unsupported. It is not possible to prove the correctness of BAP's lifting code correct because the semantics of the x86 ISA is not formally defined. Instead, we use random testing to identify any differences between the semantics of our lifted IL and behavior on a real processor.

BAP's lifting process expects to be pointed to an aligned sequence of instructions. Thus, the user must identify code locations. This can be done manually, by relying on symbol data, or by using a recursive descent analysis (such as IDA Pro). Lifting also assumes that code is static. BAP's execution trace feature can be used to reason about dynamic code.

Some analyses require indirect jumps to be resolved to concrete locations. For instance, it is not possible to generate VCs using weakest preconditions in the presence of unresolved indirect jumps, since weakest precondition is a static analysis. (It is still possible to use dynamic symbolic execution, however.)

7 Conclusion

BAP is a flexible binary analysis framework that enables program analysis and verification on binary code. BAP explicitly represents side effects of instructions in a simple, formally defined IL. A number of analyses, optimizations, and verification techniques are already built into BAP, and adding new ones is easy. The source code for BAP is periodically released at `http://bap.ece.cmu.edu`.

References

1. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
2. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison (August 2007)

[2] We chose these postconditions because they are non-trivial and can be applied to all functions.

3. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: Codesurfer/x86 - a platform for analyzing x86 executables. In: Proceedings of the International Conference on Compiler Construction (April 2005)
4. Binary Analysis Platform (BAP), `http://bap.ece.cmu.edu`
5. BitBlaze binary analysis project (2007), `http://bitblaze.cs.berkeley.edu`
6. Brumley, D.: `http://security.ece.cmu.edu`
7. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proceedings of the Conference on Computer Aided Verification, pp. 524–536 (July 2007)
8. Jager, I., Brumley, D.: Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab (February 2010)
9. 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)
10. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: Proceedings of the Network and Distributed System Security Symposium (February 2011)
11. Microsoft. Phoenix framework, `http://research.microsoft.com/phoenix/` (url checked April 21, 2011)
12. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: Proceedings of the Third Workshop on Runtime Verification, Boulder, Colorado, USA (July 2003)
13. Paradyn/Dyninst. Dyninst: An application program interface for runtime code generation, `http://www.dyninst.org` (url checked April 21, 2011)
14. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 317–331 (May 2010)
15. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)

HMC: Verifying Functional Programs Using Abstract Interpreters

Ranjit Jhala[1], Rupak Majumdar[2], and Andrey Rybalchenko[3]

[1] UC San Diego
[2] MPI-SWS and UC Los Angeles
[3] TU München

Abstract. We present Hindley-Milner-Cousots (HMC), an algorithm that reduces verification of safety properties of typed higher-order functional programs to interprocedural analysis for first-order imperative programs. HMC works as follows. First, it uses the type structure of the functional program to generate a set of logical refinement constraints whose satisfaction implies the safety of the source program. Next, it transforms the logical refinement constraints into a simple first-order imperative program and an invariant that holds iff the constraints are satisfiable. Finally, it uses an invariant generator for first-order imperative programs to discharge the invariant. We have implemented HMC and describe preliminary experimental results using two imperative checkers – ARMC and INTERPROC – to verify OCAML programs. By composing type-based reasoning grounded in program syntax and state-based reasoning grounded in abstract interpretation, HMC enables the *fully automatic* verification of programs written in modern programming languages.

1 Introduction

Automatic verification of semantic properties of modern programming languages is an important step toward reliable software systems. For higher-order functional programming languages with inductive and polymorphic datatypes, the main verification tool has been type systems. These type systems traditionally capture only coarse data-type properties (*e.g.,* functions expecting `ints` are only passed `ints`), but not more precise, semantic properties (*e.g.,* that an array index is within bounds). For first-order imperative programming languages, automatic tools based on abstract interpretation, such as ASTREE [3], SLAM [1], BLAST [7], *etc.*, can infer program invariants and prove many semantic properties of practical interest. While these tools faithfully model the semantics of base values like `ints`, they are overly imprecise on modern programming features such as closures, higher-order functions, inductive datatypes or polymorphism.

We present Hindley-Milner-Cousots (HMC), an algorithm that combines type-based reasoning for higher-order languages with invariant generation for first-order languages to prove semantic properties of programs fully automatically and without additional programmer annotations.

The link between types and invariants is the notion of *refinement type checking* [2, 14, 20, 26], a type-based analogue of Floyd-Hoare logic. A refinement type is a type whose "leaves" are base types decorated with *refinement predicates*. For example, the refinement type $\{\texttt{x:int} \mid \texttt{x} < 100\}$ `list` describes a list of integers, each

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 470–485, 2011.

of which is smaller than 100 and $\texttt{int} \rightarrow \{\texttt{x:int} \mid \texttt{x} \neq 0\} \rightarrow \texttt{int}$ describes a function (*e.g.,* integer division) whose second (curried) argument must be non-zero. By riding atop type-structure, refinements can express sophisticated data structure invariants as well [4, 5, 12]. While refinement type checking can be used to verify functional programs [2], the programmer must manually provide the refinements which is analogous to the burden of writing loop-invariants together with pre- and post-conditions in the imperative setting. HMC eliminates the need for programmer annotations and thereby enables automatic checking via a three-step process.

Step 1: Constraint Generation. HMC generates a set of refinement constraints whose satisfaction implies the safety of the source program. To verify safety of a functional program, we need to compute safe overapproximations of the sets of values that various expressions can evaluate to (*i.e.,* the functional analogue of "reachable states" in the imperative setting). With refinement types, overapproximation is formalized via subtyping. Thus, in the first phase, HMC makes a syntax directed pass over the functional program to generate a set of subtyping constraints over *refinement templates* that represent the unknown refinement types for various program expressions. The templates employ *refinement variables* κ as placeholders for the unknown refinement predicates, which are analogous to program invariants, that decorate the leaves of the complex types. Crucially, as the overapproximation is structured via types, we can use the standard rules for subtyping complex types to reduce the complex subtyping constraints to a set of simple implication constraints [13, 22], whose satisfaction implies program safety.

Step 2: Constraint Translation. Next, HMC transforms the implication constraints into a first-order imperative program that is safe if and only if the constraints are satisfiable. This translation – our main technical contribution – is founded upon two key insights. First, the refinement variables κ, normally viewed as placeholders for (unknown) refinement predicates, semantically represent (unknown) n-ary relations over the value being defined by the refinement type and the $n-1$ variables that are in scope at the point where the type is defined. Second, the constraints on each κ can be used to encode a simple *first-order imperative* function $\texttt{F}_\kappa$ whose input-output semantics corresponds to an n-ary relation that satisfies the constraints on κ. The $n-1$ components of the relation are treated as input parameters of the function $\texttt{F}_\kappa$, and the value component of the relation becomes the output of the function. Using these insights we design an algorithm that translates type-bindings into function calls and implications into assignments/assumes, respectively. Thus, we obtain a first-order imperative program that is safe if and only if the constraints are satisfiable, *i.e.,* whose safety implies the safety of the source functional program.

Thus, the two-step HMC algorithm uses type-structure to reduce the safety of a higher-order functional program to the safety of a first-order imperative program.

Step 3: Verification of Obtained First-Order Program. In a third step, we use existing invariant generation tools for first-order imperative programs to automatically verify the safety of the imperative program produced during the second step.

The most immediate dividend of our approach is that HMC allows one to readily apply any of the well-developed semantic imperative program analyses to the verification of modern software with polymorphism, inductive datatypes, and higher-order functions. More importantly, HMC provides a "separation-of-concerns" that can open the door to a suite of precise model checkers and abstract interpreters capable of handling languages with advanced features. Using HMC, the analysis designer can factor the analysis into two parts: a syntactic, type-system based component that analyzes macroscopic language concerns like collections, inductive types, polymorphism, closures, *etc.*, and a semantic, abstract interpretation-based component that analyzes microscopic language concerns like invariant relationships between primitive integers or booleans. Thus, HMC provides a simple way to incorporate independent progress in type systems for specifying complex control as well as dataflow and in invariant generation techniques into the verification flow. For example, one can tune the precision and scalability of an analysis either by changing the amount of context-sensitivity in the type system (*e.g.,* via intersection types) or by using more/less precise abstract domains (*e.g.,* using polyhedra instead of octagons).

To demonstrate the feasibility of our approach, we have developed two safety verifiers for ML programs, HMC(ARMC) and HMC(INTERPROC), which use the CEGAR-based ARMC [21] software model checker and the Polyhedra-based INTERPROC [17] analyzer, respectively, to verify the translated programs. This allows *fully automatic* verification of a set of OCAML benchmarks for which previous approaches either required manual annotations (either the refinement types [26] or their constituent predicates [22]), or an elaborate customization and adaptation of the counterexample-guided abstraction refinement paradigm [24].

Related Work. Our starting point was the vast body of work in the verification of imperative programs (see, e.g., [10] for a survey), including tools such as SLAM [1], BLAST [8], and ASTREE [3], and to "lift" the techniques to higher-order programming languages. We were influenced by work on refinement types [6, 13] implemented in dependent ML [26] and, more recently, combined with predicate abstraction [12, 22], but wanted to eliminate the need for explicit annotations (or predicates).

Kobayashi [15, 16] gives an algorithm for model checking higher order programs by a reduction to model checking for higher-order recursion schemes (HORS) [19], which has been augmented to perform CEGAR [24, 25]. For safety verification, HMC shows a promising alternative, enabling us to reuse the vast literature on invariant generation for first order programs (using abstract interpreters or model checkers).

While we have implemented our tool for ML, our constraint language is generic and can express refinement constraints arising out of other expressive source languages, such as F7 [2] or C [23], which include module signatures, recursive and contextual types, mutable state, *etc.* Thus, through the collaboration of types and invariants, HMC opens the door to the automatic safety verification of complex properties of programs in modern languages.

2 Overview

We outline the steps of the HMC algorithm. For lack of space, we omit the full formalization and correctness proofs and provide the main insights through an example. The formalization can be found in [11].

```
let rec iteri i xs f =
  match xs with
  | []     -> ()
  | x::xs' -> f i x;
              iteri (i+1) xs' f

let mask a xs =
  let g j y = a.(j) <- y && a.(j) in
  if Array.length a = List.length xs then
    iteri 0 xs g
```

Fig. 1. ML Example

An ML Example. Figure 1 shows a simple ML program that updates an array a using the elements of the list xs. The program comprises two functions. The first function is a higher-order list *indexed-iterator*, iteri, that takes as arguments a starting index i, a (polymorphic) list xs, and an iteration function f. The iterator goes over the elements of the list and invokes f on each element and the index corresponding to the element's position in the list. The second function is a client, mask, of the iterator iteri that takes as input a boolean array a and a list of boolean values xs, and if the lengths match, calls the indexed iterator with an iteration function g that masks the $\mathtt{j}^{th}$ element of the array.

Suppose that we wish to statically verify the safety of the array reads and writes in function g; that is to prove that whenever g is invoked, $0 \leq \mathtt{j} < \mathtt{len}(\mathtt{a})$. As this example combines higher-order functions, recursion, data-structures, and arithmetic constraints on array indices, it is difficult to analyze automatically using either existing type systems or abstract interpretation implementations in isolation. The former do not infer handle arithmetic constraints on indices, and the latter do not precisely handle higher-order functions and are often imprecise on data structures. We show how our technique can automatically prove the correctness of this program.

Refinement Types. To verify the program, we compute program invariants that are expressed as *refinements* of ML types with predicates over program values [2, 13, 22]. The predicates are additional constraints that must be satisfied by every value of the type. We work with a fixed set of *base types* β, comprising int for *integer* values, bool for *boolean* values, and ui, a family of *uninterpreted types* that encode complex source language types such as products, sums, recursive types *etc.*A base value, say of type β, can be described by the refinement type $\{\nu\!:\!\beta \mid p\}$ where ν is the value variable of the refinement type that names the value being defined, and p is a *refinement predicate* which constrains the range of ν to a subset of β. For example, $\{\nu\!:\!\mathtt{int} \mid 0 \leq \nu < \mathtt{len}(\mathtt{a})\}$ denotes the set of integers that are between 0 and the value of the expression $\mathtt{len}(\mathtt{a})$. Thus, the (unrefined) type int abbreviates $\{\nu\!:\!\mathtt{int} \mid \mathit{true}\}$. Base types can be combined to construct *dependent function types*, where the value variable for the input type, *i.e.,* the name of the formal parameter, can appear in the refinement predicates in the output type, thereby expressing a "post-condition" that relates the function's outputs with its inputs. For example, $\{\mathtt{x}\!:\!\mathtt{int} \mid 0 \leq \mathtt{x}\} \rightarrow \{\nu\!:\!\mathtt{int} \mid \nu = \mathtt{x} + 1\}$ is the type of a function which takes a non-negative integer parameter and returns an output which is one more than the input. Thus, the input and output types describe pre- and post-conditions for the function. In the following, we write $\mathtt{x}\!:\!\beta$ for the type $\{\mathtt{x}\!:\!\beta \mid \mathit{true}\}$, and $\mathtt{x}\!:\!r$ for $\{\mathtt{x}\!:\!\beta \mid r\}$, when β is clear from the context,

Safety Specification. Refinement types can be used to *specify* safety properties by encoding pre-conditions into primitive operations of the language. For example, consider the array read `a.(j)` (resp. write `a.(j) <- e`) in g which is an abbreviation for `get a j` (resp. `set a j e`.) By giving `get` and `set` the types

$$\mathtt{a}\!:\!\alpha\ \mathtt{array} \to \{\mathtt{i}\!:\!\mathtt{int} \mid 0 \le \mathtt{i} < \mathtt{len}(\mathtt{a})\} \to \alpha\ ,$$
$$\mathtt{a}\!:\!\alpha\ \mathtt{array} \to \{\mathtt{i}\!:\!\mathtt{int} \mid 0 \le \mathtt{i} < \mathtt{len}(\mathtt{a})\} \to \alpha \to \mathtt{unit}\ ,$$

we can specify that in any program the array accesses must be within bounds. More generally, arbitrary safety properties can be specified [22] by giving `assert` the refinement type $\{\mathtt{p}\!:\!\mathtt{bool} \mid \mathtt{p} = \mathit{true}\} \to \mathtt{unit}$.

We assume that the call-by-value dynamic semantics of ML programs is formalized using a standard small-step (contextual) operational semantics.[1] We write $\rightsquigarrow$ for the single evaluation step relation for expressions, and write $\overset{*}{\rightsquigarrow}$ to describe the reflexive, transitive closure of $\rightsquigarrow$. To capture program errors, we assume there is a special Err value, and if a constant is applied to a value that is not in the domain of the constant (*e.g.,* accessing an array out of bounds, or calling `assert` with `false`), then the application reduces to Err. For a program e, we say that e *safe* if there is no derivation of the form $\mathtt{e} \overset{*}{\rightsquigarrow} \mathsf{Err}$. In other words, a program is safe if it never reduces to Err.

Safety Verification. The ML type system is too imprecise to prove the safety of the array accesses in our example as it infers that g has type $\mathtt{j}\!:\!\mathtt{int} \to \mathtt{y}\!:\!\mathtt{bool} \to \mathtt{unit}$, *i.e.,* that g can be called with *any* integer j. If the programmer manually provides the refinement types for all functions and polymorphic type instantiations, refinement-type checking [2, 5, 26] can be used to verify that the provided types were consistent and strong enough to prove safety. This is analogous to providing pre- and post-conditions and loop invariants for verifying imperative programs. For our example, a refinement type system could check the program if the programmer provided the types:

$$\begin{aligned}\mathtt{iteri} :: \mathtt{i}\!:\!\mathtt{int} \to \{\mathtt{xs}\!:\!\alpha\ \mathtt{list} \mid 0 \le \mathtt{len}(\mathtt{xs})\} \to&\\ (\{\mathtt{j}\!:\!\mathtt{int} \mid \mathtt{i} \le \mathtt{j} < \mathtt{i} + \mathtt{len}(\mathtt{xs})\} \to \alpha \to \mathtt{unit}) \to \mathtt{unit}&\\ \mathtt{g} :: \{\mathtt{j}\!:\!\mathtt{int} \mid 0 \le \mathtt{j} < \mathtt{len}(\mathtt{a})\} \to \mathtt{bool} \to \mathtt{unit}&\end{aligned}$$

Automatic Verification via HMC. As even this simple example illustrates, the annotation burden for verification can be quite high. Instead, we show how our algorithm combines type-based reasoning for complex language features and abstract interpretation for first-order control flow to automatically verify the program without requiring refinement annotations. Our HMC algorithm proceed as follows. First, we use the *source* program to generate a set of constraints which is satisfiable if the program is safe. Second, we translate the constraints into a first-order *imperative* target program which is safe iff the set of constraints is satisfiable. After these two steps, we can analyze the target program with any first-order safety analyzer. If the analyzer determines the target is

[1] Our formalization of the algorithm actually uses a core fragment μML of ML. We keep the following discussion simple by simply referring to ML programs.

safe, we can soundly conclude that the constraints are satisfiable, and hence, the source program is safe. Next, we illustrate these steps using the source program from Figure 1.

Notation. Let X be a set of variables. We use ν, $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ and subscripted versions thereof to range over X. A *state* σ is a partial map from variables X to values in the universe $\mathcal{U}(\beta)$ of values of type β. We lift states to maps from expressions to values and predicates to boolean values in the standard manner. We write $[\cdot]$ for the state with empty domain, and write $\sigma[\mathtt{z} \mapsto v]$ for the state that maps the variable $\mathtt{z}$ to v and all other variables y to $\sigma(y)$.

2.1 Step 1: Constraint Generation

First, we generate a system of *refinement constraints* for the source program [13, 22]. To do so, we (a) build templates that refine the ML types with refinement variables that stand for the unknown refinements, and (b) make a syntax-directed pass over the program to generate subtyping constraints that capture the flow of values.

Templates. For the functions iteri and g from Figure 1, with the respective ML types

$$\mathtt{i}\!:\!\mathtt{int} \rightarrow \mathtt{xs}\!:\!\alpha\ \mathtt{list} \rightarrow (\mathtt{j}\!:\!\mathtt{int} \rightarrow \alpha \rightarrow \mathtt{unit}) \rightarrow \mathtt{unit}$$
$$\mathtt{j}\!:\!\mathtt{int} \rightarrow \mathtt{bool} \rightarrow \mathtt{unit}$$

we generate the templates

$$\mathtt{i}\!:\!\mathtt{int} \rightarrow \mathtt{xs}\!:\!\{0 \leq \mathtt{len}(\mathtt{xs})\} \rightarrow (\mathtt{j}\!:\!\kappa_1(\mathtt{j},\mathtt{i},\mathtt{xs}) \rightarrow \alpha \rightarrow \mathtt{unit}) \rightarrow \mathtt{unit}$$
$$\mathtt{j}\!:\!\kappa_2(\mathtt{j},\mathtt{a},\mathtt{xs}) \rightarrow \mathtt{bool} \rightarrow \mathtt{unit}$$

The templates refine the ML types with *parameterized refinement variables* that represent the unknown refinements. $\kappa_1(\mathtt{j},\mathtt{i},\mathtt{xs})$ represents the unknown refinement that describes the values passed as the first input to the function f that is used by the iterator iteri. The values are the first elements of tuples belonging to a ternary relation between the values of j and the two other program variables *in-scope* at that point, namely i and xs. $\kappa_2(\mathtt{j},\mathtt{a},\mathtt{xs})$ represents the unknown refinement that describes the values passed as the first input to g. In this case, the values belong to a ternary relation over j: the formal and the two variables a and xs in scope at that program point.

For clarity of exposition, we have use the trivial refinement *true* for some variables (*e.g.,* for α and bool); HMC would automatically infer these refinements. We model the length of lists (resp. arrays) with an uninterpreted function len from the lists (resp. arrays) to integers, and (again, for brevity) add the refinement stating xs has a non-negative length in the type of iteri.

Constraints. Informally, refinement constraints reduce the flow of values within the program into subtyping relationships that must hold between the source and target of the flow. A *refinement* r is either a *concrete predicate* p drawn from some predicate language or a parameterized refinement variable $\kappa(\mathtt{x}_0,\ldots,\mathtt{x}_n)$, where κ is a refinement variable of arity n. We assume, without loss of generality, that each κ has a fixed arity. A *refinement type binding* ρ is a triple $\{\mathtt{x}\!:\!\beta \mid r\}$ comprising a *variable* x that is being bound, a base type β describing the base type of $\mathtt{x}$, and a refinement r that describes

Predicates	$\sigma \models p$	iff $\sigma(p) = \mathit{true}$
Environments	$\Sigma, [\cdot] \models \emptyset$	
	$\Sigma, \sigma \models G;\{x{:}\beta \mid r\}$	iff $\Sigma, \sigma \setminus x \models G$ and $\Sigma, \sigma \models \{x{:}\beta \mid r\}$
Refinements	$\Sigma, \sigma \models \{x{:}\beta \mid p\}$	iff $\sigma \models p$
	$\Sigma, \sigma \models \{x{:}\beta \mid \kappa(y_0, \ldots, y_n)\}$	iff $(\sigma(y_0), \ldots, \sigma(y_n)) \in \Sigma(\kappa)$
Constraints	$\Sigma \models G \vdash \{\mathtt{x}_1{:}\beta \mid r_1\} \prec \{\mathtt{x}_2{:}\beta \mid r_2\}$	iff For all σ : $\Sigma, \sigma \models G;\{\nu{:}\beta \mid r_1\}$ implies $\Sigma, \sigma \models \{\mathtt{x}_1{:}\beta \mid r_2[\mathtt{x}_1/\mathtt{x}_2]\}$

Fig. 2. Constraint Satisfaction

an invariant satisfied by all the values bound to x. A *refinement environment* G is a sequence of refinement bindings. A *refinement constraint* $G \vdash \{\mathtt{x}{:}\beta \mid r_1\} \prec \{\mathtt{x}{:}\beta \mid r_2\}$ states that when the program variables satisfy the invariants described in G, the type r_1 must be a subtype of r_2, that is, the set of values described by the refinement r_1 *must be included in* the set of values described by the refinement r_2.

Satisfaction. A *relational interpretation* for κ of arity n is a subset of $\mathcal{U}(\beta_1) \times \ldots \mathcal{U}(\beta_n)$ for appropriate types $\beta_1, \ldots, \beta_n$ for the parameters of κ. A *relational model* Σ is a map from refinement variables κ to relational interpretations. Figure 2 formalizes the notion of when a relational interpretation Σ *satisfies* a constraint. A state satisfies a predicate if the predicate evaluates to true in the state. A state satisfies a predicate refinement binding if the tuple of values of relevant variables belongs to the relation corresponding to the refinement. A state satisfies an environment if it satisfies each binding in the environment. A relational interpretation satisfies a constraint if every state that satisfies the LHS of the constraint also satisfies the RHS of the constraint. A relational interpretation satisfies a set of constraints if it satisfies each constraint in the set.

Constraint Generation. Let $\mathsf{Generate}(e)$ denote the procedure that takes as input a program e and uses the type structure of the program to generate constraints. $\mathsf{Generate}(e)$ proceeds syntactically over the structure of the program. We omit the full description of the procedure $\mathsf{Generate}(e)$, which is similar to the constraint generation procedure for refinement type constraints (*e.g.,* [2, 6, 13, 22]). Theorem 1 summarizes the main property of $\mathsf{Generate}(e)$.

Theorem 1. *If* $\mathsf{Generate}(e)$ *is satisfiable then e is safe.*

For our example, the following subtyping constraints are generated syntactically from the code. First consider the function `iteri`. The call to `f` generates

$$G \vdash \{\nu{:}\mathtt{int} \mid \nu = \mathtt{i}\} \prec \{\nu{:}\mathtt{int} \mid \kappa_1(\nu, \mathtt{i}, \mathtt{xs})\} \qquad \text{(c1)}$$

where ν is the parameter's value, and the environment bindings are

$$\begin{aligned} G \doteq\ & \mathtt{i}{:}\mathtt{int};\ \{\mathtt{xs}{:}\alpha\ \mathtt{list} \mid 0 \leq \mathtt{len}(\mathtt{xs})\}; \\ & \mathtt{x}{:}\alpha; \{\mathtt{xs}'{:}\alpha\ \mathtt{list} \mid 0 \leq \mathtt{len}(\mathtt{xs}') = \mathtt{len}(\mathtt{xs}) - 1\} \end{aligned}$$

The constraint ensures that at the call-site, the refinement of the actual is included in (*i.e.,* a subtype of) the refinement of the formal. The bindings in the environment are simply the refinement templates for the variables in scope at the point the value flow occurs. The refinement type system yields the information that the length of $\mathtt{xs}'$ is one less than $\mathtt{xs}$ as the former is the tail of the latter [12, 26]. Similarly, the recursive call to `iteri` generates

$$G \vdash \mathtt{j}\!:\!\kappa_1(\mathtt{j},\mathtt{i},\mathtt{xs}) \rightarrow \alpha \rightarrow \mathtt{unit} \prec (\mathtt{j}\!:\!\kappa_1(\mathtt{j},\mathtt{i},\mathtt{xs}) \rightarrow \alpha \rightarrow \mathtt{unit})[\mathtt{i}+1/\mathtt{i}][\mathtt{xs}'/\mathtt{xs}]$$

which states that type of the actual `f` is a subtype of the third formal parameter of `iteri` after applying substitutions $[\mathtt{i}+1/\mathtt{i}]$ and $[\mathtt{xs}'/\mathtt{xs}]$ that represent the passing in of the actuals $\mathtt{i}+1$ and $\mathtt{xs}'$ for the first two parameters respectively. By pushing the substitutions inside and applying the standard rules for function subtyping this constraint simplifies to

$$G \vdash \mathtt{j}\!:\!\kappa_1(\mathtt{j},\mathtt{i}+1,\mathtt{xs}') \prec \mathtt{j}\!:\!\kappa_1(\mathtt{j},\mathtt{i},\mathtt{xs}) \qquad \text{(c2)}$$

Next, consider the function `mask`. The array accesses in g generate

$$G'; \mathtt{j}\!:\!\kappa_2(\mathtt{j},\mathtt{a},\mathtt{xs}); \mathtt{y}\!:\!\mathtt{bool} \vdash \{\nu = \mathtt{j}\} \prec \{0 \leq \nu < \mathtt{len}(\mathtt{a})\} \qquad \text{(c3)}$$

a "bounds-check" constraint where G' has bindings for the other variables in scope, namely $\mathtt{a}\!:\!\mathtt{bool\ array}$ and $\{\mathtt{xs}\!:\!\mathtt{bool\ list} \mid 0 \leq \mathtt{len}(\mathtt{xs})\}$. Finally, the flow due to the third parameter for the call to `iteri` yields

$$G'; \mathtt{len}(\mathtt{a}) = \mathtt{len}(\mathtt{xs}) \vdash \mathtt{j}\!:\!\kappa_2(\mathtt{j},\mathtt{a},\mathtt{xs}) \rightarrow \mathtt{bool} \rightarrow \mathtt{unit} \prec \mathtt{j}\!:\!\kappa_1(\mathtt{j},0,\mathtt{xs}) \rightarrow \mathtt{bool} \rightarrow \mathtt{unit}$$

where, on the RHS, we have substituted the actuals 0 and `xs` for the formals `i` and `xs`. The last conjunct in the environment represents the guard from the `if` under whose auspices the call occurs. By standard function subtyping, the above reduces to

$$G'; \mathtt{len}(\mathtt{a}) = \mathtt{len}(\mathtt{xs}) \vdash \mathtt{j}\!:\!\kappa_1(\mathtt{j},0,\mathtt{xs}) \prec \mathtt{j}\!:\!\kappa_2(\mathtt{j},\mathtt{a},\mathtt{xs}) \qquad \text{(c4)}$$

By Theorem 1, if the set of constraints given by (c1), (c2),(c3), and (c4) is satisfiable, then the program is safe.

2.2 Step 2: Translation to Imperative Program

Our main contribution is a translation from the satisfiability problem for a set of refinement constraints to the safety verification problem for an imperative program.

Imperative Programs. We write imperative programs in μC, a first-order imperative language with variables ranging over base types β. An *instruction* is either an assignment $\mathtt{x} \leftarrow e$, an assume `assume` p, an assert `assert` p, or the sequencing `I; I` or non-deterministic choice $\mathtt{I} [\!] \mathtt{I}$ of two instructions. An *assignment* to a target variable is of one of three kinds. Either (1) $\mathtt{x} \leftarrow e$: an expression e over the variables is evaluated and assigned to the target variable `x`, or, (2) $\mathtt{x} \leftarrow \mathtt{nondet}()$: an arbitrary non-deterministically chosen value of the appropriate base type is assigned to `x`, *i.e.,* the target variable is "havoc-ed", or (3) $\mathtt{x} \leftarrow \mathtt{F}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$: a function $\mathtt{F}$ is called, and its return value is assigned to the target variable $\mathtt{x}$. A *function definition* has a name $\mathtt{F}$, a sequence of formal parameters $\mathtt{z}_1, \ldots, \mathtt{z}_n$, a body instruction $\mathtt{I}$, and a return variable $\mathtt{z}_0$. A *program* is a set of functions including a distinguished *entry* function $\mathtt{F}_0$ that takes no arguments.

Imperative Semantics. We formalize the call-by-value semantics of $\mu\mathsf{C}$ programs using a standard big-step transition relation between program configurations. A *configuration* is either a state, *i.e.,* a partial map from variables X to values, or a special unsafe configuration Err. All the variables in an $\mu\mathsf{C}$ program are *local*. That is, the variables of each (state) configuration describe the values of the variables of a single "stack-frame". The transition relation is described by the judgment $\mathtt{P}, \mathtt{I} \vdash \sigma \hookrightarrow \sigma'$ that stipulates that in the program $\mathtt{P}$, the execution of the instruction $\mathtt{I}$ causes the machine to move from a configuration σ to the configuration σ'.

The expression and havoc assignments update the target variable with the RHS and a non-deterministically chosen value respectively. The call assignment updates the target value with any of the possible values returned by the callee (*i.e.,* the value of the return variable of the callee in the exit configuration of the callee.) Dually, the return instruction simply assigns the return value into the return variable $\mathtt{z}_0$. The assume instruction proceeds without updating the state only if the corresponing predicate holds (and otherwise, the program halts). Thus, $\mu\mathsf{C}$ eschews if-then-else instructions in favor of the the more general assume and choice instructions. The assert instruction is like the assume, but if the predicate does not hold, the system transitions into the configuration Err (in which it remains forever.)

Let $\mathtt{P}$ be an $\mu\mathsf{C}$ program whose entry function $\mathtt{F}_0$ has the body $\mathtt{I}_0$. We say that $\mathtt{P}$ *is $\mu\mathsf{C}$-safe* if there is no transition $\mathtt{P}, \mathtt{I}_0 \vdash [\cdot] \hookrightarrow \mathsf{Err}$.

Translation. The constraints generated in Step 1 encode the semantics of program computations. In the second step, we define a translation $[\![C]\!]$ that takes a set of constraints C and constructs an $\mu\mathsf{C}$-program such that C is satisfiable iff $[\![C]\!]$ is safe. Our translation is based on two observations.

Refinements are Relations. The first observation is that refinement variables in the constraints stand for *relations* between program variables: the set of values denoted by a refinement type $\{\mathtt{x}_0 : \beta_0 \mid p\}$ where p is a predicate that refers to the program variables $\mathtt{x}_0, \ldots, \mathtt{x}_n$ of base types $\beta_0, \ldots, \beta_n$ is equivalent to

$$\{t_0 \mid \exists (t_1, \ldots, t_n) \text{ s.t. } (t_0, \ldots, t_n) \in R_p \wedge_{i=1}^{n} t_i = \mathtt{x}_i\}$$

where R_p is an $(n+1)$-ary relation in $\beta_0 \times \ldots \times \beta_n$ defined by p. For example, $\{\nu : \mathtt{int} \mid \nu \leq \mathtt{i}\}$ is equivalent to the set $\{t_0 \mid \exists t_1 \text{ s.t. } (t_0, t_1) \in R_{\leq} \wedge t_1 = \mathtt{i}\}$, where $R_{\leq}$ is the standard $\leq$-ordering relation over the integers. In other words, each parameterized refinement variable $\kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ can be seen as the projection on the first co-ordinate of a $(n+1)$-ary relation over the variables $(\mathtt{x}_0, \ldots, \mathtt{x}_n)$. Thus, the problem of determining the satisfiability of the constraints is analogous to the problem of determining the existence of appropriate relations.

From Relations to Imperative Programs. The second observation is that the problem of finding appropriate relations can be reduced to the problem of analyzing a simple imperative program, which encodes each refinement variable with a function whose input-output semantics correspond to the relation described by the refinement variable.

The imperative program derived from a set of constraints C consists of a set of mutually recursive functions, one for each parameterized refinement variable in C, together with a "main" function that checks the safety property. In particular, for the variable κ_i with arity $n + 1$, the imperative program has a function $\mathtt{F}_i$ that enjoys the following *function property*: $\mathtt{F}_i$ takes n arguments $v_1, \ldots, v_n$ and (non-deterministically) returns a value v_0 iff the tuple $v_0, \ldots, v_n$ is in the relation corresponding to κ_i in *every* relational model that satisfies C. Following this intuition, an environment binding $\mathtt{x}{:}\kappa_i(\mathtt{y}_1, \ldots, \mathtt{y}_n)$ can be encoded as a function call $\mathtt{x} \leftarrow \mathtt{F}_i(\mathtt{y}_1, \ldots, \mathtt{y}_n)$ and each lower-bound constraint on $kvar_i$, *i.e.,* where κ_i appears on the RHS can be encoded as a return from $\mathtt{F}_i$ after a prefix of instructions that establishes the conditions of the LHS of the lower-bound constraint. Next, we outline the translation $[\![C]\!]$, summarized in Figure 3.

Functions from Refinement Variables. Figure 4 shows the imperative program translated from the constraints for our running example. There are two functions $\mathtt{F}_1$ and $\mathtt{F}_2$, corresponding to the refinement variables κ_1 and κ_2. The function $\mathtt{F}_1$ encodes the function property for κ_1. The formals $\mathtt{z}_1, \mathtt{z}_2$ encode the second and third elements of the relation κ_1. The return value encodes the first element of the relation κ_1. The body of the function is the non-deterministic choice between a set of two blocks which encode κ_1's lower-bound constraints (c1) and (c2) respectively. Similarly, the function $\mathtt{F}_2$ encodes the function property for κ_2, via a single block that encodes κ_2's only lower-bound constraint (c4).

Bound Translation. The translation gathers all the constraints whose RHS have concrete refinements into a set

$$C\!\downarrow\!\bot \;\doteq\; \{c \in C \mid c \equiv \_ \vdash \_ \prec p\}$$

and translates these constraints into the entry function $\mathtt{f}_0$. Intuitively, in such constraints the RHS defines a concrete "upper bound" on the set of tuples that satisfy LHS. In the translated μC program, the entry function enforces the upper bound via `assert` instructions as described below. The main function $\mathtt{F}_0$, in which execution starts, encodes the concrete-upper-bound (*i.e.,* "bounds-check") constraint (c3) which stipulates that the value of the variable $\mathtt{j}$ is within bounds. The body of $\mathtt{F}_0$ translates the constraint to an assertion over the corresponding variables. As with the other functions, the main function is the non-deterministic choice of all the blocks that encode the individual upper-bound constraints.

Blocks. To ensure that $\mathtt{F}_i$ satisfies the function property, we first gather the set $C\!\downarrow\!\kappa_i$ of constraints where κ_i appears on the RHS of the constraint. Formally,

$$C\!\downarrow\!\kappa_i \;\doteq\; \{c \in C \mid c \equiv \_ \vdash \_ \prec \kappa_i(\_)\}$$

Each constraint in the set $C\!\downarrow\!\kappa_i$ is individually translated into a block of straight-line assignments and assumes that has the *block property* that the state at the end of the block, maps the formals $\mathtt{z}_1, \ldots, \mathtt{z}_n$ and the return value $\mathtt{z}_0$ to a tuple of values that must belong in every relational model of κ_i that satisfies the constraint. Thus, the body instruction of $\mathtt{F}_i$, *i.e.,* the choice composition of all the blocks is such that each tuple of inputs and output of $\mathtt{F}_i$ belongs in every relational interpretation of κ_i.

To ensure that the translation of each constraint $G \vdash \{\mathtt{x}_1 : \beta \mid r_1\} \prec \{\mathtt{x}_2 : \beta \mid r_2\}$ in $C \downarrow \kappa_i$ has the block property, we translate the constraint into a straight-line block of instructions with three parts: a sequence of instructions that establishes the environment bindings ($[\![G]\!]$), a sequence of instructions that "gets" the values corresponding to the LHS ($[\![\{\mathtt{x} : \beta \mid r_1\}_{get}]\!]$) and a sequence of instructions that "sets" the return value of $\mathtt{F}_i$ appropriately ($[\![\{\mathtt{x} : \beta \mid r_2\}]\!]_{set}$).

Get Instructions. Each environment binding is encoded as a local variable, and gets translated as a "get" operation that defines the local variable as follows. Bindings with unknown refinements $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ are translated into calls $\mathtt{x}_0 \leftarrow \mathtt{F}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ to $\mathtt{F}_i$ with arguments $\mathtt{x}_1, \ldots, \mathtt{x}_n$, with the return value assigned to $\mathtt{x}_0$. Bindings with concrete refinements p are translated into non-deterministic assignments followed by an `assume` enforcing that the non-deterministically assigned values satisfy p.

Set Instructions. Each RHS refinement is translated into a "set" operation as follows. A concrete refinement p is translated into an `assert` which enforces that the RHS refinement is indeed an upper bound on the values populating the corresponding type in the inclusion constraints. A parameterized refinement $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ is translated into an `assume` that establishes the equalities between each x_i and the formal $\mathtt{z}_i$ representing the i^{th} tuple element, followed by a `return` $\mathtt{x}_0$. Thus, the translation guarantees that any execution that reaches the end of the block is such that the tuple of values of the return variable and formals of $\mathtt{F}_i$ satisfies the constraint to which the RHS refinement (over κ_i) belongs.

Correctness of Translation. The correctness of the key translation step of the HMC algorithm is stated by Theorem 2. Due to lack of space we defer the proof to [11].

Theorem 2. *[Translation] C is satisfiable iff $[\![C]\!]$ is μC-safe.*

Consider the constraint (c2) which is translated to the second block in $\mathtt{F}_1$ (*i.e.,* the block after the non-deterministic choice $[\!]$). The (trivial) environment binding $\mathtt{i}$:`int`, is encoded as a non-deterministic assignment $\mathtt{i} \leftarrow \mathtt{nondet}()$ followed by the (elided) assume `assume` *true*. The (non-trivial) environment binding $\{\mathtt{xs} : \alpha\ \mathtt{list} \mid 0 \leq \mathtt{len}(\mathtt{xs})\}$ is encoded as

$$\mathtt{xs} \leftarrow \mathtt{nondet}();\ \mathtt{assume}\ (0 \leq \mathtt{len}(\mathtt{xs}))$$

where in the encoded program `xs` takes on values of a basic uninterpreted type `ui`, and `len` is an uninterpreted function from `ui` to `int`. Similarly $\mathtt{xs}'$ gets assigned an arbitrary value, that has non-negative length and whose length is 1 less than that of `xs`. The LHS of (c2) corresponds to the environment binding $\mathtt{j} : \kappa_1(\mathtt{j}, \mathtt{i}+1, \mathtt{xs}')$. Thus, in the encoded block, the local `j` is defined via a (recursive) call to $\mathtt{F}_1(\mathtt{i}+1, \mathtt{xs}')$. The block is terminated by returning the value `j`, after assuming that function parameters $\mathtt{z}_1$ and $\mathtt{z}_2$ equal the tuple elements `i` and `xs` of the RHS parameterized refinement, thereby ensuring that the right set of tuples populate corresponding refinement κ_1.

The HMC Algorithm. The HMC algorithm takes the ML program, generates constraints (Step 1, $\mathsf{Generate}(\cdot)$) and translates them into an imperative program (Step 2,

$$
\begin{array}{lll}
C\downarrow\kappa & \doteq & \{c \in C \mid c \equiv \_ \vdash \_ \prec \kappa_i(\_)\} \\
C\downarrow\bot & \doteq & \{c \in C \mid c \equiv \_ \vdash \_ \prec p\} \\
[\![C]\!] & \doteq & \textbf{let } \kappa_1, \ldots, \kappa_m = \text{Ref. vars. of } C \textbf{ in} \\
 & & [\![0, 0, C\downarrow\bot]\!], \\
 & & [\![1, \text{arity } \kappa_1, C\downarrow\kappa_1]\!] \\
 & & , \ldots, \\
 & & [\![m, \text{arity } \kappa_m, C\downarrow\kappa_m]\!] \\
[\![i, a, \{c_1, \ldots, c_n\}]\!] & \doteq & \mathtt{f}_i(\mathtt{z}_1, \ldots, \mathtt{z}_a)\{[\![c_1]\!] \parallel \ldots \parallel [\![c_n]\!]\} \\
[\![G \vdash \{\mathtt{x}_1 : \beta \mid r_1\} \prec \{\mathtt{x}_2 : \beta \mid r_2\}]\!] & \doteq & [\![G; \{\mathtt{x}_1 : \beta \mid r_1\}]\!]_{get}; \\
 & & [\![\{\mathtt{x}_1 : \beta \mid r_2[\mathtt{x}_1/\mathtt{x}_2]\}]\!]_{set} \\
[\![\{\mathtt{x} : \beta \mid r\}; G]\!]_{get} & \doteq & [\![\{\mathtt{x} : \beta \mid r\}]\!]_{get}; [\![G]\!]_{get} \\
[\![\emptyset]\!]_{get} & \doteq & \mathtt{skip} \\
[\![\{\mathtt{x} : \beta \mid p\}]\!]_{get} & \doteq & \mathtt{x} \leftarrow \mathtt{nondet}(); \\
 & & \mathtt{assume}\ p \\
[\![\{\mathtt{x}_0 : \beta \mid \kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)\}]\!]_{get} & \doteq & \mathtt{x} \leftarrow \mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) \\
[\![\{\mathtt{x} : \beta \mid p\}]\!]_{set} & \doteq & \mathtt{assert}\ p \\
[\![\{\mathtt{x}_0 : \beta \mid \kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)\}]\!]_{set} & \doteq & \mathtt{assume}\ (\wedge_{j=1}^{n} \mathtt{x}_j = \mathtt{z}_j) \\
 & & \mathtt{return}\ x
\end{array}
$$

Fig. 3. Translating Constraints To μC Programs

$[\![\cdot]\!]$). After this, it runs an off-the-shelf abstract interpretation or invariant generation tool on the translated program, and uses the result of this analysis to determine whether the original ML program is safe.

A *safety verifier* V is a procedure that takes an input program and returns Safe or Unsafe. V is *sound* for a language if for each program x in the language, $\mathsf{V}(x) = \mathsf{Safe}$ implies that x is safe. HMC converts a verifier for the (first-order, imperative) language μC to a verifier for the (higher-order, functional) language ML in the following way:

$$\mathsf{HMC}(\mathsf{V}) \doteq \lambda e.\mathsf{V}([\![\mathsf{Generate}(e)]\!])$$

The correctness of HMC follows by combining Theorems 1 and 2.

Theorem 3. *[*HMC *Algorithm] If* V *is a sound verifier for* μC*, then* $\mathsf{HMC}(\mathsf{V})$ *is a sound verifier for* ML*.*

For the translated program shown in Figure 4, the CEGAR-based software model checker ARMC [21] or the abstract interpretation tool INTERPROC [17] finds that the assertion is never violated. From the invariants computed by the tools, we can find solutions to the refinement variables:

$$\kappa_1 \doteq \mathtt{i} \leq \nu < \mathtt{i} + \mathtt{len(xs)} \quad \kappa_2 \doteq 0 \leq \nu < \mathtt{len(a)}$$

```
F0 (){
    a ← nondet();
    xs ← nondet(); assume $(0 \le$ len(xs)$)$;
    j ← F2(a, xs);
    assert $(0 \le$ j $<$ len(a)$)$;
}
```

```
F2 (z1, z2){
    a ← nondet();
    xs ← nondet(); assume $(0 \le$ len(xs)$)$;
    assume (len(a) = len(xs));
    j ← F1(0, xs);
    assume (z1 = a $\wedge$ z2 = xs);
    return j;
}
```

```
F1 (z1, z2){
    i ← nondet();
    xs ← nondet(); assume $(0 \le$ len(xs)$)$;
    xs′ ← nondet(); assume $(0 \le$ len(xs′) = len(xs) $- 1)$;
    j ← nondet(); assume (j = i);
    assume (z1 = i $\wedge$ z2 = xs);
    return j;
[]
    i ← nondet();
    xs ← nondet(); assume $(0 \le$ len(xs)$)$;
    xs′ ← nondet(); assume $(0 \le$ len(xs′) = len(xs) $- 1)$;
    j ← F1(i + 1, xs′);
    assume (z1 = i $\wedge$ z2 = xs);
    return j;
}
```

Fig. 4. Translated Program

which suffice to typecheck the original ML program. Indeed, these predicates are easily shown to satisfy the constraints (c1), (c2), (c3) and (c4).

By exploiting the refinement type structure, HMC reduces verification of programs with advanced language features to verification of simple imperative programs that are amenable to analysis by a wide variety of analysis algorithms and tools.

3 Experiments

To demonstrate the feasibility of HMC, we have instantiated it for OCAML with two off-the-shelf imperative verifiers. We use the liquid types types infrastructure implemented in DSOLVE [22] to generate refinement constraints from OCAML programs. The implementation uses OCAML's implementation of Hindley-Milner type inference to obtain the ML types for each expression, after which the refinement constraints are generated via a syntax-directed pass. Instead of parameterized refinement variables, these constraints have variables with pending substitutions and a separate set of well-formedness (WF) constraints that define the scope of each κ. In a first post-processing step, we use the WF constraints to introduce parametrized refinement variables in place of the pending substitutions. In a second post-processing step, we perform constraint simplifications like constant propagation and resolution.

We use two back-end imperative verifiers to verify the translated programs: ARMC [21], a counterexample-guided software model checker based on predicate abstraction and interpolation-based refinement, and INTERPROC [17], a static analyzer for recursive programs that uses a set of numerical domains such as polyhedra and octagons to compute invariants over numeric variables. In our experiments, we invoked INTERPROC with a polyhedral domain implemented using the Polka library [9]. For each benchmark, the invariants computed by ARMC and INTERPROC could be used to synthesize refinement types for the original source ML program.

Results. Table 1 shows the results of running the two verifiers on suite of small OCAML examples. In addition to the running time, we report the number of predicates discovered by ARMC. The rows with prefix `na_` are a subset of the array manipulating programs from [22], where the safety objective is to prove array accesses are within bounds. The other rows correspond to the benchmark suite used to evaluate the DEPCEGAR verifier [24], where each program contains a set of assertions designed to enforce safety. For each program we created a buggy version that contains a manually inserted safety violation. We observe that despite our blackbox treatment of ARMC and INTERPROC we obtain running times that are competitive with DEPCEGAR on most of the examples (DEPCEGAR uses a customized procedure for unfolding constraints and creating interpolation queries that yield refinement types).

Refinements Discovered. Most of the atomic predicates discovered by ARMC and INTERPROC fall into the two-variables-per-inequality fragment. However, the example MASK from Section 2 required a predicate that refers to three variables, and thus could not be verified using a simpler domain (*e.g.,* octagons). In this case, INTERPROC determined the following relationship between the input and output variables of $\mathtt{F}_1$ and $\mathtt{F}_2$ (after existentially eliminating local variables):

$$\begin{aligned}&\mathtt{F}_1 :: \mathtt{z}_1 \leq \nu \leq \mathtt{z}_1 + \mathtt{len}(\mathtt{z}_2) - 1\\&\mathtt{F}_2 :: 0 \leq \nu \leq \mathtt{len}(\mathtt{z}_1) - 1 \wedge \mathtt{len}(\mathtt{z}_1) = \mathtt{len}(\mathtt{z}_2)\end{aligned}$$

These invariants are sufficient to show that the assertion in $\mathtt{F}_0$ always holds.

4 Discussions: Completeness

Since the safety verification problem for higher-order programs is undecidable, the sound HMC cannot also be complete in general. Even in the finite-state case, in which each base type has a finite domain (*e.g.,* Booleans), completeness depends on the generation of refinement constraints.

For example, in our current formulation, we employ a *context insensitive* form of constraint generation where we use the same template for a (monomorphic) function at different call points. It has been shown through practical benchmarks that since the types themselves capture relations between the inputs and outputs, the context-insensitive constraint generation suffices to prove a variety of complex programs safe [2, 12, 22]. Nevertheless, there can be a loss of information. Consider the program

Table 1. Experimental Results: ARMC (s) denotes the time taken (in seconds) by ARMC to analyze the translated program in its correct and buggy version; DNF indicates that the tool did not finish on the benchmark. **ARMC Preds.** denotes the number of predicates iteratively found by ARMC in order to verify the safe benchmarks. **INTERPROC (s)** denotes the time (in seconds) taken by INTERPROC to analyze the translated program in its correct and buggy version; *_ means INTERPROC was not precise enough to prove all assertions, *i.e.,* raised false alarms.

Program	**ARMC (s)**	**ARMC Preds.**	**INTERPROC (s)**
	correct / buggy		**correct / buggy**
`na_dotprod-m`	0.04 / 0.04	2	0.55 / 0.56
`na_arraymax-m`	0.32 / 0.05	6	0.40 / 0.23
`na_bcopy-m`	0.09 / 5.94	3	0.33 / 0.38
`na_bsearch-m`	0.91 / 0.10	11	9.73 / 2.76
`na_insertsort`	0.03 / 0.03	0	40.11 / 7.38
`na_heapsort`	DNF / DNF		*27.99 / 28.26
`boolflip`	0.23 / 0.19	5	0.05 / 0.09
`lock`	0.03 / 0.03	0	*0.19 / 0.23
`mult-cps-m`	0.03 / 0.03	0	0.08 / 0.12
`mult-all-m`	0.03 / 0.03	1	0.13 / 0.07
`mult`	0.03 / 0.03	2	0.08 / 0.06
`sum-all-m`	0.03 / 0.03	1	0.10 / 0.08
`sum`	0.03 / 0.03	2	0.02 / 0.02
`sum-acm-m`	0.04 / 0.03	2	*0.10 / 0.13

```
let check f x y = assert (f x = y) in
check (fun a -> a) false false ;
check (fun a -> not a) false true
```

For `check`, our constraint generation produces the template

$$(\{\texttt{x:bool} \mid \kappa_1\} \to \{\kappa_2\}) \to \{\kappa_3\} \to \{\kappa_4\} \to \texttt{unit}$$

which is too weak to show safety as the template "merges" the two call sites for `check`. However, we can regain sensitivity via the following *refined intersection type* [5, 6, 15, 18], for `check`:

$$\bigwedge \begin{array}{l} (\texttt{x} : \texttt{bool} \to \{\nu = \texttt{x}\}) \to \{\neg\nu\} \to \{\neg\nu\} \to \texttt{unit} \\ (\texttt{x} : \texttt{bool} \to \{\nu = \neg\texttt{x}\}) \to \{\neg\nu\} \to \{\nu\} \to \texttt{unit} \end{array}$$

It is important to note that our translation works holds for *any* set of implication constraints (Theorem 2). Thus, one can improve the precision of HMC, by using a more expressive refinement type system to generate the constraints, without having to modify the back-end invariant generation. For example, to recover completeness in the finite-state case, we can use intersection type system of [15] that uses a finite number of "contexts" to generate the implication constraints, after which a finite-state checker *e.g.,* BEBOP [1] would suffice to give a complete verification procedure. (We omit this in our current implementation as there can be a super-exponential number of implication constraints, and the relational refinements were sufficient for our experiments.)

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3. ACM, New York (2002)
2. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: CSF (2008)
3. 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: PLDI, pp. 196–207 (2003)
4. Cui, S., Donnelly, K., Xi, H.: ATS: A language that combines programming with theorem proving. In: FroCos (2005)
5. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2007)
6. Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI (1991)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004. ACM, New York (2004)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
9. 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)
10. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surveys (2009)
11. Jhala, R., Majumdar, R., Rybalchenko, A.: Refinement type inference via abstract interpretation. CoRR, abs/1004.2884 (2010)
12. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
13. Knowles, K., Flanagan, C.: Type reconstruction for general refinement types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 505–519. Springer, Heidelberg (2007)
14. Knowles, K.W., Flanagan, C.: Hybrid type checking. ACM TOPLAS 32(2) (2010)
15. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL (2009)
16. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to modal μ-calculus model checking of recursion schemes. In: LICS (2009)
17. Lalire, G., Argoud, M., Jeannet, B.: Interproc, http://bit.ly/8Y310m
18. Naik, M., Palsberg, J.: A type system equivalent to a model checker. ACM Trans. Program. Lang. Syst. 30(5) (2008)
19. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS (2006)
20. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: IFIP TCS, pp. 437–450 (2004)
21. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL (2007)
22. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
23. Rondon, P., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL (2010)
24. Terauchi, T.: Dependent types from counterexamples. In: POPL. ACM, New York (2010)
25. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP (2009)
26. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL (1999)

A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations*

Ajith K. John[1] and Supratik Chakraborty[2]

[1] Homi Bhabha National Institute, BARC, Mumbai, India
[2] Dept. of Computer Sc. & Engg., IIT Bombay, India

Abstract. We present a layered bit-blasting-free algorithm for existentially quantifying variables from conjunctions of linear modular (bit-vector) equations (LMEs) and disequations (LMDs). We then extend our algorithm to work with arbitrary Boolean combinations of LMEs and LMDs using two approaches – one based on decision diagrams and the other based on SMT solving. Our experiments establish conclusively that our technique significantly outperforms alternative techniques for eliminating quantifiers from systems of LMEs and LMDs in practice.

1 Introduction

Quantifier elimination (henceforth called QE) is the process of converting a formula containing existential and/or universal quantifiers in a suitable logic into a semantically equivalent quantifier-free formula. Formally, let A be a quantifier-free formula over a set X of free variables in a first-order theory $\mathcal{T}$. Consider the quantified formula $Q_1y_1\, Q_2y_2\, \ldots Q_my_m. A$, where $Y = \{y_1, \ldots y_m\}$ is a subset of X, and $Q_i \in \{\exists, \forall\}$ for $i \in \{1, \ldots m\}$. QE computes a quantifier-free formula A' with free variables in $X \setminus Y$ such that $A' \equiv_{\mathcal{T}} Q_1y_1\, Q_2y_2\, \ldots Q_my_m. A$, where $\equiv_{\mathcal{T}}$ denotes semantic equivalence in theory $\mathcal{T}$. This has a number of important applications in formal verification and program analysis. Example applications include computing abstractions of symbolic transition relations, computing strongest postconditions of program statements and computing interpolants in CEGAR frameworks. Since $\forall y. \varphi \equiv \neg\exists y. \neg\varphi$ in all first-order theories, it suffices to focus on algorithms for eliminating existential quantifiers. This paper presents one such algorithm for a fragment of the theory of bit-vectors that we have found useful in verification of word-level RTL designs.

Currently, the most popular technique for eliminating quantifiers from bit-vector formulae involves *blasting* bit-vectors into individual bits (Boolean variables), followed by quantification of the blasted Boolean variables. This approach has some undesirable features. For example, blasting involves a bitwidth-dependent blow-up in the size of the problem. This can present scaling problems in the usage of Boolean reasoning tools (e.g. BDD based tools), especially when reasoning about wide words. Similarly, given an instance of the QE problem,

* This work was supported by a research grant from Board of Research in Nuclear Sciences, India.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 486–503, 2011.

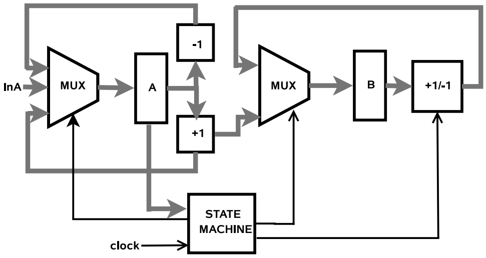
blasting variables that are quantified may transitively require blasting other variables (that are not quantified) as well. This can cause the quantifier-eliminated formula to appear like a propositional formula on blasted bits, instead of being a bit-vector formula. Since reasoning at the level of bit-vectors is often more efficient in practice than reasoning at the level of bits, QE using bit-blasting might not be the best option if the quantifier-eliminated formula is intended to be used in further bit-vector level reasoning. This motivates us to ask if we can efficiently eliminate quantifiers in the theory of bit-vectors without resorting to bit-blasting (or model enumeration) in practice. Ideally, we would like to obtain such a QE procedure for the entire theory of bit-vectors. Unfortunately, we do not have this yet. We therefore focus on a fragment of the theory, namely Boolean combinations of equations and disequations of bit-vectors, that we have found useful in word-level verification of RTL designs.

Since bit-vector arithmetic is the same as modular arithmetic on integers, our algorithm can also be viewed as one for existentially quantifying variables from a Boolean combination of linear modular integer equations and disequations. A Linear Modular Equation (LME) is an equation of the form $c_1 \cdot x_1 + \cdots + c_n \cdot x_n = c_0 \pmod{2^p}$ where p is a positive integer constant, $x_1, \ldots, x_n$ are p-bit non-negative integer variables, and $c_0, \ldots, c_n$ are integer constants in $\{0, \ldots, 2^p - 1\}$. Similarly, a Linear Modular Disequation (LMD) is a disequation of the form $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \neq c_0 \pmod{2^p}$. Conventionally, 2^p is called the modulus of the LME or LMD. For notational convenience, we will henceforth use "LMC" to refer to a Linear Modular Constraint, i.e. an LME or LMD. Since every variable in an LMC $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$, where $\bowtie \in \{=, \neq\}$, represents a p-bit integer, it follows that a set of LMCs sharing a variable must have the same modulus. However, there are applications where we need to consider Boolean combinations of LMCs that do not share any variable, and have different moduli. In such cases, we propose to appropriately shift the moduli of LMCs, so that all LMCs have the same modulus. This can always be done since the LMCs $\lambda_1 \equiv c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0 \pmod{2^p}$ and $\lambda_2 \equiv 2^q \cdot c_1 \cdot x'_1 + \cdots + 2^q \cdot c_n \cdot x'_n \bowtie 2^q \cdot c_0 \pmod{2^{p+q}}$ are related in the following way: every solution of λ_1 can be bit-extended to give a solution of λ_2, and every solution of λ_2 can be bit-truncated to give a solution of λ_1. Hence, using λ_2 in place of λ_1 suffices for checking satisfiability and also for finding solutions of Boolean combinations of LMCs. In the remainder of this paper, we will assume without loss of generality that whenever we consider a set of LMCs, all of them have the same modulus.

Our primary motivation for studying QE of LMCs comes from bounded model checking (BMC) of word-level RTL designs. As an example, consider the synchronous circuit shown in Fig. 1, with the relevant part of its functionality described in VHDL in the right half of the figure. The thick shaded arrows and the thin solid arrows represent 8-bit words and 1-bit lines respectively. The circuit comprises a controller and two 8-bit registers, A and B. The controller switches between two states, 0 and 1, depending on the value of A. In state 0, A works as a down-counter until it reaches 0x00[1], in which case A loads itself with an

[1] We use the 0x prefix to denote hexadecimal values.

input value from InA and the controller switches to state 1. In state 1, A works as an up-counter until it reaches 0xff, in which case it loads the value from InA and the controller switches to state 0. Register B is always loaded with the value of $A + 1$ except when A has the value 0xff. If this happens in state 0 (down-counting state), B decrements its previously stored value; otherwise, B increments its previously stored value.



```
   ....
if (clock'event and clock = '1') then
 case state is
  when '0' =>
   if (A = x"00") A <= InA; state <= '1';
   else  A <= A-1; state <= '0'; end if;
   if  (A = x"ff") B <= B-1;
   else B <= A+1; end if;
  when others =>
   if  (A = x"ff") A <= InA; state <= '0';
                   B <= B+1;
   else A <= A+1; state <= '1'; B <= A+1;
   end if;
 end case;
end if;
   ....
```

Fig. 1. An Example Circuit

A word-level transition relation, R, for this circuit can be obtained by conjoining the following three equality relations, where all operations on A and B are assumed to be modulo 2^8.

$$\begin{aligned} \mathsf{state}' &= \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0x00},\ 1,\ 0),\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ 0,\ 1)) \\ \mathsf{A}' &= \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0x00},\ \mathsf{InA},\ \mathsf{A} - 1),\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{InA},\ \mathsf{A} + 1)) \\ \mathsf{B}' &= \mathsf{ite}(\mathsf{state} = 0,\ \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{B} - 1,\ \mathsf{A} + 1), \mathsf{ite}(\mathsf{A} = \mathsf{0xff},\ \mathsf{B} + 1,\ \mathsf{A} + 1)) \end{aligned}$$

In the above relations, state', A' and B' refer to values of state, A and B after the next rising edge of the clock. Note also that A, A', B and B' are 8−bit wide bit-vector variables and state and state' are propositional variables. Since R is a conjunction of equalities involving ite, and since $a = ite(b, c, d)$ represents $(b \wedge (a = c)) \vee (\neg b \wedge (a = d))$, R is essentially a Boolean combination of LMCs.

The above circuit has the property that once started in state 0, it never reaches state 1 with 0x00 in register B. Suppose we wish to use BMC to prove that this property holds for the first N cycles of operation. This can be done by unrolling the transition relation N times, conjoining the unrolled relation with the negation of the property, and then checking for satisfiability of the resulting constraint using an SMT solver that can reason about bit-vectors. Since R contains all variables (in unprimed and primed versions) that appear in the RTL description, unrolling R a large number of times gives a constraint with a large number of variables. This problem is particularly acute for circuits with a large number of internal state variables. While the number of variables in a constraint is not the only factor that affects the performance of an SMT solver, for large enough values of N, the increased variable count indeed has an adverse effect on the performance of the solver, as indicated by our experiments.

In order to alleviate the above problem, one can use an abstract transition relation R' that relates only a chosen subset of variables relevant to the property being checked, while abstracting the relation between the other variables. In our example, we can compute such an R' by existentially quantifying the bit-vector variables A and A' from R. This gives R' as:

$$\begin{aligned}((\mathsf{state}' = 1) \wedge (\mathsf{B}' = \mathsf{0x01})) \vee \\ ((\mathsf{state}' = 0) \wedge (\mathsf{B}' = \mathsf{ite}(\mathsf{state} = 0, \mathsf{B} - 1, \mathsf{B} + 1))) \vee \\ ((\mathsf{state}' = \mathsf{state}) \wedge (\mathsf{B}' \neq \mathsf{0x00}) \wedge (\mathsf{B}' \neq \mathsf{0x01}))\end{aligned}$$

On careful examination, it can be seen that if we unroll R' (instead of R) during BMC, we can still prove that the circuit never reaches state 1 with $\mathsf{0x00}$ in B, if it starts in state 0. Since R' contains fewer variables than R, the constraint obtained by unrolling R' has fewer variables. In general, this can lead to significantly better performance of the back-end SMT solver, as demonstrated in our experiments.

The example presented above is representative of a more general scenario. In general, Boolean combinations of LMCs arise when building transition relations of RTL designs and/or embedded systems containing conditional statements that check for equalities of words/registers. Building an abstract transition relation in such cases requires existentially quantifying variables from Boolean combinations of LMCs. Obtaining the abstract transition relation at the word-level is particularly appealing since it allows word-level reasoning to be applied to the abstraction. This motivates us to study the problem of eliminating quantifiers from Boolean combinations of LMCs without resorting to bit-blasting (or model enumeration) in practice.

Contributions. There are two primary contributions of this paper. First, we describe a bit-blasting-free algorithm for eliminating quantifiers from conjunctions of LMCs. The algorithm is based on a layered approach, i.e., the cheaper layers are invoked first and more expensive layers are called only when required. Later, we extend this to an algorithm for eliminating quantifiers from Boolean combinations of LMCs. While our algorithm uses a final layer of model enumeration for the sake of theoretical completeness, extensive experiments indicate that we never need to invoke this layer in practice. Our second contribution is an extensive set of carefully conducted experiments that not only demonstrate the effectiveness of our approach over alternative techniques, but also allows us to identify criteria for choosing the right QE technique for a given problem instance.

Related Work. Several interesting approaches have been proposed earlier for reasoning about LMEs (e.g., [6,7]). Although our study indicates that non-trivial counts of LMDs appear in constraints arising from real verification problems, LMDs have traditionally received relatively less attention. Jain et al [7] showed that the satisfiability problem for a conjunction of LMCs is NP-hard. However, their work subsequently focused on systems of LMEs and Linear Diophantine Equations and Disequations, and discussed algorithms to compute interpolants in such systems. Bit-blasting [3] followed by bit-level QE is arguably the dominant technique used in practice for eliminating quantifiers from bit-vector constraints. As discussed earlier, this approach, though simple, destroys the word-level structure of the problem and does not scale well for LMCs with large modulus. Since LMEs and LMDs can be expressed as formulae in Presburger Arithmetic (PA) [3], QE techniques for PA (e.g. those in [5]) can also be used to eliminate quantifiers from Boolean combinations of LMCs. Similarly, automata-theoretic approaches for eliminating quantifiers from PA formulae [8] can also be used. However, converting the results obtained as PA formulae back to Boolean combinations of LMCs is often difficult. Also, empirical studies have shown that using PA techniques to eliminate quantifiers from Boolean combinations of LMCs often blows up in practice [3]. The work that is most closely related to our work is that of Ganesh and Dill [6]. The authors of [6] present a technique for reducing LMEs to a solved form by selecting variables in a specific order. While this does not directly give us a technique to eliminate a user-specified variable from a conjunction of LMEs, their work can be extended to achieve this. More importantly, [6] does not consider the problem of eliminating variables in constraints involving LMDs. This problem is addressed in our work.

2 Quantifier Elimination for a Conjunction of LMCs

The problem we wish to solve in this section can be formally stated as follows. Given a set of LMCs over variables $x_1, \ldots, x_n$, let A denote the conjunction of the LMCs. Without loss of generality, we wish to compute $A' \equiv \exists x_1 \cdots \exists x_t. A$, where A' is a Boolean combination of LMCs. For reasons of succinctness, we also require that A' contains no ground terms other than integer constants, and no ground (sub-)formulas other than true and false. This problem is easily seen to be NP-hard. This follows from the facts: (i) the satisfiability problem for a conjunction of LMCs is NP-hard, even when all moduli are a priori fixed to 4 (see [7]), and (ii) a conjunction of LMCs A over $x_1, \ldots, x_n$ is satisfiable iff an algorithm for computing $A' \equiv \exists x_1 \cdots \exists x_n. A$ returns true (due to the succinctness requirement of A').

Since an algorithm for computing $\exists x_i. A$ can be used in an iterative way to compute $\exists x_1 \cdots \exists x_t. A$, we will initially focus on the (seemingly simpler) problem of computing $\exists x_i. A$ in the subsequent discussion. All LMCs considered in the remainder of this section have modulus 2^p, for some positive integer p, unless stated otherwise. For notational clarity, we will therefore omit mentioning " (mod 2^p)" with LMCs in the following discussion. We have skipped the proofs of most lemmas and details of some procedures due to lack of space. For the interested reader, these details can be found in [13].

Lemma 1. *An LMC $c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie c_0$ can be equivalently expressed as $2^{k_1} \cdot x_1 \bowtie t_1$, where $\bowtie \in \{=, \neq\}$, t_1 is a term free of x_1, and k_1 is an integer such that $0 \leq k_1 \leq p-1$.*

Example: All LMCs in this example have modulus 8. Consider the LME $6x + 4y = 0$. Rearranging the terms modulo 8, we get $3 \cdot 2^1 x = 4y$. Multiplying by 3 (multiplicative inverse of 3 modulo 8) and simplifying gives, $2^1 x = 4y$.

Henceforth whenever we express an LMC as $2^{k_i} \cdot x_1 \bowtie t_i \pmod{2^p}$ where $\bowtie \in \{=, \neq\}$, it will be implicitly understood that "t_i is a term free of x_1 and k_i is an integer such that $0 \leq k_i \leq p-1$". Lemma 1 ensures that there is no loss of generality in doing this.

Lemma 2. $\exists x_1 . (2^{k_1} \cdot x_1 = t_1) \pmod{2^p} \equiv (2^{p-k_1} \cdot t_1 = 0) \pmod{2^p}$

Example: All LMCs in this example have modulus 8. $\exists y . (2^1 . y = 5.x + 2) \equiv (2^{3-1}.(5.x+2) = 0) \equiv (4.x = 0)$

Lemma 3. *Let A be the conjunction of m LMEs of the form $2^{k_i} \cdot x_1 = t_i$, where i ranges from 1 through m. Then $\exists x_1 . A$ can be equivalently expressed as a conjunction of LMEs each of which is free of x_1.*

Example: All LMCs in this example have modulus 8. Consider the problem of computing $\exists y . ((2^1 y = 5x + 2) \wedge (2^2 y = 5x + 6z) \wedge (2^1 y = 2x + 4))$. This can be equivalently expressed as $\exists y . ((2y = 5x + 2) \wedge (2 \cdot (5x+2) = 5x + 6z) \wedge (5x + 2 = 2x + 4))$. Simplifying modulo 8, we get $\exists y . ((2y = 5x + 2)) \wedge (5x + 2z = 4) \wedge (3x = 2)$. Using Lemma 2, we obtain the final result as $(4x = 0) \wedge (5x + 2z = 4) \wedge (3x = 2)$.

Lemma 4. *Let A be the conjunction of r LMCs of the form $2^{k_i} \cdot x_1 \bowtie t_i$, where $\bowtie \in \{=, \neq\}$ and i ranges from 1 through r. Let $2^{k_1} \cdot x_1 = t_1$ be the LME with the minimum k_i among all LMEs in A. Then $\exists x_1 . A \equiv \psi_1 \wedge \exists x_1 . \psi_2$, where ψ_1 is a conjunction of LMCs independent of x_1, and ψ_2 is a conjunction of LMDs and at most one LME i.e., $2^{k_1} \cdot x_1 = t_1$. In addition, ψ_2 contains only those LMDs from A in which the coefficient of x_1 is of the form 2^{k_i}, where $k_i < k_1$.*

Example: All LMCs in this example have modulus 8. Consider the problem of computing $\exists y . ((2^1 y = 5x + 2) \wedge (2^2 y = 5x + 6z) \wedge (2^1 y \neq 2x + 4) \wedge (2^0 y \neq 6x + 7z))$. This can be equivalently expressed as $\exists y . ((2y = 5x+2) \wedge (2 \cdot (5x+2) = 5x + 6z) \wedge (5x + 2 \neq 2x + 4) \wedge (y \neq 6x + 7z))$. Simplifying modulo 8, we get $(5x + 2z = 4) \wedge (3x \neq 2) \wedge \exists y . ((2y = 5x + 2) \wedge (y \neq 6x + 7z))$. Note that ψ_1 and ψ_2 here are $(5x + 2z = 4) \wedge (3x \neq 2)$ and $(2y = 5x + 2) \wedge (y \neq 6x + 7z)$ respectively.

For the remainder of the paper, we adopt the convention that algorithms for eliminating a single variable will have names starting with "*QE1_*",while those for eliminating multiple variables will have names starting with "*QE_*".

Lemmas 1 through 4 yield two simple algorithms: (a) *QE1_LME* that takes an LME and a variable to quantify out, and returns the equivalent quantifier-free formula (based on Lemma 2), and (b)*QE1_Layer1* that takes a conjunction of LMCs and a variable x_1 to quantify out and returns the equivalent conjunction of ψ_1 and $\exists x_1 . \psi_2$ (as given by Lemma 4). As we will soon see, *QE1_Layer1* forms the core of the first layer of our layered QE algorithm.

If the k_i's of all LMDs in A are such that $k_1 \leq k_i$, then $\exists x_1 . \psi_2$ reduces to $\exists x_1 . (2^{k_1} \cdot x_1 = t_1)$. According to Lemma 2, this is equivalent to $2^{p-k_1} \cdot t_1 = 0$. Hence, in this case, algorithms *QE1_Layer1* and *QE1_LME* suffice to compute $\exists x_1 . A$. In general, however, ψ_2 may contain LMDs containing x_1 that require further processing before x_1 is eliminated. We describe techniques for doing this in the following subsections.

2.1 Dropping Unconstraining LMDs

We now consider the problem of simplifying $\exists x_1 . \psi_2$ obtained above, when $\exists x_1 . \psi_2$ contains LMDs. Let $\psi_2 \equiv \xi \wedge \lambda$, where λ is an LMD and ξ is a conjunction of LMCs. We say that λ is *unconstraining* in $\exists x_1 . \psi_2$ iff $\exists x_1 . (\xi \wedge \lambda) \equiv \exists x_1 . \xi$. Unconstraining LMDs can simply be dropped from $\exists x_1 . \psi_2$, thereby simplifying the task of QE. Unfortunately, identifying all unconstraining LMDs from ψ_2 involves invoking an SMT solver for quantified bit-vector formulas. In this subsection, we present a sound technique for identifying a subset of unconstraining LMDs in $\exists x_1 . \psi_2$. Our approach exploits the fact that an LMD is satisfied even if a single bit in the left-hand side of the LMD differs from the corresponding bit in the right-hand side. We therefore propose to identify LMDs in $\exists x_1 . \psi_2$ that can be satisfied by selectively assigning values to specific bits of x_1, without causing any other LME or LMD in $\exists x_1 . \psi_2$ to be violated. Since x_1 is existentially quantified, these LMDs are effectively unconstraining in $\exists x_1 . \psi_2$. We illustrate this idea below through an example.

Consider $\exists x. (\xi \wedge \lambda)$, where $\xi \equiv (4x = 6y + 2z) \wedge (2x \neq 2y + 4z) \wedge (2x \neq 6y + 6z)$ and $\lambda \equiv (x \neq y + z)$, and all LMCs have modulus 8. For clarity of exposition, we use the notation $x[i]$ to denote the i^{th} bit of a bit-vector x, and adopt the convention that $x[0]$ denotes the least significant bit of x. We claim that any solution of ξ can be "engineered" by possibly modifying the value of $x[2]$ to give a solution of $\xi \wedge \lambda$, and vice versa. In order to see why this is true, note that the LME $4x = 6y + 2z$ constrains only $x[0]$ and the LMDs $(2x \neq 2y + 4z)$, $(2x \neq 6y + 6z)$ constrain only $x[0]$ and $x[1]$. Therefore, the value of $x[2]$ does not affect satisfaction of ξ. Any solution of ξ can therefore be engineered to become a solution of $\xi \wedge \lambda$ by ensuring that $x[2]$ differs from the most-significant bit of $y + z$. Hence, $\exists x. (\xi) \Rightarrow \exists x. (\xi \wedge \lambda)$. The converse, i.e. $\exists x. (\xi \wedge \lambda) \Rightarrow \exists x. (\xi)$ obviously holds. Hence in this example, $(x \neq y + z)$ is an unconstraining LMD in $\exists x. (\xi \wedge \lambda)$.

```
DropLMDSimple(E, D, $x_1$)
 core ← E;
while(core ≠ E ∪ D)
  if (isExt(core, E ∪ D, $x_1$))
    return core;
  else
    d ← getLstCnstr(D\core);
    core ← core ∪ d;
return core;
```

```
DropImpliedLMD(E, D, $x_1$)
 while(true)
  impl ← NULL;
  for each LMD d ∈ D
    if (E ∪ (D\d) ⇒d)
      impl ← d; break;
  if (impl = NULL)
    break;
  D ← D\impl;
return E∪D;
```

Fig. 2. Algorithms to drop unconstraining LMDs

The above idea leads to a simple algorithm, called *DropLMDSimple*, shown in Fig. 2. This algorithm takes as inputs a set of LMEs E, a set of LMDs D, and a variable x_1 to be quantified from the conjunction of all LMCs in $E \cup D$.

Algorithm *DropLMDSimple* returns a subset of LMCs in $E \cup D$ such that the result of quantifying x_1 from the conjunction of LMCs in this subset is equivalent to the result of quantifying x_1 from the conjunction of LMCs in $E \cup D$.

Algorithm *DropLMDSimple* computes the desired subset in a variable *core* that is initialized to E. Subsequently, it determines if any solution to the conjunction of LMCs in *core* can be engineered by modifying specific bits of x_1 to give a solution to the conjunction of LMCs in $E \cup D$. This is achieved by invoking a function *isExt*. If such an engineering is indeed possible, then all LMDs not in *core* are unconstraining, and algorithm *DropLMDSimple* returns *core*. Otherwise we use the function *getLstCnstr* to identify the LMDs in $D \setminus core$ whose truth depends on the least number of bits of x_1. Intuitively, these LMDs are the most difficult ones to satisfy among the LMDs in $D \setminus core$. These LMDs are then included in *core* and the process repeats. Clearly, algorithm *DropLMDSimple* terminates since *core* cannot have more LMCs than those in $E \cup D$.

Since each LMD is of the form $2^{k_i} \cdot x_1 \neq t_i$, the LMD with the largest k_i is the one whose truth depends on the least number of bits of x_1. This gives a simple implementation of function *getLstCnstr*. One possible implementation of *isExt* is through the use of an SMT solver that checks if one quantified formula implies another quantified formula. However, this is inefficient in general. Instead, we propose an implementation of *isExt* based on the following Lemma.

Lemma 5. *Let k_{core} be the smallest among the k_i's of all LMCs $2^{k_i} \cdot x \bowtie t_i$ in core. Let $D \setminus core$ be expressed as $\{(2^{k_1} \cdot x \neq t_1), \ldots, (2^{k_n} \cdot x \neq t_n)\}$. If $2^{k_{core}} - \sum_{i=1}^{n} 2^{k_i} \geq 1$, any solution to the conjunction of LMCs in core can be engineered to give a solution to the conjunction of LMCs in $E \cup D$.*

We give a sketch of the proof of Lemma 5. Let C_1 be the conjunction of LMCs in *core* and C_2 be the conjunction of LMDs outside *core*. Let π be any solution of C_1. Clearly π constrains only bits $x[0]$ through $x[p - k_{core} - 1]$ of x. Hence there are $2^{k_{core}}$ ways in which bits $x[p - k_{core}]$ through $x[p-1]$ can be assigned values without affecting the truth of any LMC in C_1. It can be shown that $\eta = 2^{k_{core}} - \sum_{i=1}^{n} 2^{k_i}$ under-approximates the number of ways in which bits $x[p - k_{core}]$ through $x[p-1]$ can be assigned values in the solution π of C_1 such that we obtain a solution of C_2 while still satisfying C_1. Therefore if $\eta \geq 1$, there exists at least one assignment of values to bits $x[p - k_{core}]$ through $x[p-1]$ such that π can be engineered to be a solution of the conjunction of LMCs in $E \cup D$.

DropLMDSimple may not be able to identify all the unconstraining LMDs in $\exists x_1 . \psi_2$. For example, consider the problem $\exists x.\, ((2x = y) \wedge (x \neq 2y) \wedge (x \neq y))$, where all LMCs have modulus 8. Here *core* is $\{2x = y\}$, $k_{core} = 1$, $k_1 = k_2 = 0$. Therefore, $\eta = 0$ and *DropLMDSimple* concludes that it is not possible to engineer a solution of $(2x = y)$ to give a solution of $(2x = y) \wedge (x \neq 2y) \wedge (x \neq y)$ by assigning values to specific bits of x. Hence, *DropLMDSimple* cannot identify any LMD to drop. However, it can be seen that $(2x = y) \wedge (x \neq 2y) \Rightarrow (x \neq y)$. Hence $\exists x.\, ((2x = y) \wedge (x \neq 2y) \wedge (x \neq y)) \equiv \exists x.\, ((2x = y) \wedge (x \neq 2y))$. Once $x \neq y$ is dropped, *DropLMDSimple* can further simplify $\exists x.\, ((2x = y) \wedge (x \neq 2y))$

to $\exists x. (2x = y)$. Based on this idea, we present an algorithm to drop implied LMDs called *DropImpliedLMD* (see Fig. 2). The notation used in this algorithm is the same as that used in algorithm *DropLMDSimple*. The implication check in *DropImpliedLMD* requires invoking an SMT solver, in general.

We now present an algorithm *QE1_Layer3* which drops LMDs from $\exists x_1. \psi_2$ using *DropLMDSimple* and *DropImpliedLMD*. Given $\exists x_1. \psi_2$, *QE1_Layer3* initially invokes *DropLMDSimple* to drop unconstraining LMDs. If one or more LMDs remain, *DropImpliedLMD* is invoked to identify the implied LMDs and drop them. If there exist LMDs in the output of *DropImpliedLMD*, we invoke *DropLMDSimple* once again. Thus finally, we are left with a conjunction of LMCs ψ_2' with possibly fewer LMDs vis-a-vis to ψ_2, while guaranteeing that $\exists x_1. \psi_2 \equiv \exists x_1. \psi_2'$.

The algorithms *QE1_Layer1*, *DropLMDSimple* and *QE1_Layer3* form the first three layers of our layered QE algorithm. We present in Fig. 3 a procedure *QE1_Layers1To3* that tries to compute $\exists x_1. A$ using these layers. Initially *QE1_Layer1* is called to reduce $\exists x_1. A$ to $\psi_1 \wedge \exists x_1. \psi_2$. If ψ_2 is free of LMDs, *QE1_LME* is called to compute $\exists x_1. \psi_2$; hence $\exists x_1. A$ gets computed by the first layer itself. If ψ_2 is not free of LMDs, *QE1_Layers1To3* initially calls *DropLMDSimple* and later on *QE1_Layer3* (if required) to drop the LMDs. If all the LMDs in $\exists x_1. \psi_2$ are dropped by *DropLMDSimple* (or *QE1_Layer3*), $\exists x_1. A$ gets computed in the second (or third) layer. Otherwise, *QE1_Layers1To3* returns $\psi_1 \wedge \exists x_1. \psi_2'$ such that $\psi_1 \wedge \exists x_1. \psi_2' \equiv \exists x_1. A$. The techniques required to compute such (harder) instances of $\exists x_1. A$ are presented in the following subsection.

2.2 Splitting and Model Enumeration

Let us have a closer look at those instances of $\exists x_1. A$ that cannot be computed by *QE1_Layers1To3*. The difficulty in QE in such cases arises from the fact that there are some bits x_1 constrained by the LMDs but not by any LME. For example, consider the problem of computing $\exists x. ((2x = a) \wedge (x \neq b) \wedge (x \neq c))$ where all the LMCs have modulus 8. The LME $(2x = a)$ constrains only bits $x[1]$ and $x[0]$ of x, whereas the LMDs constrain bits $x[0], x[1]$ and $x[2]$. It can be observed that in this example, QE cannot be performed by *QE1_Layers1To3*. We now describe two techniques to compute such instances of $\exists x_1. A$, namely *Splitting* and *Model Enumeration*[2].

Splitting is based on the observation that each LMD $2^{k_i} \cdot x_1 \neq t_i$ can be equivalently expressed as the disjunction of two constraints : an LMD $(2^k \cdot x_1 \neq 2^{k-k_i} \cdot t_i)$ and a conjunction $((2^k \cdot x_1 = 2^{k-k_i} \cdot t_i) \wedge (2^{k_i} \cdot x_1 \neq t_i))$ where $k_i < k$. Repeated application of this converts A into $A_1 \vee \ldots \vee A_n$ where each A_i is a conjunction of LMCs. Thus, $\exists x_1. A$ is equivalent to $\exists x_1. A_1 \vee \ldots \vee \exists x_1. A_n$ where each subproblem $\exists x_1. A_i$ is potentially "simpler" to solve than the original problem $\exists x_1. A$. For example, in the previous problem, the LMD $(x \neq b)$ can

[2] For all the benchmarks we have experimented with, *Splitting* and *Model Enumeration* were never required to eliminate quantifiers. Hence they are only briefly described here.

be split into $(2x \neq 2b) \vee ((2x = 2b) \wedge (x \neq b))$ converting the problem into $\exists x. ((2x = a) \wedge (2x \neq 2b) \wedge (x \neq c)) \vee \exists x. ((2x = a) \wedge (2x = 2b) \wedge (x \neq b) \wedge (x \neq c))$.

Model Enumeration is based on the observation that $\exists x_1. A$ can be equivalently expressed as $A|_{x_1 \leftarrow 0} \vee \ldots \vee A|_{x_1 \leftarrow 2^p - 1}$ (where $A|_{x_1 \leftarrow i}$ denotes A with x_1 replaced by constant i).

We call (i) the procedure that makes use of *Splitting* and *Model Enumeration* to compute $\exists x_1. A$ as *QE1_Layer4* and (ii) the procedure that makes use of *QE1_Layer4* to compute $\exists x_1 \cdots \exists x_t. A$ as *QE_Layer4*.

We present in Fig. 3 the algorithm *QE_LMC* that computes $\exists x_1 \cdots \exists x_t. A$ using *QE1_Layers1To3* and *QE_Layer4*. *QE_LMC* initially tries to eliminate the quantified variables $x_1, \ldots, x_t$ one by one by invoking *QE1_Layers1To3*. Variables that cannot be eliminated by *QE1_Layers1To3* are collected in a set Y. It can be observed that after the **for** loop in *QE_LMC*, $\exists x_1 \cdots \exists x_t. A$ can be equivalently expressed as $\varphi_1 \wedge \exists Y. \varphi_2$ where φ_1 and φ_2 are conjunctions of LMCs. This is achieved using the function *scopeReduce* in Fig. 3. Finally $\exists Y. \varphi_2$ is computed by *QE_Layer4*. The result is conjoined with φ_1 to obtain the final result. *QE_Layer4* computes $\exists Y. \varphi_2$ as a disjunction of conjunctions of LMCs. Hence the final result is, in general, a Boolean combination of LMCs.

```
QE1_Layers1To3(A, $x_1$)
$\psi_1 \wedge \exists x_1. \psi_2 \leftarrow$ QE1_Layer1($A$, $x_1$);
if ($\psi_2$ is free of LMDs)
  return ($\psi_1 \wedge$ QE1_LME($\psi_2, x_1$));
else
  $e \leftarrow$ LME in $\psi_2$;
  $\psi_{D,2} \leftarrow$ set of LMDs in $\psi_2$;
  if(DropLMDSimple($\{e\}, \psi_{D,2}, x_1) = \{e\}$)
    return ($\psi_1 \wedge$ QE1_LME($e, x_1$));
  else
    $\psi'_2 \leftarrow$ QE1_Layer3($\{e\}, \psi_{D,2}, x_1$);
    if($\psi'_2 = e$)
      return ($\psi_1 \wedge$ QE1_LME($e, x_1$));
    else return ($\psi_1 \wedge \exists x_1. \psi'_2$);
```

```
QE_LMC(A, $\{x_1, \ldots, x_t\}$)
$Y \leftarrow \{\}$;
for each $x_i \in \{x_1, \ldots, x_t\}$
  $A' \leftarrow$ QE1_Layers1To3($A$, $x_i$);
  if($A'$ is free of $x_i$)
    $A \leftarrow A'$;
  else /*$A' \equiv \psi_1 \wedge \exists x_i. \psi'_2$*/
    $A \leftarrow \psi_1 \wedge \psi'_2$;
    $Y \leftarrow Y \cup \{x_i\}$;
$\varphi_1 \wedge \exists Y. \varphi_2 \leftarrow$ scopeReduce($A, Y$);
$\varphi'_2 \leftarrow$ QE_Layer4($\varphi_2$, $Y$);
return $\varphi_1 \wedge \varphi'_2$;
```

Fig. 3. Procedures *QE1_Layers1To3* and *QE_LMC*

3 Boolean Combinations of LMCs

Algorithm *QE_LMC* described above eliminates a set of variables from a conjunction of LMCs. In this section, we explore two approaches for extending *QE_LMC* to Boolean combinations of LMCs. Specifically we investigate a *Decision Diagram (DD)* based approach and a *DAG* based (or *SMT solving* based) approach.

3.1 DD Based Approach

We introduce a data structure called Linear Modular Decision Diagram (LMDD) that represents Boolean combinations of LMCs. LMDDs are like BDDs [4], but with nodes labeled by LMEs. The problem we wish to solve in this subsection can be formally stated as follows. Given an LMDD f representing a Boolean combination of LMCs over a set of variables X, we wish to compute an LMDD $g \equiv \exists V.\, f$ where $V \subseteq X$.

The algorithms presented in this subsection use the following helper functions: a) *createLMDD* for creating an LMDD from a DAG representing a Boolean combination of LMCs, b) *isUnsat* to determine if the conjunction of LMCs in the given set is unsatisfiable, d) *getConjunct* to compute the conjunction of LMCs in a given set φ, e) *AND*, *OR*, *NOT*, *ITE* to perform the basic operations on LMDDs indicated by their names. We denote a non-terminal LMDD node f as $(P(f), H(f), L(f))$ where $P(f)$ is the LME labeling the node and $H(f)$ and $L(f)$ are the high child and low child respectively as defined in [4].

A straightforward procedure to compute $\exists V.\, f$ is to apply *QE_LMC* to each path originating at the node f similar to Black-box QE on Linear Decision Diagrams described in [1]. However, as observed in [1], this technique is not amenable to dynamic programming and the number of recursive calls to the procedure is linear in the number of paths in f (which is can be exponential in the number of nodes).

In the following discussion we present a more efficient procedure *QuaLMoDE* to compute $\exists V.\, f$. *QuaLMoDE* makes use of a procedure called *QE1_LMDD* that eliminates a single variable v from f (see Fig. 4). To compute $\exists v.\, f$, we call *QE1_LMDD* with arguments f, $\{\,\}$, $\{\,\}$ and v. *QE1_LMDD* performs a recursive traversal of the LMDD rooted at f collecting the set of LMEs E, and the set of LMDs D, containing v that it encountered along the path from f.

In general, if E denotes the set of LMEs and D denotes the set of LMDs, *QE1_LMDD*(f, E, D, v) computes an LMDD for $\exists v.\, (f \wedge C_E \wedge C_D)$, where C_E and C_D denote the conjunctions of LMEs in E and LMDs in D, respectively. Using Lemma 1, E can be expressed as $\{(2^{k_1} \cdot v = t_1), \ldots, (2^{k_n} \cdot v = t_n)\}$. Without loss of generality, let k_1 be the smallest among $k_1, \ldots, k_n$. Let g be an internal non-terminal node of f. Thus g can be represented as $(P(g), H(g), L(g))$. Suppose $P(g)$ is $(2^k \cdot v = t)$ where $k \geq k_1$. It can be observed that g can then be simplified to $\big((2^{k-k_1} \cdot t_1 = t), H(g), L(g)\big)$ using the LME $(2^{k_1} \cdot v = t_1)$. Procedures *selectLME* and *simplifyLMDD* (see Fig. 4) respectively perform the selection of LME with the minimum k among the LMEs in E and simplification of f using the selected LME as described above. The procedure *applyL1* in Fig. 4 returns an LME equivalent to the argument LME using Lemma 1.

It can be observed if the same LMDD node is encountered with the same LME following two different paths, the results of the calls to *simplifyLMDD* must be the same. Hence *simplifyLMDD* can be implemented with dynamic programming.

Note that if *simplifyLMDD* is successful in eliminating all occurrences of variable v using the LME selected, *QE1_LMDD* returns without any further recursive

```
QE1_LMDD(f, E, D, v)
 if ($f = 0 \vee isUnsat(E \cup D)$)
   return 0;
if ($f = 1$)
   return createLMDD(QE_LMC
   (getConjunct($E \cup D$), {v}));
if ($E \neq \phi$)
   $e_1 \leftarrow$ selectLME(E);
   f' $\leftarrow$ simplifyLMDD(f, v, $e_1$);
   if (f' is free of v)
     return AND(f', createLMDD
     (QE_LMC(getConjunct($E \cup D$), {v})));
else
   f' $\leftarrow$ f;
e $\leftarrow$ P(f');
if (e is free of v)
   return ITE(e, QE1_LMDD(H(f'), E, D, v),
   QE1_LMDD(L(f'), E, D, v));
else
   return OR
   (QE1_LMDD(H(f'), $E \cup \{e\}$, D, v),
   QE1_LMDD(L(f'), E, D $\cup\{\neg e\}$, v));
```

```
simplifyLMDD(f, v, $e_1$)
 if (f = 1 or f = 0)
   return f;
e $\leftarrow$ P(f);
if (e is free of v)
   return ITE(e,
   simplifyLMDD(H(f), v, $e_1$),
   simplifyLMDD(L(f), v, $e_1$));
else
   applyL1(e, v);
   /* gives ($2^k \cdot v = t$) */
   applyL1($e_1$, v);
   /* gives ($2^{k_1} \cdot v = t_1$) */
   if ($k \geq k_1$)
      return ITE($2^{k-k_1} \cdot t_1 = t$,
      simplifyLMDD(H(f), v, $e_1$),
      simplifyLMDD(L(f), v, $e_1$));
   else
      return ITE(e,
      simplifyLMDD(H(f), v, $e_1$),
      simplifyLMDD(L(f), v, $e_1$));
```

Fig. 4. Algorithms *QE1_LMDD* and *simplifyLMDD*

calls. The procedure *QE1_LMDD* can be repeatedly invoked to compute $\exists V.\, f$. This is implemented in the procedure *QuaLMoDE*.

3.2 DAG Based Approach

The problem we wish to solve in this subsection is the following. Given a DAG f representing a Boolean combination of LMCs over a set of variables X, we wish to compute a DAG $g \equiv \exists V.\, f$ where $V \subseteq X$.

We present an algorithm *Monniaux* to compute $\exists V.\, f$, that is a simple extension of the algorithm EXISTELIM in [2]. EXISTELIM as given in [2] computes $\exists V.\, f$ where f is a Boolean combination of linear inequalities over reals. A naive way of computing this is by converting f to DNF by enumerating all satisfying assignments, and by using a QE technique for conjunctions of linear inequalities. EXISTELIM improves upon this by generalizing a satisfying assignment to obtain a cube of satisfying assignments, and by projecting the cube on the remaining variables (not in V) before its complement is conjoined with f and further satisfying assignments are found.

The algorithm *Monniaux* designed by us is an extension of the algorithm EXISTELIM, with the following changes: a) The predicates are LMCs, not linear inequalities over reals, b) the projection algorithm PROJECT (see [2]) is replaced by *QE_LMC*, and c) the algorithm GENERALIZE2 (see [2]) for generalization of conjunctions is replaced by an algorithm *GENERALIZE2_LMC*.

Given a formula G and a conjunction M of literals of G such that $M \Rightarrow \neg G$, the algorithm GENERALIZE2 described in [2] removes unnecessary literals from M and returns M' such that $M \Rightarrow M'$ and $M' \Rightarrow \neg G$. However, in our experiments with LMCs, we have found that GENERALIZE2 is prohibitively time consuming as it involves a large number of SMT solver calls.We therefore designed algorithm *GENERALIZE2_LMC* that works in the following way. Given a conjunction of literals M, we effectively have an assignment of Boolean value to each atomic predicate in the formula $\neg G$. We evaluate the propositional skeleton (DAG representation of the propositional structure) P of $\neg G$ using these Boolean values assigned to the atomic predicates. This assigns a Boolean value b_n to each node n in P. We now find the subset S_n of literals in M that is sufficient to evaluate n to b_n. Let S_r be the set of literals found in this way for the root r of P. Let M' be the conjunction of literals in S_r. It is easy to see that $M \Rightarrow M'$ and $M' \Rightarrow \neg G$. We illustrate this idea with a simple example. Let $\neg G$ be the formula $ite(A, B, C) \vee ite(D, E, F)$ and let M be $A \wedge B \wedge \neg C \wedge \neg D \wedge \neg E \wedge F$ where A, B, C, D, E and F are LMCs. It is easy to see that the set of literals $\{A, B\}$ is sufficient to cause $ite(A, B, C)$ to evaluate to true. Similarly $\{\neg D, F\}$ is sufficient to cause $ite(D, E, F)$ to evaluate to true. Hence, $\{A, B\}$ (or $\{\neg D, F\}$) is sufficient to cause $\neg G$ to evaluate to true. Hence *GENERALIZE2_LMC* would therefore return $A \wedge B$ (or $\neg D \wedge F$) as M'.

4 Experimental Results

We performed three sets of experiments to achieve the following goals: a) evaluate the performance of *QuaLMoDE*, *Monniaux* and *QE_LMC*, b) compare the performance of *QE_LMC* with alternative QE techniques and c) evaluate the utility of our QE algorithms in word-level RTL verification.

The experiments were performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Ubuntu 8.04. We implemented our own LMDD package for carrying out QE experiments using the DD based approach. In our implementation, we convert LMDs with modulus 2 to equivalent LMEs as a simplification step. Hence, in this section “LMD” refers to LMDs with modulus greater than 2.

Evaluation of *QuaLMoDE*, *Monniaux* and *QE_LMC*: In order to evaluate *QuaLMoDE* and *Monniaux*, we used a benchmark suite consisting of 210 *real* benchmarks and 212 *artificial* benchmarks. The *real* benchmarks were obtained in the following manner. We took a set of real word-level VHDL designs and derived their symbolic transition relations. One set of *real* benchmarks was obtained by quantifying out all the internal variables (i.e. neither input nor output of the top-level module) from these symbolic transition relations. Effectively this gives abstract transition relations of the designs. The second set of *real* benchmarks were obtained by applying iterative squaring to the symbolic transition relations for 3-5 steps. Each step of iterative squaring involves quantifying out one copy of all state variables in the symbolic transition relations. We observed a significant number of LMDs in these benchmarks when expressed in Negation

Normal Form (NNF) (see Fig. 5(a)). In order to generate the *artificial* benchmarks, we selected some of our *real* benchmarks and some SMTLib benchmarks from the category QF_BV/bruttomesso/simple_processor/ [10] and used random choices for the set of variables to be eliminated[3]. The total number of variables (N), number of variables to be eliminated (E) and the number of bits to be eliminated in the entire set of benchmarks range from 3 to 175, 1 to 170 and 1 to 1265 respectively.

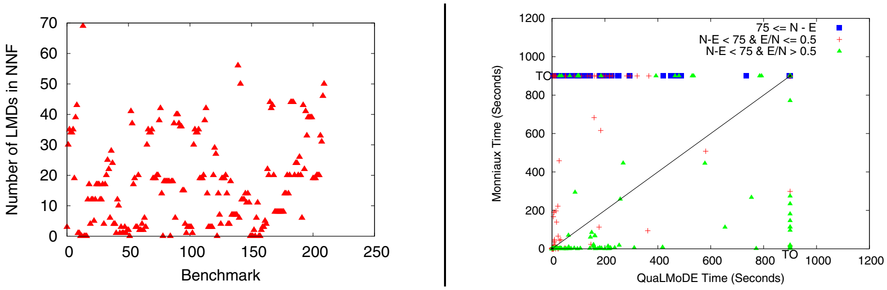


Fig. 5. Plots showing (a) significant number of LMDs in the *real* benchmarks. (b) *QuaLMoDE* Time Vs *Monniaux* Time (TO : > 900 seconds)

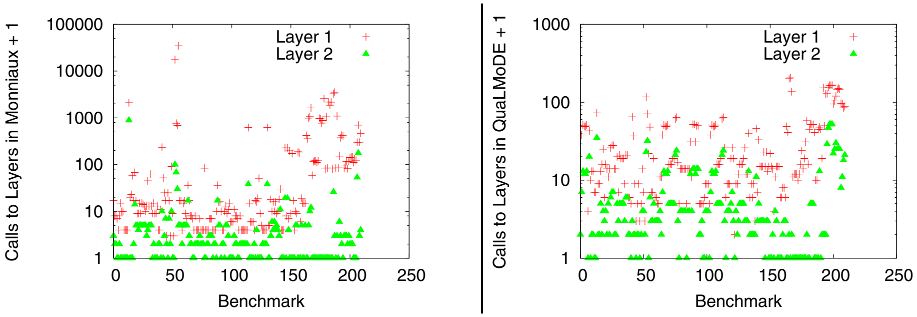


Fig. 6. Contribution of the layers in *QE_LMC*

We measured the QE time by *QuaLMoDE* and *Monniaux* for each benchmark (for *QuaLMoDE*, this includes the time taken to build LMDDs). It was observed that (see Fig. 5(b)) for benchmarks with $N - E$ below a certain threshold t_1 and E/N above a certain threshold t_2, *Monniaux* outperformed *QuaLMoDE* in most cases. For our benchmark suite, t_1 and t_2 were empirically estimated as 75 and 0.5 respectively. For the other benchmarks, we observed that *QuaLMoDE* outperformed *Monniaux*. It was also observed that, for benchmarks with $t_1 \leq N-E$, *Monniaux* timed out irrespective of E/N. We figured out that the different behaviours of *Monniaux* and *QuaLMoDE* were due to the following reasons. (i) For

[3] The SMTLib benchmarks contain bit-vector operators like selection and concatenation, which our work does not address. We introduced a fresh variable to denote the result of each such operator.

benchmarks with low $N - E$ and high E/N, the interleaving of projection inside model enumeration in *Monniaux* simplified the problem considerably whereas for the other benchmarks this simplification was not substantial. (ii) The single variable elimination strategy in *QuaLMoDE* resulted in a large number of calls to *QE1_LMDD* for benchmarks with low $N - E$ and high E/N.

The number of calls to *QE_LMC* from *QuaLMoDE* and from *Monniaux* while performing QE for the *real* benchmarks ranged from 1 to 205 and from 1 to 3842 respectively. We observed that a considerable number of these calls contained LMDs. The average number of LMDs in *QE_LMC* calls from *QuaLMoDE* and from *Monniaux* ranged from 0 to 12.2 and 0 to 18.8, respectively. The average of the ratio of the number of LMEs to the number of LMDs ranged from 0 to 1 and from 0.19 to 23.4 respectively.

We evaluated the roles of different layers of *QE_LMC* in performing QE for the *real* benchmarks. It was observed that all quantifiers were eliminated by the first two layers, without even a single call to *QE1_Layer3* or *QE_Layer4*. A large fraction of the calls to *QE1_Layers1To3* were solved by the first layer itself and the remaining were solved by the second layer (see Fig. 6)[4].

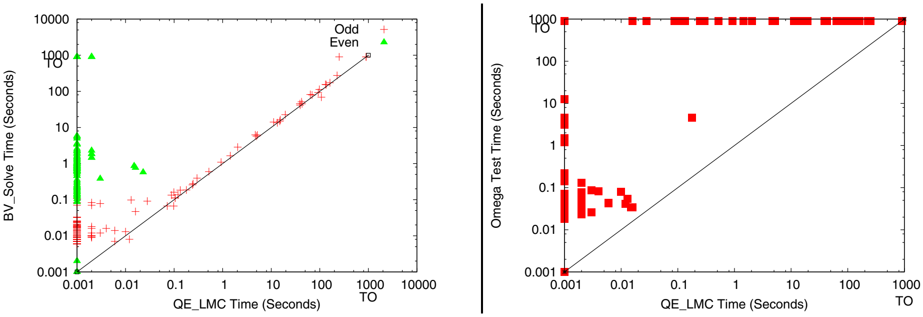


Fig. 7. Plots comparing (a) *QE_LMC* with *BV_Solve* (b) *QE_LMC* with Omega Test (TO : > 900 seconds)

Comparison of *QE_LMC* with alternative QE techniques : We have compared the performance of *QE_LMC* with QE based on Presburger Arithmetic using Omega Test and with QE based on bit-blasting (see Fig. 7). In the latter case, we implemented a procedure *BV_Solve* that first quantifies out variables appearing with odd coefficients in LMEs using the ideas described in [6] and then uses bit-blasting and BDD based bit-level QE [11] for the remaining variables. We used a set of 405 benchmarks that are instances of the QE problem for conjunction of LMCs; 371 of these arise from calls from *QuaLMoDE*/*Monniaux* when QE is performed on the *real* benchmarks and the remaining 34 are randomly generated. Our results clearly demonstrate that *QE_LMC* outperforms both alternative QE techniques. In Fig. 7(a), a benchmark is labeled "Odd" if

[4] Note that the y-axis of both plots are in log-scale. One is added to the y-values to include the points with no calls to the second layer.

Table 1. Experimental Results on VHDL Programs

Design	LOC	SS	TR	UNR=500		
				NA	QL	QB
machine_1	363	8	(371, 20, 547)	TO(TO)	98(4, 27)	TO(TO, -)
machine_2	373	6	(371, 19, 341)	TO(TO)	70(2, 0)	TO(TO, -)
machine_3	383	7	(395, 22, 344)	TO(TO)	75(3, 3)	TO(TO, -)
machine_4	253	4	(235, 19, 515)	1497(1418)	79(1, 0)	TO(TO, -)
machine_5	253	4	(235, 19, 387)	1527(1451)	76(1, 0)	TO(TO, -)
machine_6	363	4	(242, 15, 56)	122(80)	41(0, 0)	52(2, 3)
machine_7	379	5	(270, 20, 61)	206(152)	52(3, 1)	66(3, 5)
machine_8	251	2	(170, 13, 83)	225(195)	30(1, 1)	35(4, 1)
machine_9	251	3	(170, 13, 323)	TO(TO)	30(1, 1)	53(28, 1)
machine_10	363	5	(242, 15, 356)	TO(TO)	40(1, 0)	63(13, 3)
machine_11	363	6	(352, 22, 96)	TO(TO)	97(1, 7)	98(2, 24)
machine_12	363	5	(242, 15, 356)	TO(TO)	478(8, 427)	TO(TO, -)
board_1	404	4	(265, 13, 163)	1455(1426)	51(24, 0)	TO(TO, -)
board_2	373	3	(283, 13, 163)	TO(TO)	66(49, 0)	TO(TO, -)
board_3	503	4	(284, 13, 190)	TO(TO)	67(44, 0)	TO(TO, -)
board_4	415	3	(272, 11, 31)	362(229)	111(10, 3)	215(104, 13)

All times are in seconds. **TO** : > 1800 seconds, **LOC** : Lines of code, **SS** : Symbolic simulation time, **TR** : Transition relation details (dag size, number of variables, number of bits), **NA** : Without abstraction : total time (simplifyingSTP time), **QL** : With *QuaLMoDE* for abstraction : total time (*QuaLMoDE* time, simplifyingSTP time), **QB** : With *QBV_Solve* for abstraction : total time (*QBV_Solve* time, simplifyingSTP time) (for **NA**, **QL** and **QB** most of the remaining time is spent in slicing - we use a naive implementation of slicer), **UNR** : Number of BMC unrollings

each quantified variable in it appears with odd coefficient in at least one LME and "Even" otherwise. Our results demonstrate that *BV_Solve* performs comparable to *QE_LMC* for the "Odd" benchmarks, but not for the "Even" ones. This is not surprising since *BV_Solve* uses the technique from [6] to eliminate variables whenever possible before bit-blasting. Hence it is able to eliminate variables without any bit-blasting for all "Odd" benchmarks. In contrast, *BV_Solve* has to bit-blast for "Even" benchmarks, thereby performing poorly.

Utility of our QE algorithms in verification : In order to evaluate the utility of our QE algorithms, we used *QuaLMoDE* to compute abstract transition relations when checking safety properties of a set of word-level VHDL designs using BMC. We first derived the symbolic transition relation R of each design. For each BMC frame i, we then used slicing to obtain a slice R_i of R containing only the relevant part of R for this frame. Next, we eliminate a chosen subset of variables (subset of internal variables) from R_i to obtain R_i' using *QuaLMoDE* as well as *QBV_Solve* (an extension of *BV_Solve* using the DD based approach to handle Boolean combinations of LMCs). The final unrolled constraint is a conjunction of the different R_i's computed by *QuaLMoDE*/*QBV_Solve*. This is conjoined with the negation of the safety property being checked and given to an SMT solver for checking satisfiability. The SMT solver used is simplifyingSTP [12][5]. Table 1 gives a summary of these results. The designs in our experiments machine_1 to machine_12 are modified versions of publicly available benchmarks obtained from [9]. The remaining designs are proprietary and

[5] We selected simplifyingSTP because (i) it is the winner of SMT-COMP 2010 bit-vector category and (ii) it has a variable eliminator implemented as per [6].

were obtained from safety critical applications used in nuclear reactors. They are control-oriented designs with wide data paths. Our results clearly demonstrate (i) the significant performance benefit of using abstract transition relations computed by *QuaLMoDE* in these verification exercises, and (ii) the performance benefits of *QuaLMoDE* over *QBV_Solve* in computing the abstract transition relations particularly for designs involving constant multiplications with even coefficients and large bit widths.

Our QE algorithms can be used in principle for checking the satisfiability of Boolean combinations of LMCs. This can be done by quantifying out all variables. However preliminary experiments suggest that this approach is not competitive with DPLL-style SMT solvers or with bit-blasting followed by QBF solving. Thus, the intended usage of our algorithms is for eliminating some (but not all) variables from Boolean combinations of LMCs.

5 Conclusion

In this paper, we addressed the QE problem for LMCs. Our main contributions are: (i) a bit-blasting-free QE algorithm for conjunctions of LMCs that is later extended to a QE algorithm for Boolean combinations of LMCs, and (ii) comparison of our approach with alternative techniques and the identification of a simple-to-use criteria for choosing the right QE approach for a given problem instance. We propose to study QE for linear modular inequalities and non-linear modular equalities as part of future work.

Acknowledgements . We would like to thank Trevor Hansen and Vijay Ganesh for providing us with the latest version of simplifyingSTP. We also thank Mukesh Sharma, Ashutosh Kulkarni, Rajkumar Gajavelly and Nachiket Vaidya for their valuable support. We convey our special acknowledgement to Anup Bhattacharjee and S.D.Dhodapkar for their indispensable help and support.

References

1. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: FMCAD (2009)
2. Monniaux, D.: A quantifier elimination algorithm for linear real arithmetic. In: LPAR (2008)
3. Kroening, D., Strichman, O.: Decision procedures: an algorithmic point of view. Texts In Theoretical Computer Science. Springer, Heidelberg (2008)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
5. Pugh, W.: The Omega Test: A fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 102–114 (1992)
6. 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)

7. Jain, H., Clarke, E., Grumberg, O.: Efficient craig interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
8. Ganesh, V., Berezin, S., Dill, D.: Deciding Presburger arithmetic by model checking and comparisons with other methods. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 171–186. Springer, Heidelberg (2002)
9. ITC 99 benchmarks, `http://www.cad.polito.it/downloads/tools/itc99.html`
10. SMTLib website, `http://goedel.cs.uiowa.edu/smtlib/`
11. CUDD release 2.4.2 website, `vlsi.colorado.edu/~fabio/CUDD`
12. STP website, `http://sites.google.com/site/stpfastprover/`
13. John, A., Chakraborty, S.: A quantifier elimination algorithm for linear modular equations and disequations, Technical Report TR-11-33, CFDVS, IIT Bombay, `http://www.cfdvs.iitb.ac.in/reports/reports/ajith_report.pdf`

Bug-Assist: Assisting Fault Localization in ANSI-C Programs*

Manu Jose[1] and Rupak Majumdar[1,2]

[1] University of California, Los Angeles,
Department of Computer Science, USA
[2] Max Planck Institute for Software Systems,
Kaiserslautern, Germany

Abstract. Several verification tools exist for checking safety properties of programs and reporting errors. However, a large part of the program development cycle is spend in analyzing the error trace to isolate locations in the code that are potential causes of the bug. Currently, this is usually performed manually, by stepping through the error trace in a debugger. We describe Bug-Assist, a tool that assists programmers localize error causes to a few lines of code. Bug-Assist takes as input an ANSI-C program annotated with assertions, performs bounded model checking to find potential assertion violations, and for each error trace returned by the model checker, returns a set of lines of code which can be changed to eliminate the error trace. Bug-Assist formulates error localization as a MAX-SAT problem and uses scalable MAX-SAT solvers. In experiments on a set of C benchmarks, Bug-Assist was able to reduce error traces to only a few lines of code. Bug-Assist is available as an Eclipse plug-in, enabling its easy deployment in the code development phase.

1 Introduction

Quality assurance is a major component of the software development cycle. Recent years have seen an explosion in static and dynamic analysis tools that can automatically look for classes of program errors. These tools take (unmodified) code as input, and produce error traces (and test cases) to demonstrate how assertions about correct behavior can be violated. However, after such tools report error traces, it is usually up to the programmer to debug and locate the faulty lines in the program, before modifying the program to eliminate the error trace. Thus, even with the support of automatic verification tools, a large part of the development cycle is spent in debugging, where the programmer looks at a long, failing, trace and tries to localize the problem to a few lines of source code that elucidate the cause of the problem.

We present Bug-Assist, a tool for fault localization for ANSI-C programs. The tool and the user manual can be downloaded at http://bugassist.mpi-sws.org. The input to our tool is a C program, instrumented with assertions which specify

* This research was funded in part by the NSF grant CCF-0546170.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 504–509, 2011.

the correct behavior of the program, and either a test case failing an assertion or an option to run a model checker looking for assertion violations. If a violation is found, the output is a minimal set of program statements such that there exists a way to replace these statements to correct the program execution.

Bug-Assist is available as a command-line tool and through a graphical user interface (GUI) as a plugin inside the popular integrated development environment (IDE) Eclipse. The GUI is similar to existing GUI based debugging tools inside Eclipse IDE, which are familiar to software engineers. So the programmer need not worry about the command line options or the internal symbolic analysis instead can look in to the highlighted potential buggy lines in the source code window.

Internally, Bug-Assist constructs a Boolean formula from a failing test case, and uses a MAX-SAT solver to identify minimal sets of instructions whose modification can eliminate the failing execution. The failing test case is obtained either directly from the testsuite for the program, or obtained using a bounded model checker (CBMC) integrated within Bug-Assist.

We evaluated Bug-Assist using programs from the Siemens set of benchmarks with injected faults [4]. In each case, Bug-Assist can efficiently and precisely determine the exact (to the human) lines of code that constitute the reason of the "bug". The TCAS program in the testsuite is run with all the faulty versions in detail to illustrate the completeness of the tool. Each of these test cases consisted of 173 lines of code and the maximum time to localize the bug locations was 0.136 seconds with the average number of buggy lines came to 8% of the total lines of code [7]. The other 4 programs are used to show the scalability of the tool by using error trace reduction methods for real world programs. This shows the effectiveness of using the tool in parallel with the code development process.

While there has been a lot of research in fault localization [1,5,6], to the best of our knowledge, there is no publicly available tool for pin-pointing error locations. Delta-debugging tools, that minimize the input for a failing run, exist [9], but have a somewhat orthogonal emphasis: reducing the failure causing input. Bug-Assist fills in the gap between error traces and bug-fixing by pointing out exact locations of the code where corrections should be made instead of minimal code fragments demonstrating failure. We believe Bug-Assist can be effectively used in conjunction with a delta-debugging tool.

2 Tool Description

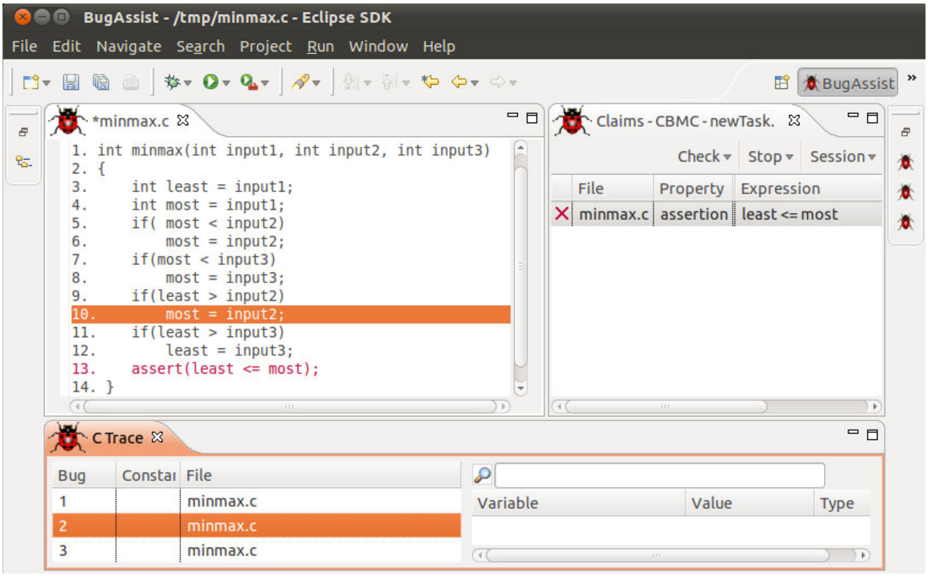
2.1 User Guide

The tool is available as an eclipse plugin and as a command line tool for linux platform. It takes as input an ANSI-C program with properties specified as *assert* statements. The tool can also generate implicit assertions for array bound checks, null pointer checks, etc. If the program violates the specifications, it outputs the potential lines of code which are the repair candidates for the program. For localizing error for a faulty program file.c, we can run Bug-Assist as follows:

./bugassist file.c –ba –maxsat <solver location>

where the option *–ba* outputs the potential repair locations. The *–maxsat* gives user the flexibility to choose the MAX-SAT solver by providing the solver executable location as Bug-Assist generates the MAX-SAT instances following the general MAX-SAT competition format. The detailed list of options can be obtained with the option *–help*.

2.2 The Integrated Debugging Environment


BugAssist - /tmp/minmax.c - Eclipse SDK
File Edit Navigate Search Project Run Window Help
BugAssist
*minmax.c
Claims - CBMC - newTask.
Check Stop Session
C Trace

```
1. int minmax(int input1, int input2, int input3)
2. {
3.     int least = input1;
4.     int most = input1;
5.     if( most < input2)
6.         most = input2;
7.     if(most < input3)
8.         most = input3;
9.     if(least > input2)
10.        most = input2;
11.    if(least > input3)
12.        least = input3;
13.    assert(least <= most);
14. }
```

	File	Property	Expression
✗	minmax.c	assertion	least <= most

Bug	Constai	File
1		minmax.c
2		minmax.c
3		minmax.c

Variable	Value	Type

Fig. 1. A screenshot of the tool localizing error for a simple program

To improve the usability of our tool, we have built an Eclipse plugin and a GUI to help the programmer find bug locations during the development process. During the code development phase, the programmers might only be interested in modules or files under the current development branch. Therefore Bug-Assist IDE creates a new project which gives the users the flexibility to pick the files and functions for which they want to check for property violations and debug the problems only within those files.

Figure 1 shows Bug-Assist IDE analyzing a simple program "minmax.c", which computes the minimum and maximum among its input arguments. There is an error in the program at line 10. During the first run of Bug-Assist, it lists out the assertions which are specified in the system. In the example program there is an explicit assertion claiming "least" should be less than or equal to "most" and is listed in the claims window. Additionally, one can specify implicit assertions for null pointer checks or array-bounds checks, by setting properties of the model checker.

When the model checker returns the violated assertions, the programmer can now double click on each of the violated assertions to find the potential lines of code that lead to the violation of that property. The three potential bug locations analyzed by the tool is given in the "C Trace" window in Figure 1. Selecting each of these bug locations highlights the corresponding location in the source code viewer, helping the programmer to effectively visualize the problem cause and to provide a fix. After fixing the error, (in this case it is to change the variable "most" to "least" at line 10) the user can run the model checker again to check if the fix is correct and does not break for other inputs.

2.3 Tool Architecture

Internally, our algorithm leverages symbolic analysis of software based on Boolean satisfiability, and reduces the problem to *maximum Boolean satisfiability*. Maximum Boolean satisfiability (MAX-SAT) is the problem of determining the maximum number of clauses of a given Boolean formula in CNF that can be satisfied by any given assignment.

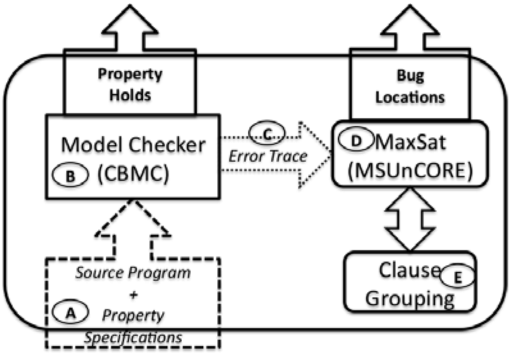


Fig. 2. The tool architecture

Figure 2 shows the tool architecture with the program flow. We describe the tool internals informally using the same program shown in Figure 1. The function `minmax` sets the "least" and "most" variables to the first arguments and then eventually sets those variable to the minimum and maximum value respectively from the input variables (shown in lines 5–12). After comparing "least" with "input2", instead of updating the "least" variable the program updates the "most" variable.

Bug-Assist performs the following two steps to localize the bug. First, from the program, it uses bounded model checking [2,3] (using the tool CBMC [3]) to construct a symbolic *trace formula*: a Boolean formula in conjunctive normal form such that the formula is satisfiable iff the program execution is feasible. For the example, CBMC (label B in Figure 2) gives a counter example with

$\texttt{input1} = \texttt{10}$, $\texttt{input2} = \texttt{1}$, and $\texttt{input3} = \texttt{5}$, which makes $\texttt{most} = \texttt{1}$ and violates the assertion at line 13. The trace formula TF encoding the program semantics is:

$$\begin{aligned}\mathsf{TF} \equiv\, & \texttt{least}_1 = \texttt{input1}_1 \wedge \texttt{most}_1 = \texttt{input1}_1 \wedge \texttt{guard}_1 = \texttt{most}_1 < \texttt{input2}_1 \wedge \\ & \texttt{most}_2 = (!\texttt{guard}_1?\texttt{most}_1 : \texttt{input2}_1) \wedge \texttt{guard}_2 = \texttt{most}_2 < \texttt{input3}_1 \wedge \\ & \texttt{most}_3 = (!\texttt{guard}_2?\texttt{most}_2 : \texttt{input3}_1) \wedge \texttt{guard}_3 = \texttt{least}_1 > \texttt{input2}_1 \wedge \\ & \texttt{most}_4 = (!\texttt{guard}_3?\texttt{most}_3 : \texttt{input2}_1) \wedge \texttt{guard}_4 = \texttt{least}_1 > \texttt{input3}_1 \wedge \\ & \texttt{least}_2 = (!\texttt{guard}_4?\texttt{least}_1 : \texttt{input3}_1)\end{aligned}$$

We then extend the trace formula by conjoining it with constraints so that the final states ensure the program post-condition:

$$\Phi \equiv \underbrace{\texttt{input1}_1 = 10 \wedge \texttt{input2}_1 = 1 \wedge \texttt{input3}_1 = 5 \wedge}_{\text{test input}} \underbrace{\mathsf{TF}}_{\text{trace formula}} \wedge \underbrace{\texttt{least}_2 \leq \texttt{most}_4}_{\text{assertion}}$$

The extended trace formula essentially states that starting from the initial condition, executing the program trace leads to a state satisfying the post-condition. Notice that the extended trace formula for a failing execution must be unsatisfiable.

Second, it feeds the extended trace formula in conjunctive normal form (CNF) to a *partial maximum satisfiability solver* (label D in Figure 2). In partial MAX-SAT, the input clauses can be marked *hard* or *soft*, and the MAX-SAT instance finds the maximum number of soft clauses that can be satisfied by an assignment which satisfies every hard clause. In our algorithm, we mark the input constraints (that ensure that the input is a failing test) as well as the post-condition as hard. This is necessary: otherwise, the MAX-SAT algorithm can trivially return that changing an input or changing the post-condition can eliminate the failing execution.

In the case of Φ, we mark the constraints coming from the test input and the assertion as hard, and leave the clauses in the trace formula soft. We use an off-the-shelf MAX-SAT solver [8] to compute a maximal set of clauses of the extended trace formula that can be satisfied, and take the complement of this set as a candidate set of clauses that can be changed to make the entire formula satisfiable. Since each clause in the extended trace formula can be mapped back to a statement in the code, this identifies a candidate localization of the error in terms of program statements. Note that there may be several minimal sets of clauses that can be found in this way, and we enumerate each minimal set as candidate localizations for the user.

The clauses returned by MAX-SAT point to line 9 as a potential error location in the first iteration of our program. In the next iteration of MAX-SAT, we make the clauses arising out of line 9 hard and ask for any other correction locations. We repeat this process until MAX-SAT gives the problem to be unsatisfiable. All the three error locations reported by tool is shown in Figure 1.

In our implementation, we group clauses (label E) arising out of the same program statement together making the resulting MAX-SAT instance simple.

This ensures our algorithm localizes errors at the program statement level. Error localization can be performed at other levels of granularity as well. For example, to localize bugs at the function or module level, we can group clauses coming from the same function or module in the MAX-SAT instance.

A formal definition of the localization algorithm as well as experimental evaluations can be found in [7].

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL 2003: Principles of Programming Languages, pp. 97–105. ACM, New York (2003)
2. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC 1999: Design Automation Conference, pp. 317–320. ACM, New York (1999)
3. Clarke, E., 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)
4. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4), 405–435 (2005)
5. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. 8(3), 229–247 (2006)
6. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE 2005: Automated Software Engineering, pp. 273–282. ACM, New York (2005)
7. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: PLDI 2011: Programming Language Design and Implementation, ACM, New York (2011)
8. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: DATE 2008: Design, Automation and Test in Europe, pp. 408–413. ACM, New York (2008)
9. Zeller, A.: Isolating cause-effect chains from computer programs. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 1–10. Springer, Heidelberg (2002)

Synthesis of Distributed Control through Knowledge Accumulation*

Gal Katz[1], Doron Peled[1], and Sven Schewe[2]

[1] Department of Computer Science,
Bar Ilan University, Ramat Gan 52900, Israel
[2] Department of Computer Science,
University of Liverpool, Liverpool, UK

Abstract. In distributed systems, local controllers often need to impose global guarantees. A solution that will not impose additional synchronization may not be feasible due to the lack of ability of one process to know the current situation at another. On the other hand, a completely centralized solution will eliminate all concurrency. A good solution is usually a compromise between these extremes, where synchronization is allowed for in principle, but avoided whenever possible. In a quest for practicable solutions to the distributed control problem, one can constrain the executions of a system based on the pre-calculation of knowledge properties and allow for temporary interprocess synchronization in order to combine the knowledge needed to control the system. This type of control, however, may incur a heavy communication overhead. We introduce the use of simple *supervisor* processes that *accumulate* information about processes until *sufficient knowledge* is collected to allow for safe progression. We combine the knowledge approach with a game theoretic search that prevents progressing to states from which there is no way to guarantee the imposed constraints.

1 Introduction

Designing concurrent systems such that they satisfy a given specification is a highly difficult task that requires a lot of creativity. Automatic synthesis is very desirable, but unfortunately undecidable [16,10,11,5,20]. Separating the design of the system such that global constraints can be later imposed on a distributed system can be a very powerful tool. The hardness of imposing distributed control is derived from the fact that the processes can only rely on their individual observations of the system state. Such limited local information can be described as *knowledge* that processes have at each point of the execution [4,7].

In the classical approach to the synthesis of distributed control [24,18,19], memory is added to each process. This memory is updated based on the observation made by the respective process and, based on the state of this memory, the processes can support transitions or block them. Checking whether a distributed system can be controlled in

* The research of the 1st author was funded by Israeli Science Foundation (ISF) grant 1252/09. The research of the 2nd author was funded by Royal Society Grant TG100660 'Synthesising Permissive Monitors', The research of the 3rd author was funded by Engeering and Physical Science Research Council (EPSRC), grant EP/H046623/1.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 510–525, 2011.

this way to satisfy a given property is undecidable [22,23], even when the imposed property is an invariant on states and subsequent transitions [6].

In this paper, we study the problem of controlling distributed systems to satisfy an invariant property by adding supervisory processes that can communicate asynchronously with several processes, helping them to make a safe progress decision. We show that in this case controllability is decidable; it is EXPTIME complete, and PSPACE complete for systems with or without uncontrollable transitions, respectively.

In order to decide controllability and determine a control strategy, we first perform a global *strategy search* in order to avoid being trapped into a state where the invariant cannot be satisfied. Then, the obtained reduced state space is redistributed according to the original processes based on model checking the processes *knowledge* [13,1,2,6]. The information gathered during the model checking stage is used as a basis for a program transformation that controls the execution of the system by adding constraints on the enabledness of transitions. This does not produce new program executions or deadlocks, and consequently preserves all stuttering closed [14] linear temporal logic properties of the system [12].

When a process does not have enough local knowledge to progress safely, it can *hang* on a *supervisor* process. Supervisors accumulate information about multiple processes that are hung on them until sufficient knowledge is acquired. In a series of solutions, we show strategies to minimize the amount of overhead in terms of additional communication and synchronization, and of blocking of concurrent occurrence of transitions. There are several contributions in this paper:

- While most of the results on distributed synthesis are negative, we provide a constructive and efficient solution. We introduce supervisor processes that, while running asynchronously with the controlled system, accumulate the knowledge from several processes until a decision to safely execute a transition can be made.
- Knowledge has been shown to be a useful tool to impose distributed control [1,2,6,18]. Synchronization [6] or message passing [18] can be used to prevent blocking (deadlock) due to lack of knowledge. However, the controlled system may still block by reaching a state where all the possible enabled transitions would violate the imposed property. We solve this by calculating knowledge based on the result of a game theoretic strategy search.
- We introduce a detailed and realistic concurrency model that distinguishes the part of the global state that a process can observe from the smaller part it can control.

2 Preliminaries

The model used in this paper is Petri Nets. It was chosen due to its visual representation. The method and algorithms developed here can equally apply to other models, e.g., transition systems, communicating automata, etc.

Definition 1. *A* (1-safe) Petri Net *N is a tuple (P,T,E,s_0) where*

- *P is a finite set of* places*,*
- *the* states *are defined as $S = 2^P$ where $s_0 \in S$ is the* initial state*,*

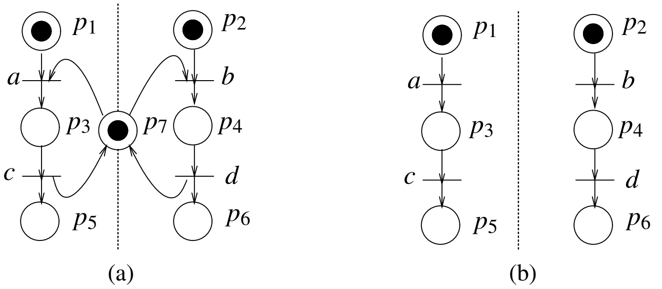


Fig. 1. Petri Nets: (a) without priorities and (b) with priorities $a \ll d$ and $b \ll c$

- *T is a finite set of* transitions*, and*
- *$E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between the places and the transitions.*

For a transition $t \in T$, we define the set of input places *${}^\bullet t$ as $\{p \in P \mid (p,t) \in E\}$, and* output places *$t^\bullet$ as $\{p \in P \mid (t,p) \in E\}$.*

Definition 2. *A transition t is* enabled *in a state s, denoted $s[t\rangle$, if ${}^\bullet t \subseteq s$ and $t^\bullet \cap s \subseteq^\bullet t$ (we allow self loops). A state s is in* deadlock *if there is no enabled transition from it.*

Definition 3. *A transition t can be* fired *(or* executed*) from state s to state s', denoted by $s[t\rangle s'$, when t is enabled at s. Then, $s' = (s \setminus^\bullet t) \cup t^\bullet$.*

Definition 4. *Two transitions t_1 and t_2 are* dependent *if $({}^\bullet t_1 \cup t_1{}^\bullet) \cap ({}^\bullet t_2 \cup t_2{}^\bullet) \neq \emptyset$. Let $D \subseteq T \times T$ be the* dependence *relation. Two transitions are* independent *if they are not dependent.*

Transitions are visualized as lines, places as circles, and the relation E is represented using arrows. In Figure 1(a), there are places $p_1, p_2, \ldots, p_7$ and transitions a, b, c, d. We depict a state by putting full circles, called *tokens*, inside the places of that state. In the example in Figure 1(a), the initial state s_0 is $\{p_1, p_2, p_7\}$. The transitions that are enabled from the initial state are a and b. If we fire transition a from the initial state, the tokens from p_1 and p_7 will be removed, and a token will be placed in p_3. In this Petri Net, all transitions are dependent on each other, since they all involve the place p_7. Removing p_7, see Figure 1(b), makes both a and c become independent from both b and d.

Definition 5. *An* execution *of a Petri Net N is a maximal (i.e., it cannot be extended) alternating sequence of states $s_0 t_1 s_1 t_2 s_2 \ldots$, where s_0 is the initial state, such that, for each states s_i in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$. We denote these executions by $exec(N)$.*

For convenience, we sometimes use as executions just the sequence of states, or just the sequence of transitions, as will be clear from the context. A state is *reachable* in a Petri Net if it appears on at least one of its executions. We denote the reachable states of a Petri Net N by $reach(N)$. In order to simplify the presentation and not distinguish between finite and infinite maximal executions, a special new element ε is added to T; it has no input and output places and can be fired exactly in states s with deadlocks (where no other transition is enabled), thence $s[\varepsilon\rangle s$. Yet, these states are still distinguished as deadlocks.

We use places also as state predicates. As usual, we write $s \models p_i$ iff $p_i \in s$ and extend this in the standard way to Boolean combinations on state predicates. For a state s, we denote by φ_s the formula that is a conjunction of the places in s and the negated places not in s. Thus, φ_s is satisfied exactly by the state s. For the Petri Net in Figure 1(a), the initial state s_0 satisfies $\varphi_{s_0} = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge p_7$. For a set of states $Q \subseteq S$, we call $\bigvee_{s \in Q} \varphi_s$, or any logically equivalent propositional formula φ_Q, a *characterizing formula* of Q. As usual in logic, when φ_Q and $\varphi_{Q'}$ characterize sets of states Q and Q', respectively, then $Q \subseteq Q'$ exactly when $\varphi_Q \rightarrow \varphi_{Q'}$.

An invariant [3] of N is a subset of the states $Q \subseteq 2^S$; a net N satisfies the invariant Q if $reach(N) \subseteq Q$. A *generalized invariant* of N is a set of pairs $I \subseteq S \times T$; a net N satisfies I if, whenever $s[t\rangle$ for a reachable s, then $(s,t) \in I$. This covers the above simple case of an invariant when pairing up every state that appears in Q with *all* transitions T.

Definition 6. *An execution of a Petri Net N* restricted *with respect to a set $I \subseteq S \times T$, denoted $exec_I(N)$, is the set of executions $s_0 t_1 s_1 t_2 s_2 \ldots \in exec(N)$ such that, for each pair $s_i t_{i+1}$ in the sequence, $(s_i, t_{i+1}) \in I$ holds. The set of states reachable in $exec_I(N)$ is denoted $reach_I(N)$.*

Lemma 1. *$reach_I(N) \subseteq reach(N)$ and $exec_I(N) \subseteq exec(N)$.*

Definition 7. *A* process *π of a Petri Net N is a subset of the transitions T satisfying that, for each pair $t_1, t_2 \in \pi$ of independent (i.e., $(t_1, t_2) \notin D$) transitions in π, there is no reachable state s in which both t_1 and t_2 are enabled.*

We will represent the separation of transitions of a Petri Net into processes using dotted lines. We assume a given set of processes $\mathcal{C}$ that *covers* all transitions (except ε) of the net, i.e., $\bigcup_{\pi \in \mathcal{C}} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Let $proc(t) = \{\pi \mid t \in \pi\}$ be the set of processes to which t belongs. For the Petri Net in Figure 1(a), there are two executions: *acbd* and *bdac*. There are two processes: the *left* process $\pi_l = \{a, c\}$ and the *right* process $\pi_r = \{b, d\}$. We use the same partitioning of transitions to processes in Figure 1(b).

The *neighborhood* of a set of processes Π includes all places that are either inputs or outputs to transitions of Π.

Definition 8. *The* neighborhood *$ngb(\pi)$ of a process π is the set of places $\bigcup_{t \in \pi} ({}^\bullet t \cup t^\bullet)$. For a set of processes $\Pi \subseteq \mathcal{C}$, $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$.*

A set of processes Π *owns* the places in their neighborhood that cannot gain or loose a token by a transition that is not exclusively in Π.

Definition 9. *The places* owned *by a set of processes (including a singleton process) Π, denoted $own(\Pi)$, is $ngb(\Pi) \setminus ngb(\mathcal{C} \setminus \Pi)$.*

When a notation refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π, e.g., we write $own(\pi)$. Note that $ngb(\Pi_1) \cup ngb(\Pi_2) = ngb(\Pi_1 \cup \Pi_2)$, while $own(\Pi_1) \cup own(\Pi_2) \subseteq own(\Pi_1 \cup \Pi_2)$. The neighborhood of process π_l is $\{p_1, p_3, p_5, p_7\}$. Place p_7 in Figure 1(a) is neither owned by π_l, nor by π_r, but it is owned by $\{\pi_l, \pi_r\}$. It belongs to the neighborhood of both processes and acts as a semaphore. It can be captured by the execution of a or of b, guaranteeing that $\neg(p_3 \wedge p_4)$ is an invariant of the system.

Our goal is to control the system to satisfy a generalized invariant by restricting some of its transitions from some of the states. We may be allowed to control only part of the transitions $ct(T) \subseteq T$, called *controllable* transitions. The other transitions, $uc(T) = T \setminus ct(T)$, are *uncontrollable*. Note that we may be at some state where either some uncontrollable transitions, or all enabled transitions, violate the generalized invariant. Being in such states is therefore "too late"; part of the controlling task is to avoid reaching such states.

As a running example for a generalized invariant we use a property of *priorities*.

Definition 10. *A* Petri Net with priorities *is a pair* $(N, \ll)$*, where* N *is a Petri Net and* $\ll$ *is a partial order relation among the transitions* T *of* N*. For the set* $I_{\ll} = \{(s,t) \mid$ t *is maximal among the enabled transitions in* s *w.r.t.* $\ll\}$*, the set of* prioritized executions $exec_{I_{\ll}}(N)$ *of* $(N, \ll)$ *is the set of executions restricted to* $I_{\ll}$*.*

The executions of the Petri Net M in Figure 1(b) (when the priorities $a \ll d$ and $b \ll c$ are *not* taken into account) include $abcd, acbd, bacd, badc$, etc. However, the prioritized executions of $(M, \ll)$ are the same as the executions of the Net N in Figure 1(a).

Enforcing prioritized executions in a completely distributed way may be impossible. In Figure 1(b), a and c belong to the left process π_l, and b and d belong to the right process π_r, with no interaction between the processes. The left process π_l, upon having a token in p_1, cannot locally decide whether to execute a; the priorities dictate that a can be executed if d is not enabled, since a has a lower priority than d. But this information is not locally available to π_l, which cannot distinguish between the cases where π_r has a token in p_2, p_4, or p_6.

The local information of Π at a given state consists of the restriction of the state to the neighborhood of the transitions of Π.

Definition 11. *The* local information *of a set of processes* Π *of a Petri Net* N *in a state* s *is* $s\lceil_{\Pi} = s \cap nbg(\Pi)$*.*

In the Petri Net in Figure 1(a), the local information of π_l in any state s consists of the restriction of s to the places $\{p_1, p_3, p_5, p_7\}$. In the depicted initial state, the local information is $\{p_1, p_7\}$. The local state restricts the places to the ones changeable only by a process (or set of processes).

Definition 12. *The* local state *of a set of processes* Π *of a Petri Net* N *in a state* s *is* $s\lfloor_{\Pi} = s \cap own(\Pi)$*.*

The local state of π_l in Figure 1(a) is $\{p_1\}$. It is always the case that $s\lfloor_{\Pi} \subseteq s\lceil_{\Pi}$.

Lemma 2. *If* $\pi \notin \Pi$ *then* $s\lfloor_{\Pi \cup \{\pi\}}$ *is the (disjoint) union of* $s\lfloor_{\Pi}$ *and* $s\lceil_{\pi} \cap own(\Pi \cup \{\pi\})$*.*

3 A Globally Controlled System

Before providing a solution to the distributed control problem we need to provide a solution to a related global control problem. Some reachable states are not allowed according to the generalized invariant. In order not to reach these states, we may need to avoid some transitions that lead to such states from previous states. This is done using game theoretical search.

The game is played between a *constructor*, who wants to preserve the generalized invariant I indefinitely, and a *spoiler*, who has the opposite goal. The game is played on the states S of a net, starting from the initial state s_0. In each round, the constructor player chooses a nonempty subset of enabled transitions that must include all enabled uncontrollable transitions. Subsequently, the spoiler chooses a transition from this set, which is then executed. The spoiler wins as soon as she can choose a transition that violates I, i.e., $(s,t_s) \not\in I$, while the constructor wins if he can avoid this for ever.

Let B (for "bad") be the reachable states that we do not want to reach, namely,

$$B = \{s \mid s \in \mathit{reach}(N) \wedge (\exists t \in \mathit{uc}(T)\,(s[t\rangle \wedge (s,t) \not\in I) \vee \forall t \in T\, s[t\rangle \rightarrow (s,t) \not\in I\})$$

We may be forced into B in a single step from every state that has an uncontrollable transition into B or for which all enabled transitions allowed according to I lead to states in B.

Let $\mathit{attr}(A)$ be states s that satisfy that one of the following conditions:

- $s \in A$, or
- there exists an uncontrollable transition t enabled[1] in s with $s[t\rangle s'$ and $s' \in A$, or
- for all transitions t enabled in s, such that $s[t\rangle s'$ and $(s,t) \in I$, it holds that $s' \in A$.

As usual, we define $\mathit{attr}^n(A) = \mathit{attr}(\mathit{attr}^{n-1}(A))$, where $\mathit{attr}^1(A) = \mathit{attr}(A)$. Because of the monotonicity of the $\mathit{attr}(A)$ operator (with respect to set inclusion) and the finiteness of the state space, there is a least fixpoint $\mathit{attr}^*(A)$, which is $\mathit{attr}^n(A) = \mathit{attr}^{n+1}(A)$ for some (smallest) n.

Now, let $G = \mathit{reach}(N) \setminus \mathit{attr}^*(B)$. These are the "good" reachable states in the sense that they are allowed by I *and* the system can be controlled to henceforth adhere to I.

Definition 13. *Let $R = \{(s,t) \in I \mid \exists s'\, s[t\rangle s' \wedge s, s' \in G\}$ be the* safe transition relation.

If the initial state is good (or, equivalently, G is nonempty), then the constructor can win by playing according to R. If, on the other hand, s_0 is in the attractor $\mathit{attr}^*(B)$ of the bad states, then s_0 is in $\mathit{attr}^n(B)$ for some $n \leq |S|$. But by the definition of $\mathit{attr}^n(B)$, the spoiler can force the game to $\mathit{attr}^{n-1}(B)$ in the next step, then to $\mathit{attr}^{n-2}(B)$, and so forth, and thus make sure the bad states are reached within n steps.

Lemma 3. *The constructor can force a win if, and only if, $s_0 \in G$.*

This game can obviously be evaluated quickly on the *explicit* game graph, and hence in time exponentially in the number of places. EXPTIME completeness can be demonstrated by a simple reduction from the PEEK-G_5 [21] game. An instance of PEEK-G_5 consists of two disjoint sets of Boolean variables, X (owned by a safety player) and Y (owned by a reachability player), a subset $Z \subseteq X \cup Y$ of them that are initially *true*, and a Boolean formula φ over $X \cup Y$ that the reachability player wants to become *true* eventually. The game is played in turns between the safety and the reachability player

[1] If $s[t\rangle$ and $s \not\in A$ then $(s,t) \in I$.

(say, with reachability player moving first), who in their turn can change the truth value of up to one of their variables.

To reduce the PEEK-G_5 game to the question if the initial state s_0 of a Petri Net is good, we introduce two places for each variable in $X \cup Y$, where a token is always in exactly one of them, one for true and one for false. Initially, the tokens are placed in accordance with Z. We also have introduce two control places, indicating that it is the reachability and safety player's move, respectively. (We again always have a token, called the control token, in exactly one of these places.) The invariant I represents the formula $\neg\varphi$.

For each variable $v \in X$ (resp. $v \in Y$), we have a transition that takes the control token from the safety (resp. reachability) place and the token from the *true*/*false* place of v, and puts the control token to the reachability (resp. safety) place and toggles the value of v by putting a token into the *false*/*true* place of v. We also transitions that only take the control token from the safety (resp. reachability) place and put it into the reachability (resp. safety) place. (In total, we have $2|X \cup Y| + 2$ transitions.) Exactly the $2|Y| + 1$ transitions that move the control token from the reachability to the safety place are uncontrollable. The initial state is the state encoding Z with the control token in the reachability place.

The constructor can play the role of the safety player in the PEEK-G_5 game by choosing a singleton set of transitions when the control pebble is in the safety place, while the spoiler can play the role of the reachability player by choosing among the enabled (uncontrollable) transitions when the control pebble is in the reachability place.

Lemma 4. *Deciding if the constructor can force a win is EXPTIME complete.*

In case all transitions are controllable, checking whether or not $s \in G$ (or $(s,t) \in R$) can be done in PSPACE using a binary search for an ultimately periodic cycle in $reach_I(N)$ that includes the state s (followed immediately by the transition t, respectively). For hardness, it is easy to reduce the halting problem of a deterministic space-bounded Turing machine to this decision problem by representing the tape explicitly.

Lemma 5. *Deciding if the constructor can force a win is PSPACE complete for Petri Nets with only controllable transitions.*

4 Knowledge Based Distributed Control

In control theory, the transformation that takes a system and allows blocking some transitions adds a supervisor process [17,24,19], which is usually an automaton that runs synchronously with the controlled system. This (finite state) automaton observes the controlled system, progresses according to the transitions it observes, and blocks some of the enabled transitions, depending on its current state [24]. This is often insufficient for obtaining distributed control [18]. We propose a control mechanism with supervisors that run asynchronously with the controlled processes.

In the following definitions, we can often use either the local information or the local state. When this is the case, we will use $s|_\Pi$ instead of either $s\lceil_\Pi$ or $s\lfloor_\Pi$.

Definition 14. *Let $\Pi \subseteq \mathcal{C}$ be a set of processes. Define an equivalence relation $\equiv_\Pi \subseteq reach(N) \times reach(N)$ such that $s \equiv_\Pi s'$ when $s|_\Pi = s'|_\Pi$.*

As $s|_\Pi$ can stand for either $s\lceil_\Pi$ or $s\lfloor_\Pi$, this gives two different equivalence relations. When it is important to distinguish between them, we denote the one based on "$\lceil$" as $\equiv^w_\Pi$ (weak equivalence) and the one based on "$\lfloor$" as $\equiv^s_\Pi$ (strong equivalence). It is easy to see that the enabledness of a transition depends only on the local information of a process that contains it, as stated in the following lemma.

Lemma 6. *If $t \in \pi$ and $s \equiv^w_\pi s'$ then $s[t\rangle$ if, and only if, $s'[t\rangle$.*

This lemma does not hold when we replace $\equiv^w_\pi$ by $\equiv^s_\pi$. In the Prioritized Petri Net in Figure 1(b), e.g., we have that $\{p_1, p_2\} \equiv^w_{\pi_l} \{p_1, p_4\}$, since in both states π_l has the same local information $\{p_1\}$. In the state $\{p_1, p_2\}$, a is a maximal priority enabled transition (incomparable with b), while in $\{p_1, p_4\}$, a is not maximal anymore, as we have that $a \ll d$, and both a and d are now enabled.

Definition 15. *The processes Π (jointly)* know *a property ψ in a state s, denoted $s \models K_\Pi \psi$, if, for each s' such that $s \equiv_\Pi s'$, we have that $s' \models \psi$.*

We distinguish knowledge based on strong equivalence $\equiv^s_\Pi$ (and hence on local states), denoted $K^s_\Pi \varphi$, from knowledge based on weak equivalence $\equiv^w_\Pi$ (and hence local information), denoted $K^w_\Pi \varphi$. Since there can be multiple local informations for a single local state, we have $K^s_\Pi \varphi \rightarrow K^w_\Pi \varphi$. Therefore the knowledge based on the local state (resp. local information) is called *s*trong (resp. *w*eak) knowledge.

In order to make choices (to execute a transition) that take into account knowledge based on local information, a process, or a set of processes, needs to have some guarantee that the local information will not be changed by other processes while it is making the decision. For a single process, this may be achieved by the underlying hardware. On the other hand, it is unreasonable to require such a guarantee for a set of processes. Thus, for decisions involving a set of processes, knowledge based on the joint local state is used instead.

The classical definition of knowledge is based on the relations $\equiv_\Pi$ over the reachable states. However, when using knowledge to control a system to satisfy a generalized invariant, one may calculate the equivalences and the knowledge based on the states that appear in the executions of the original system that satisfy this general invariant [2]. This (cyclic looking) claim is proved [2] by induction on the progress of the execution in the controlled system: the controlled system allows at any state to execute only transitions that preserve the generalized invariant. For the same argument, when our allowed transitions will be restricted according to the safe transition relation R, we can further restrict the states space used to calculate the knowledge to $G \subseteq reach(N)$.

The definition of knowledge that we use assumes that the processes do not maintain a log with their history. The use of knowledge with such a log, called *knowledge with perfect recall* [13], is discussed in [1]. It is shown there that updating such a log can require enormous complexity for each process. We henceforth use knowledge formulas combined with Boolean operators and propositions. For a detailed syntactic and semantic description of logics with knowledge one can refer, e.g., to [4]. Once $s \models K_\Pi \psi$ is defined, ψ can also be a knowledge property, hence $s \models K_{\Pi'} K_\Pi \psi$ (knowledge about knowledge) is also defined.

Lemma 7. *If $s \models K_\Pi \varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi \varphi$.*

Lemma 8. *Knowledge is monotonic: if $\Pi' \subseteq \Pi$ then $K_{\Pi'}\varphi \rightarrow K_\Pi \varphi$.*

We will use the following propositional formulas, with propositions that are the places of the Petri Net, to explain the approach and the implementation:

- The good states G: φ_G.
- The states where a transition t is enabled: $\varphi_{en(t)}$.
- At least one transition is enabled, i.e., there is no deadlock: $\varphi_{df} = \bigvee_{t \in T} \varphi_{en(t)}$.
- Transition t is allowed from the current state by the safe transition relation R: $\varphi_{good(t)}$
- The local information (resp. local state) of processes Π at state s: $\varphi_{s\lceil_\Pi}$ (resp. $\varphi_{s|_\Pi}$).

The corresponding sets of states can easily be computed by model checking and stored in a compact way, e.g., using BDDs.

Now, given a Petri Net, one can perform model checking in order to calculate whether $s \models K_\pi \psi$. The processes Π know ψ at state s exactly when $(\varphi_G \wedge \varphi_{s|_\Pi}) \rightarrow \psi$ is a propositional tautology. We can also check properties that include nested knowledge by simply checking first the innermost knowledge properties and marking the states with additional propositions for these innermost properties.

Model checking knowledge using BDDs, is *not* the most space efficient way of checking knowledge properties, since φ_G can be exponentially big in the size of the Petri Net. In a (polynomial) space efficient check (which has a higher *time* complexity), we enumerate all states s' such that $s \equiv_\pi s'$, check reachability of s' using binary search, and, if reachable, check whether $s' \models \psi$. This can apply also to nested knowledge formulas, where inner knowledge properties are recursively reevaluated each time they are needed. The PSPACE complexity is subsumed by the EXPTIME complexity in the general case algorithm for the safe transition relation R.

Definition 16. *An* extended Petri Net *[9] is a Petri Net with a finite set of variables V_π over a finite domain per each process $\pi \in \Pi$. In addition, a transition t can be augmented with a predicate en_t on the variables $V_t = \cup_{\pi \in proc(t)} V_\pi$ and a transformation function $f_t(V_t)$. In order for t to fire, en_t must hold in addition to the basic Petri Net enabling condition on the input and output places of t. When t fires, in addition to the usual changes to the tokens, the variables V_t are updated according to the transformation f_t.*

We transform a Petri Net N and a generalized invariant I into an extended Petri Net N' that allows only the executions of N controlled according to I.

Definition 17. *A* controlling *transformation obeys the following conditions:*

- *New transitions and places can be added.*
- *The input and output places of the new transitions are disjoint from the existing places.*
- *Variables, conditions and transformations, as in Definition 16 can be added to existing transitions.*
- *Existing transitions will remain with the same input and output places.*
- *It is not possible to fire from some point an infinite sequence of added transitions.*

The added transitions are grouped into new (supervisory) processes. The added variables will represent some knowledge-dependent finite memory for controlling the system, and some interprocess communication between the original processes and the added ones. Processes from the original net will have disjoint sets of variables from one another; intuitively, the independence between the original transitions is preserved by the transformation, although some coordination may be enforced indirectly through the interaction with the new supervisory processes.

As a natural extension of Definition 11, $s\lceil_C$ maps a state s of the transformed version N' into the places of the original version N by projecting out additional variables and places that N' may have on top of the places of N. In this way, we will be able to relate the sets of states of the original and transformed version. Firing of a transitions added by the controlling transformation does not change $s\lceil_C$ and is not considered to violate I (the requirement that (s_i,t_{i+1}) in Definition 6 is imposed only when t_{i+1} is from the original net N). Note that our restrictions on the transformation implies that the sets $ngb(\Pi)$ and $own(\Pi)$ for $\Pi \subseteq C$ are not affected by the transformation.

Definition 18. *Two sequences of states σ and σ' are equivalent up to stuttering [14] when, by replacing any finite adjacent repetition of the same state by a single occurrence in either σ or σ', we obtain the same sequence. Let $stutcl(\Gamma)$ be the stuttering closure of a set Γ of sequences, i.e., all sequences that are stuttering equivalent to some sequences in Γ.*

Lemma 9. *A controlling transformation produces an extended Petri Net N' from N such that $exec(N')\lceil_C \subseteq stutcl(exec(N))$.*

Lemma 10. *Given that $s \models K_\Pi\varphi$ in some basic Petri Net N, then $s \models K_\Pi\varphi$ also in a transformed version N'.*

5 Control Using Knowledge Accumulating Supervisors

According to the knowledge based approach to distributed control [1,6,2,18], model checking of knowledge properties is used at a preliminary stage to find when, depending on the local information, an enabled transition can be safely fired. In our case, this means checking $s \models K^w_\pi\varphi_{good(t)}$ (by Lemma 7, the satisfaction only depends on $s\lceil_\pi$). At runtime, a process *supports* a transition in every local information where this holds. The following *support policy* uses this information at runtime:

> A transition t can be fired (is enabled) in a state when, in addition to its original enabledness condition, at least one of the processes in $proc(t)$ supports it.

Lemma 11. *If $t \in \pi \cap uc(T)$ and $(s,t) \in R$, then $s \models K^w_\pi\varphi_{good(t)}$.*

This follows from the observation that the safe transition relation does not restrict the uncontrolled transition. This means that an uncontrolled transition can always be supported by any processes that contains it.

It is possible that, in some (non deadlock) states of G, no process has enough local knowledge to support an enabled transition and, furthermore, no uncontrollable transitions are enabled. We may count such global states as additional “bad states”, and repeat the strategy calculation to refine the state space further, as done in Section 3.

Still, there are cases where the local knowledge will not be sufficient, as in the priority example in Figure 1(b). Then, we may need to synchronize several processes to collect more knowledge. Then, a process can decide, based on its current knowledge, whether it needs to hang on a supervisor and send it its local state; the supervisor can make a decision, based on accumulated joined knowledge of several hung processes, that one of them can support an enabled transition. For example, in Figure 1(b), in the initial state the local information (and also the local state) of π_l is $\{p_1\}$. Thus, π_l does not have enough knowledge to support any transition. Similarly, the local information of π_r is $\{p_2\}$, which also is not sufficient to support any transition. After they both hang on a supervisor, it has enough information to support a or b.

To simplify the presentation, we will describe the solution in several steps. Each solution is an improvement over the previous solutions.

Solution 1. [Completely centralized; Waiting for all processes]

First, we assume that there is a single supervisor process $\mathcal{T}$, which is responsible for all processes. At this point, assume that no uncontrollable transitions are allowed. Dealing with them is deferred until Solution 3. A process hangs on a supervisor, when the following property *does not* hold:

$$\kappa_1^{\pi} = \bigvee_{t \in \pi} K_{\pi}^{w} \varphi_{good(t)}$$

When $\mathcal{T}$ sees that *all* processes are hanging on it, then it can acquire the complete global state from the individual processes. Since all processes are hung, none of them can progress. Each process then reports its local information to the supervisor $\mathcal{T}$. Now, $\mathcal{T}$ has enough information to instruct some process to support an enabled transition, according to the safe transition relation R. When a transition is supported and fired, all processes are freed from being hung.

This solution prevents new deadlocks in the system that did not occur under the original Petri Net: if no process has the local knowledge that is required to progress, no process π has the knowledge according to κ_1^{π}; eventually all processes will hang on $\mathcal{T}$, and $\mathcal{T}$ has the capability to break the deadlock.

Solution 2. [Supervisor can decide based on a subset of processes]

The supervisor $\mathcal{T}$ keeps the updated joint local state of the hung processes Π. When a process π hangs, it updates this view by transmitting to $\mathcal{T}$ its local information $s\lceil_{\pi}$, from which $\mathcal{T}$ keeps (according to Lemma 2) $s\lceil_{\pi} \cap own(\Pi \cup \{\pi\})$. Since all processes in $\Pi' = \Pi \cup \{\pi\}$ are now hung, no other process can change these places. Then the knowledge $K_{\Pi'}^{s} \varphi_{good(t)}$ can be used to support a transition t. Recall that knowledge based decisions of a single process use weak knowledge (based on the local information), while multiple processes use strong knowledge (i.e., based on the joint local state).

The supervisor process $\mathcal{T}$ can identify (precalculated) joint local states where enough knowledge is available to decide that a transition t is allowed by the safe transition relation R and make one of the processes in $proc(t)$ support it. The supervisor $\mathcal{T}$ does not need to wait for all processes to hang on it to make such a decision. Once t is fired, $\mathcal{T}$ frees all hung processes.

Solution 3. [Hanging processes may still progress]

Consider the following cases:

1. After the decision of a process π to hang on $\mathcal{T}$, other processes make changes to π's local information that allow it to support some transition t.
2. A transition t with $\{\pi, \pi'\} \subseteq proc(t)$ is supported by π' while π is hung.
3. An uncontrollable transition can always be executed, even if it belongs to a hung process.

In all of these cases, we allow π to notify $\mathcal{T}$ that it has decided not to hang on it anymore. Moreover, $\mathcal{T}$, which acquired information about the hung processes Π, will have to forget the information about the places $own(\Pi) \setminus own(\Pi \setminus \{\pi\})$.

We can now weaken the condition κ_1^π for a process not to hang into:

$$\kappa_2^\pi = \bigvee_{t \in \pi} K_\pi^w \varphi_{good(t)} \vee K_\pi^w \bigvee_{\pi' \neq \pi} \bigvee_{t \in \pi'} K_{\pi'}^w \varphi_{good(t)}$$

That is, a process does neither hang on the supervisor when it has enough knowledge to support a transition, nor if it knows that some other process has such knowledge. In the latter case, it does not actually need to distinguish which process has that knowledge.

The ability of processes to hang on a supervisor but also to progress independently before the supervisor has made any supporting choice requires some protocol between the processes and the supervisor. The α-core protocol [15,8] can be adopted here.

Solution 4. [Multiple supervisors]

Instead of having a single supervisor $\mathcal{T}$, we use several supervisors $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$. Each supervisor $\mathcal{T}_i$ takes care of a set of processes $proc(\mathcal{T}_i)$. These sets are pairwise disjoint and do not necessarily cover all the processes.

An effectively checkable criterion to determinte if at least one process or supervisor will be able to provide a progress from any nondeadlock state in G is as follows:

$$(\varphi_G \wedge \varphi_{df}) \rightarrow \big(\bigvee_{t \in \pi \in C} K_\pi^w \varphi_{good(t)} \vee \bigvee_{i \in 1 \ldots k} \bigvee_{t \in \pi \in proc(\mathcal{T}_i)} K_{proc(\mathcal{T}_i)}^s \varphi_{good(t)} \big)$$

It is interesting to compare Solution 3 with Solution 4. The former is more flexible in the sense that knowledge accumulated is not limited to a fixed partitioning of the processes. On the other hand, the latter solution allows true concurrency between transitions supported with the help of supervisors, whereas the former one interleaves the firing of such transitions. A solution that allows supervisors for a nondisjoint set of processes, which achieves the flexibility of Solution 3 *and* allows concurrent (independent) support of transitions by different supervisors as in Solution 4, is also possible. At first glance, this looks like gaining from both worlds; in practice, however, allowing processes to hang on *multiple* supervisors requires a large amount of overhead.

Lemma 12. *Under our transformations from a Petri Net N to an extended Petri Net N', $exec(N') \lceil_C \subseteq stutcl(exec_I(N))$ holds.*

This is proved by induction on prefixes of the execution and using Lemma 9.

Lemma 13. *N' satisfies all stuttering invariant temporal properties of N.*

This follows from the fact that our transformation does not introduce new deadlocks.

6 Implementing Supervisors

Processes hang on a supervisor in some arbitrary order. The supervisor needs to decide, based on the part of the global state that it sees, whether there is enough information to support some transition. We consider first the case of a single supervisor.

Definition 19. *Let $L = \{s\lfloor_\Pi \times \Pi \mid s \in G, \Pi \subseteq \mathcal{C}\}$ denote the set of joint local states, each paired up with the set of relevant processes (then $G \times \mathcal{C} \subseteq L$). We define $\sqsubseteq \subseteq L \times L$ (and, symmetrically, $\sqsupseteq$) as follows: $q \sqsubseteq q'$ if $q = (s\lfloor_{\Pi_1}, \Pi_1), q' = (s\lfloor_{\Pi_2}, \Pi_2)$ (i.e., both are part of the same global state s) and $\Pi_1 \subseteq \Pi_2$. We say that q'* subsumes *q.*

Definition 20. *The support function $supp : L \to 2^T$ returns, for each $q \in L$, the transitions that are allowed by R from all states that subsume q. Formally, $supp(q) = \bigcap_{(s,\mathcal{C}) \sqsupseteq q}\{t \mid t \in T, (s,t) \in R\}$.*

That is, for $q = (s\lfloor_\Pi, \Pi)$, $t \in supp(q)$ iff $s \models K^s_\Pi \varphi_{good(t)}$. If $t \in supp(q) \cap ct(T)$, then the supervisor can select a process in $proc(t)$ to support t. Obviously, when $q \sqsubseteq q'$, $supp(q) \subseteq supp(q')$. There is no need for a supervisor to store in the domain of *supp* elements $q = (s\lfloor_\Pi, \Pi)$ where $|\Pi| < 2$; in this case, when $supp(q) \neq \emptyset$, the process with that local state can locally support a transition without the help of a supervisor.

Definition 21. *Let $\rightsquigarrow \subseteq L \times L$ be such that $q \rightsquigarrow q'$ if $q = (s\lfloor_\Pi, \Pi)$ and $q' = (s\lfloor_{\Pi \cup \{\pi\}}, \Pi \cup \{\pi\})$, where $\pi \notin \Pi$ (i.e., q' extends q according to exactly one process).*

In the example from Figure 1(b), we have that $(\{p_1\}, \{\pi_l\}) \rightsquigarrow (\{p_1, p_2\}, \{\pi_l, \pi_r\})$ and $(\{p_2\}, \{\pi_r\}) \rightsquigarrow (\{p_1, p_2\}, \{\pi_l, \pi_r\})$. The supervisor updates its view about the joint local state of the processes according to the relation $\rightsquigarrow$: when moving from q to q' by acquiring the relevant information about a new processor π; consequently, its knowledge grows and it can decide to support one of the transitions in $supp(q')$.

Definition 22. *A joint local state q is* minimal supporting *if $supp(q) \neq \emptyset$ and, for each q' such that $q' \rightsquigarrow q$, $supp(q') = \emptyset$.*

Definition 23. *The* upward closure *$\uparrow U$ of a subset of the joint local states $U \subseteq L$ is $\{q \in L \mid \exists q' \in U\, q' \sqsubseteq q\}$.*

Because processes hang on a supervisor one by one, there is no need to calculate and store *all* the cases of the function *supp*.

Lemma 14. *A sufficient condition for restricting the domain $U \subseteq L$ of supp for a supervisor, without introducing new deadlocks, is that $G \times \{\mathcal{C}\} \subseteq \uparrow U$.*

This suggests the following algorithm for calculating the representation table for *supp*: perform DFS such that if $q \rightsquigarrow q'$, then q is searched before q'; backtrack when visiting q again, or when $supp(q) \neq \emptyset$. This algorithm can be used also for multiple supervisors, when restricting the search to the joint local states of $\Pi \subseteq proc(\mathcal{T}_i)$ for each $\mathcal{T}_i$.

In order to reduce the set of local states that a supervisor needs to keep in the support table, one may decide that a supervisor will not always support transitions as soon as the joint local state of the hung processes allows that. This introduces further delays

Table 1. Tests and Results

Processes	States	Global table size	Avg. num. of hung procs.	Duration (sec)
5	164	147	3.24	0.0002
8	3344	3296	4.06	0.2
9	9136	8449	4.37	1.3
10	24960	21371	4.65	6.8

in decisions, where the supervisor waits for more processes to hang even when it can already support some transitions. On the other hand, in this case the set of supported transitions may be larger, allowing more nondeterminism.

The size of the global state space of a Petri Net is $O(2^{|P|})$. Since we need to keep also the joint local states, the size of the support table that we store in a supervisor, is $O(2^{|P|+|C|})$ (which is the size of L). However, by Lemma 14, the representation may be much more succinct. In theory, when there are no uncontrollable transitions, a (particularly slow) supervisor can avoid storing the support table, and perform the PSPACE binary search each time it needs to make a decision on a joint local state.

7 Experimental Results

We implemented a prototype tool that allows to define Petri Nets with priorities and then simulate their executions according to Solution 3 described above. The tool uses model checking for building the local support tables for each process, as well as a central support table allowing the supervisor to base its support on accumulated knowledge. The tests were performed on a standard PC with Intel Core$^{\text{TM}}$2 Duo Processor and 2GB RAM.

In order to measure the efficiency of the suggested solution, the tests were repeated for several Petri Nets with various numbers of processes, places, and transitions. In each test, multiple random execution sequences of $10,000$ steps were generated, allowing to gather reliable performance statistics. Table 1 summarizes some tests and their results. The table presents, for each test, the number of global states, the size of the table used by the supervisor, the average number of hanged processes needed until the supervisor could support a transition, and the model checking duration.

8 Conclusions

We have developed a simple and effective algorithm for synthesizing distributed control. The resulting control strategy uses communication and knowledge collection without blocking the processes unnecessarily. One strength of our approach is that it is complete in the sense that, provided a centralized solution exists, it finds a solution. However, this does not come at the cost of centralizing the control completely. To the contrary, the system can progress without the support of a global or regional supervisor as soon as the local information suffices to do so.

Our solution for the distributed control of systems uses knowledge to construct a distributed controller for a global constraint. In [1,2], it is demonstrated that the local knowledge may be insufficient to construct a controller. Knowledge of perfect recall [13], which depends not only on the local state (information), but on the gathered visible history, can alleviate some, but not all, of these situations. The use of interprocess communication to obtain joint knowledge is suggested in [18]; however, no systematic algorithm for collecting such knowledge, or for evaluating when enough knowledge has been collected, was provided there. In [6], joint knowledge is calculated through temporary multiprocess synchronization. However, such synchronization is expensive, and multiple interactions (including different interactions of the same set of processes) may require a separate synchronizing process. We proposed here a practical solution for distribute control where a small number of (or even a single) supervisor(s) run(s) concurrently with the controlled system.

The algorithms in [1,2,6] guarantee a solution only under the assumption that the required constraint on a distributed system may not cause a deadlock (i.e., is nonblocking); accordingly, at the limit, one may synchronize all the processes and then make a decision how to progress given the global state. These solutions work, e.g., in the case of imposing priorities without uncontrollable transitions. However, when imposing the priorities when uncontrollable transitions are allowed, such control synthesis may fail: a state where the uncontrollable transitions have minimal priority among the enabled ones would be too late, and needs to be eliminated in advance. Our solution solves this problem by calculating the knowledge based on a reduced state space that avoids being blocked in such situations.

An interesting research direction is to deal with more general temporal properties. The difficulty lies with the need to use global memory for controlling the system and then distributing it according to the original architecture.

References

1. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
2. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge based controlling of distributed systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 52–66. Springer, Heidelberg (2010)
3. Clarke, E.M.: Synthesis of Resource Invariants for Concurrent Programs. ACM Transactions on Programming Languages and Systems 2(3), 338–358 (1980)
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
5. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: LICS 2005, Chicago, IL, pp. 321–330 (2005)
6. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control through Model Checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
7. Halpern, J.Y., Zuck, L.: A little knowledge goes a long way: knowledge based derivation and correctness proof for a family of protocols. Journal of the ACM 39(3), 449–478 (1992)

8. Katz, G., Peled, D.: Code Mutation in Verification and Automatic Code Correction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 435–450. Springer, Heidelberg (2010)
9. Keller, R.M.: Formal Verification of Parallel Programs. Communications of the ACM 19, 371–384 (1976)
10. Kupferman, O., Vardi, M.Y.: Synthesizing Distributed Systems. In: LICS 2001, Boston, MA (2001)
11. Madhusudan, P., Thiagarajan, P.S.: Distributed Controller Synthesis for Local Specifications. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001)
12. Manna, Z., Pnueli, A.: How to Cook a Temporal Proof System for Your Pet Language. In: POPL 1983, Austin, TX, pp. 141–154 (1983)
13. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140, 115–157 (1980)
14. Peled, D., Wilke, T.: Stutter-Invariant Temporal Properties are Expressible without the Text Time Operator. Information Processing Letters 63, 243–246 (1997)
15. Pérez, J.A., Corchuelo, R., Toro, M.: An Order-based Algorithm for Implementing Multiparty Synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
16. Pnueli, A., Rosner, R.: Distributed Reactive Systems are Hard to Synthesize. In: FOCS 1990, St. Louis, Missouri, pp. 746–757 (1990)
17. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25(1), 206–230 (1987)
18. Rudie, K., Ricker, S.L.: Know means no: Incorporating knowledge into discrete-event control systems. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
19. Rudie, K., Wonham, W.M.: Think globally, act locally: descentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
20. Schewe, S., Finkbeiner, B.: Synthesis of Asynchronous Systems. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 127–142. Springer, Heidelberg (2007)
21. Stockmeyer, L.J., Chandra, A.K.: Provably Difficult Combinatorial Games. SIAM Journal of Computing 8, 151–174 (1979)
22. Thistle, J.G.: Undecidability in Descentralized Supervision. Systems and Control Letters 54, 503–509 (2005)
23. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters 90(1), 21–28 (2004)
24. Yoo, T.S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dynamic Systems, Theory & Applications 12(3), 335–377 (2002)

Language Equivalence for Probabilistic Automata*

Stefan Kiefer[1], Andrzej S. Murawski[2], Joël Ouaknine[1],
Björn Wachter[1], and James Worrell[1]

[1] Department of Computer Science, University of Oxford, UK
[2] Department of Computer Science, University of Leicester, UK

Abstract. In this paper, we propose a new randomised algorithm for deciding language equivalence for probabilistic automata. This algorithm is based on polynomial identity testing and thus returns an answer with an error probability that can be made arbitrarily small. We implemented our algorithm, as well as deterministic algorithms of Tzeng and Doyen et al., optimised for running time whilst adequately handling issues of numerical stability. We conducted extensive benchmarking experiments, including the verification of randomised anonymity protocols, the outcome of which establishes that the randomised algorithm significantly outperforms the deterministic ones in a majority of our test cases. Finally, we also provide fine-grained analytical bounds on the complexity of these algorithms, accounting for the differences in performance.

1 Introduction

Probabilistic automata were introduced by Michael Rabin [16] as a variation on non-deterministic finite automata. In a probabilistic automaton the transition behaviour is determined by a stochastic matrix for each input letter. Probabilistic automata are one of a variety of probabilistic models, including Markov chains and Markov decision processes, that are of interest to the verification community. However a distinguishing feature of work on probabilistic automata is the emphasis on language theory and related notions of equivalence as opposed to temporal-logic model checking [1,4,8,12,13,14].

Probabilistic automata accept weighted languages: with each word one associates the probability with which it is accepted. In this context two automata $\mathcal{A}$ and $\mathcal{B}$ are said to be equivalent if each word is accepted with the same probability by both $\mathcal{A}$ and $\mathcal{B}$. It was shown by Tzeng [21] that equivalence for probabilistic automata is decidable in polynomial time, although this fact can also be extracted from results of Schützenberger [17] on minimising weighted automata. By contrast the natural analog of language inclusion—that $\mathcal{B}$ accepts each word with probability at least as great as $\mathcal{A}$—is undecidable [6] even for automata of fixed dimension [2].

* Research supported by EPSRC grant EP/G069158. The first author is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 526–540, 2011.

The motivation of Tzeng to study equivalence of probabilistic automata came from learning theory. Here our motivation is software verification as we continue to develop APEX [12], an automated equivalence checker for probabilistic imperative programs. APEX is able to verify contextual equivalence of probabilistic programs[1], which in turn can be used to express a broad range of interesting specifications.

APEX works by translating probabilistic programs into automata-theoretic representations of their game semantics [8]. The translation applies to open programs (i.e., programs with undefined components) and the resultant automata are highly abstracted forms of the program operational semantics. Intuitively, only externally observable computational steps are visible in the corresponding automata and internal actions such as local-variable manipulation remain hidden. Crucially, the translation relates (probabilistic) contextual equivalence with (probabilistic) language equivalence [14]. Accordingly, after two programs have been translated into corresponding probabilistic automata, we can apply any language equivalence testing routine to decide their contextual equivalence. In some of our case studies, we shall use automata generated by APEX for benchmarking the various algorithms discussed in the paper.

In conjunction with the ability of the APEX tool to control which parts of a given program are externally visible, the technique of verification by equivalence checking has proven surprisingly flexible. The technique is particularly suited to properties such as obliviousness and anonymity, which are difficult to formalise in temporal logic.

In this paper we develop a new randomised algorithm for checking equivalence of probabilistic automata (more generally of real-valued weighted automata). The complexity of the algorithm is $O(nm)$, where n is the number of states of the two automata being compared and m is the total number of labelled transitions. Tzeng [21] states the complexity of his algorithm as $O(|\Sigma| \cdot n^4)$; the same complexity bound is likewise claimed by [5,10] for variants of Tzeng's procedure. Here we observe that the procedure can be implemented with complexity $O(|\Sigma| \cdot n^3)$, which is still slower than the randomised algorithm.

This theoretical complexity improvement of the randomised algorithm over the deterministic algorithm is borne out in practice. We have tested the algorithms on automata generated by APEX based on an anonymity protocol and Herman's self-stabilisation protocol. We have also generated test cases for the equivalence algorithms by randomly performing certain structural transformations on automata. Our results suggest that the randomised algorithm is around ten times faster than the deterministic algorithm. Another feature of our experiments is that the performance of both the randomised and deterministic algorithms is affected by whether they are implemented using forward reachability from the initial states of the automata or using backward reachability from the accepting states of the automata.

[1] Two programs are contextually equivalent if and only if they can be used interchangeably in any context.

2 Algorithms for Equivalence Checking

Rabin's probabilistic automata [16] are *reactive* according to the taxonomy of Segala [19]: the transition function has type $Q \times \Sigma \to \mathrm{Dist}(Q)$, where Q is the set of states, Σ the alphabet and $\mathrm{Dist}(Q)$ the set of probability distributions on Q. Another type of probabilistic automata are the so-called *generative* automata in which the transition function has type $Q \to \mathrm{Dist}(\Sigma \times Q)$. The automata produced by APEX combine both reactive and generative states. In this paper we work with a notion of $\mathbb{R}$-*weighted automaton* which encompasses both reactive and generative transition modes as well as negative transition weights, which can be useful in finding minimal representations of the languages of probabilistic automata [17]. In the context of equivalence checking the generalisation from probabilistic automata to $\mathbb{R}$-weighted automata is completely innocuous.

Definition 1. *An* $\mathbb{R}$-weighted automaton $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$ *consists of a positive integer* $n \in \mathbb{N}$ *representing the number of states, a finite alphabet* Σ*, a map* $M : \Sigma \to \mathbb{R}^{n \times n}$ *assigning a transition matrix to each alphabet symbol, an initial (row) vector* $\alpha \in \mathbb{R}^n$*, and a final (column) vector* $\eta \in \mathbb{R}^n$*. The automaton* $\mathcal{A}$ *assigns each word* $w = \sigma_1 \cdots \sigma_k \in \Sigma^*$ *a* weight $\mathcal{A}(w) \in \mathbb{R}$*, where* $\mathcal{A}(w) := \alpha M(\sigma_1) \cdots M(\sigma_k)\eta$*. Two automata* $\mathcal{A}, \mathcal{B}$ *over the same alphabet* Σ *are said to be* equivalent *if* $\mathcal{A}(w) = \mathcal{B}(w)$ *for all* $w \in \Sigma^*$*.*

From now on, we fix automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$, and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$, and define $n := n^{(\mathcal{A})} + n^{(\mathcal{B})}$. We want to check $\mathcal{A}$ and $\mathcal{B}$ for equivalence. It is known that if $\mathcal{A}$ and $\mathcal{B}$ are not equivalent then there is a short word witnessing their inequivalence [15]:

Proposition 2. *If there exists a word* $w \in \Sigma^*$ *such that* $\mathcal{A}(w) \neq \mathcal{B}(w)$ *then there exists a word* w *of length at most* n *such that* $\mathcal{A}(w) \neq \mathcal{B}(w)$*.*

It follows immediately from Proposition 2 that the equivalence problem for weighted automata is in co-NP. In fact this problem can be decided in polynomial time. For instance, Tzeng's algorithm [21] explores the tree of possible input words and computes for each word w in the tree and for both automata the distribution of states after having read w. The tree is suitably pruned by eliminating nodes whose associated state distributions are linearly dependent on previously discovered state distributions. Tzeng's procedure has a runtime of $O(|\Sigma| \cdot n^4)$. Another algorithm was recently given in [10]. Although presented in different terms, their algorithm is fundamentally a backward version of Tzeng's algorithm and has the same complexity. (An advantage of a "backward" algorithm as in [10] is that repeated equivalence checks for different initial state distributions are cheaper, as results from the first equivalence check can be reused.) Variants of Tzeng's algorithm were also presented in [5] for checking linear-time equivalences on probabilistic labelled transition systems, including probabilistic trace equivalence and probabilistic failures equivalence. These variants all have run time $O(|\Sigma| \cdot n^4)$ as for Tzeng's algorithm. It has been pointed out in [7] that more efficient algorithms can actually be obtained using techniques from the 60s: One can – in linear time – combine $\mathcal{A}$ and $\mathcal{B}$ to form a weighted automaton that assigns each word w the weight $\mathcal{A}(w) - \mathcal{B}(w)$. An algorithm of Schützenberger [17] allows to compute a minimal equivalent automaton, which is the empty automaton if and only if $\mathcal{A}$ and $\mathcal{B}$ are equivalent. As sketched in [7], Schützenberger's algorithm runs in $O(|\Sigma| \cdot n^3)$, yielding an overall runtime of $O(|\Sigma| \cdot n^3)$ for checking equivalence.

2.1 Deterministic Algorithms

Since the focus of this paper is language equivalence and not minimisation, we have not implemented Schützenberger's algorithm. Instead we have managed to speed up the algorithms of [21] and [10] from $O(|\Sigma| \cdot n^4)$ to $O(|\Sigma| \cdot n^3)$ by efficient bookkeeping of the involved vector spaces. Whereas [7] suggests to maintain an LU decomposition to represent the involved vector spaces, we prefer a QR decomposition for stability reasons. We have implemented two versions of a deterministic algorithm for equivalence checking: a forward version (related to [21]) and a backward version (related to [10]). In the following we describe the backward algorithm; the forward version is obtained essentially by transposing the transition matrices and exchanging initial and final states.

Figure 1 shows the backward algorithm. The algorithm computes a basis Q of the set of state distributions

$$\left\{ \begin{pmatrix} M^{(\mathcal{A})}(\sigma_1) \ldots M^{(\mathcal{A})}(\sigma_k)\eta^{(\mathcal{A})} \\ M^{(\mathcal{B})}(\sigma_1) \ldots M^{(\mathcal{B})}(\sigma_k)\eta^{(\mathcal{B})} \end{pmatrix} \mid \sigma_1 \ldots \sigma_k \in \Sigma^* \right\}$$

represented by n-dimensional column vectors. It is clear that $\mathcal{A}$ and $\mathcal{B}$ are equivalent if and only if $\alpha^{(\mathcal{A})} u^{(\mathcal{A})} = \alpha^{(\mathcal{B})} u^{(\mathcal{B})}$ for all vectors $\begin{pmatrix} u^{(\mathcal{A})} \\ u^{(\mathcal{B})} \end{pmatrix}$ in Q.

Computing a basis Q of the set of state distributions requires a membership test to decide if a new vector is already included in the current vector space. While a naive membership test can be carried out in $O(n^3)$ time, maintaining the basis in a canonical form (here as an orthogonal set) admits a test in $O(n^2)$.

Given a set $Q = \{q_1, \ldots, q_k\}$ of $k \le n$ orthogonal vectors and another vector $u \in \mathbb{R}^n$, we denote by $u \downarrow Q$ the orthogonal projection of u along Q onto the space orthogonal to Q. Vector $u \in \mathbb{R}^n$ is included in the vector space given by Q, if u is equal to its projection into Q, i.e., the vector $u - u \downarrow Q$. Vector $u \downarrow Q$ can be computed by Gram-Schmidt iteration:

```
for $i$ from 1 to $k$
    $u := u - \frac{q_i^T u}{q_i^T q_i} q_i$
return $u$
```

Thereby q_i^T denotes the transpose of vector q_i. Note that this iteration takes time $O(nk)$.

The improvement from the $O(|\Sigma| \cdot n^4)$ bound for equivalence checking reported in [5,10,21] to $O(|\Sigma| \cdot n^3)$ arises from the fact that we keep the basis in canonical form which reduces the $O(n^3)$ complexity [5,10,21] for the membership test to $O(n^2)$. Since each iteration of the equivalence checking algorithm adds a new vector to Q or decreases the size of *worklist* by one, $|\Sigma| \cdot n$ iterations suffice for termination. Thus the overall complexity is $O(|\Sigma| \cdot n^3)$.

For efficiency, we consider an implementation based on finite-precision floating point numbers. Therefore we need to take care of numerical issues. To this end, we use the modified Gram-Schmidt iteration rather than the classical Gram-Schmidt iteration, as the former is numerically more stable, see [20].

To be more robust in presence of rounding errors, vectors are compared up to a relative error $\epsilon > 0$. Two vectors $v, v' \in \mathbb{R}^n$ are equal up to relative error $\epsilon \in \mathbb{R}$, denoted by $v \approx_\epsilon v'$, iff $\frac{\|v-v'\|}{\|v\|} < \epsilon$. The relative criterion is used to compare the weight of a word in automaton $\mathcal{A}$ and $\mathcal{B}$, and to check if a vector $u \in \mathbb{R}^n$ is included in a vector space given by a set of base vectors Q, i.e., $u \approx_\epsilon u - u \downarrow Q$.

Algorithm EQUIV$_{\leftarrow,\mathrm{det}}$

Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

```
if $\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$
    return "$\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} = \mathcal{A}(\varepsilon) \neq \mathcal{B}(\varepsilon) = \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$"
worklist:= $\{(\eta^{(\mathcal{A})}, \eta^{(\mathcal{B})}, \varepsilon)\}$
$\eta := \begin{pmatrix} \eta^{(\mathcal{A})} \\ \eta^{(\mathcal{B})} \end{pmatrix}$
if $\|\eta\| = 0$
    return "$\mathcal{A}$ and $\mathcal{B}$ are both empty and hence equivalent"
$Q := \{\eta\}$
while worklist $\neq \emptyset$
    choose and remove $(v^{(\mathcal{A})}, v^{(\mathcal{B})}, w)$ from worklist
    forall $\sigma \in \Sigma$
        $u^{(\mathcal{A})} := M^{(\mathcal{A})}(\sigma)v^{(\mathcal{A})}$; $u^{(\mathcal{B})} := M^{(\mathcal{B})}(\sigma)v^{(\mathcal{B})}$; $w' := \sigma w$
        if not $\alpha^{(\mathcal{A})}u^{(\mathcal{A})} \approx_\epsilon \alpha^{(\mathcal{B})}u^{(\mathcal{B})}$
            return "$\alpha^{(\mathcal{A})}u^{(\mathcal{A})} = \mathcal{A}(w') \neq \mathcal{B}(w') = \alpha^{(\mathcal{B})}u^{(\mathcal{B})}$"
        $u := \begin{pmatrix} u^{(\mathcal{A})} \\ u^{(\mathcal{B})} \end{pmatrix}$
        $q := u \downarrow Q$
        if not $u - q \approx_\epsilon u$
            add $(u^{(\mathcal{A})}, u^{(\mathcal{B})}, w')$ to worklist
            $Q := Q \cup \{q\}$
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent"
```

Fig. 1. Deterministic algorithm (backward version ←)

2.2 Randomised Algorithms

In this section we present a new randomised algorithm for deciding equivalence of $\mathbb{R}$-weighted automata. Recall from Proposition 2 that two $\mathbb{R}$-weighted automata with combined number of states n are equivalent if and only if they have the same *n-bounded language*, that is, they assign the same weight to all words of length at most n. Inspired by the work of Blum, Carter and Wegman on free Boolean graphs [3] we represent the *n-bounded language* of an automaton by a polynomial in which each monomial represents a word and the coefficient of the monomial represents the weight of the word. We thereby reduce the equivalence problem to polynomial identity testing, for which there is a number of efficient randomised procedures.

Let $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$ be two $\mathbb{R}$-weighted automata over the same alphabet Σ. Let $n := n^{(\mathcal{A})} + n^{(\mathcal{B})}$ be the total number of states of the two automata. We introduce a family of variables $\mathbf{x} = \{x_{\sigma,i} : \sigma \in \Sigma, 1 \leq i \leq n\}$ and associate the monomial $x_{\sigma_1,1}x_{\sigma_2,2}\dots x_{\sigma_k,k}$ with a word $w = \sigma_1\sigma_2\dots\sigma_k$ of length $k \leq n$. Then we define the polynomial $P^{(\mathcal{A})}(\mathbf{x})$ by

$$P^{(\mathcal{A})}(\mathbf{x}) := \sum_{k=0}^{n} \sum_{w \in \Sigma^k} \mathcal{A}(w) \cdot x_{\sigma_1,1}x_{\sigma_2,2}\dots x_{\sigma_k,k} \,. \tag{1}$$

The polynomial $P^{(\mathcal{B})}(\mathbf{x})$ is defined likewise, and it is immediate from Proposition 2 that $P^{(\mathcal{A})} \equiv P^{(\mathcal{B})}$ if and only if $\mathcal{A}$ and $\mathcal{B}$ have the same weighted language.

To test equivalence of $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ we select a value for each variable $x_{i,\sigma}$ independently and uniformly at random from a set of rationals of size Kn, for some constant K. Clearly if $P^{(\mathcal{A})} \equiv P^{(\mathcal{B})}$ then both polynomials will yield the same value. On the other hand, if $P^{(\mathcal{A})} \not\equiv P^{(\mathcal{B})}$ then the polynomials will yield different values with probability at least $(K-1)/K$ by the following result of De Millo and Lipton [9], Schwartz [18] and Zippel [22] and the fact that $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ both have degree n.

Theorem 3. *Let $\mathbb{F}$ be a field and $Q(x_1,\dots,x_n) \in \mathbb{F}[x_1,\dots,x_n]$ a multivariate polynomial of total degree d. Fix a finite set $\mathbb{S} \subseteq \mathbb{F}$, and let $r_1,\dots,r_n$ be chosen independently and uniformly at random from $\mathbb{S}$. Then*

$$\mathbf{Pr}[Q(r_1,\dots,r_n) = 0 \mid Q(x_1,\dots,x_n) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|} \,.$$

While the number of monomials in $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ is proportional to $|\Sigma|^n$, i.e., exponential in n, writing

$$P^{(\mathcal{A})}(\mathbf{x}) = \alpha^{(\mathcal{A})} \left(\sum_{i=0}^{n} \prod_{j=1}^{i} \sum_{\sigma \in \Sigma} x_{\sigma,j} \cdot M^{(\mathcal{A})}(\sigma) \right) \eta^{(\mathcal{A})} \tag{2}$$

it is clear that $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ can be evaluated on a particular set of numerical arguments in time polynomial in n. The formula (2) can be evaluated in a forward direction, starting with the initial state vector $\alpha^{(\mathcal{A})}$ and post-multiplying by the transition matrices, or in a backward direction, starting with the final state vector $\eta^{(\mathcal{A})}$ and pre-multiplying by the transition matrices. In either case we get a polynomial-time Monte-Carlo algorithm for testing equivalence of $\mathbb{R}$-automata. The backward variant is shown in Figure 2.

```
Algorithm EQUIV_{←,rand}
Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

if $\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$
    return "$\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} = \mathcal{A}(\varepsilon) \neq \mathcal{B}(\varepsilon) = \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$"
$v^{(\mathcal{A})} := \eta^{(\mathcal{A})}$; $v^{(\mathcal{B})} := \eta^{(\mathcal{B})}$
for $i$ from 1 to $n$ do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^\Sigma$
    $v^{(\mathcal{A})} := \sum_{\sigma \in \Sigma} r(\sigma) M^{(\mathcal{A})}(\sigma) v^{(\mathcal{A})}$
    $v^{(\mathcal{B})} := \sum_{\sigma \in \Sigma} r(\sigma) M^{(\mathcal{B})}(\sigma) v^{(\mathcal{B})}$
    if $\alpha^{(\mathcal{A})} v^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})} v^{(\mathcal{B})}$
        return "$\exists w$ with $|w| = i$ such that $\mathcal{A}(w) \neq \mathcal{B}(w)$"
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent with probability at least $(K-1)/K$"
```

Fig. 2. Randomised algorithm, backward version

```
Algorithm EQUIV_{←,rand}
Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

$v_0^{(\mathcal{A})} := \eta^{(\mathcal{A})}$; $v_0^{(\mathcal{B})} := \eta^{(\mathcal{B})}$;
for $i$ from 1 to $n$ do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^\Sigma$
    $v_i^{(\mathcal{A})} := \left(\sum_{\sigma \in \Sigma} r(\sigma) M^{(\mathcal{A})}(\sigma)\right) v_{i-1}^{(\mathcal{A})}$;
    $v_i^{(\mathcal{B})} := \left(\sum_{\sigma \in \Sigma} r(\sigma) M^{(\mathcal{B})}(\sigma)\right) v_{i-1}^{(\mathcal{B})}$;
    if $\alpha^{(\mathcal{A})} v_i^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})} v_i^{(\mathcal{B})}$
        $w := \varepsilon$;
        $u^{(\mathcal{A})} := \alpha^{(\mathcal{A})}$; $u^{(\mathcal{B})} := \alpha^{(\mathcal{B})}$;
        for $j$ from $i$ downto 1 do
            choose $\sigma \in \Sigma$ with $u^{(\mathcal{A})} M^{(\mathcal{A})}(\sigma) v_{j-1}^{(\mathcal{A})} \neq u^{(\mathcal{B})} M^{(\mathcal{B})}(\sigma) v_{j-1}^{(\mathcal{B})}$;
            $w := w\sigma$;
            $u^{(\mathcal{A})} := u^{(\mathcal{A})} M^{(\mathcal{A})}(\sigma)$;
            $u^{(\mathcal{B})} := u^{(\mathcal{B})} M^{(\mathcal{B})}(\sigma)$;
        return "$\alpha^{(\mathcal{A})} u^{(\mathcal{A})} = \mathcal{A}(w') \neq \mathcal{B}(w') = \alpha^{(\mathcal{B})} u^{(\mathcal{B})}$"
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent with probability at least $(K-1)/K$"
```

Fig. 3. Randomised algorithm with counterexamples, backward version

2.3 Runtime

The randomised algorithm is simpler to implement than the deterministic algorithm since there is no need to solve large systems of linear equations. While the deterministic algorithm, carefully implemented, runs in time $O(|\Sigma| \cdot n^3)$, the randomised algorithm runs in time $O(n \cdot |M|)$, where $|M|$ is the number of nonzero entries in all $M(\sigma)$, provided that sparse-matrix representations are used. In almost all of our case studies, below, the randomised algorithm outperforms the deterministic algorithm.

2.4 Extracting Counterexamples

We can obtain counterexamples from the randomised algorithm by exploiting a self-reducible structure of the equivalence problem. We generate counterexamples incrementally, starting with the empty string and using the randomised algorithm as an oracle to know at each stage what to choose as the next letter in our counterexample. Here the important thing is to avoid repeatedly running the randomised algorithm, which would negate the complexity advantages of this procedure over the deterministic algorithm. In fact this can all be made to work with some post-processing following a single run of the randomised procedure as we now explain.

Once again assume that $\mathcal{A}$ and $\mathcal{B}$ are weighted automata over the same alphabet Σ and with combined number of states n. To evaluate the polynomial $P^{(\mathcal{A})}$ we substitute a set of randomly chosen rational values $\mathbf{r} = \{r_{\sigma,i} : \sigma \in \Sigma, 1 \leq i \leq n\}$ into the expression (2). Here we will generalize this to a notion of *partial evaluation* $P_w^{(\mathcal{A})}(\mathbf{r})$ of polynomial $P^{(\mathcal{A})}$ with respect to values $\mathbf{r}$ and a word $w \in \Sigma^m$, $m \leq n$:

$$P_w^{(\mathcal{A})}(\mathbf{r}) = \alpha^{(\mathcal{A})} M^{(\mathcal{A})}(w) \left(\sum_{i=m}^{n} \prod_{j=m+1}^{n} \sum_{\sigma \in \Sigma} r_{\sigma,j} \cdot M^{(\mathcal{A})}(\sigma) \right) \eta^{(\mathcal{A})} \tag{3}$$

Notice that $P_\varepsilon^{(\mathcal{A})}(\mathbf{r}) = P^{(\mathcal{A})}(\mathbf{r})$ where ε is the empty word and, at the other extreme, $P_w^{(\mathcal{A})}(\mathbf{r}) = \mathcal{A}(w)$ for any word w of length n. We define $P_w^{(\mathcal{B})}$ similarly.

Proposition 4. *If $P_w^{(\mathcal{A})}(\mathbf{r}) \neq P_w^{(\mathcal{B})}(\mathbf{r})$ then either $\mathcal{A}(w) \neq \mathcal{B}(w)$ or $m < n$ and there exists $\sigma \in \Sigma$ with $P_{w\sigma}^{(\mathcal{A})}(\mathbf{r}) \neq P_{w\sigma}^{(\mathcal{B})}(\mathbf{r})$*

Proof. Considering the equation (3) and the corresponding statement for $\mathcal{B}$ the contrapositive of the proposition is obvious: if $\mathcal{A}(w) = \mathcal{B}(w)$ and $P_{w\sigma}^{(\mathcal{A})}(\mathbf{r}) \neq P_{w\sigma}^{(\mathcal{B})}(\mathbf{r})$ for each $\sigma \in \Sigma$ then $P_w^{(\mathcal{A})}(\mathbf{r}) = P_w^{(\mathcal{B})}(\mathbf{r})$.

From Proposition 4 it is clear that the algorithm in Figure 3 generates a counterexample trace given $\mathbf{r}$ such that $P^{(\mathcal{A})}(\mathbf{r}) \neq P^{(\mathcal{B})}(\mathbf{r})$.

3 Experiments

Implementation. To evaluate the deterministic and randomised algorithms, we have implemented an equivalence checker for probabilistic automata in C++ and tested their performance on a 3.07 GHz workstation with 48GB of memory. The forward and the backward algorithm have about 100 lines of code each, and the randomised one around 80, not counting supporting code. We use the Boost uBLAS library to carry out vector and matrix operations. Numbers are represented as double-precision floating point numbers. In order to benefit from the sparsity of vectors and transition matrices, we store them in a sparse representation which only stores non-zero entries.

Sparsity also plays a role in the Gram-Schmidt procedure because, in our context, the argument vector u often shares non-zeros with only a few base vectors. We have implemented an optimised version of Gram-Schmidt that only projects out base vectors that share non-zero positions with the argument. This can lead to dramatic speed-ups in the deterministic algorithms (the randomised algorithm does not use Gram-Schmidt).

3.1 Herman's Protocol

We consider Herman's self-stabilization protocol [11], in which N processes are arranged in a ring. Initially each process holds a token. The objective of Herman's protocol is to evolve the processes into a *stable* state, where only one process holds a token. The algorithm proceeds in rounds: in every round, each process holding a token tosses a coin to decide whether to keep the token or to pass it to its left neighbour; if a process holding a token receives an additional token, both of these tokens expire. We will be interested in the number of rounds that it takes to reach the stable state. The corresponding APEX model "announces" each new round by making calls to an undefined (external) procedure *round* inside the control loop of the protocol. Making the announcements at different program points within the loop (e.g. at the very beginning or the very end) results in different automata shown in Figure 4 (for $N = 3$).

A few remarks are due regarding our graphical representation of automata. Initial states are marked by a grey background. Accepting states are surrounded by a double border and labelled by a distribution over return values, e.g., the automaton on the left returns value 0 with a probability of $\frac{3}{4}$.

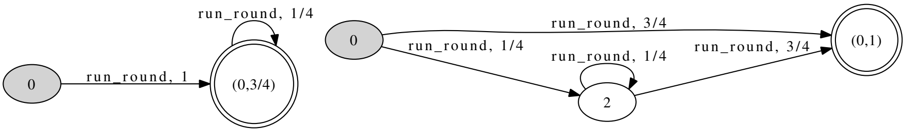


Fig. 4. Herman's protocol: automata for early (left) and late (right) announcement

These automata are structurally different and not bisimilar. However, intuitively, it should not matter at which position within the loop the announcement is made and therefore we check if the resulting automata are equivalent. Here is how the four algorithms perform in this case study.

	states		equivalence checking			
N	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
9	23	24	0.00	0.00	0.00	0.00
11	63	64	0.00	0.09	0.05	0.13
13	190	191	0.15	2.66	1.19	3.60
15	612	613	3.98	93.00	30.32	119.21

3.2 Grade Protocol

Assume that a group of students has been graded and would like to find out the sum of all their grades, e.g. to compute the average. However, none wants to reveal their individual grade in the process.

The task can be accomplished with the following randomised algorithm. Let $S \in \mathbb{N}$ be the number of students, and let $\{0, \cdots, G-1\}$ ($G \in \mathbb{N}$) be the set of grades. Define $N = (G-1) \cdot S + 1$. Further we assume that the students are arranged in a ring and that each pair of adjacent students shares a random integer between 0 and $N-1$. Thus a student shares a number l with the student on the left and a number r with the student on the right, respectively. Denoting the student's grade by g, the student announces the number $(g + l - r)\ mod\ N$. Because of the ring structure, each number will be reported twice, once as l and once as r, so the sum of all announcements (modulo N) will be the same as the sum of all grades. What we can verify is that only this sum can be gleaned from the announcements by a casual observer, i.e. the observer cannot gain access to any extra information about individual grades. This correctness condition can be formalised by a specification, in which the students make random announcements subject to the condition that their sum equals the sum of their grades.

Correct protocol. To check correctness of the Grade protocol, we check if the automaton resulting from the implementation of the protocol is equivalent to the specification automaton. Both automata are presented in Figure 5 for $G = 2$ and $S = 3$. Both automata accept words which are sequences of grades and announcements. The words are accepted with the same probability as that of producing the announcements for the given grades. Adherence to specification can then be verified via language equivalence.

In Figure 6 we report various data related to the performance of APEX and the equivalence checking algorithms. The reported times t_{spec} and t_{prot} are those needed to construct the automata corresponding to specification and the protocol respectively (we also give their sizes $s_{\text{spec}}, s_{\text{prot}}$). The columns labelled by $t_{\leftarrow,\text{rand}}, t_{\leftarrow,\text{det}}, t_{\rightarrow,\text{rand}}, t_{\rightarrow,\text{det}}$ give the running time of the language equivalence check of the respective automata using the four algorithms. The symbol $-$ means that the computation timed out after 10 minutes.

The running time of the deterministic backwards algorithm is higher than the one of the forward algorithm because the backwards algorithm constructs a different vector space with a higher number of dimensions.

Faulty protocol. Moving to test cases for inequivalence checking, we compare the specification for the Grade Protocol with a faulty version of the protocol in which the students draw numbers between 0 and $N-2$ (rather than $N-1$, as in the original protocol). The running times and automata sizes are recorded in Figure 7.

Both the deterministic and the randomised algorithms are decision algorithms for equivalence *and* can also generate counterexamples, which could serve, e.g., to repair a faulty protocol. We test counterexample generation on the faulty version of the protocol.

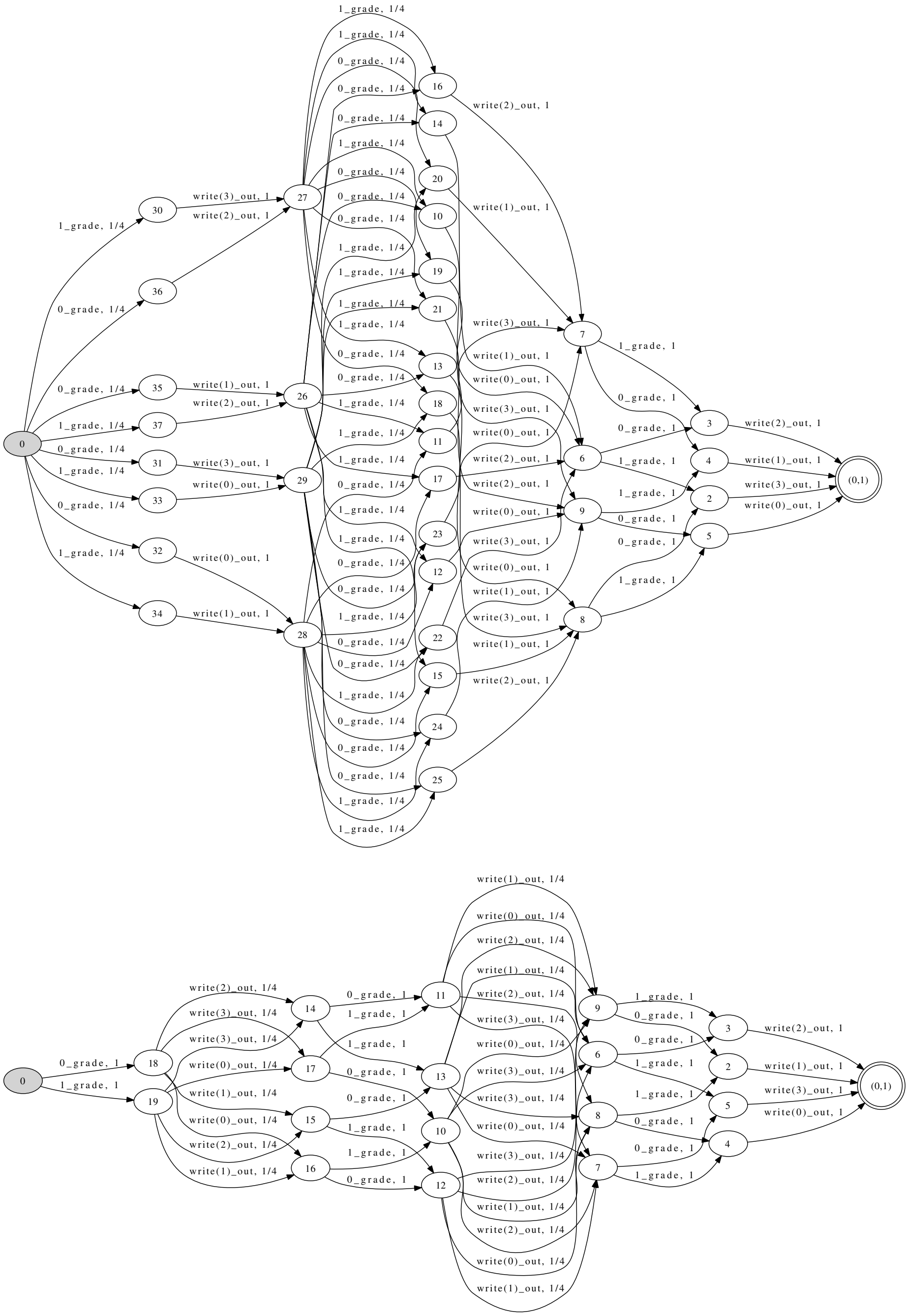


Fig. 5. Automata generated by APEX: protocol (top) and specification (bottom)

		APEX		states		equivalence checking			
G	S	t_{spec}	t_{prot}	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
2	10	0.00	0.02	202	1102	0.00	0.41	0.00	0.02
2	20	0.05	0.24	802	8402	0.10	26.09	0.03	0.26
2	50	1.75	6.46	5,002	127,502	8.08	-	1.21	10.07
2	100	25.36	82.26	20,002	1,010,002	240.97	-	10.56	159.39
2	200	396.55	-	80,002	-				
3	10	0.02	0.16	383	3803	0.02	5.79	0.01	0.09
3	20	0.22	1.64	1,563	31,203	0.69	598.78	0.23	1.33
3	50	6.99	39.27	9,903	495,003	61.76	-	9.29	63.16
3	100	102.42	-	39,803					
3	200	-	-						
4	10	0.04	0.56	564	8124	0.07	37.32	0.04	0.36
4	20	0.53	5.47	2,324	68,444	2.25	-	0.77	4.95
4	50	15.94	142.59	14,804	1,102,604	210.00	-	30.27	222.50
4	100	232.07	-	59,604					
4	200	-	-						
5	10	0.09	1.46	745	14,065	0.17	164.87	0.10	0.68
5	20	0.96	16.39	3,085	120,125	5.36	-	1.92	14.49
5	50	28.81	417.58	19,705	1,950,305	508.86	-	62.64	335.01
5	100	417.59	-	79,405					
5	200	-	-						

Fig. 6. Equivalence checks for the Grade Protocol

		APEX		states		equivalence checking					
G	S	t_{spec}	t_{prot}	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t^*_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t^*_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
2	10	0.00	0.10	202	8,845	0.02	0.02	6.71	0.01	0.01	0.26
2	20	0.05	2.24	802	151,285	1.85	1.86	-	0.12	0.14	11.28
2	50	1.75	170.40	5,002	6,120,205	-	-	-	17.38	18.60	-
3	10	0.00	0.10	383	63685	0.51	0.51	-	0.06	0.07	6.81
3	20	0.05	2.24	1,563	1,150,565	86.57	86.61	-	2.51	2.84	441
3	50	1.75	170.40	9,903	-						
4	10	0.00	0.10	564	207,125	8.04	8.04	-	0.34	0.35	60.57
4	20	0.05	2.24	2,324	3,816,145	-	-	-	17.11	17.31	-
4	50	1.75	170.40	14,804	-						
5	10	0.00	0.10	745	481,765	27.22	27.25	-	1.33	1.37	269.71
5	20	0.05	2.24	3,085	8,966,725	-	-	-	72.29	72.53	-
5	50	1.75	170.40	19,705	-						

Fig. 7. Inequivalence checks for the Grade Protocol

The deterministic algorithm stores words in the work list which immediately yield a counterexample in case of inequivalence. In the context of the randomised algorithm (as described in Section 2.4), counterexample generation requires extra work: words need to be reconstructed using matrix vector multiplications. To quantify the runtime overhead, we record running times with counterexample generation switched on and off.

The running time of the backward randomised algorithm with counterexample generation is denoted by $t^*_{\leftarrow,\text{rand}}$ and analogously for the forward algorithm. The runtime overhead of counterexample generation remains below 12%.

3.3 Randomised Inflation

Next we discuss two procedures that can be used to produce random instances of equivalent automata. We introduce two directional variants (by analogy to the forward and backward algorithms) that, given an input automaton, produce an equivalent automaton with one more state. Intuitively, equivalence is preserved because the new state is a linear combination of existing states, which are then suitably adjusted.

Backward Inflate. From a given automaton $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$, we construct an equivalent automaton $\mathcal{A}'$ with an additional state such that the automaton $\mathcal{A}'$ is not bisimilar to the original automaton $\mathcal{A}$. The automaton $\mathcal{A}' = (n+1, \Sigma, M', \alpha', \eta')$ has an additional state that corresponds to a linear combination of states in $\mathcal{A}$. To this end, the inflation procedure randomly picks several vectors:

- a single row vector $\delta \in \mathbb{R}^n$ which controls the transitions out of the new state; to obtain a substochastic transition matrix, we ensure $\delta \geq (0, \dots, 0)$ (where $\geq$ is meant componentwise) and $\delta(1, \dots, 1)^\top = 1$;
- column vectors $s(\sigma) \in \mathbb{R}^n$ for all $\sigma \in \Sigma$ that define the probability that flows into the new state; for probabilistic automata one should ensure $(0, \dots, 0)^\top \leq s(\sigma) \leq (1, \dots, 1)^\top$ and $s(\sigma)\delta \leq M(\sigma)$.

We define the transition matrices such that the new state $n+1$ is a linear combination of the old states, weighted according to δ. The transition matrix for $\sigma \in \Sigma$ is defined as folllows:

$$M'(\sigma) := \begin{pmatrix} M(\sigma) - s(\sigma)\delta & s(\sigma) \\ \delta M(\sigma) & 0 \end{pmatrix} \quad \text{and} \quad \alpha' := (\alpha \;\; 0) \quad \text{and} \quad \eta' := \begin{pmatrix} \eta \\ \delta\eta \end{pmatrix} .$$

Proposition 5. *$\mathcal{A}$ and $\mathcal{A}'$ are equivalent.*

Proof. For any word $w = \sigma_1 \cdots \sigma_k \in \Sigma^*$, define $c(w) := M(\sigma_1) \cdots M(\sigma_k)\eta$ and $c'(w) := M'(\sigma_1) \cdots M'(\sigma_k)\eta'$. We will show:

$$c'(w) = \begin{pmatrix} c(w) \\ \delta c(w) \end{pmatrix} \quad \text{for all } w \in \Sigma^*. \tag{4}$$

Equation (4) implies the proposition, because $\mathcal{A}'(w) = \alpha' c'(w) \overset{(4)}{=} \alpha c(w) = \mathcal{A}(w)$. We prove (4) by induction on the length of w. For the induction base, we have $c'(\varepsilon) = \eta' = \begin{pmatrix} \eta \\ \delta\eta \end{pmatrix} = \begin{pmatrix} c(\varepsilon) \\ \delta c(\varepsilon) \end{pmatrix}$. For the induction step, assume (4) holds for w. Let $\sigma \in \Sigma$. Then we have

$$\begin{aligned} c'(\sigma w) = M'(\sigma) c'(w) &= \begin{pmatrix} M(\sigma) - s(\sigma)\delta & s(\sigma) \\ \delta M(\sigma) & 0 \end{pmatrix} \begin{pmatrix} c(w) \\ \delta c(w) \end{pmatrix} \\ &= \begin{pmatrix} M(\sigma) c(w) \\ \delta M(\sigma) c(w) \end{pmatrix} = \begin{pmatrix} c(\sigma w) \\ \delta c(\sigma w) \end{pmatrix} , \end{aligned}$$

completing the induction proof. □

Forward Inflate. Forward Inflate is a variant obtained by "transposing" Backward Inflate. In this variant it is hard to keep the matrices stochastic. However, nonnegativity can be preserved. Construct an automaton $\mathcal{A}' = (n+1, \Sigma, M', \alpha', \eta')$ as follows. Pick randomly:

- a column vector $\delta \in \mathbb{R}^n$;
- row vectors $s(\sigma) \in \mathbb{R}^n$ for all $\sigma \in \Sigma$; for nonnegative automata one should ensure $\delta s(\sigma) \leq M(\sigma)$.

Define

$$M'(\sigma) := \begin{pmatrix} M(\sigma) - \delta s(\sigma) & M(\sigma)\delta \\ s(\sigma) & 0 \end{pmatrix} \quad \text{and} \quad \alpha' := (\alpha \;\; \alpha\delta) \quad \text{and} \quad \eta' := \begin{pmatrix} \eta \\ 0 \end{pmatrix} .$$

Proposition 6. *$\mathcal{A}$ and $\mathcal{A}'$ are equivalent.*

The proof is analogous to that of Proposition 5.

We have conducted experiments in which we compare automata with their multiply inflated versions. Our initial automaton is the doubly-linked ring of size 10, i.e. the states are $\{0, \cdots, 10\}$, there are two letters l and r such that the probability of moving from i to $(i+1)\,\mathit{mod}\,10$ on r is 0.5 and the probability of moving from i to $(i-1)\,\mathit{mod}\,10$ on l is 0.5 (otherwise the transition probabilities are equal to 0). The table below lists times needed to verify the equivalence of the ring with its backward inflations.

#inflations	$t_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$
1	0.001282	0.00209
5	0.001863	0.004165
10	0.002986	0.008560
20	0.003256	0.007932
50	0.009647	0.058917
100	0.045588	0.306103
200	0.276887	2.014591
500	3.767168	29.578838

4 Conclusion

This paper has presented equivalence checking algorithms for probabilistic automata. We have modified Tzeng's original deterministic algorithm [21]: by keeping an orthogonal basis of the underlying vector spaces, we have managed to improve the original quartic runtime complexity to a cubic bound. Further, we have presented a novel randomised algorithm with an even lower theoretical complexity for sparse automata. Our randomised algorithm is based on polynomial identity testing. The randomised algorithm may report two inequalent automata to be equivalent. However, its error probability, established by a theorem of Schwarz and Zippel, can be made arbitrarily small by increasing the size of the sample set and by repeating the algorithm with different random seeds. The randomised algorithm outperforms the deterministic algorithm on a wide range of benchmarks. Confirming the theoretical error probabilities the algorithm has detected all inequivalent automata and was able to generate suitable counterexample words.

References

1. Baier, C., Bertrand, N., Größer, M.: Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems. In: DCFS. EPTCS, vol. 3, pp. 3–16 (2009)
2. Blondel, V.D., Canterini, V.: Undecidable problems for probabilistic automata of fixed dimension. Theoretical Computer Science 36(3), 231–245 (2003)
3. Blum, M., Chandra, A., Wegman, M.: Equivalence of free boolean graphs can be decided probabilistically in polynomial time. Inf. Process. Lett. 10(2), 80–82 (1980)
4. Chatterjee, K., Doyen, L., Henzinger, T.A.: Probabilistic weighted automata. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 244–258. Springer, Heidelberg (2009)
5. Christoff, L., Christoff, I.: Efficient algorithms for verification of equivalences for probabilistic processes. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 310–321. Springer, Heidelberg (1992)
6. Condon, A., Lipton, R.: On the complexity of space bounded interactive proofs (extended abstract). In: Proceedings of FOCS, pp. 462–467 (1989)
7. Cortes, C., Mohri, M., Rastogi, A.: On the computation of some standard distances between probabilistic automata. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 137–149. Springer, Heidelberg (2006)
8. Danos, V., Harmer, R.: Probabilistic game semantics. ACM Transactions on Computational Logic 3(3), 359–382 (2002)
9. DeMillo, R., Lipton, R.: A probabilistic remark on algebraic program testing. Inf. Process. Lett. 7(4), 193–195 (1978)
10. Doyen, L., Henzinger, T., Raskin, J.-F.: Equivalence of labeled Markov chains. International Journal of Foundations of Computer Science 19(3), 549–563 (2008)
11. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. 35(2), 63–67 (1990)
12. Legay, A., Murawski, A.S., Ouaknine, J., Worrell, J.B.: On automated verification of probabilistic programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
13. Lynch, N.A., Segala, R., Vaandrager, F.W.: Compositionality for Probabilistic Automata. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 208–221. Springer, Heidelberg (2003)
14. Murawski, A.S., Ouaknine, J.: On probabilistic program equivalence and refinement. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 156–170. Springer, Heidelberg (2005)
15. Paz, A.: Introduction to Probabilistic Automata. Academic Press, London (1971)
16. Rabin, M.O.: Probabilistic automata. Information and Control 6(3), 230–245 (1963)
17. Schützenberger, M.-P.: On the definition of a family of automata. Inf. and Control 4, 245–270 (1961)
18. Schwartz, J.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM 27(4), 701–717 (1980)
19. Segala, R.: Modeling and verification of randomized distributed real -time systems. Technical report, MIT, Cambridge MA, USA (1996)
20. Trefethen, L.N., Bau, D.: Numerical Linear Algebra. SIAM, Philadelphia (1997)
21. Tzeng, W.: A polynomial-time algorithm for the equivalence of probabilistic automata. SIAM Journal on Computing 21(2), 216–227 (1992)
22. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, K.W. (ed.) EUROSAM 1979 and ISSAC 1979. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979)

Formalization and Automated Verification of RESTful Behavior

Uri Klein[1] and Kedar S. Namjoshi[2]

[1] Courant Institute of Mathematical Sciences, New York University
uriklein@courant.nyu.edu
[2] Bell Labs, Alcatel-Lucent
kedar@research.bell-labs.com

Abstract. REST is a software architectural style used for the design of highly scalable web applications. Interest in REST has grown rapidly over the past decade, spurred by the growth of open web APIs. On the other hand, there is also considerable confusion surrounding REST: many examples of supposedly RESTful APIs violate key REST constraints. We show that the constraints of REST and of RESTful HTTP can be precisely formulated within temporal logic. This leads to methods for model checking and run-time verification of RESTful behavior. We formulate several relevant verification questions and analyze their complexity.

1 Introduction

REST – an acronym for Representational State Transfer – is a software architectural style that is used for the creation of highly scalable web applications. It was formulated by Roy Fielding in [8]. The REST style provides a uniform mechanism for access to resources, thereby simplifying the development of web applications. Its structure ensures effective use of the Internet, in particular of intermediaries such as caches and proxies, resulting in fast access to applications. Over the past decade, interest in REST has increased rapidly, and it has become the desired standard for the development of large-scale web applications. The flip side to this is a considerable confusion over the principles of RESTful design, which are often misunderstood and mis-applied. This results in applications that are functionally correct, but which do not achieve the full benefits of flexibility and scalability that are possible with REST. Fielding has criticized the design of several applications which claim to be RESTful, among those are the photo-sharing application Flickr [9] and the social networking API SocialSite [10].

The criticisms show that some of the confusion is between REST and the Hypertext Transfer Protocol (HTTP) [7]. (Aside: Fielding is also a co-author of the HTTP RFC.) While RESTful applications are implemented using HTTP, not every HTTP-based application is RESTful, and not every RESTful application must use HTTP: REST is an architectural style, while HTTP is a networking protocol. Another common mistake is to call a application RESTful if it uses simpler encodings than those in the Remote Procedure Call (RPC) based SOAP/WSDL [26] mechanism. The distinction goes far beyond this superficial difference. These and other, more subtle, confusions motivate our work.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 541–556, 2011.

A question which arises naturally is whether it is possible to automatically check an application for conformance to REST. Doing so requires a precise specification of REST. In this paper, we address both questions. A formal characterization of REST has benefit beyond its use in automated analysis. It should also result in clear and effective communication about REST, and can enable deeper analysis of this elegant and effective architectural style.

We begin by formulating RESTful behavior in a general setting. A key contribution is to show that REST can be formalized within temporal logic. Two constraints define RESTful behavior. One, statelessness, is a branching-time property. The other, hypertext-driven behavior, is expressible in linear temporal logic. Both are safety properties. We then consider the common case of RESTful HTTP, and discuss how HTTP induces variants of the temporal properties.

The temporal specifications may be applied in several ways for verifying that a client-server application is RESTful. One is to model-check a fixed instance of the application [4,23]. The parameterized model checking question is also of much interest, as web applications typically handle a large number of clients. These questions presume a 'white-box' situation, where implementation code is available for analysis. A second group of questions concern run-time checking of RESTful behavior, a 'black-box' approach, where the only observable is the client-server communication. A third group of questions concern the synthesis of servers which meet a specification under RESTful constraints.

We show that, for a fixed instance, model-checking statelessness can be done in time that is linear in the size of the state-space of the instance and polynomial in the number of resources. On the other hand, checking that an instance satisfies a specification assuming hypertext-driven client behavior is PSPACE-complete in the number of resources. This property can be checked at run-time, however, in time that is polynomial in the number of clients and resources. We show decidability for parameterized model-checking under certain assumptions; the general case remains open. The full version of this paper [15] has complete proofs of theorems and further detail on RESTful HTTP.

2 REST and its Formalization

Our goal in the formalization is to stay as close as is possible to its description by Fielding in [8], which should be consulted for the rationale behind REST.

2.1 Building Blocks for Rest

REST is built around a client-server model which includes intermediate components, such as proxies and caches. An application is structured as a (conceptually) single *server component* (server, for short) and a number of *client components* (clients). All relevant communication is between a client and the server. Each *request* for a service is sent by a client to the server, which may either reject the request or perform it, returning a *response* in either case to the client. A server manages access to *resources.* A resource is an abstract unit of information with an intended meaning. Examples are a data file, a temporal service (e.g., 'current time in France'), or a collection of other resources (e.g., 'all files in a directory').

An *entity* describes the value of a resource at a given time; it can be viewed as the *state* of a resource. A resource state may be constant (e.g., 'Uri's birth date') or changing (e.g., 'current time in France'), but it must take on values which correspond to the intended meaning of the resource. A state may contain both uninterpreted data and links to other resources. This creates a 'Linked Data' view [3] of all the information under the control of an application. A *resource identifier* (resource id, for short) is a name by which a resource is identified. The mapping of names to resources is fixed and unique. In HTTP-based applications, Uniform Resource Identifiers (URIs) [27] are the resource identifiers. A *resource representation* is a description of the state of the resource at a given time. A state may have multiple representations (e.g., a web page may be represented as HTML, or by an image of its content).

A RESTful architecture has a fixed set of uniform *methods*. Hence, every application following that architecture must be based on these methods, which effectively decouples interface from implementation. In contrast, for an abstract data type or RPC model, the method set is unconstrained. Properties of a method, such as *safety* (no invocation changes server state) and *idempotence* (repeated invocation does not change server state) are required to hold uniformly, i.e., for all instantiations of the method.

2.2 Formalizing Resource-Based Applications

A resource-based application is one that is organized in terms of the previously described building blocks, which are formally defined by a *resource structure*: a tuple $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, where R is a set of resources; I is a set of resource identifiers; $B \subseteq I$, is a **finite** set of *root identifiers*; $\eta : I \mapsto R$ is a *naming function*, mapping identifiers to resources, a partial function that is injective on its domain; C is a set of *client identifiers*; D is a set of *data values*, with an *equivalence* relation $\sim \subseteq (D \times D)$; OPS is a finite set of methods; and $RETS$ is a finite set of *return codes*.

For simplicity, we use a specific form of resource representation, a pair $\langle ids; d \rangle$ in $2^I \times D$. Here, ids is a set of resource identifiers, and d a piece of data. This abstracts from HTML or XML syntax and formatting, and clearly separates resource identifiers from data values. The relation '$\sim$' may be used to ignore irrelevant portions of data, such as counters or timestamps. We extend it to resource representations as $\langle ids_1; d_1 \rangle \sim \langle ids_2; d_2 \rangle$ **iff** $ids_1 = ids_2$ and $d_1 \sim d_2$.

A *client-server communication* (a communication, for short) is represented by a 'request/response' pair, with the syntax $c{::}op(i, args)/rc(rvals)$, where: $c \in C$ is a client identifier; $op \in OPS$ is a method; $i \in I$, is a *target resource identifier*; *args* is a finite list of *arguments*; $rc \in RETS$ is a return code; and *rvals* is a finite list of *return values*. The arguments and return values are specific to the method. Both may include resource identifiers, data values, and resource representations. (We omit more complex data types for simplicity.)

With each communication m are associated two *disjoint* sets of resource identifiers, denoted $L(m)$ (linked) and $UL(m)$ (unlinked). The set $L(m)$ describes resources that are made known to the requesting client, and includes resource identifiers which are returned as results in the communication, or that are created by it. The set $UL(m)$ are identifiers which are revoked at the client.

Given a resource structure RS, a *RS-family* is a collection of client and server processes, defined over elements of RS. A *RS-instance* is a specific choice of clients and a single server from a RS-family, with the processes interacting using CCS-style synchronization [17] on communications. A *global state* of a RS-instance is given by a tuple with a local state for the server process and a local state for each client process. A *computation* is an alternating sequence of global states and *actions*, where an action is either a (synchronized) communication between a client and the server, or an internal process transition.

Caveats: In reality, requests and responses are independent events, which allows the processing of concurrent requests to overlap in time. The issue is discussed further in Section 4, as treating it directly considerably complicates the model. There is also an implicit assumption that methods have immediate effects. In practice, (e.g., HTTP DELETE) a server may return a response but postpone the effect of a request. This issue is discussed in Subsection 3.2.

A *communication sequence* σ is a (possibly infinite) sequence of communications carried out between a set of clients and the server. The *projection* of a communication sequence σ on a client c, written $\sigma|_c$, is the sub-sequence of σ which contains only those communications initiated by client c. A computation of a RS-instance *induces* a communication sequence given by the sequence of actions along that computation.

It is important to distinguish between the case where a method is successfully processed by the server, and where it is rejected without any server state change. This is done by mapping return codes to the abstract values $\{\mathsf{OK}, \mathsf{ERROR}\}$, where OK represents the first case and ERROR the second.

For a finite communication sequence σ, the set $assoc(\sigma)$ of resource identifiers defines those resources 'known' at the end of σ. For the empty communication sequence, $assoc(\lambda) = B$. Inductively, $assoc(\sigma; m)$ is $(assoc(\sigma) \cup L(m)) \setminus UL(m)$, if m has return code OK, and it is $assoc(\sigma)$, if the return code is ERROR.

For a finite computation with induced communication sequence σ, $assoc(\sigma)$ and $I \setminus assoc(\sigma)$ define the *associated* and *dissociated* resource identifiers, respectively. We associate a partial function $deref : I \mapsto 2^I \times D$ with the state of the server; $deref(i)$, if defined, is the current representation of the resource $\eta(i)$ (which must be defined if $deref(i)$ is defined).

2.3 Formalization of RESTful Behavior

For this section, fix a structure $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, and consider RS-instances. The two temporal properties discussed below define whether the behavior of an RS-instance is RESTful. It is usually more convenient to describe the failure cases, and also more helpful for the purpose of automatic verification. In the temporal formulas, we use a modified next-time operator,

$\mathsf{X}_{\langle a \rangle}$, where a is an action. Its semantics is defined on a sequence with atomic propositions on each state and an action label on each transition. For a sequence σ and position i, define $\sigma, i \models \mathsf{X}_a(\phi)$ to hold if $\sigma, i+1 \models \phi$ and the transition from step i to step $i+1$ is labeled with a.

Before diving into the specifics, it is worthwhile to point out a couple of important considerations. First, as in any formalization of a hitherto informal concept, there may be subtle differences between an informal idea and its formalization; we point out those that we are aware of. Second, a large part of the usefulness of a formalization lies in the testability of these properties. It is helpful to make a distinction between formal properties which can be tested given complete information of the implementation of clients and the server (a 'white-box' view), and those which can be tested only on the observable sequences of interaction between clients and the server (a 'black-box' view). The first viewpoint is interesting for model-checking; the second for run-time verification. Since we are targeting both approaches, we present the properties from both points of view, making it clear if one leads to a weaker test than the other. This distinction is important only for the safety and idempotence properties.

1. Stateless Behavior. In ([8], Chapter 5), this property is described as follows: *"... each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."* We formalize it by requiring that the server response to a request be functional; i.e., independent of client history or identity. (A 'client' should be understood to be a machine, rather than a user.) Failure of statelessness is shown by a finite computation followed by a two-way fork, where for some **distinct** client identifiers c, d, one branch of the fork contains the communication $c{::}op(i,a)/r_1(v_1)$, and the other branch contains the communication $d{::}op(i,a)/r_2(v_2)$, and either $r_1 \neq r_2$, or $v_1 \neq v_2$. This failure specification captures the situation where, given an identical history, the same method carried out by different clients has distinct results.

This is a branching-time property. The failure case is expressed as follows in a slight modification of Computation Tree Logic (CTL) [5], which allows the operator $\mathsf{EX}_{\langle a \rangle}$, for an action a.

$$(\exists c, d\ \exists i, op, a, r_1, v_1, r_2, v_2 : c \neq d\ \wedge\ (r_1 \neq r_2\ \vee\ v_1 \neq v_2) \wedge$$
$$\mathsf{EF}(\mathsf{EX}_{\langle c::op(i,a)/r_1(v_1)\rangle}(true)\ \wedge\ \mathsf{EX}_{\langle d::op(i,a)/r_2(v_2)\rangle}(true)))$$

The property suffices to detect the common cases of hidden per-client state. One subtlety is that the the property is based on *observable*, semantic effects of a hidden state, not its syntactic presence. Hence, it holds of a server which retains auxiliary per-client information – such as a request counter – but does not use that information to influence the response to a request.

The formalization is also slightly stronger than the intended informal notion of statelessness, in the following sense. Consider a server which implements a method as "`if (client=c) then return 3 else return 4`". This has no hidden state, yet the method has different results for distinct clients c and d, and fails the property.

2. Hypertext-driven behavior. Informally, this property requires a client to access a resource only by 'navigating' to it from a root identifier. It is also referred to by the acronym HATEOAS, which stands for "Hypertext/Hypermedia As The Engine Of Application State". The failure specification is a finite computation with induced communication sequence of the form $\sigma; c{::}op(\ldots, i, \ldots)/rc(\ldots)$, for some σ, return code rc, method op, and resource identifier i among the arguments of op, such that all of the following hold: $i \notin assoc(\sigma|_c)$, and if L is the linked set of the last communication, then $i \notin L$. The return code and values are not important. It suffices that the identifier i is currently not associated from the perspective of the client c.

This condition can be expressed in Linear Temporal Logic (LTL) [20], most conveniently by using past temporal operators [16] to express the condition $i \notin assoc(\sigma|_c)$. The past-LTL formula for failure, denoted φ_{HT}, can be built up as shown below.

In the following, the predicate $by(m, c)$ is true if communication m is by client c; $\mathsf{OK}(m)$ is true if m has return code OK; $arg(m, i)$ is true if resource id i is an argument to the request in m; Y is the 1-step predecessor operator with variant $\mathsf{Y}_{\langle a\rangle}$ (formally, $\sigma, i \models \mathsf{Y}_{\langle a\rangle}(\phi)$ if $(i \geq 1)$ and $\sigma, (i-1) \models \phi$, and the transition from step i to step $i+1$ is labeled by a); and $p\ \mathsf{S}\ q$ is the 'since' operator which holds if q holds in the past, and p holds since then. Precisely, $\sigma, i \models p\ \mathsf{S}\ q$ **iff** $(\exists k : 0 \leq k \leq i : \sigma, k \models q\ \wedge\ (\forall j : k < j\ \wedge\ j \leq i : \sigma, j \models p))$. Note that $\neg Y(true)$ is true only at the initial state of a sequence.

$$
\begin{aligned}
\varphi_{HT} =& (\exists c, i : \mathsf{F}(access(c,i)\ \wedge\ \neg\ inassoc(c,i))),\ \text{where} \\
access(c,i) =& (\exists m : \mathsf{X}_{\langle m\rangle}(true)\ \wedge\ by(m,c)\ \wedge\ arg(m,i)\ \wedge\ i \notin L(m)),\ \text{and} \\
inassoc(c,i) =& (\neg\ revoked(c,i))\ \mathsf{S}\ granted(c,i),\ \text{where} \\
revoked(c,i) =& (\exists m : \mathsf{Y}_{\langle m\rangle}(true)\ \wedge\ \mathsf{OK}(m)\ \wedge\ by(m,c)\ \wedge\ i \in UL(m)),\ \text{and} \\
granted(c,i) =& (\exists m : \mathsf{Y}_{\langle m\rangle}(true)\ \wedge\ \mathsf{OK}(m)\ \wedge\ by(m,c)\ \wedge\ i \in L(m))\ \vee \\
& (\neg \mathsf{Y}(true)\ \wedge\ i \in B)
\end{aligned}
$$

3. Safety and idempotence. REST explicitly includes intermediaries in the model, such as caches and proxies. It is encouraged to have methods which are uniformly idempotent or safe, as intermediaries can more effectively use these methods to reduce latency or mask temporary server failures. While these properties are not required of REST methods, they can be formalized in LTL and model-checked. Unlike the two main properties, the formalization of safety and idempotence is different in the white-box and black-box views.

For the black-box setting, we require the following additional constructs. We suppose that there is a distinguished method, $\mathsf{READ}(i)$, where i is the target resource identifier. It returns either ERROR or $\mathsf{OK}(deref(i))$, the representation of the resource identified by i. The linked and unlinked sets are empty. We extend the equivalence relation '$\sim$' is to a list of return values: for lists a and b, $a \sim b$ holds if the lists have the same length and corresponding elements have the same types and are related by '$\sim$'. In the following, we also assume that one can identify whether a communication affects a resource; this information is typically available for specific instances of REST, such as RESTful HTTP.

- **Safety of a method.** A method is considered safe if it does not modify resources. In the black-box view, changes to resources can be detected by means of READ methods. A failure for the safety of method *op* is a finite computation with communications c_1::$\mathsf{READ}(i)/\mathsf{OK}(r_1)$ and c_2::$\mathsf{READ}(i)/\mathsf{OK}(r_2)$ occurring in that order, with $r_1 \not\sim r_2$, where no intervening communication modifies or dissociates the resource $\eta(i)$ but includes at least one communication using *op*. Informally, failure of safety is signaled by a difference in the representation of the resource identified by i before and after method *op*.
- **Idempotence of a method.** For a method to be idempotent, repeated invocation should have no additional effect on resources. In the black-box view, such changes can be detected by means of READ methods. A failure for the idempotence of method *op* is a finite computation where the communications c_1::$op/rc(rv_1)$, c_2::$\mathsf{READ}(i)/\mathsf{OK}(r_1)$, c_3::$op/rc(rv_2)$, and c_4::$\mathsf{READ}(i)/\mathsf{OK}(r_2)$ occur in that order, $r_1 \not\sim r_2$ and the communications occurring between these distinguished ones do not dissociate i or modify the resource $\eta(i)$. Informally, the property detects failure by detecting a difference in the representation of a resource identified by i before and after the second instance of a communication with method *op*.

Both black-box properties are weaker than their white-box counterparts. For instance, it is possible that method *op* changes the server state of a resource – perhaps by incrementing an auxiliary counter – but this change is not propagated to the representation, and is hence unobservable by a READ. This violates safety in the white box view, but not in the black-box view.

Naming Independence. We present an interesting consequence of the RESTful properties, which shows that the specific choice of naming function does not matter, if client-server behaviors are hypertext-driven. To make this precise, consider structures RS and RS' which are identical except for the naming functions. The naming functions, η and η', are required to map each base name to the same resource. The functions induce a name correspondence: a name i in an RS-instance corresponds to a name j in an RS'-instance if both map to the same resource, i.e., if $\eta(i) = \eta'(j)$.

If clients C_i and C'_i in the hypothesis of the theorem are based on the same program text, a sufficient condition for bisimularity up to naming is that names are used *opaquely*: i.e., no constant names are present, names can only be stored to and copied from variables, and the only relational test allowed for names is equality of name variables. The proof of the theorem is given in the full version.

Theorem 1. *Consider an RS-instance M with clients $C_1, \ldots, C_k$ and server S, and an RS'-instance M' with clients $C'_1, \ldots, C'_k$ and server S'. Suppose that, for each i, clients C_i and C'_i are bisimular up to the naming correspondence, as are S and S'. Then, for each hypertext-driven computation σ of M, there is a hypertext-driven computation σ' of M' such that global states $\sigma(i)$ and $\sigma'(i)$ are* *bisimular, for each i, and the induced communication sequences match up to the naming correspondence.*

3 Rest on Http, and Variations

In this section, we show how the property templates from Section 2 can be instantiated for a concrete protocol, HTTP, which is the primary protocol used for constructing RESTful applications. The result is a formal definition of RESTful HTTP behavior.

3.1 Formal RESTful HTTP

HTTP is a networking protocol for distributed, collaborative, hypermedia information systems [7]. The bulk of the interest in REST among developers is in the context of HTTP-based applications. We start by demonstrating how HTTP satisfies the framework requirements described in Subsection 2.1.

HTTP is typically used in a client-server model. HTTP resources are uniquely identified using their Uniform Resource Identifiers (URIs) [27]. For HTTP applications, the fields of a resource representation $\langle uris \in 2^I; d \in D\rangle$ are used as follows: *uris* is a set of URIs, *links* that exist in the resource, and *data* is any data, of any type, that is contained in a resource. It may include auxiliary data, such as counters, which is relevant to server-internal processes, but has no relevance to client behavior. Such data can be elided through an appropriate definition of '$\sim$'. The HTTP RFC [7] defines nine methods. We present here the four main methods, the remaining five have no impact on resources. To represent the HTTP concept of *subordinate* resources, we use a partial mapping, $S : I \mapsto 2^I$, which maps each resource identifier to the set of resource identifiers for its subordinate resources, if any. We only describe successfully processed communications, which return the abstract return code OK, all other codes map to ERROR. The main HTTP methods, with their linked and unlinked sets, are as follows.

- $\mathsf{GET}(i)/\mathsf{OK}(deref(i))$: The method returns the current entity (resource representation) of the resource identified by i from the server. Both L and UL are empty.
- $\mathsf{DELETE}(i)/\mathsf{OK}$: The method dissociates the resource identifier i on the server, resulting in $deref(i)$ bring undefined. Here, L is empty, and $UL = \{i\}$. The HTTP RFC actually only requires that the server 'intends' to dissociate it [7]. We discuss this more complex scenario in Subsection 3.2.
- $\mathsf{PUT}(i, \langle uris; d\rangle)/\mathsf{OK}$: The method associates a resource identified by i, if it is not already associated, and assigns a value to its corresponding entity so that $deref(i) = \langle uris; d\rangle$. If this is a new association, then $S(i) = \{\}$. Here, UL is empty, while $L = \{i\}$.
- $\mathsf{POST}(i, \langle uris; d\rangle)/\mathsf{OK}(j)$: The method associates a fresh resource, which is identified by j, and sets $S(j) = \{\}$ and $deref(j) = \langle uris; d\rangle$. The resource identified by j becomes a subordinate of the resource identified by i, and j is added to $S(i)$. Here, UL is empty, while $L = \{j\}$.

Instantiating the REST property templates from Subsection 2.3 with the HTTP methods results in a formal definition of RESTful HTTP. This is a rather technical, mostly straightforward translation, and is given in the full paper.

3.2 Variations on RESTful HTTP Properties

In this section we mention several common or reasonable modifications of the HTTP model from Subsection 3.1, and discuss how they impact the RESTful HTTP properties. Complete descriptions of these modifications and the corresponding changes to the properties are in the full paper.

Cascade of DELETE Methods by Subordination. One side-effect of the POST method is the creation of a subordination relation from the target resource identifier to the newly associated one. A common feature in many HTTP applications is the requirement that when a resource identifier is dissociated through a DELETE call, its subordinates are deleted as well (which, in turn, may trigger more dissociations of resource identifiers with higher degrees of subordination to the originally deleted one). In our model, this would translate into a (recursive) modification to the linked set of DELETE communications. In RESTful HTTP, this change would require modifying all properties whose definition relies on the unlinked sets of communications (the hypertext-driven sequences property and any idempotence properties) to use more complex, yet easily computed, definitions of the unlinked sets.

Subordination Expressed as a Link. Here, subordination is expressed as a link, i.e., for every $i \in I$ such that $\mathit{deref}(i) = \langle \mathit{uris}; d \rangle$, if $S(i)$ is defined, then $S(i) \subseteq \mathit{uris}$. In this case, a side effect of the communication $c{::}\mathsf{POST}(i, r)/\mathsf{OK}(j)$ would be the modification of the resource identified by i to include j in uris. The idempotence property should account for this case by considering that successful POST methods modify existing resources.

Background Data Modifications by the Server. In some cases, where the semantics of the domain D are such that it is is (partially or fully) dynamic by nature, HTTP allows the server to modify the data field of resource representations arbitrarily, in accordance with their semantics. An example is a ‘current time’ resource, whose value is updated by the server. Successive GET’s on this resource would result in different values for the time, potentially violating the safety property of GET. This case can be handled by a proper definition of the data equivalence relation to ignore such changes.

Delayed Executions of Completed DELETE Communications. In the HTTP RFC ([7]) it is said that when the server successfully processes a DELETE request it merely means that “at the time the response is given, it intends to delete the resource or move it to an inaccessible location.” Our interpretation of this quote is that the single resource identifier that is in the unlinked set of successfully processed DELETE communications is dissociated only after some arbitrary, **finite**, delay, unless it is associated in the meantime by another communication. Although this behavior makes our definition of $assoc(s)$ irrelevant, it does not complicate the HTTP application of the hypertext-driven sequences property, due to the fact that HTTP clients must 'assume' anyway that resources on which they performed successful DELETE methods are no longer accessible for them. All other properties, however, need to be modified to account for the fact that DELETE methods are not as well-behaved as assumed earlier. The HTTP statelessness property, for instance, would have to disallow 'temporal forks' that include DELETE methods performed by different clients which take effect at different times.

3.3 Distinguishing REST from HTTP

Following are some interesting hypothetical applications which clarify the differences between HTTP and REST, and which address some common misunderstandings regarding RESTful HTTP.

Consider an application which uses only two HTTP methods: PUT and GET. A client encodes methods in the *uri* argument of $\mathsf{PUT}(uri, junk)$ requests, where *junk* - a resource representation - is a meaningless constant. A GET communication is used by a client to examine the state of the server. This application is compliant with the HTTP RFC, as there is no restriction on the PUT communications' return values. However, it is non-RESTful, since it would either have to include an infinite set of root identifiers (each *uri* argument being one), or it would violate the hypertext-driven behavior property. The Flickr API is non-RESTful for a similar reason.

Consider an application which relies entirely on POST communications, and uses a single root identifier, *base*, for all such communications (one may consider $B = \{base\}$). In any $\mathsf{POST}(base, \langle base; data\rangle)/\mathsf{OK}(uri)$ communication, clients encode methods in the *data* field. We consider two variants:

1. The return value of an method is encoded in the newly associated URI *uri*, returned as a result of POST. This is compliant with the HTTP RFC, but it goes against the notion of dividing information into distinct resources, as the base URI must be treated as a single resource. As there is no division into resources (which would be created by – and used to identify – different clients), this application is likely to violate the HTTP statelessness property. Moreover, it is also likely to violate the resource identifier opaqueness assumption from Subsection 2.3, as a program must interpret the URI strings returned by POST. While the opaqueness assumption is not an essential part of REST, it is important to simplify program development and maintenance.

2. The newly associated URI *uri* is used to point to a resource whose representation is the result of the method, and which is later retrieved by a GET on the *uri*. This violates the HTTP RFC, which requires that the result of POST identifies a resource with the supplied data as its representation. As in the previous case, this application is also likely to violate the HTTP statelessness property.

4 Automated Verification of RESTful Behavior

In this section, we formulate and discuss questions relevant to the automated verification of RESTful behavior. We give preliminary results and point to questions that are still open.

4.1 Computation Model

The somewhat informal model used previously can be made precise as follows. Client and server processes are modeled as labeled transition systems. A communication is modeled as a CCS synchronization [17]. Hence, in a communication of the form 'request/response', a client offers this communication at its state, the server offers to accept it, and the two are synchronized to effect the communication. Processes may have internal actions, including internal non-determinism. The CCS model is appealing for its simplicity but assumes atomic communication. We formulate problems and solutions in this model. Subsequently, we discuss how the atomicity requirement may be relaxed, which brings the analysis closer to real implementation practice.

4.2 Fundamental Questions

The two properties of REST, statelessness and hypertext-driven behavior, lead to the following key verification questions.

ST. Does a client-server application M satisfy the statelessness property?
HT1. For a client-server application M, does its specification, φ, hold for all computations where client behavior is hypertext-driven?
HT2. For a client-server application M, do all computations which are not hypertext-driven satisfy a 'safe-behavior' property ξ?

These fundamental questions may be asked for a program with a fixed set of clients and resources, or in the parameterized sense. One may also ask if violations of these properties can be detected using run-time monitors. Another interesting question is whether, given an application specification, one can synthesize a server which satisfies it (again, fixed or parameterized).

4.3 Automata Constructions

A nondeterministic automaton which detects a *failure* of the hypertext-driven behavior property works as follows. For a given input word, the automaton guesses the client and resource identifier with which to instantiate the failure specification, then keeps track of whether the resource id belongs to the current *assoc* for that client. It accepts if, at some point, there is a request by the client using the resource id, but the id is not part of the current *assoc* set. Keeping track of whether a resource id belongs to the *assoc* set for a client does not require computing the *assoc* set. A simple two-state machine suffices, with states $In(c, i)$ and $Out(c, i)$. If the current communication m is by client c and is successful, a transition is made from $In(c, i)$ to $Out(c, i)$ if $i \in UL(m)$, and from $Out(c, i)$ to $In(c, i)$ if $i \in L(m)$. Otherwise, the state is unchanged. The number of automaton states, therefore, is polynomial in $|I|$ and $|C|$.

The deterministic form of this automaton must track all clients and resource ids simultaneously. Thus, the size of a state of the deterministic automaton is $O(|I| \cdot |C|)$, and its state space is exponential: $O(2^{|I| \cdot |C|})$. Non-deterministic failure automata for safety and idempotence can be derived similarly from their failure specifications; these are described in the full paper.

4.4 Model-Checking for Fixed Instances

A fixed instance has a fixed set of resources and clients. The parameters of interest are the sets in the underlying resource structure: the clients, C, the resource identifiers, I, and the data domain, D.

Statelessness is expressed in a slight variant of CTL, as described previously. (The extension does not affect model-checking complexity.) The indexed property expands out to a propositional formula which is polynomial in the sizes of I and D. Hence, using standard CTL model-checking algorithms [4,23], the ST property can be verified in time linear in the overall application state space and polynomial in the resource structure parameters.

Property HT1 can be verified as follows. A violation of HT1 is witnessed by a computation where all clients are hypertext-driven but φ is false. This can be checked using automata-theoretic model checking [24] by forming the product of the application process with (1) a Büchi automaton for the negation of φ, and (2) an automaton which checks that all clients follow hypertext-driven behavior. The property is verified **iff** the product has an empty language. The second automaton is the deterministic automaton from Section 4.3, with negated acceptance condition.

Property HT2 can be verified by forming the product of the application process with (1) a Büchi automaton for the negation of ξ, and (2) an automaton which checks for failure of hypertext-driven behavior by some client. The property is verified **iff** the product has an empty language. The second automaton is the non-deterministic failure automaton from Section 4.3. The verification takes polynomial time if the size of the application state space is polynomial in the parameter sizes. The verification of HT1 is significantly more difficult.

Theorem 2. *Verification of HT1 for a fixed instance is* PSPACE*-hard in the number of resources. It is in* PSPACE *if a state of the application and of the negated specification automaton can be described in space polynomial in the parameter sizes.*

Proof Sketch. Membership in PSPACE is straightforward, by observing that the automaton used to describe the hypertext-driven property for HT1 has a state size which is polynomial in the the parameter sizes.

PSPACE-hardness for HT1 holds under severe restrictions: a single client, where client, server, and negated specification automaton have a state-space with size polynomial in the parameters' sizes. The reduction is from the question of deciding, given a Turing Machine (TM) M and input x, whether M accepts x within the first $|x| + 1$ tape cells, which is a PSPACE-complete problem (IN-PLACE ACCEPTANCE in [19]). The reduction uses the server state to store the TM head position, while a TM configuration is encoded in the implicitly defined *assoc* set for the client, using resources to represent tape cell contents. □

4.5 Parameterized Verification

The parameterized verification question has particular importance, as web applications usually handle a large number of clients and resources. Since statelessness is not a given, it is necessary to assume a server which stores information about each client, which implies that the state space of the server is also unbounded. Nonetheless, the problem can be solved under certain assumptions.

Suppose that clients have a finite state space, X, and that the state space of the server can be written as $Y \times [C \rightarrow Z]$, where Y and Z are finite sets. Thus, a global state of an instance with N clients is a triplet (c, a, b), where c is an array of client states, of size N, a is the finite part of the server state, and b is an array of N server-side entries. Assume further that on receiving a request from client i, the server update depends only on, and may only modify, the components a and $b(i)$; i.e., the new entry for client i does not depend on the entries of the other clients. Then, by a change of viewpoint, one may combine the entry $b(i)$ on the server with the state $c(i)$ of client i, obtaining an equivalent application where the new client space is $X \times Z$, and the server space is Y. Both spaces are now finite, although there is still an unbounded number of clients. This situation fits the model in [11], where an algorithm is given for checking linear-temporal properties. The algorithm has very high worst-case complexity, however, so it may be more fruitful to try alternative methods, such as the method of invisible invariants [21,18], or methods based on upward-closed sets [1].

Several questions remain open. The modeling above implicitly assumes a bounded set of resources and data values. Moreover, the suggested algorithm applies only to linear-time properties and cannot, therefore, be used to check statelessness.

4.6 Run-Time Monitoring

Perhaps the most promising immediate application of the formalization is run-time monitoring. In this setting, the client-server communications are captured by an intermediate proxy, which passes them through analysis automata. This method can be applied to the properties HT1 and HT2; statelessness, being a branching-time property, cannot be checked at run-time, unless some form of backtracking is implemented. The automata described in Section 4.4 for model-checking HT1 and HT2 can be used for run-time verification of safety specifications. The non-deterministic automata used for checking hypertext-driven behavior must be determinized for run-time analysis. This can be done on the fly, as is the case for implementations of the Unix `grep` command (cf. [2]). The size of the deterministic automaton state is $O(|I| \cdot |C|)$, so the required storage is $O(|I| \cdot |C| \cdot K)$, where K is the state-size of the negated specification automaton. For each communication, the update of the automaton state requires time proportional to the size of the state, and is hence polynomial in the resource parameters. An alternative to run-time verification is off-line testing of a logged communication sequence.

4.7 Synthesizing Servers

A particularly intriguing question is the possibility of synthesizing RESTful servers. A specific question is the following: given a resource structure and a specification φ, synthesize a **stateless** server which satisfies φ. We show below that, under certain assumptions, the statelessness constraint can be dropped. We define a server specification φ to be universally synthesizable if there exists a server implementation which satisfies φ given any set of clients. A sufficient condition for φ to be universally synthesizable is if it is insensitive to client ids and is synthesizable for a single, arbitrary client. Insensitivity means that for sequences σ, δ which agree up to client ids in communications, $\sigma \models \varphi$ **iff** $\delta \models \varphi$.

Theorem 3. *Consider a server temporal logic specification φ. The specification φ is deterministically and universally synthesizable* **iff** *$ST \wedge \varphi$ is deterministically and universally synthesizable.*

Proof Sketch. For the left-to-right direction, given a deterministic server M implementing φ, one can direct all communications to it through an intermediary which replaces all client ids with a single, dummy, client id. By the universality of M, this combination satisfies φ; as M 'sees' only a single client id and is deterministic, the combination is stateless. □

The synthesis problem for LTL specifications, assuming a bounded state-space, was solved in [22]. Implementing the intermediary adds constant complexity.

Adding the assumption that client interactions are hypertext-driven may make an otherwise-unsynthesizable specification synthesizable, but it may also add significantly to the specification complexity.

4.8 Relaxing the Atomicity of Communications

So far, we have assumed that communications are atomic. In real implementations, however, a request and its response are distinct actions. This allows requests from different clients to overlap in time. To handle this concurrency, we assume that the server is linearizable [12]. Every computation produces results which are equivalent to one where each method takes effect atomically.

Hypertext-driven behavior is formulated entirely in terms of the request and response parameters. If clients are not allowed to issue concurrent requests, hypertext-driven behavior holds of a computation **iff** it holds of its linearization. Assume that the service specification is also defined on communication sequences, and has the same property. Then, it suffices to check properties over the linearized subset of computations, which corresponds to the atomic communication model. This reasoning does not apply to statelessness, which is a branching property, and thus outside the scope of linearizability. Further work is necessary to formulate a notion like linearizability for branching-time properties.

5 Related Work and Conclusions

There is surprisingly little in the literature on formal definitions and analysis of REST. In [13], the authors describe a pi-calculus model of RESTful HTTP. This model, however, comes across as a mechanism for programming a specific type of RESTful HTTP application. The paper does not consider the general properties of REST: statelessness and hypertext-following, nor does it describe a methodology for checking that arbitrary implementations satisfy these properties. There are also a number of books and expository articles on REST, but those do not include formal specifications, nor do they consider analysis questions.

Our work appears to be – to the best of our knowledge – the first to precisely formulate the key properties of REST, and to demonstrate interesting consequences, such as naming independence and the PSPACE-hardness of verification. This work also opens up a number of interesting questions. One is to use the formalization as a basis to investigate questions about REST itself: for instance, how to combine authentication with REST, and how to extend REST to executable representations [6]. We have argued that the parameterized model-checking and synthesis questions are especially relevant for web applications using REST. Constructing a practically usable verifier for REST properties is itself a non-trivial task. We have experimented with simple examples verified using SPIN [14]. An effort to use JPF [25] to verify applications written in the JAX-RS extension of Java was unsuccessful, however, as JPF currently lacks support for key libraries in JAX-RS. Our current focus is on creating a runtime checker, which has the advantage of being independent of implementation language.

To summarize, the formal modeling of REST clarifies its definition, and also raises several challenging questions, both in modeling and in automated analysis.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS (1996)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, 2nd edn. Addison-Wesley, Reading (2007)
3. Bizer, C., Heath, T., Idehen, K., Berners-Lee, T.: Linked data on the web (LDOW2008). In: WWW, pp. 1265–1266 (2008), talk by Tim Berners-Lee at TED (2009), http://www.w3.org/2009/Talks/0204-ted-tbl/
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, Springer, Heidelberg (1982)
5. Emerson, E., Clarke, E.: Proving correctness of parallel programs using fixpoints. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, Springer, Heidelberg (1980)

6. Erenkrantz, J.R., Gorlick, M.M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC/SIGSOFT FSE, pp. 255–264 (2007)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: W3C RFC 2616 (June 1999), http://www.w3.org/Protocols/rfc2616/rfc2616.html
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irving (2000)
9. Fielding, R.T.: (2008), http://roy.gbiv.com/untangled/2008/no-rest-in-cmis#comment-697
10. Fielding, R.T.: (2008), http://roy.gbiv.com/untangled/2008/rest-apis-must\discretionary-be-hypertext-driven
11. German, S., Sistla, A.: Reasoning about systems with many processes. Journal of the ACM (1992)
12. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
13. Hernández, A.G., García, M.N.M.: A formal definition of RESTful semantic web services. In: WS-REST, pp. 39–45 (2010)
14. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003), http://spinroot.com
15. Klein, U., Namjoshi, K.S.: Formalization and Automated Verification of RESTful Behavior. Tech. rep., Bell Labs; Courant Institute of Mathematical Sciences, NYU TR2011-938 (2011)
16. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Proc. of the Conf. on Logics of Programs (1985)
17. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1990)
18. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
19. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
20. Pnueli, A.: The temporal logic of programs. In: FOCS (1977)
21. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
23. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, Springer, Heidelberg (1982)
24. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: IEEE Symposium on Logic in Computer Science (1986)
25. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003), http://babelfish.arc.nasa.gov/trac/jpf
26. SOAP version 1.2 part 1: Messaging framework (second edition). W3C Recommendation (2007), http://www.w3.org/TR/soap12-part1/
27. Uniform Resource Identifier (URI): Generic Syntax. W3C RFC 3986 (2005)

Linear Completeness Thresholds for Bounded Model Checking*

Daniel Kroening[1], Joël Ouaknine[1], Ofer Strichman[2], Thomas Wahl[1], and James Worrell[1]

[1] Department of Computer Science, Oxford University, UK
[2] Technion, Israel

Abstract. *Bounded model checking* is a symbolic bug-finding method that examines paths of bounded length for violations of a given LTL formula. Its rapid adoption in industry owes much to advances in SAT technology over the past 10–15 years. More recently, there have been increasing efforts to apply SAT-based methods to *unbounded* model checking. One such approach is based on computing a *completeness threshold*: a bound k such that, if no counterexample of length k or less to a given LTL formula is found, then the formula in fact holds over all infinite paths in the model. The key challenge lies in determining sufficiently small completeness thresholds. In this paper, we show that if the Büchi automaton associated with an LTL formula is *cliquey*, i.e., can be decomposed into clique-shaped strongly connected components, then the associated completeness threshold is *linear* in the recurrence diameter of the Kripke model under consideration. We moreover establish that all unary temporal logic formulas give rise to cliquey automata, and observe that this group includes a vast range of specifications used in practice, considerably strengthening earlier results, which report manageable thresholds only for elementary formulas of the form $\mathsf{F}\,p$ and $\mathsf{G}\,q$.

1 Introduction

LTL *bounded model checking* (BMC) [4,3] is a symbolic bug-finding method that searches for lasso-shaped counterexamples to an LTL formula in a given Kripke structure. Within three or four years following its introduction, it was found to have almost entirely replaced BDD-based model checkers in the hardware industry, owing to the fact that many users care more about finding bugs quickly than about formal proofs of their absence, especially as the latter often require vast amounts of memory and time. This major success can be attributed mostly to the impressive advances made in SAT technology over the past 10 to 15 years.

The fundamental approach underpinning BMC is to look for counterexamples, or bugs, of bounded length. As such, an absence of counterexample is inconclusive; a genuine bug could still lurk deeper in the system. For this reason, from the very inception of the technique, researchers have attempted to turn BMC into a

* Supported by the EU FP7 STREP PINCETTE.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 557–572, 2011.

complete method with the ability also to guarantee the absence of counterexamples of any length. See, for instance, the original work of Biere *et al.* [4], or the 2008 Turing Award lecture of Ed Clarke [7], in which the problem is described as a topic of active research.

In [4], Biere *et al.* observed that for safety properties of the form $\mathsf{G}\,p$, a *completeness threshold* is given by the *diameter* (longest distance between any two states) of the Kripke structure under consideration: indeed, if no counterexample to $\mathsf{G}\,p$ of length at most the diameter of the system can be found, then no counterexample of any length can possibly exist. Likewise, for liveness properties such as $\mathsf{F}\,q$, the *recurrence diameter* (longest loop-free path) of the Kripke structure can be seen to be an adequate completeness threshold. But the general problem of determining reasonably tight completeness thresholds for arbitrary LTL formulas remains wide open to this day.

Note that the diameter (for safety properties) and the recurrence diameter (for liveness properties) are not merely sound bounds, they are also worst-case tight. In other words, no smaller completeness threshold expressible strictly in terms of the diameters can be achieved. Of course, in any particular situation the least completeness threshold may well be orders of magnitude smaller than the diameter, but determining its value is clearly at least as hard as solving the original model-checking problem in the first place, and we must therefore be content with sound but reasonably tight over-approximations.

In this paper, we describe an efficient technique for obtaining fairly tight, *linear* completeness thresholds for a wide range of LTL formulas, as a function of the diameter and recurrence diameter of any Kripke structure under consideration. All Büchi automata that are *cliquey*, i.e., that can be decomposed into clique-shaped strongly connected components, admit linear completeness thresholds. Moreover, we show that such automata subsume *unary linear temporal logic*, and indeed comprise a wide range of formulas used in practice, including, for example, the vast majority of specifications appearing in Manna and Pnueli's classic text on the specification of reactive and concurrent systems [12].[1] We also show that computing these linear completeness thresholds can be done in time linear in the size of the given Büchi automata. Finally, we exhibit some simple (non-cliquey) Büchi automata, and corresponding LTL formulas, having *superpolynomial* and even *exponential* completeness thresholds.

In the past, researchers have been able to achieve completeness thresholds by studying the *product* structure of the Kripke model and the Büchi automaton corresponding to the specification of interest; see, e.g., [6,1]. Such thresholds are in general incomparable with the ones we present in this paper. Moreover, a significant disadvantage of the earlier approach is that it requires one to investigate a structure which is often much too large and unwieldy to construct, let alone perform any calculations upon. Another benefit of the present approach is that, once the diameter and recurrence diameter of a given Kripke structure are known (or over-approximated), they can be put to use against any

[1] For instance, specifications such as *conditional safety*, *guarantee*, *obligation*, *response*, *persistence*, *reactivity*, *justice*, *compassion*, etc., all fall within our framework.

number of specifications, whereas the earlier approach requires a fresh calculation of the diameters of each of the different product automata, possibly resulting in prohibitive computation costs.

Orthogonal research directions aiming to achieve completeness of bounded model checking include cube enlargement techniques [13], circuit co-factoring [8], induction [16], and Craig interpolation [14].

2 Definitions

By LTL\X we denote standard propositional linear temporal logic [5] without the *next-time* operator X; all the results in this paper also hold if backwards temporal operators are included as well. Every LTL\X formula φ is *invariant under stuttering*, meaning that any two stuttering-equivalent paths either both satisfy or both violate φ (see [5] for a precise definition).[2]

Let AP be a finite set of atomic propositions. A *Kripke structure* is a tuple $M = (S, S_0, R, L)$ with finite set of states S, set of initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$, and state-labelling function $L \colon S \to 2^{AP}$, which assigns to each state the set of atomic propositions considered to be true in that state. A *path* through M is a sequence $(\pi_i)_{i=0}^{l}$ ($l \in \mathbb{N} \cup \{\infty\}$) of states such that for $i < l$, $(\pi_i, \pi_{i+1}) \in R$. By $|\pi|$ we denote the number l of edges in π. Thus, if π is finite, its last vertex is $\pi_{|\pi|}$. An infinite path π is *lasso-shaped and k-bounded* if there exist two finite paths u and v such that $\pi = u.v^{\omega}$ and $|u|+|v| \leq k$. Here, v^{ω} denotes infinite repetition of v, and u and v^{ω} are concatenated. For concatenating two paths, we require that the last state of the first path be identical to the first state of the second path. Thus the definition of *k-bounded* implies that $u_{|u|} = v_0 = v_{|v|}$. The concepts of 'reachable' (existence of a connecting path) and 'distance between reachable states' (length of shortest connecting path) are defined in the standard way.

A *(generalised) Büchi automaton* is a tuple $B = (S, S_0, R, L, \mathcal{A})$ with finite set of states S, set of initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$, state-labelling function $L \colon S \to \mathcal{B}(AP)$, and family $\mathcal{A} \subseteq 2^S$ of accepting sets of states; here $\mathcal{B}(AP)$ denotes the set of all Boolean combinations of atomic propositions in AP. Note that states (rather than transitions) are labelled, namely by such Boolean combinations of atomic propositions. An infinite path π through B is *accepting* if, for each state set $T \in \mathcal{A}$, π visits a state of T infinitely often.

The *product* of Kripke structure $M = (S, S_0, R, L)$ with Büchi automaton $B = (S', S'_0, R', L', \mathcal{A}')$, denoted $M \times B$, is defined as the Büchi automaton $(S'', S''_0, R'', L'', \mathcal{A}'')$ with

- $S'' = \{(s, s') \in S \times S' \mid L(s) \vDash L'(s')\}$[3]
- $S''_0 = (S_0 \times S'_0) \cap S''$

[2] Many computer scientists, starting with Lamport in the 1980s [11], have argued that high-level specifications of computer systems always ought to be stuttering-invariant.

[3] By $L(s) \vDash L'(s')$, we mean that the Boolean formula $L'(s')$ evaluates to *true* if all variables in $L(s)$ are assigned *true* and all other variables are assigned *false*.

- $R'' = \{((s_1, s_1'), (s_2, s_2')) \in S'' \times S'' \,|\, (s_1, s_2) \in R$ and $(s_1', s_2') \in R'\}$
- $L''\colon S'' \to 2^{AP} \times \mathcal{B}(AP)$ with $L''(s, s') = (L(s), L'(s'))$
- $\mathcal{A}'' = \{(S \times T') \cap S'' \,|\, T' \in \mathcal{A}'\}$.

Note that the labelling functions of M and B determine which states exist (are valid) in the product $M \times B$. There is a transition in the product iff corresponding transitions are present in both components. For our purposes, the labelling of states in the product automaton is irrelevant. Finally, the acceptance set family $\mathcal{A}''$ is derived from that of the Büchi automaton.

The product construction is related to LTL model checking as follows:

Theorem 1 ([9]). *Let M be a Kripke structure and φ an LTL formula. There exists a generalised Büchi automaton $B_{\neg\varphi}$ such that $M \models \varphi$ exactly if $M \times B_{\neg\varphi}$ has no accepting path.*

In figures, we represent Büchi automata as directed graphs. Initial states have an incoming edge without source. Accepting states are drawn as filled discs (our illustrating examples all have a singleton acceptance set family, in other words they are simple Büchi automata), and other states are drawn as hollow circles. In Kripke structures (cf. Figure 4), we depict the label of a state as a set of propositions, omitting the braces {}.

For a Kripke structure M, we write $M \models_k \varphi$ to denote that every lasso-shaped k-bounded path π in M satisfies φ. A *completeness threshold for M and φ* is an integer k such that

$$M \models_k \varphi \quad \Rightarrow \quad M \models \varphi\,.$$

This definition reflects the intuition behind bounded model checking: assuming that there is no counterexample to φ of length at most k, φ should hold in M. We can generalise this definition to Büchi automata as follows: a *completeness threshold for a Kripke structure M and a Büchi automaton B* is any integer k such that, if $M \times B$ has *any* accepting path, then it has a k-bounded lasso-shaped accepting path. With these definitions, an integer k is a completeness threshold for a Kripke structure M and formula φ precisely if it is a completeness threshold for M and $B_{\neg\varphi}$, where $B_{\neg\varphi}$ is the result of translating $\neg\varphi$ into any equivalent generalised Büchi automaton.

The following are key notions in this paper:

Definition 2. *Let M be a Kripke structure. The* ***distance*** *from a state s to a state t is the length of a shortest path from s to t (or ∞ if there is no such path). The* ***diameter*** *of M, denoted $d(M)$, is the largest distance between any two reachable states ('longest shortest path'). The* ***recurrence diameter*** *of M, denoted $rd(M)$, is the length of a longest simple (loop-free) path through M.*

3 Büchi Automata with Linear Completeness Thresholds

Given a Kripke structure and an LTL formula, it is clear that determining the *smallest* completeness threshold is at least as hard as the model-checking problem itself, and is thus not something we are aiming to achieve. Rather, the goal of this paper is to establish a class of LTL formulas admitting completeness thresholds that are *linear* in the diameter and the recurrence diameter of any given Kripke structure. In this section, we first introduce a class of generalised Büchi automata, termed *cliquey*, with this property. We also present an algorithm which, given a cliquey automaton B, produces a symbolic arithmetic expression $ct(d, rd)$ such that, for any Kripke structure M, $ct(d(M), rd(M))$ is a valid completeness threshold for M and B. Moreover, the expression ct is linear in d and rd. In Section 4, we exhibit a class of LTL formulas that have cliquey Büchi representations, namely *unary linear temporal logic* formulas.

Let $sap(B) = \min\{k \mid B \text{ has a lasso-shaped } k\text{-bounded accepting path}\}$, for a non-empty Büchi automaton B. An $sap(B)$-bounded accepting path is called a *shortest accepting path*, SAP for short. If B is empty, let $sap(B) = -\infty$.

Definition 3. *A generalised Büchi automaton B* ***has a linear completeness threshold*** *if there exists $c \in \mathbb{N}$ such that for all Kripke structures M, $sap(M \times B) \leq c \cdot rd(M)$.*

We are now ready to define a class of Büchi automata that admit linear completeness thresholds.

3.1 Cliquey Büchi Automata

The class of automata we consider in this section is characterised by a particular structure of the underlying transition graph:

Definition 4. *A directed graph is* ***cliquey*** *if every maximal strongly connected component (SCC) is a bidirectional clique, i.e. any two nodes of an SCC are connected by an edge in either direction. In particular, every node has a self-loop.*

We say that a generalised Büchi automaton is *cliquey* if its underlying graph structure (ignoring the vertex labelling and the accepting condition) is cliquey. The Büchi automaton in Figure 1 (a) is cliquey, whereas that in (b) is not cliquey. Moreover, we shall see in Section 5.1 that the latter has no equivalent cliquey representation.

Theorem 5. *Every cliquey generalised Büchi automaton admits a linear completeness threshold.*

Proof: Let B be cliquey, M an arbitrary Kripke structure with recurrence diameter rd, and $\Pi = M \times B$. We show $sap(\Pi) \leq c \cdot rd$, for a number c that can be chosen independently of M. If Π has no accepting path, then $sap(\Pi) = -\infty$, so there is nothing to prove. Otherwise, let π be an SAP of Π, and let $C_1, \ldots, C_s$ be the sequence of SCCs of B that π traverses, in this order. We now bound the length of π in each SCC.

Consider a non-final SCC C_i (i.e., $i < s$), and let π^i be the segment of π that traverses C_i (in other words, π^i is a maximal segment of π such that the projection of its states to B is a path in C_i). Suppose $|\pi^i| \geq rd + 2$. The prefix of π^i of length $|\pi^i| - 1$ exceeds M's recurrence diameter rd. Thus we can find two product states of the form (m, b) and (m, b') along this segment. Let (m'', b'') be the successor of (m, b') along π^i (note that (m'', b'') still belongs to π^i):

$$(m, b) \quad \rightsquigarrow \quad (m, b') \rightarrow (m'', b'')\,.$$

From $(m, b') \rightarrow (m'', b'')$ in Π, we conclude $m \rightarrow m''$ in M. Since C_i is a clique, we conclude $b \rightarrow b''$ in B. Hence, $(m, b) \rightarrow (m'', b'')$ in Π. This, however, contradicts the fact that π is an SAP through Π. Therefore, $|\pi^i| \leq \mathit{rd} + 1$.

Consider now the final SCC C_s, and let the family of accepting sets of B be $\mathcal{A} = \{A_1, \ldots, A_n\}$. The segment π^s of π traversing C_s visits each A_i infinitely often. Since each A_i is finite, there exists in fact a fixed state $a_i \in A_i$ in each of them that is visited infinitely often. Segment π^s thus looks like this:

$$(m, b) \quad \rightsquigarrow \quad (m_1, a_1) \quad \rightsquigarrow \quad (m_2, a_2) \quad \rightsquigarrow \quad \ldots \quad \rightsquigarrow \quad (m_n, a_n) \quad \rightsquigarrow \quad (m, b)\,.$$

Using the same argument as in the non-final case, each segment abbreviated by $\rightsquigarrow$ has length at most $\mathit{rd} + 1$ (otherwise, a shorter accepting path could be constructed). As a result, $|\pi^s| \leq (n + 1)(\mathit{rd} + 1)$.

In total, $|\pi| \leq (s - 1)(\mathit{rd} + 1) + (n + 1)(\mathit{rd} + 1)$, clearly inducing a linear completeness threshold, for example with constant $c = 2(s+n)$ (note that s and n are parameters of B and do not depend on M). □

3.2 Computing Completeness Thresholds of Cliquey Automata

The proof in Section 3.1 establishes linearity of the completeness threshold for cliquey Büchi automata. It is, however, very coarse. Among others, the argument ignores the structure of the SCC quotient graph. In the following, we give a higher-order algorithm that takes a cliquey Büchi automaton B as input and returns a function ct over two arguments. When supplied with the diameter d and the recurrence diameter rd of a Kripke structure M, this function returns a completeness threshold for M and B: $\mathit{sap}(M \times B) \leq \mathit{ct}(d, \mathit{rd})$. Exploiting the fact that B is cliquey, $\mathit{ct}(d, \mathit{rd})$ will be linear in d and rd.

The algorithm proceeds in two stages. In the first stage, each clique in the SCC quotient graph of B is assigned a cost, as a function of d and rd, of traversing it in the product automaton $M \times B$, namely the maximum length a path segment can 'spend' in this clique, given that the path is an SAP. In the second stage, the algorithm traverses the SCC quotient graph, in order to *symbolically* compute respective longest paths from initial cliques to all cliques, using the cost measures computed during the first stage. The result returned by function ct is then the maximum path length computed, over all cliques that could potentially serve as the clique visited last along an accepting path.

The Cost of Traversing a Clique. For a generalised Büchi automaton B with accepting sets $A_1, \ldots, A_n$, call a clique C in B *accepting* if for each $i \in \{1, \ldots, n\}$, $C \cap A_i \neq \emptyset$. Such a clique contains a state from each accepting set and is thus eligible as a *final* clique, visited infinitely often, as $M \times B$ is traversed. Further, we say C is *vacuously labelled* if, for each Büchi state b in C, $L(b) = \mathit{true}$. The significance of this condition is that such a Büchi state b can be paired with any Kripke state m: the condition $L(m) \vDash L(b)$ holds for any m. This will be instrumental in redirecting certain non-optimal paths, as we shall see below.

We denote the cost of traversing C as a non-final clique by $cost[C]$, and the cost of traversing C as a final clique by $cost_f[C]$. These values are assigned according to Table 1, depending on whether C is vacuously labelled or not, and whether C is accepting or not. Note that an accepting clique may well be traversed as a non-final clique, whereas a non-accepting clique cannot be traversed as a final clique.

Table 1. Over-approximating the cost of traversing a clique

C vacuously labelled?	C accepting?	$cost[C]$	$cost_f[C]$
no	no	$rd + 1$	∞
no	yes	$rd + 1$	$(n + 1)(rd + 1)$
yes	no	d	∞
yes	yes	d	$(n + 1)d$

To discuss these figures, consider first the typical case in which C is not vacuously labelled. We have seen in the proof of Theorem 5 in Section 3.1 that any SAP segment within C, visited as a non-final clique, is of length at most $rd + 1$; it does not matter here whether C is accepting or not. When visited as the final clique, however, C must be accepting; in this case said segment is of length at most $(n + 1)(rd + 1)$, as shown in the same proof.

If C is vacuously labelled, we can strengthen the argument from Section 3.1: let $s = (m_0, b_0) \rightsquigarrow (m_1, b_1)$ be a path segment of an SAP whose projection to the B component runs within C. Suppose the length of s is greater than M's diameter d. Then there is a path π from m_0 to m_1 in M with $|\pi| < |s|$. Path π can be used to form a path through C in the product that is shorter than s: pair every state along π with b_0, except the last state m_1, which is paired with b_1 as above. The product states (m, b) thus created *vacuously* satisfy the condition $L(m) \vDash L(b)$, since $L(b) = true$. The B-components of the new path form a path in B, since they run within a clique, with self-loops on all states. These findings contradict the fact that s is a segment of an SAP through $M \times B$. Therefore, $|s| \leq d$, so d is the cost of traversing C as a non-final clique. For traversing C as the final clique (assuming it is accepting), the cost is $(n + 1)$ times higher, as before: we apply the diameter argument to each subsegment of s between two accepting states.

The Cost of Traversing the SCC Quotient Graph. The second stage is to 'collect' the costs we have computed per clique in stage 1. Which cliques of B are visited in an actual SAP in $M \times B$ of course depends on M. For our results to hold over any Kripke structure, we determine a *longest* path through the SCC quotient graph. This quotient graph is acyclic, so that the single-source longest path problem can be solved in time linear in the number of quotient edges, by traversing the graph in topological order.

A complication is that, since we do not have the concrete Kripke structure at hand, the costs of moving from clique to clique are given symbolically, by expressions of the form appearing in Table 1. Thus, when comparing the lengths of paths to a particular clique found so far, instead of recording the new length as the numerical maximum of the two given lengths, we record it as the *symbolic maximum* of the two length expressions. The final result reported by the function will thus be an expression involving the parameters d and rd of the unknown Kripke structure, as well as **linear** operators connecting them, such as addition, constant multiplication, and max.

The traversal of the SCC quotient graph is shown in Algorithm 1. It assumes the Büchi automaton has a unique initial clique C_0 (i.e., a clique containing initial states of B); we handle the general case below. The algorithm keeps the cost of traversing a clique, as computed in Table 1, in an array *cost*, and the cost of reaching and traversing a clique in an array *reach*, both as a non-final and final clique (the latter stored in arrays with subscript f). The *reach* values are initialised to 0. For the initial clique, these values are set to the cost to traverse it (Line 4).

Algorithm 1. Maximum length of an SAP in $M \times B$

```
Input: $B$ with initial clique $C_0$
 0: for each clique $C$ do
 1:     initialise $cost[C]$, $cost_f[C]$ as in Table 1
 2:     $reach[C] := reach_f[C] := 0$
 3: end for
 4: $reach[C_0] := cost[C_0]$, $reach_f[C_0] := cost_f[C_0]$
 5: for each clique $C$ of $B$ in a topological order, starting at $C_0$ do
 6:     for each successor clique $D$ of $C$ do
 7:         $reach[D] := \max\{reach[D], reach[C] + cost[D]\}$
 8:         if $D$ is accepting then
 9:             $reach_f[D] := \max\{reach_f[D], reach[C] + cost_f[D]\}$
10:         end if
11:     end for
12: end for
13: return $\max\{reach_f[C] \mid C \text{ is accepting}\}$
```

The algorithm traverses the cliques C of B in some topological order, starting with C_0, and examines all of C's successor cliques D. Value *reach* is updated to the maximum of its current value and the value obtained by reaching D via C. Value $reach_f$ is updated analogously, but only if D is accepting. After processing all cliques this way, the algorithm returns the maximum of the values $reach_f[C]$ over all accepting cliques.

If B has several initial cliques, the algorithm is performed for each of them in turn; in this case we return the maximum over all values obtained, as the maximum length of an SAP, for any Kripke structure M.

Complexity. The applications of + and max in the algorithm are to be understood as symbolic operations. That is, the result returned in Line 13 is an expression, involving d, rd and the linear operators +, max, and constant multiplication (the latter come from the base expressions in Table 1). In the worst case, the expression can contain a number of max applications that is linear in the number of edges of the SCC quotient graph. In many cases, however, the symbolic maximum can be evaluated to one of its arguments. For example, $\max\{rd, d\} = rd$, while $\max\{rd, 2d\}$ cannot be simplified.

The algorithm itself also has linear time complexity, as a (slight modification of the) standard algorithm to compute longest paths in a directed acyclic graph.

4 Linear Temporal Logic and Cliquey Automata

We now turn to temporal-logic model checking, and investigate the relationship with cliquey Büchi automata. In this section, we write bold-face **LTL**, **LTL\X**, and **CL** to denote the set of ω-regular languages that correspond to LTL formulas, LTL\X formulas, and cliquey Büchi automata respectively.

Lemma 6. CL $\subseteq$ LTL: *every cliquey generalised Büchi automaton can be encoded as an equivalent LTL formula.*

Proof: Given a cliquey automaton B, we show how to express its language in LTL. Recall that every strongly connected component of B is a clique, and that an *accepting clique* is one that has a non-empty intersection with each accepting set of B. There are only a finite number of SCC paths that go from a initial clique to an accepting clique. Moreover, for each pair of neighbouring cliques C_i, C_{i+1} along such a path, there is only a finite number of edges from C_i to C_{i+1}. We can therefore write the language of B as a finite union of 'path languages' each of which encodes the words along a path π to an accepting clique such that any successive cliques C_i, C_{i+1} along π are connected by a unique edge.

Each path language can be written as an ω-regular expression over the alphabet 2^{AP}, of the form

$$\underbrace{L(i_0).L(C_0)^*.L(o_0)}_{\text{first clique}}\ \underbrace{L(i_1).L(C_1)^*.L(o_1)}_{\text{second clique}}\ L(i_2)\ \ldots\ L(o_{f-1})\ \underbrace{L(i_f).L(C_f)^\omega}_{\text{final clique}} \quad (1)$$

where, for clique number j, $L(i_j)$ is the labelling of the unique in-point (entry) of C_j coming from C_{j-1}, $L(o_j)$ is the labelling of the unique out-point (exit) of C_j to C_{j+1}, and $L(C_j)$ is the union of the labellings of all Büchi states in C_j. For example, a clique of three states, with the entry state labelled a, exit state labelled b and a third state labelled c, where $a, b, c \in 2^{AP}$, is encoded as a regular expression $a.\{a, b, c\}^*.b$.

Expression (1) can be turned into a star-free ω-regular expression by replacing the subexpressions $L(C_l)^*$ by $\overline{\overline{\emptyset}.\overline{L(C_l)}.\overline{\emptyset}}$, where $\overline{}$ denotes complementation. Being star-free, it is well-known that this expression is equivalent to a suitable LTL formula [10,15]. □

Lemma 7. **CL $\not\subseteq$ LTL\X** : *there exist cliquey automata that cannot be encoded as LTL\X formulas.*

Proof: Consider the Büchi automaton B in Figure 1 (a). B is cliquey: the two p-labelled states form one SCC, the q-labelled state forms the other SCC, and both are cliques. B does not, however, correspond to any LTL\X formula: B's language contains the word $\{p\}.\{p\}.\{q\}^\omega$, but not the word $\{p\}.\{q\}^\omega$. Since these two words are stuttering equivalent, an encoding of B as an LTL\X formula would violate the stuttering closure of LTL\X. □

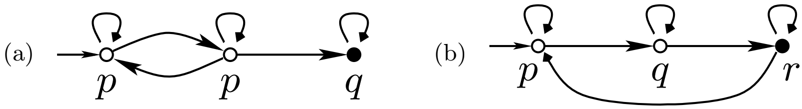


Fig. 1. (a) A cliquey Büchi automaton that does not correspond to any LTL\X formula; (b) A non-cliquey Büchi automaton with linear completeness threshold

Lemma 8. **LTL\X $\not\subseteq$ CL** : *not all LTL\X formulas have a cliquey automaton encoding.*

Proof: Let $AP = \{p, q, r\}$, and let $p!$ be a short-hand notation for $p \wedge \neg q \wedge \neg r$, and similarly for $q!$ and $r!$. Consider the LTL\X formula

$$\varphi \;=\; p! \;\wedge\; \mathsf{G}\;(\,(p! \Rightarrow (p! \,\mathsf{U}\, q!)) \;\wedge\; (q! \Rightarrow (q! \,\mathsf{U}\, r!)) \;\wedge\; (r! \Rightarrow (r! \,\mathsf{U}\, p!))\,)\;. \quad (2)$$

To prove that φ does not have a cliquey Büchi encoding, we first show:

Property 9. *Any cliquey Büchi automaton over $AP = \{p, q, r\}$ that accepts the word $(\{p\}.\{q\}.\{r\})^\omega$ also accepts some word in $(2^{AP})^*.\{q\}.\{p\}.(2^{AP})^*$.*

Proof: Let B be cliquey and accept $w := (\{p\}.\{q\}.\{r\})^\omega$. We show that B also accepts some word with the substring $\{q\}.\{p\}$. Any path π in B along which w is accepted contains infinitely many states with a label that is satisfied by $\{p\}$. Since B has finitely many states, these states are not all different; let b be a state with such a label that occurs twice along π. Let c be the state following the first occurrence of b; the label of c is satisfied by $\{q\}$. Since c is between two occurrences of b along π, states b and c belong to the same SCC. As B is cliquey, b and c are in fact part of one clique; thus there is an edge from c to b in B. Now consider the path π' that is identical to π, except that one occurrence of the edge $b \rightarrow c$ is replaced by the segment $b \rightarrow c \rightarrow b \rightarrow c$. Path π' is a valid path in B. It is also accepting, since we have only added two edges to the accepting path π. Finally, π' accepts a word that contains the substring $\{q\}\{p\}$. □

We can now prove Lemma 8: any automaton B that encodes φ in equation (2) accepts $(\{p\}.\{q\}.\{r\})^\omega$, but does not accept any word with substring $\{q\}.\{p\}$, since a path with such a trace would violate φ. It follows that B is not cliquey. □

We have so far shown that while **CL** is contained in **LTL**, **LTL\X** and **CL** are incomparable. Of particular interest is the intersection of the latter two: LTL\X formulas that have a cliquey representation. To narrow in on this class, consider the *unary temporal logic* fragment of LTL\X, denoted UTL\X, i.e., LTL\X without the *until* operator U (and, if one is using past temporal operators, without the past counterpart of U either). The formulas of UTL\X are built from atomic propositions using Boolean connectives and the unary temporal operators F (*eventually*), and $\overleftarrow{\mathsf{F}}$ (*sometime in the past*)—the dual operators G and $\overleftarrow{\mathsf{G}}$ are derived in the usual way. Formally, UTL\X is defined by the following grammar:

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathsf{F}\,\varphi \mid \overleftarrow{\mathsf{F}}\,\varphi\,,$$

where p is any atomic proposition. Naturally, we denote the associated class of languages **UTL\X**.

We now have:

Lemma 10. **UTL\X** $\subseteq$ **CL** : *every UTL\X formula can be encoded as a generalised cliquey Büchi automaton.*

Proof: We prove this lemma by constructing, for any UTL\X formula φ, a cliquey automaton A_φ that encodes it. Define the *closure* of φ to be the set $cl(\varphi)$ of all subformulas of φ and their negations, where we identify $\neg\neg\psi$ with ψ.

Say that $\mathbf{s} \subseteq cl(\varphi)$ is a *complete type* if (i) for each formula $\psi \in cl(\varphi)$ precisely one of ψ and $\neg\psi$ is a member of $\mathbf{s}$; (ii) $\psi_1 \wedge \psi_2 \in \mathbf{s}$ iff $\psi_1 \in \mathbf{s}$ and $\psi_2 \in \mathbf{s}$; (iii) $\psi \in \mathbf{s}$ implies $\mathsf{F}\,\psi \in \mathbf{s}$ and $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{s}$. Given types $\mathbf{s}$ and $\mathbf{t}$, write $\mathbf{s} \sim \mathbf{t}$ if $\mathbf{s}$ and $\mathbf{t}$ agree on all formulas whose outermost connective is a temporal operator, i.e., $\mathbf{s} \sim \mathbf{t}$ exactly if for all formulas ψ we have $\mathsf{F}\,\psi \in \mathbf{s}$ iff $\mathsf{F}\,\psi \in \mathbf{t}$, and $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{s}$ iff $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{t}$. Write tp_φ for the set of complete types for φ.

An ω-word w over alphabet 2^{AP} naturally extends to an ω-word $\overline{w} = \overline{w}_0\overline{w}_1\ldots$ over alphabet tp_φ, where $\overline{w}_i = \{\psi \in cl(\varphi) \,|\, (w,i) \models \psi\}$.

Recall that a generalised Büchi automaton has a family $\mathcal{A} = \{A_1,\ldots,A_n\}$ of accepting sets such that an accepting run must visit each A_i infinitely often. We define a generalised Büchi automaton $A_\varphi = (S, S_0, R, L, \mathcal{A})$ that accepts $\{w \in (2^{AP})^\omega \,|\, (w,0) \models \varphi\}$. The set of states is $S = tp_\varphi$, with the set S_0 of initial states consisting of those $\mathbf{s} \in tp_\varphi$ such that (i) $\varphi \in \mathbf{s}$ and (ii) $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{s}$ only if $\psi \in \mathbf{s}$. The state-labelling function $L\colon S \to \mathcal{B}(AP)$ is defined by $L(\mathbf{s}) = \bigwedge(\mathbf{s} \cap AP) \wedge \bigwedge\{\neg p \,|\, p \in AP \setminus \mathbf{s}\}$. The transition relation R consists of those pairs $(\mathbf{s},\mathbf{t})$ such that

(i) $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{t}$ iff either $\psi \in \mathbf{t}$ or $\overleftarrow{\mathsf{F}}\,\psi \in \mathbf{s}$,
(ii) $\mathsf{F}\,\psi \in \mathbf{s}$ and $\psi \notin \mathbf{s}$ implies $\mathsf{F}\,\psi \in \mathbf{t}$, and
(iii) $\neg\,\mathsf{F}\,\psi \in \mathbf{s}$ implies $\neg\,\mathsf{F}\,\psi \in \mathbf{t}$.

The accepting set family is $\mathcal{A} = \{A_{\mathsf{F}\,\psi} \,|\, \mathsf{F}\,\psi \in cl(\varphi)\}$, where $A_{\mathsf{F}\,\psi} = \{\mathbf{s} \,|\, \psi \in \mathbf{s} \text{ or } \mathsf{F}\,\psi \notin \mathbf{s}\}$. This completes the definition of A_φ.

We finally argue that automaton A_φ is cliquey: by the definition of the transition relation of A_φ, states $\mathbf{s}$ and $\mathbf{t}$ are in the same connected component iff $\mathbf{s} \sim \mathbf{t}$. Any two states $\mathbf{s}$ and $\mathbf{t}$ with $\mathbf{s} \sim \mathbf{t}$ are connected by a transition. □

Combining Theorem 5 and Lemma 10 yields one of our main results:

Theorem 11. *Every UTL\X formula admits a linear completeness threshold.*

Finally, one may wonder whether LTL\X formulas that have a cliquey representation are in fact always equivalent to some UTL\X formula. The answer is no, as our next result shows:

Lemma 12. LTL\X $\cap$ CL $\not\subseteq$ UTL\X : *there exist LTL\X formulas that do have a cliquey representation yet are not equivalent to any UTL\X formula.*

Proof (sketch)**:** Let a, b, c be distinct elements of 2^{AP}, and consider the language $L = (a + b + c)^*.a.a^*.b.(a + b + c)^\omega$. L is captured by the LTL\X formula $\mathsf{F}(a \wedge (a \,\mathsf{U}\, b))$, and it is also clear that L is cliquey. Using the results of [17], one can show that this language is inexpressible in UTL (let alone UTL\X). For example, one can compute the syntactic monoid associated with L and invoke the characterisation of syntactic monoids of UTL-definable languages from [17] to obtain the desired result. We omit the details. □

Figure 2 summarises our expressiveness results. All inclusions are strict.

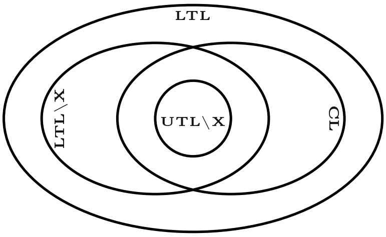


Fig. 2. Relationships among various classes of ω-regular languages

5 Beyond Cliqueyness

Two natural questions arise as to whether cliqueyness is necessary in order to achieve a linear completeness threshold, and whether there actually are any ω-regular languages that fail to have linear completeness thresholds. We answer the first question negatively and the second one positively. In fact, we show that ω-regular languages can be engineered to have completeness thresholds bounded below in the worst case by superpolynomial and even exponential functions of the recurrence diameter of Kripke structures.

5.1 Linear Completeness Thresholds without Cliqueyness

Consider the Büchi automaton B depicted in Figure 1 (b). It is clearly not cliquey and is in fact *semantically* non-cliquey, i.e., not equivalent to any cliquey Büchi automaton. To see this, observe that B accepts the word $w := (\{p\}.\{q\}.\{r\})^\omega$, yet no word with the substring $\{q\}.\{p\}$. By Property 9, B cannot be equivalent to a cliquey automaton.

We claim that B nonetheless has a linear completeness threshold: namely, for any Kripke structure M, $\mathit{sap}(M \times B) \leq \mathit{rd}(M) + 1$. Indeed, if an SAP had length greater than $\mathit{rd}(M) + 1$, its projection onto M would have to exhibit an 'inner' loop that for some reason could not be cut out. A straightforward case analysis then quickly leads to a contradiction.

5.2 Büchi Automata with Non-linear Completeness Thresholds

On the other hand, not every LTL formula and in fact not every LTL\X formula has a linear completeness threshold. Consider the non-cliquey automaton B in Figure 3, which encodes the LTL\X formula

$$\begin{array}{rcllllll} \varphi & = & p \wedge \neg r \wedge \mathsf{G}\ (& (p \wedge \neg r) \Rightarrow & (\ (p \wedge \neg r)\ \mathsf{U} & (q \wedge \neg r) &) \wedge \\ & & & (q \wedge \neg r) \Rightarrow & (\ (q \wedge \neg r)\ \mathsf{U} & r! &) \wedge \\ & & & r! \Rightarrow & (\ \ r!\ \mathsf{U} & (p \wedge \neg r) \vee \mathsf{G}\, r!) &)\,. \end{array}$$

Again, the notation $r!$ is short for $r \wedge \neg p \wedge \neg q$.

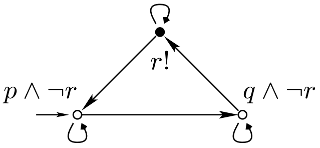


Fig. 3. A non-cliquey Büchi automaton with superpolynomial completeness threshold

To show that B has no linear completeness threshold, we construct a collection $(M_i)_{i=1}^{\infty}$ of Kripke structures such that, for each i, we have $\mathit{sap}(M_i \times B) \geq i/4 \cdot \mathit{rd}(M_i)$.

The construction is depicted in Figure 4. M_i contains i copies of a q-labelled loop, 'q-loop' for short. Each q-loop comprises i states. Consecutive occurrences of the q-loop are connected via an r-labelled state, or r-state for short. The final r-state has a self-loop.

Let us compute M_i's recurrence diameter: a longest loop-free path starts at the first state of the first q-loop (a successor of the first p, q-state), follows that loop around to the first p, q-state, then follows the baseline path—skipping all intermediate q-loops—all the way to the final q-loop. It then enters that loop and follows it to the last of its states (*before* the loop is closed). M_i thus has a recurrence diameter of at most $4i$.

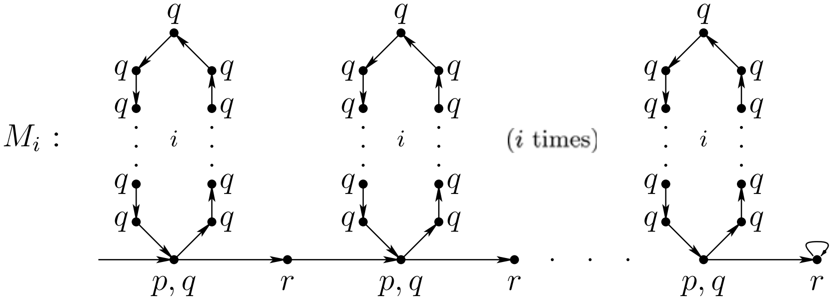


Fig. 4. Kripke structure family $(M_i)_{i=1}^{\infty}$ witnessing a non-linear completeness threshold

An SAP of $\Pi = M \times B$, however, must take **all** q-loops. To see this, consider the initial state of Π, which is labelled $(\{p, q\}, p \wedge \neg r)$. B does not allow an r-state as successor (both possible transitions [one of which is a B-self-loop] require successors satisfying $\neg r$). Thus the joint path must enter the first q-loop. During this loop, B stays in the $(q \wedge \neg r)$-state, up to and including the time when M finishes the loop and arrives back at the p, q-state. At this time the shortest path continues at the state labelled $(\{r\}, r!)$, followed by the state labelled $(\{p, q\}, p \wedge \neg r)$, at which point it is forced into the next q-loop of M_i. Note that, for this path to be accepting, it has to visit an r-state of M_i infinitely often, which is only possible via the self-loop reachable *after* all the q-loops have been taken.

Having to go through i loops each of size i, an SAP of Π has length at least i^2. Combining this with the size of the recurrence diameter of at most $4i$, we see that the completeness threshold for B is at least *quadratic* in the recurrence diameter of Kripke structures. □

It is not difficult to see our family of Kripke structures can in fact be modified to exhibit a *cubic* completeness threshold for our very same automaton B, by modifying the loops slightly and grafting a further additional family of loops onto each of them. In this vein, one sees that completeness thresholds exceeding any given polynomial can in fact be achieved, so that our formula φ and Büchi automaton B have *superpolynomial* completeness threshold.

In fact, even *exponential* completeness thresholds can be achieved for LTL formulas.[4] Consider a family of Kripke structures, each of which resembles a full binary tree, with bidirectional edges between every parent and child. The recurrence diameter of any such structure is the length of a longest loop-free path from one leaf to another, and is therefore logarithmic in the size of the structure. These structures can however be instrumented in such a way that a certain LTL formula forces the unique accepting path to perform a depth-first traversal of the entire tree, resulting in a path of length exponential in the recurrence diameter. To achieve this, atomic propositions are used to keep track of the depth of nodes modulo 3, and further propositions label the root, leaves, and left and right

[4] We are grateful to one of the anonymous referees for this observation.

children accordingly. A traversal of the tree is then orchestrated by requiring that (i) whenever an interior node is entered from above (which is determined by knowledge of the depths modulo 3 of the present node and that of the previous one), then the left child should be visited next; (ii) whenever a non-leaf node is returned to from a left child, then the right child should be visited next; and (iii) whenever a non-leaf node is returned to from a right child, then the parent node should be visited next. Finally, the rightmost leaf is labelled with a special proposition which the formula requires to hold eventually.

6 Concluding Remarks

We have presented a method for calculating fairly tight, linear completeness thresholds for a large class of LTL specifications. The algorithm we propose is highly efficient, running in time linear in the size of the Büchi automaton. Several potential bottlenecks however remain, including the following two:

- Computing the diameter and recurrence diameter of a large Kripke structure can be computationally prohibitive; one possible remedy might be to settle for tractable over-approximations of the diameters, as in [2], in a trade-off which would likely require careful consideration.
- It has often been empirically observed that bounded model checking computations tend not to scale up very well. Since many Kripke structures have deep recurrence diameters (of the order of the total number of states, for example), one can expect that exploring the system to the required depth prove in certain cases to be intractable.

Nonetheless, this is an area of active research in which progress is being made on several fronts. Our hope is that the techniques presented here may prove beneficial not only to practitioners, but also to other researchers whose technology it might potentially complement.

Alongside these practical considerations, two interesting theoretical questions arise: (i) is it decidable whether a given LTL formula (or more generally a given ω-regular language) has a linear completeness threshold; and (ii) is the completeness threshold of an ω-regular language always either linear or superpolynomial? We leave these questions as further research.

References

1. Awedh, M., Somenzi, F.: Proving more properties with bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 96–108. Springer, Heidelberg (2004)
2. Baumgartner, J., Kuehlmann, A., Abraham, J.A.: Property checking via structural analysis. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 151. Springer, Heidelberg (2002)
3. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)

4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
6. Clarke, E., Kröning, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
7. Clarke, E.M., Allen Emerson, E., Sifakis, J.: Model checking: Algorithmic verification and debugging. CACM 52(11), 75–84 (2008)
8. Ganai, M., Gupta, A., Ashar, P.: Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: ICCAD, pp. 510–517 (2004)
9. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV, pp. 3–18 (1995)
10. Kamp, H.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California (1968)
11. Lamport, L.: What good is temporal logic. In: IFIP Congress, pp. 657–668 (1983)
12. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems—Specification. Springer, Heidelberg (1991)
13. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 250. Springer, Heidelberg (2002)
14. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
15. Schützenberger, M.-P.: On finite monoids having only trivial subgroups. Information and Control 8(2), 190–194 (1965)
16. 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)
17. Thérien, D., Wilke, T.: Over words, two variables are as powerful as one quantifier alternation. In: STOC, pp. 234–240 (1998)

Interpolation-Based Software Verification with WOLVERINE*

Daniel Kroening[1] and Georg Weissenbacher[2]

[1] Computer Science Department, Oxford University
[2] Department of Electrical Engineering, Princeton University

Abstract. WOLVERINE is a software verification tool using Craig interpolation to compute invariants of ANSI-C and C++ programs. The tool is an implementation of the *lazy abstraction* approach, generating a reachability tree by unwinding the transition relation of the input program and annotating its nodes with interpolants representing safe states. WOLVERINE features a built-in interpolating decision procedure for equality logic with uninterpreted functions which provides limited support for bit-vector operations. In addition, it provides an API enabling the integration of other interpolating decision procedures, making it a valuable source of benchmarks and allowing it to take advantage of the continuous performance improvements of SMT solvers. We evaluate the performance of WOLVERINE by comparing it to the predicate abstraction-based verifier SATABS on a number of verification conditions of Linux device drivers.

1 Introduction

The last decade has seen significant progress in the area of automated software verification, manifesting itself in a number of impressive verification tools. A recent and comprehensive survey of software verification techniques is provided in [1] and a comparison of verification tools can be found in [2]. One approach that received particular attention is predicate abstraction [3], a technique that constructs a conservative abstraction of the original program using a finite set of first-order-logic predicates to track relevant facts about the program variables.

The performance of such predicate abstraction-based software model checkers is contingent on suitable predicates. Contemporary verification tools (e.g., Microsoft's SLAM [4]) derive these predicates from spurious counterexamples in an iterative manner [5,6]. Recent incarnations of this technique (such as BLAST [7]) rely on Craig interpolation to derive predicates, taking advantage of the inherent properties of interpolants which enable concise abstractions.

Some flavours of this interpolation-based abstraction mechanism avoid the use of predicate abstraction to construct an abstract transition relation altogether [8,9,10]. This omission has several advantages:

* Supported by the EU FP7 STREP MOGENTES and a gift from the Intel Labs Academic Research Office.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 573–578, 2011.

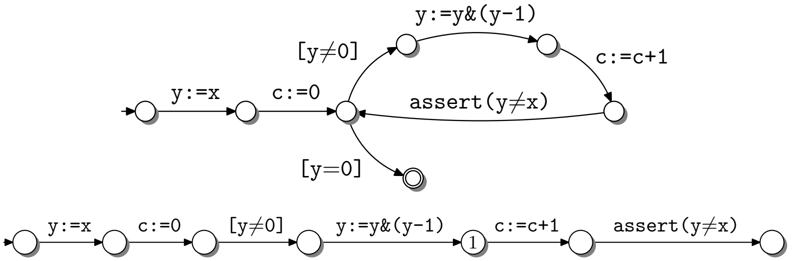


Fig. 1. A program and one of its execution traces

- It eliminates computationally expensive calls to a theorem prover that predicate abstraction-based verifiers require to construct an abstraction, and
- it decreases the size and the complexity of the implementation of the verification tool significantly (by about two thirds in our experience).

The algorithm presented in [8] (and our implementation Wolverine) essentially follows the *lazy abstraction* paradigm [11], but retains the abstract transition function at the coarsest level (determined by the control-flow-graph of the program under scrutiny). The reachability tree obtained by unwinding this transition function is annotated with interpolants representing an over-approximation of the reachable states. A fixed point of these annotations constitutes an invariant establishing the correctness of the program with respect to a given safety property. Section 2 provides more details.

Contributions. Wolverine is a software verification tool checking reachability properties stated in terms of assertions and implements the algorithm described in [8]. It is, to the best of our knowledge, the first freely available[1] software model checker for C/C++ programs based on this algorithm.

Wolverine features a built-in interpolating decision procedure for equality logic with uninterpreted functions which provides limited support for bit-vector operations, while many comparable verification tools use linear arithmetic to approximate semantics of the program. In addition, it provides a programming interface (described in Section 3) enabling the integration of other interpolating decision procedures, allowing it to take advantage of the continuous performance improvements of interpolating SMT solvers (see, for instance, [12,13,14]).

We present an evaluation of our implementation in Section 4 and provide a tool to generate additional benchmarks on the website of Wolverine.

2 Implementation

The implementation of Wolverine is based on the CProver framework(written in C++), which also forms the foundation of the verification tools

[1] Source available under a BSD-style license on `http://www.cprover.org/wolverine`.

CBMC [15], SATABS [16], and IMPACT [8]. We describe the implementation of WOLVERINE using the example program in Figure 1. The program implements Wegner's algorithm, determining how many bits of x are set to one. The assertion assert(y≠x) is a naïve safeguard against non-termination. WOLVERINE uses symbolic simulation to construct a reachability tree. To this end, it unwinds the control flow graph of the program until an assertion is reached. The shortest execution trace of our example program reaching the assertion is shown at the bottom of Figure 1. In order to check whether the trace violates the assertion, WOLVERINE transforms it into static single assignment form:

$$\overbrace{(\mathtt{y}_1 = \mathtt{x}_0) \wedge (\mathtt{y}_1 \neq 0) \wedge (\mathtt{y}_2 = \mathtt{y}_1\&(\mathtt{y}_1 - 1))}^{\text{prefix}} \overset{①}{\wedge} \overbrace{(\mathtt{y}_2 = \mathtt{x}_0)}^{\neg\text{ assertion}} \qquad (1)$$

Using slicing, we eliminate the assignments to c, since they are not relevant to the correctness of the program. The effect of negating the asserted condition is that every satisfying assignment of Formula (1) represents a witness for an assertion violation. Note, however, that the formula is unsatisfiable and therefore this execution cannot violate the assertion. Accordingly, the sub-formula tagged "prefix" in Formula (1) represents a set of reachable states (at location ① in the trace) that is *safe* with respect to the assertion.

WOLVERINE splits the symbolic representation of the execution trace into n partitions $A_1, \ldots, A_n$, one for each basic block traversed by the trace. It passes these n formulas on to an *interpolating* decision procedure (by default the built-in algorithm described in [17,18]), which returns $n-1$ interpolants $I_1, \ldots, I_{n-1}$ that satisfy the following conditions [8]:

1. For all $1 \leq j \leq n$, $(I_{j-1} \wedge A_j)$ implies I_j (with $I_0 = \mathsf{true}$ and $I_n = \mathsf{false}$), and
2. for all $1 \leq j < n$, I_j refers only to SSA variables that occur in $A_1, \ldots, A_j$ as well as in $A_{j+1}, \ldots, A_n$.

The first condition guarantees that each interpolant I_j represents a set of safe states at the respective program location from which no state violating the assertion is reachable via the given trace (i.e., the interpolants and the program statements in the trace form Hoare triples). The second condition above guarantees that the interpolants refer only to SSA variables that are *live* at the corresponding location in the trace. For instance, a valid sequence of interpolants for the formulas $(\mathtt{y}_1 = \mathtt{x}_0)$, $(\mathtt{y}_1 \neq 0)$, $(\mathtt{y}_2 = \mathtt{y}_1\&(\mathtt{y}_1 - 1))$, and $(\mathtt{y}_2 = \mathtt{x}_0)$ would be $\mathtt{y}_1 = \mathtt{x}_0$, $(\mathtt{y}_1 = \mathtt{x}_0) \wedge (\mathtt{y}_1 \neq 0)$, and $(\mathtt{x}_0 \neq 0) \wedge (\mathtt{y}_2 \leq \mathtt{x}_0 - 1)$.

After mapping the SSA variables back into the original program context, WOLVERINE annotates the corresponding path in the reachability tree accordingly, e.g., the node ① is labelled $(\mathtt{x} \neq 0) \wedge (\mathtt{y} \leq \mathtt{x} - 1)$.

WOLVERINE continues expanding the reachability tree until each leaf is either fully expanded or *covered* by a previously discovered node. A node is covered if its (or one of its predecessors') annotation implies the annotation of a previously discovered node associated with the same program location.[2] For instance, if

[2] The fact that interpolation is non-monotonic imposes some restrictions on the covering relation, which are described in more detail in [8,19].

Wolverine annotates a node ② (which succeeds ① in the reachability tree and also corresponds to the program location following the assignment `y:=y&(y-1)`) with $(\mathtt{x} \neq 0) \wedge (\mathtt{y} \leq \mathtt{x} - 1)$, then ② is covered by ①.

The built-in decision procedure supports bit-vector operations using a limited set of inference rules (such as $(t_1 = t_2 \,\&\, t_3) \vdash (t_1 \leq t_2)$ and $(1 > t_1) \vdash (t_1 = 0)$ for terms t_i of type unsigned integer). Details are provided in [19,17]. Moreover, it uses eager bit-blasting and a SAT solver to identify an unsatisfiable core before invoking the "word-level" interpolating decision procedure [19,18]. If the decision procedure fails to provide an interpolant, Wolverine falls back on using the weakest precondition. Finally, using the option `--interpolator smt-out`, Wolverine is able to print the SSA instances in the SMT-LIB format, enabling the generation of benchmarks.

3 Interface for Interpolating Decision Procedures

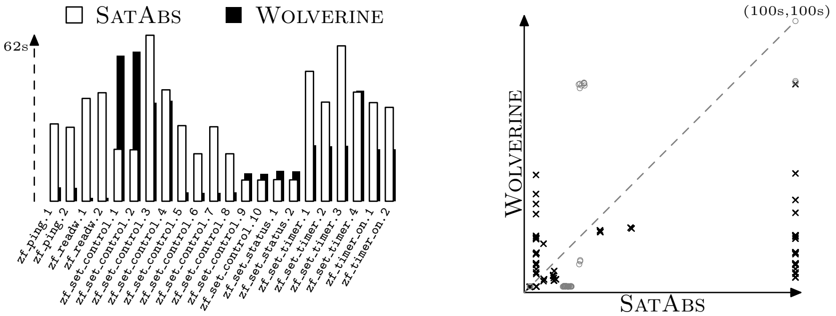
Wolverine provides a C++ interface for calling external interpolating decision procedures. In order to integrate an external solver into Wolverine, the programmer has to implement a class inheriting from `external_interpolatort`:

```
class external_interpolatort: public wolver_interpolatort {
     ...
     virtual bool initialise();
     virtual bool process_options(const optionst&);
  protected:
     virtual bool translate(const expr_listt&)=0;
     virtual decision_proceduret::resultt solve()=0;
     virtual bool read_interpolants(expr_listt&)=0;      };
```

The public methods `initialise` and `process_options` provide an opportunity to initialise the external tool and to deal with command line parameters. Wolverine provides the class `external_processt`, which supports the execution of and communication with command line tools.

The methods `translate` and `read_interpolants` are required to convert formulas between the representation used by the external interpolator and expressions in the CProver format. CProver expressions (represented by the class `exprt`) are annotated syntax trees with typing information. The class `typet` is used to store types. Wolverine expects the interpolants returned to be typed. Accordingly, an interpolator which discards the typing information needs to restore it before returning a result to Wolverine. The CProver framework provides support for this task in form of the methods `c[pp]_typecheck`.

The method `solve` returns `D_UNSATISFIABLE`, `D_SATISFIABLE`, or `D_ERROR`. In the latter case, Wolverine provides the option to fall back on computing interpolants using the weakest precondition. If the instance is satisfiable, the trace represents a valid counterexample and is reported. Otherwise, the method `read_interpolants` is expected to return in its parameter a sequence of typed expressions which satisfy the conditions stated in Section 2.



(a) `machzwd`: IO port assertions

(b) `machzwd` and `nbd`: IO & bounds

Fig. 2. Performance results Wolverine vs SatAbs on DDVerify drivers

4 Checking Linux Device Drivers

Figure 2 provides a comparison of Wolverine with the predicate-abstraction based verifier SatAbs on a number of sequential device driver benchmarks (generated with DDVerify [20], which provides a harness and an OS model annotated with assertions) on a 3GHz Intel Core i7 CPU with 4GB RAM.[3] Figure 2(a) shows the run-time of Wolverine and SatAbs on 22 assertions related to the usage of IO ports for the `machzwd` device driver [20]. Wolverine performs better than SatAbs in all but 6 cases. The performance gain is particularly impressive for the assertions `zf_readw.`[1,2], for which both tools report a counterexample. We attribute this to the lower overhead of the search algorithm of Wolverine. In the case of the claims `zf_set_control.`[1,2], we observe a large number of coverage checks in Wolverine for different branches of the reachability tree, and the *eager* abstraction approach of SatAbs prevails.

The scatter-plot in Figure 2(b) shows the run-time for 73 array bound checks for the `machzwd` driver (displayed using ×) and 78 array bound and IO properties for the driver `nbd` (indicated by ○). SatAbs exceeded the time-out of 100 seconds for 23 properties of `machzwd`, and SatAbs as well as Wolverine timed out in 22 cases for `nbd`. Our results suggest that, while SatAbs is significantly faster when few predicates are sufficient to prove an assertion correct, Wolverine's lazy approach is more robust as the number of predicates increases.

5 Conclusion

Wolverine is a freely available implementation of the interpolation-based lazy abstraction algorithm presented in [8]. Its modular design enables the integration of modern interpolating SMT solvers, making it future-proof and (when combined with DDVerify [20]) a valuable source for benchmarks. Our experimental evaluation shows that our implementation is competitive when compared to existing predicate-abstraction based verification tools. As future work, we intend to integrate and study the performance impact of different interpolation techniques.

[3] Performance results for the device drivers presented in [7] are reported in [19].

References

1. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys 41(21), 1–21 (2009)
2. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27, 1165–1178 (2008)
3. 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)
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
6. Ball, T., Rajamani, S.: Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report 2002-09, Microsoft Research (2002)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM, New York (2004)
8. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
9. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482. ACM, New York (2010)
10. Caniart, N.: MERIT: An interpolating model-checker. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 162–166. Springer, Heidelberg (2010)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM, New York (2002)
12. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
13. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The openSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
14. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Transactions on Computational Logic (2010) (to appear)
15. Clarke, E., 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)
16. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
17. Kroening, D., Weissenbacher, G.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 150–168. Springer, Heidelberg (2011)
18. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: FMCAD, pp. 85–89. IEEE, Los Alamitos (2007)
19. Weissenbacher, G.: Program Analysis with Interpolants. PhD thesis, Oxford University (2010)
20. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: ASE, pp. 501–504. IEEE, Los Alamitos (2007)

Synthesizing Biological Theories

Hillel Kugler, Cory Plock, and Andy Roberts

Microsoft Research, Cambridge, UK
hkugler@microsoft.com, coryp@acm.org, i-anrobe@microsoft.com

Abstract. Studies of biological systems are often facilitated by diagram models that summarize the current understanding of underlying mechanisms. The increasing complexity of our understanding of biology necessitates computational models that can extend these representations to include their dynamic behavior. We present here a new tool we call *Synthesizing Biological Theories* which enables biologists and modelers to construct high-level theories and models of biological systems, capturing biological hypotheses, inferred mechanisms, and experimental results within the same framework. Among the key features of the tool are convenient ways to represent several competing theories and the interactive nature of building and running the models using an intuitive, rigorous scenario-based visual language. The definition of the modeling language is geared towards enabling formal verification and analysis.

1 Introduction

Biological systems and their ability to function robustly and perform complicated processes have been a source of fascination and scientific enquiry for many years. Recently, significant advances in biology have enabled a deeper mechanistic understanding of many fundamental underlying processes. Computational modeling is gaining prominence in the biological community as it allows us to formulate hypotheses and reason about the biological implications of these models in a precise and formal way that could lead to an improved scientific understanding. Performing certain experiments in the biological system is often not possible, or may require a lot of work to overcome technical hurdles. Yet, running a simulation and analyzing the underlying model is possible and holds a potential of guiding experimental work and thinking about the consequences of performing certain experiments even before they become feasible.

In this paper we present a new tool for biological modeling we call *Synthesizing Biological Theories* (SBT) which enables biologists and modelers to construct high-level theories and models of biological systems, capturing biological hypotheses, inferred mechanisms, and experimental results within the same framework using an intuitive, rigorous scenario-based visual language. The tool is publicly available at [SBT11] together with a user manual, documentation and sample models. In this paper we present the main challenges we tried to address while building the tool, its design principles, highlights of the main functionality and also briefly discuss several of the future directions.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 579–584, 2011.

2 Main Challenges and Design Principles

We set out to build a tool that would be intuitive for experimental biologists, yet powerful to allow modeling, simulation and analysis of large and complex biological systems. We use scenarios as the basic modeling unit for specifying behavior. In SBT Version 1.0 we support a scenario-based language inspired by live sequence charts [DH01]. The reason for adopting a scenario based language is the ability to break up a complex model specifications into many charts (scenarios). Each chart can represent an explicit hypothesis about the system, a certain property that holds for the system, or a representation of an experimental result. The ability to view a chart in isolation from the entire model and examine its assumptions is important for biological modeling; the utility and impact of the model depends critically on all the biological assumptions it makes and the ability of experimental biologists to understand the representation in detail.

A user interface allows the user to build the model by interacting with a simple browser to define classes, types, objects and their properties and methods. Navigating complex models with many charts and filtering relevant objects is supported, where the object name serves as the key and all relevant items starting with a given prefix are displayed. The motivation for using this modeling scheme rather than coding custom software applications is to provide an easier entry point for experimental biologists with the goal of making such tools useful for a wider community. For this reason we use a visual representation for the scenario modeling language and provide a simple-to-use interface to construct scenarios, using the same visual representation during execution to provide meaningful feedback to the user. The tool was designed to support complex models and has been tested with models containing thousands of objects and hundreds of charts.

3 The SBT Language and Tool

We now describe the main features of our tool *Synthesizing Biological Theories* (SBT), with emphasis on the novel aspects of our work. The current version of SBT is 1.0, publicly available at [SBT11]. We also mention some extensions that we implemented in experimental versions of the tool but are not part of the first version, which we plan to make available in next versions. SBT uses a specification language inspired by live sequence charts (LSCs) [DH01].

3.1 Classes, Objects, Types

We adopt an object oriented framework, where a model is composed of a set of classes which are restricted to single inheritance. An object is an instance of a class, and a model contains a definition of static objects that are present at runtime when the model starts executing. In addition dynamic objects can be created and deleted during runtime as specified in the model scenarios.

A *class* is a structural template from which physical and conceptual entities called *objects* originate. In a biological model a class can, for example, represent a certain type of cell, while objects of this class are the actual cells. Classes, which can be added by the user in SBT, consist of *methods*, *messages*, and *properties*. Methods are parameterless communications that may be received by the class or its objects. Messages are similar to methods, except with parameters. Properties are variables belonging to the class and collectively describe the state of the class and its objects. Properties can be modified using messages called *setters*.

Properties and message parameters have *types* which describe their domain. User-defined types can be constructed by parameterizing the SBT primitives `range` and `enumeration`, `bound`, and `boolean`.

3.2 Generalized Charts

The LSC definition [DH01] makes a distinction between two types of charts, *universal* and *existential*, but the SBT execution engine maps these to a single type of chart we call a *generalized chart*. This single chart type is made possible through the use of a third *warm* temperature [Plo08]. The intuition is that a cold LSC location is generally understood to denote that no progress is required beyond the location, whereas a hot location means that progress is required. Applying this notion to precharts, however, is not straightforward: strictly interpreted, this means that the chart would never have to progress beyond the initial locations of the chart and therefore the chart is vacuously satisfied without ever having left the initial state. What is actually intended is a similar but somewhat looser interpretation whereby progress is still not required, but advancement is required if an enabled event occurs at any prechart location. Our use of generalized charts and warm temperatures allows SBT to capture our intended semantics in an elegant way while avoiding special cases in the underlying execution engine.

3.3 Subcharts

Subcharts are charts which can be embedded in a parent chart. The execution engine internally maintains subchart executions as their own individual execution, which can satisfy or violate just as any other chart. Subcharts follow an orthogonal approach whereby any chart in the requirements may be added as a subchart to any other LSC[1].

Upon termination, subchart termination status propagates up to the parent chart, which will respond accordingly. For example, if a subchart is located within the prechart of its parent and it violates, the parent will close without violation. If the subchart is located in the main chart of the parent, however, the parent chart will also violate. When the subchart is satisfied, its parent may progress beyond the subchart.

3.4 Symbolic Instances

Symbolic instances are those that can be bound to objects of a specified class at runtime when the first visible event on the instance line occurs. A quantifier

[1] The user interface of SBT 1.0 does not fully support this currently although the execution engine does.

expression over the class properties is associated with the symbolic instance. For each object satisfying the quantifier expression at binding-time, a new chart is created in which the symbolic instance is replaced with the concrete object.

The set can be universally or existentially quantified: all executions in the set must satisfy for universal instances, whereas only one must satisfy for existential instances. The symbolic instances are numbered such that charts with multiple symbolic instance lines will be quantified in a specific order.

Symbolic instances can also be used to create new objects dynamically from a class definition using the special SBT method `create`. Once created, dynamic objects can be referenced using symbolic instances and even destroyed.

4 Execution Modes

SBT presents a set of novel language constructs and runtime execution semantics geared towards biological modeling, we highlight some of these in the following section.

4.1 Deterministic vs. Nondeterministic Execution

Scenarios consist of *visible* events which are observable and *hidden* events which are non-observable, where SBT (on behalf of the biological system) picks and executes events internally. The choice of which internal events the biological system picks can have a great impact on whether or not the chart is eventually satisfied (i.e., accepts the behaviors.) SBT supports both a deterministic and nondeterministic mode:

According *deterministic* mode, a visible event is executed and then the execution engine determines whether any hidden events are enabled. If so, it arbitrarily selects and executes one at the same instant[2] as the closest preceding visible event. This is repeated until no more hidden events are enabled.

On the other hand, *nondeterministic* mode attempts all possibilities by effectively splitting a single run into multiple threads of parallel execution, of which one must eventually be satisfied. The LSC is considered satisfied if any run in the set accepts. This mode operates across two axes of nondeterminism: it selects not only the choice of which hidden event to execute next, but also the choice of whether to proceed to then next hidden event (if one exists) or remain in the present location.

For nondeterministic mode we found that the visualization of progress along the charts is useful to follow the nondeterminism. Nondeterminism is essential for modeling of biological systems as it provides an easy way for the modeler to represent a certain assumption when the precise mechanism is still unknown. Additional experience still needs to be gained to identify the most useful way to specify this nondeterminism and to restrict it in cases where it is not needed to simplify progress visualization and improve performance.

[2] The word “instant” is used here in the sense of reactive systems operating under the synchrony hypothesis.

4.2 Stepping Mode

Stepping mode specifies how events are to be generated. There are two modes:

Interactive mode allows the user to execute any event by selecting it from a list of available events. Both *system* and *environment* events are available for selection, where system events are those controlled by the biological model and environment events are external. *Super-stepping* mode allows the user to execute a visible *environment* event, and the biological system will respond with a multi-event response called a *super-step*, which terminates when no system events are enabled or the main charts of all universal charts have closed.

In super-stepping mode, the event selection policy can either be *arbitrary* (default) or *random*, where the engine will select only non-violating events according to the chosen policy.

Violating executions are removed from the nondeterministic execution set. The set itself violates when and if the last execution in the set violates.

4.3 Configurations

A user can run simulations and investigate the system behavior at any stage during construction of the model. The user can set a configuration by selecting which charts in the model are used and which are not, allowing to ignore charts that are still "in progress." Several different configurations per model can be maintained, one of them is designated as the active configuration in use, the typical usage of multiple configurations in biological modeling is for maintaining several competing hypotheses that share most of the scenarios in the model but differ on certain specific charts thus representing different variants of the model.

4.4 Display Options

The basic visualization provided by SBT shows property values of all objects in the model (i.e., all static objects and dynamic objects that have been created but not yet deleted) and the progress of all active charts. The user can set any of these options to be active or not. Runs of simulations can be recorded and later replayed for more careful investigation. The execution engine was designed to run independently from the user interface to ensure a clean separation between the modules and allowing to connect other components to the execution engine, version 1.0 supports connecting the executed models to a processing visualization environment.

5 Biological Examples and Future Directions

A scenario-based approach using LSCs and the Play-Engine [HM01] to model vulval development in *C. elegans* was presented in [KKM08]. The model focused on six cells and how their fate is specified. This is conceptually an important process in developmental biology, where developing up-to-date realistic and predictive models for this system is a major challenge requiring development of powerful tools and methodologies [FH07].

To deal with models with large cell populations, SBT supports dynamic creation and destruction of objects. It is now being actively used to study stem cell population dynamics in the *C. elegans* germ line, embryogenesis and emergent behavior of bacteria populations [KLH10]. As these models and new ones are published, they will be made part of the tool's online samples and made available as a resource to the research community.

We hope that in the future, similar competitions to those conducted for SMT solvers in the SMT-competitions can be organized for biological models. Since the format of the models and the properties are still under investigation, we believe that making our tool and other such tools available will help allow CAV and the formal verification community to become familiar with the challenges and special characteristics of such models. SBT experimental versions support verification and analysis of models [KS09, KPP09, MK11], the inherent difficulties in scaling such methods to large-scale realistic biological models can provide challenging research opportunities.

References

[DH01] Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)

[FH07] Fisher, J., Henzinger, T.A.: Executable Cell Biology. Nature Biotechnology 25(11), 1239–1249 (2007)

[HM01] Harel, D., Marelly, R.: Specifying and executing behavioral requirements: The play-in/play-out approach. In: Software and System Modeling, SoSyM (2003)

[KKM08] Kam, N., Kugler, H., Marelly, R., Appleby, L., Fisher, J., Pnueli, A., Harel, D., Stern, M.J., Hubbard, E.J.A.: A scenario-based approach to modeling development: A prototype model of C. elegans vulval fate specification. Developmental Biology 323(1), 1–5 (2008)

[KPP09] Kugler, H., Plock, C., Pnueli, A.: Controller Synthesis from LSC Requirements. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer, Heidelberg (2009)

[KS09] Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 77–91. Springer, Heidelberg (2009)

[MK11] Milicevic, A., Kugler, H.: Model Checking Using SMT and Theory of Lists. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 282–297. Springer, Heidelberg (2011)

[Plo08] Plock, C.: Synthesizing Executable Programs from Requirements. PhD thesis, New York Univ. (2008)

[SBT11] Microsoft Research Cambridge, Synthesizing Biological Theories (2011), `http://research.microsoft.com/SBT/endthebibliography`

PRISM 4.0: Verification of Probabilistic Real-Time Systems

Marta Kwiatkowska[1], Gethin Norman[2], and David Parker[1]

[1] Department of Computer Science,
University of Oxford, Oxford, OX1 3QD, UK
[2] School of Computing Science,
University of Glasgow, Glasgow, G12 8RZ, UK

Abstract. This paper describes a major new release of the PRISM probabilistic model checker, adding, in particular, quantitative verification of (priced) probabilistic timed automata. These model systems exhibiting *probabilistic*, *nondeterministic* and *real-time* characteristics. In many application domains, all three aspects are essential; this includes, for example, embedded controllers in automotive or avionic systems, wireless communication protocols such as Bluetooth or Zigbee, and randomised security protocols. PRISM, which is open-source, also contains several new components that are of independent use. These include: an extensible toolkit for building, verifying and refining abstractions of probabilistic models; an explicit-state probabilistic model checking library; a discrete-event simulation engine for statistical model checking; support for generation of optimal adversaries/strategies; and a benchmark suite.

1 Introduction

This paper describes a major new release, version 4.0, of the PRISM probabilistic model checker. This adds, in particular, formal modelling and analysis capabilities for systems with *probabilistic*, *nondeterministic* and *real-time* characteristics, through support for verification of (priced) *probabilistic timed automata*.

PRISM already provides model checking for several types of probabilistic models: discrete- and continuous-time Markov chains and Markov decision processes, as well as a modelling language in which to express them. The tool has been widely taken up (downloaded more than 20,000 times) and used for *quantitative verification* in a broad spectrum of application domains, from wireless communication protocols to quantum cryptography to systems biology. In many cases, flawed or anomalous behaviour has been identified, from worst-case performance conditions for Bluetooth [2] to behavioural predictions (later validated experimentally) for biological signalling pathways [4].

Increasingly, though, new application domains are dictating the need for quantitative verification techniques and tools for richer classes of models. Embedded systems, such as in multimedia devices or avionic systems, exhibit stochastic behaviour and also operate under constraints on timing and other resources. PRISM 4.0 supports (priced) probabilistic timed automata, a natural model for

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 585–591, 2011.

Time	Nondeterminism	Probabilistic Models
discrete	no	discrete-time Markov chains (DTMCs)
	yes	Markov decision processes (MDPs) probabilistic automata (PAs)
continuous	no	continuous-time Markov chains (CTMCs)
	yes	**probabilistic timed automata (PTAs)** **priced probabilistic timed automata (PPTAs)**

Fig. 1. The types of probabilistic models currently supported by PRISM, classified by modelling of time and the presence of nondeterminism; boldface denotes new additions

such systems. It also incorporates several new components, including engines for *quantitative abstraction-refinement* [9] and *statistical model checking* [14,5]. The components are designed to be extensible and re-usable. As well as improving scalability for existing model types and adding support for (infinite-state) PTAs, they are targeted at facilitating verification of more expressive classes of models such as probabilistic and stochastic hybrid systems.

1.1 Functionality Overview

We begin with a brief overview of the current functionality of the PRISM tool. Items in boldface denote new or improved features in version 4.0, which are described in more detail in the remainder of the paper.

- modelling and construction of many types of probabilistic models (see Fig. 1 for a summary), now including **probabilistic timed automata**; all can be augmented with costs or rewards, in the case of PTAs yielding the model of **priced probabilistic timed automata**;
- model checking of a wide range of quantitative properties, expressed in a language that subsumes the temporal logics PCTL, CSL, LTL and PCTL*, as well as extensions for quantitative specifications and costs/rewards;
- multiple model checking engines, both symbolic (BDD-based) and **explicit-state**; and a variety of probabilistic verification techniques, such as symmetry reduction and **quantitative abstraction refinement**;
- a **discrete-event simulator**, with support for **statistical model checking** methods, including confidence-level approximation and acceptance sampling;
- model import options, e.g. from SBML (systems biology markup language);
- **optimal adversary/strategy** generation for nondeterministic models;
- a GUI, with model editor, simulator and graphing, or command-line tool;
- a **benchmark suite** of probabilistic models and associated properties.

2 Probabilistic Timed Automata (PTAs)

Probabilistic timed automata (PTAs) [6,12] are finite-state automata enriched with real-valued clocks, in the style of timed automata, and with discrete probabilistic choice, in the style of Markov decision processes (MDPs).

Clocks are real-valued variables, whose values increase simultaneously over time. Predicates over clock variables called *guards* and *invariants* are assigned to transitions and states, respectively, imposing restrictions on when transitions can occur and how long can be spent in a state. For ease of modelling, we can also add finite-ranging *data variables* to a PTA. Transitions between states can reset clocks (to integer values) and update data variables. This is done *probabilistically*: the target state, clock resets and variable updates are specified by a discrete probability distribution. The choice between multiple transitions, as well as the elapse of time (subject to invariant satisfaction) are both *nondeterministic*.

Fig. 2 (left) shows a simple example of a PTA, modelling repeated attempts to transmit a message over an unreliable channel. The system tries to send for between 1 and 2 time-units. With probability 0.1, this fails, in which case a delay of between 3 and 5 time-units elapses before retrying (up to N times).

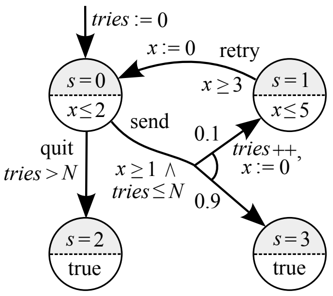
PTAs can be augmented with information about costs incurred or, equivalently, rewards gained (PRISM uses the latter terminology). This model, often known as *priced* PTAs, allows reasoning about a wide range of additional properties, such as energy consumption or resource usage. PRISM supports *linearly* priced PTAs, where costs/rewards are accumulated at a rate proportional to the elapse of time, with the rate depending on the current state (and data variables).

Finally, we mention that PTAs also support *parallel composition* (as for timed automata and MDPs), in which multiple PTAs operate concurrently, synchronising on transitions with matching labels. For precise details, see [11].

Modelling PTAs. PRISM uses a uniform modelling language for all the probabilistic models that it supports, including PTAs. This is a textual language, based on guarded command notation. To support PTAs, PRISM 4.0 adds a new `clock` datatype. Clock variables can appear (as convex expressions) in guards, on the left-hand side of a command, and can be reset, like any other variable, with an update on the right-hand side. A new `invariant` keyword is introduced to allow expression of invariants. Fig. 2 (right) gives a PRISM modelling language description for the example PTA described above. It also shows an example of a PRISM reward structure, labelled "energy", to create a priced PTA which assigns a reward rate of 2.5 when s=0, i.e. during message transmission.

PTA verification techniques. PRISM analyses two main classes of properties for PTAs: (i) the minimum/maximum probability of reaching a target, possibly within a time bound (e.g. "the maximum probability of an airbag failing to deploy within 0.02 seconds"); and (ii) the minimum/maximum expected reward accumulated until a target is reached (e.g. "the maximum expected time for the protocol to terminate"). Two verification methods are implemented:

- *Quantitative abstraction refinement* [9] constructs and analyses a series of probabilistic abstractions, automatically refining at each step to produce more precise results (see also Section 3). By using *stochastic two-player game* abstractions, defined in terms of *zones*, this yields an effective technique for exact verification of probabilistic reachability properties of PTAs [10]. Since experimental results in [10] show that this method generally outperforms all others currently available, this is the default engine in PRISM.



```
pta
const int N;
module transmitter
    // Local variables
    s : [0..3] init 0;
    tries : [0..N+1] init 0;
    x : clock;
    invariant
        (s=0 ⇒ x≤2) & (s=1 ⇒ x≤5)
    endinvariant
    // Guarded commands
    [send] s=0 & x≥1 & tries≤N  →  0.9 : (s′=3)
            + 0.1 : (s′=1)&(tries′=tries+1)&(x′=0);
    [retry] s=1 & x≥3  →  (s′=0)&(x′=0);
    [quit] s=0 & tries>N  →  (s′=2);
endmodule
rewards "energy" (s=0) : 2.5; endrewards
```

Fig. 2. Left: A PTA, with clock x and integer variable *tries*, modelling attempted message transmission over an unreliable channel. Right: corresponding PRISM code.

– *Digital clocks* [11] performs an automatic model translation to an equivalent finite-state, discrete-time model (with integer-valued clocks)[1] and then uses PRISM's existing MDP model checking techniques. For (non-probabilistic) timed automata, such methods are usually much less efficient than on-the-fly zone-based reachability (as, e.g., in UPPAAL [13]). For PTAs, which lack such on-the-fly methods, the digital clocks approach remains competitive, especially in combination with PRISM's symbolic (BDD-based) implementation. This method also has the widest applicability, supporting both probabilistic reachability properties and expected cumulative rewards for (linearly) priced probabilistic timed automata [11].

Related tools. UPPAAL [13] is the leading verification tool for timed automata. A recent extension, UPPAAL-PRO [15], adds support for PTAs, but currently only analyses maximum probabilistic reachability properties. Fortuna [1] supports the same class but also allows the inclusion of reward-bounds when (linearly) priced PTAs are considered. Another tool aimed at verification of probabilistic real-time systems is `mcpta` [3], which translates a subset of the modelling language Modest into PRISM using *digital clocks* [11]. Finally, MRMC [7], a probabilistic model checker for Markov chains and Markov reward models, has recently also added support for continuous-time MDPs, another model combining nondeterministic, probabilistic and real-time features. For a more detailed list of probabilistic verification tools, including other tools for abstraction refinement (such as RAPTURE, PASS) and statistical model checking, see [18].

3 Other New PRISM Components and Features

Quantitative abstraction refinement toolkit. As described above, PRISM's default PTA verification technique uses *quantitative abstraction refinement* [9]. This can be seen as a quantitative analogue of classical counterexample-guided

[1] Under slight restrictions, e.g. *strict* clock comparisons (such as $x<1$) are not allowed.

abstraction refinement. It provides a fully automatic approach to verification of large or infinite-state probabilistic systems, by iteratively building, analysing and refining increasingly precise probabilistic abstractions. In addition to PTAs [10], the same approach has been applied to verification of probabilistic software (using predicate abstraction and SAT) [8] and to finite-state MDPs [9]. While these implementations all build abstractions of MDPs as stochastic two-player games, the same approach can be used to, for example, build abstractions of Markov chains as Markov decision processes. Quantitative abstraction refinement is implemented in PRISM in the form of an extensible toolkit, with support for multiple model types, a choice of refinement strategies and various configurable optimisations.

Explicit-state probabilistic model checking library. PRISM already features several model checking engines (called "MTBDD", "sparse", and "hybrid"), all either fully or partially *symbolic* (i.e. BDD-based). The tool now incorporates a new, entirely *explicit-state* probabilistic model checking library, implemented in Java and based on sparse matrix data structures. It supports stochastic two-player games, Markov decision processes and discrete- and continuous-time Markov chains. The code is designed to serve as a general purpose library, either for inclusion in other techniques or for prototyping new model checking algorithms. For example, the library is used in the abstraction-refinement toolkit, in which probabilistic models need to be constructed and modified on-the-fly, a task not well-suited to symbolic implementations.

Simulation engine and statistical model checking. Version 4.0 of PRISM incorporates a newly rewritten version of its *discrete-event simulation engine.* This provides efficient random generation of paths through PRISM models, both for the purposes of debugging models and to support so-called *statistical* (or *approximate*) model checking techniques [14,5]. PRISM now offers two types of such analysis. For *quantitative* properties (e.g. `P=?[·]` in PRISM notation), it either generates a confidence interval (based on a given confidence level) or a probabilistic guarantee of correctness, using the Chernoff-Hoeffding bound [5]. For *bounded* properties (e.g. `P<0.1[·]`), it uses *acceptance sampling* [14], implementing Wald's sequential probability ratio test (SPRT). Statistical model checking offers significantly improved scalability, in comparison to conventional probabilistic model checking techniques, and applies to a broader class of models.

Optimal adversary (strategy) generation. PRISM's MDP verification implementation now includes the ability to generate *optimal adversaries* (also known as strategies). This means that, when PRISM computes the minimum or maximum value for a probabilistic reachability (or expected reward) property, it can also generate an adversary (resolution of nondeterminism in the model) that produces it. This can be used to debug or analyse the results of model checking, for example in order to generate probabilistic counterexamples, or to produce an optimal solution for a scheduling problem. Furthermore, by using the digital clocks engine, optimal adversaries can also be generated for PTAs.

The PRISM benchmark suite. There are a large number of existing PRISM case studies, distributed with the tool, included in publications and on the tool website [16]. These are widely used, for example to evaluate new model checking techniques, or to compare model checking implementations and tools. Unfortunately, there are often several different variants of each model and it is not always easy to locate a particular class of models or properties. The *PRISM benchmark suite* [17] aims to provide a comprehensive source of freely-available benchmarks for probabilistic model checking. It includes a large selection of probabilistic models, of varying types and sizes, and corresponding properties for model checking, grouped by type. External contributions are also welcomed.

Technical details and availability. PRISM is free and open source (GPL). It is coded in a mix of Java and C++, and runs on all major operating systems. It is available for download from `http://www.prismmodelchecker.org/`.

Acknowledgments. The authors are part supported by ERC Advanced Grant VERIWARE, EU-FP7 project CONNECT, DARPA project PRISMATIC and EPSRC grant EP/D07956X. For a full list of PRISM contributors, see [16].

References

1. Berendsen, J., Jansen, D., Vaandrager, F.: Fortuna: Model checking priced probabilistic timed automata. In: Proc. QEST 2010, pp. 273–281 (2010)
2. Duflot, M., Kwiatkowska, M., Norman, G., Parker, D.: A formal analysis of Bluetooth device discovery. STTT 8(6), 621–632 (2006)
3. Hartmanns, A., Hermanns, H.: A Modest approach to checking probabilistic timed automata. In: Proc. QEST 2009, pp. 187–196 (2009)
4. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. TCS 319(3), 239–257 (2008)
5. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
6. Jensen, H.: Model checking probabilistic real time systems. In: Proc. 7th Nordic Workshop on Programming Theory, pp. 247–261 (1996)
7. Katoen, J.P., Hahn, E.M., Hermanns, H., Jansen, D., Zapreev, I.: The ins and outs of the probabilistic model checker MRMC. In: Proc. QEST 2009, pp. 167–176 (2009)
8. Kattenbelt, M., Kwiatkowska, M., 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)
9. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. In: FMSD, vol. 36(3) (2010)
10. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009)
11. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. FMSD 29, 33–78 (2006)

12. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. TCS 282, 101–150 (2002)
13. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
14. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)
15. `http://www.cs.aau.dk/~arild/uppaal-probabilistic/`
16. `http://www.prismmodelchecker.org/`
17. `http://www.prismmodelchecker.org/benchmarks/`
18. `http://www.prismmodelchecker.org/other-tools.php`

Program Analysis for Overlaid Data Structures[⋆]

Oukseh Lee[1,2], Hongseok Yang[3], and Rasmus Petersen[2]

[1] Hanyang University, South Korea
[2] Queen Mary University of London, United Kingdom
[3] University of Oxford, United Kingdom

Abstract. We call a data structure overlaid, if a node in the structure includes links for multiple data structures and these links are intended to be used at the same time. In this paper, we present a static program analysis for overlaid data structures. Our analysis implements two main ideas. The first is to run multiple sub-analyses that track information about non-overlaid data structures, such as lists. Each sub-analysis infers shape properties of only one component of an overlaid data structure, but the results of these sub-analyses are later combined to derive the desired safety properties about the whole overlaid data structure. The second idea is to control the communication among the sub-analyses using ghost states and ghost instructions. The purpose of this control is to achieve a high level of efficiency by allowing only necessary information to be transferred among sub-analyses and at as few program points as possible. Our analysis has been successfully applied to prove the memory safety of the Linux deadline IO scheduler and AFS server.

1 Introduction

Recent advances in verification research have resulted in successful industrial-strength software verifiers, such as Microsoft SDV and Astrée. These tools do verification-by-static-analysis, where the tools work fully automatically without asking the user to insert loop invariants or procedure specifications. But these tools cannot approach many parts of operating systems, because of their inaccurate or unsound treatment of the heap. In fact, the heap is one of the outstanding problems holding back verification-by-static-analysis (or software model checking). Although there have been works approaching verification of the heap in real-world systems programs [3,14], fundamental problems remain, and one of the most fundamental is the presence of nontrivial, but not unrestricted, sharing. The not unrestricted aspect gives some hope that techniques might be found that do not immediately run into an efficiency brick wall.

In this paper, we consider the automatic verification of overlaid data structures, which show such nontrivial but not unrestricted sharing. We call a data

[⋆] We want to thank Gilad Arnold, Patrick Cousot, Peter Hawkins, Peter O'Hearn, Martin Rinard, Noam Rinetzky, Xavier Rival, and John Wickerson for helpful comments. This work was supported by EPSRC, and Lee by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2011-0000968).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 592–608, 2011.

structure overlaid, if a node in the structure includes links for multiple data structures and these links are intended to be used at the same time. These overlaid data structures are frequently used in systems code in order to impose multiple types of indexing structures over the same set of nodes. For instance, the deadline IO scheduler of Linux has a queue whose nodes have links for a doubly-linked list as well as links for a red-black tree. The linked list is used to record the order in which nodes are inserted in the queue, and the red-black tree provides an efficient indexing structure on the sector fields of the nodes.

Our goal is to build an efficient yet precise program analysis for overlaid data structures, capable of verifying the memory safety or shape properties of real-world programs. The objective here is not to verify toy problems of overlaid data structures, but to verify real-world examples. In fact, we created an analyser in 2008 that could prove the memory safety of toy examples, but this analyser could not scale to verify real code like the deadline IO scheduler for several fundamental reasons (see Section 7). Also, there have been other papers that take on toy programs using overlaid data structures or graphs, but they are all too imprecise or too expensive to verify serious programs [9,7,4,13].

In this paper, we present a new program analysis for overlaid data structures, which can verify the memory safety and shape properties of medium sized real-world examples from Linux. Our analysis implements two main ideas:

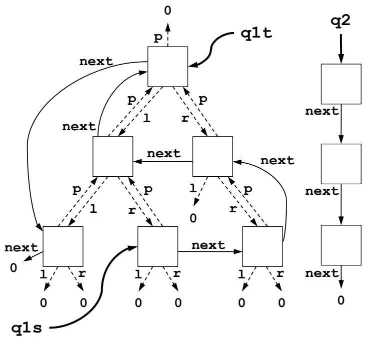
1. Run multiple sub-analyses that track information about standard data structures, such as lists: Each sub-analysis infers shape properties of only one component of an overlaid data structure, but the results of these sub-analyses are later combined to derive the desired safety properties about the whole overlaid data structure. This is reminiscent of cartesian abstraction [2].
2. Control the communication among the sub-analyses using ghost states and ghost instructions: We found that to prove the memory safety of programs using overlaid data structures, the sub-analyses need to transfer information among themselves (using a form of reduction [5]); the memory safety of the programs often relies on the fact that components of an overlaid data structure use the same set of nodes. Our analysis controls this information transfer in order to achieve a high level of efficiency. It aims at allowing only necessary information to be transferred among sub-analyses and only at as few program points as possible. To achieve the aim, the analysis uses ghost states, special instructions for modifying ghost states, and algorithms that insert those instructions before or during the main phase of the analysis.

Related work. We discuss three further related works here. The first is the synthesis approach by Hawkins *et al.* [8], where a programmer specifies an overlaid data structure using a high level specification in the style of a relational database. This approach focuses on generating new correct programs using overlaid data structures, and it is complementary to the results of this paper. The second is the general meet algorithm [1] for finding intersections of heap abstractions in TVLA. The algorithm is related to our operator for transferring information among sub-analyses, but it aims at computing the exact meet, not an efficient

```
struct node { struct node *next;
              struct node *p,*l,*r;
              int key; };
struct node *q1s, *q1t, *q2;

void move_request() {
 struct node *c;
 c = list_remove_first(&q1s);
 if(c==0) return; //trans(list->tree)(c)
 tree_remove(c);  //move(c,gamma)
 list_add_first(&q2,c);
 c = 0;           //moveRgn(gamma,beta)
}
```

(a) C code



(b) a snapshot of data structure

Fig. 1. Baby IO scheduler

over-approximation of the meet as in this paper. The last is the Hob system by Kuncak *et al.* [10], where one can apply different analysis plug-ins for different data structures, combine the analysis results, and verify that values stored in these data structures are properly related. This system regards an overlaid data structure as a single data structure, and requires a plug-in for its analysis. This requirement can be met by the analysis in this paper.

2 Informal Description

We start with an informal description of our analysis using the baby IO scheduler in Figure 1(a), which is modelled on the Linux deadline IO scheduler.

Our baby IO scheduler schedules IO requests using two disjoint queues. When a request arrives, it is stored in the first queue. Later the request is selected according to a scheduling policy, processed, and moved to the second queue. In order to help the performance of the scheduling, the first queue uses an overlaid data structure with list and tree components. The list component is a singly-linked list starting from `q1s`, and it keeps requests in FIFO order. The tree component is a binary search tree with parent pointers. The address of the root of the tree is stored in `q1t`, and the tree provides an efficient search mechanism on the `key` field of requests. The second queue is, on the other hand, a simple linked list from `q2`, storing processed requests in FIFO order. A concrete example of both queues is shown in Figure 1(b).

The `move_request` function in Figure 1 shows a typical example of exploiting both components of an overlaid data structure. This function removes the first node of the list component `q1s` of the overlaid data structure. Then, it switches to the tree component, removes the node from the tree, and adds it to `q2`. One important aspect is that the removal from the tree exploits the correlation between components of the overlaid data structure—both the list `q1s` and the tree `q1t` use the same set of nodes. Although the node `c` is found using the list part, the correlation ensures that the node is in the tree as well. Hence, the removal from the tree can be performed safely without traversing the tree.

The main challenge for automatically proving the memory safety or shape properties of the baby IO scheduler is to find a good representation of the overlaid data structure $(\texttt{q1s}, \texttt{q1t})$, which enables the design of an efficient yet precise program analysis. Although nodes in this data structure are highly shared, this sharing has a pattern, i.e., it is generated by the overlay of a list and a tree. Furthermore, our baby scheduler, like the original Linux IO scheduler, relies only on the correlation between the list and tree components found in the `move_request` function—both components are formed using exactly the same set of nodes. We would like the representation to exploit fully the pattern of $(\texttt{q1s}, \texttt{q1t})$, and to express only this relatively weak correlation of its two components.

Our solution is to use the conjunction of two types of assertions $\varphi \wedge \psi$, where φ describes the heap only in terms of list fields and ψ does the same but using only the fields from the tree (including `key`). To express that the components of an overlaid data structure use the same set of nodes, φ and ψ use what we call region variables α, β, γ, which denote sets of memory addresses. Concretely, our analysis infers that the data structures of our IO scheduler normally satisfy the following assertion:

$$(\mathsf{ls}(\texttt{q1s})_\alpha * \mathsf{ls}(\texttt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\texttt{q1t})_\alpha * \mathsf{true}_\beta). \tag{1}$$

The predicate $\mathsf{ls}(x)$ means a singly-linked list starting from the address x, and $\mathsf{tr}(y)$ a tree rooted at y. The separating conjunction $P * Q$ means that the heap consists of two disjoint sub-heaps described by P and Q.

The first conjunct in (1) says that the heap contains two disjoint singly-linked lists `q1s` and `q2`. Using the subscripts $-_\alpha$ and $-_\beta$, it also states that the addresses of the nodes in the list `q1s` form the set α, and those of the nodes in the list `q2` the set β. The second conjunct, on the other hand, talks about tree-related properties of the heap. According to this conjunct, the heap contains a tree with root address `q1t`. Furthermore, the addresses of nodes in the tree form the set α, while the addresses of all the other nodes make the set β. Note that each conjunct has its own characterisations of α and β. To be consistent, both characterisations of α should mean the same, which implies that the list and the tree use the same set of nodes. This is exactly the type of correlation that we want to express for the overlaid data structure $(\texttt{q1s}, \texttt{q1t})$.

This representation enables an interesting strategy for analyzing a client program of an overlaid data structure. The strategy is to run multiple sub-analyses that are designed for tracking information about standard non-overlaid data structures, such as lists and trees. Each of these sub-analyses infers shape properties of only one component of the overlaid data structure, hence handling only one conjunct in our representation. The desired memory properties of the program are then proved by combining the results of the sub-analyses.

Our analysis implements a real-world adjustment of this strategy. Note that in our example, the sub-analyses cannot be completely independent. They need to communicate during (not after) analysis, because of the above-mentioned correlation among components of an overlaid data structure; in the function `move_request`, the removal of `c` from the tree cannot be inferred to be safe without looking at the list. To address this concern while keeping the communication cost of the sub-analyses low, our analysis uses ghost instructions for region variables. It runs the sub-analyses independently most of the time, except at a few program points where the memory safety proof demands communication among the sub-analyses. At these program points, the analysis inserts ghost instructions that initiate communication among sub-analyses. Furthermore, even in those communication points, the analysis tries to keep the communicated information as simple as possible, using region variables.

We illustrate the analysis using the `move_request` function in Figure 1. In this case, our analysis runs the list and tree sub-analyses, which update the conjunct for list and that for tree, respectively. The first step of our analysis is a pre-analysis that inserts ghost instructions for changing the values of region variables or for transferring information between the list and tree sub-analyses. For our `move_request` example, the pre-analysis inserts $\mathsf{trans}_{\mathsf{list}\to\mathsf{tree}}(\mathtt{c})$ and $\mathtt{move}(\mathtt{c}, \gamma)$ before and after `tree_remove`, as shown in Figure 1. The first instruction $\mathsf{trans}_{\mathsf{list}\to\mathsf{tree}}(\mathtt{c})$ tells the tree sub-analysis to get information about cell `c` from the list sub-analysis, and it is a so-called reduction operator in program analysis [5]. The instruction is inserted here, because the pre-analysis conjectures that information about cell `c` at this program point will be necessary for verification. The second instruction $\mathtt{move}(\mathtt{c}, \gamma)$ tells the analysis to manage the values of region variables by moving the address `c` from its current region to the region γ. We defer the details of the pre-analysis to Section 5.

The second step is to run the `move_request` function symbolically, starting from the assertion in (1), while abstracting away unnecessary information from time to time. This symbolic abstract execution is done by invoking the corresponding routines of the sub-analyses. The command `list_remove_first(&q1s)` is run first in this manner, and results in the assertion

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathsf{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathsf{q2})_\beta) \;\wedge\; (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathsf{true}_\beta) \tag{2}$$

for the true branch of the following conditional statement. Compared to the original in (1), the assertion has additionally $\mathsf{c} \mapsto \{\}_\alpha$ in the first conjunct, and this additional predicate describes the cell `c` removed from the list `q1s`. In this abstract execution, our analysis runs only the list sub-analysis not the tree one, because it detects that `list_remove_first(&q1s)` is equivalent to skip as far as the tree sub-analysis is concerned.

Note that only the first conjunct of (2) knows the allocatedness of cell `c` in α. The next instruction $\mathsf{trans}_{\mathsf{tree}\to\mathsf{list}}(\mathtt{c})$ makes the analysis transfer the information about cell `c` from the first to the second conjunct, which gives the assertion:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathsf{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathsf{q2})_\beta) \;\wedge\; (\varphi(\mathtt{q1t}, \mathsf{c}, \alpha) * \mathsf{true}_\beta). \tag{3}$$

Here $\varphi(\mathtt{q1t}, \mathsf{c}, \alpha)$ is an assertion with free variables $\mathtt{q1t}, \mathsf{c}, \alpha$, and it describes a tree with root `q1t` and a normal node `c` such that all nodes of the tree form the set α.[1] This refinement of assertions is how our analysis enables the communication between sub-analyses, this time from the list to the tree sub-analysis.

[1] Concretely, $\varphi(\mathtt{q1t}, \mathsf{c}, \alpha)$ is $\exists uvwxy.\ \mathsf{tseg}(\mathtt{q1t}, 0, \mathsf{c}, u)_\alpha * \mathsf{c} \mapsto \{\mathsf{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}x\}_\alpha * \mathsf{tseg}(v, \mathsf{c}, 0, w)_\alpha * \mathsf{tseg}(x, \mathsf{c}, 0, y)_\alpha$ where tseg is a tree segment predicate explained in Section 4.

The transferred information allows the analysis to prove the memory safety of the following instruction `tree_remove(c)`, which is handled by the tree sub-analysis only, and to over-approximate the instruction's output states by the assertion:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{true}_\beta). \tag{4}$$

This assertion has $\mathtt{c} \mapsto \{\}_\alpha$ in both conjuncts, hence confirming that the node `c` is indeed removed from both the list `q1s` and the tree `q1t`.

The next instruction is the ghost instruction $\mathtt{move}(\mathtt{c}, \gamma)$ inserted by the pre-analysis. This instruction simply changes the subscript of $\mathtt{c} \mapsto \{\}$ from α to γ:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\gamma * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathtt{c} \mapsto \{\}_\gamma * \mathsf{true}_\beta). \tag{5}$$

Semantically, this change means that the allocated cell `c` is moved from the set α to the set γ, which only contains `c`. The decision for singling out `c` and putting it in a separate set γ is made because the pre-analysis detected a possibility of moving cell `c` between two different data structures. This possibility is indeed realized in the program, because the following two instructions `list_add_first(&q2,c)` and `c = 0` move the cell `c` to the second queue `q2`. The analysis tracks the move of the cell, using its list sub-analysis, and transforms (5) to the assertion:

$$(\exists a.\ \mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{q2} \mapsto \{\mathtt{next}{:}a\}_\gamma * \mathsf{ls}(a)_\beta) \ \wedge \ (\exists b.\ \mathsf{tr}(\mathtt{q1t})_\alpha * b \mapsto \{\}_\gamma * \mathsf{true}_\beta). \tag{6}$$

The variable a has the old value of `q2`, and b the old value of `c`.

Note that the sub-formula $\mathtt{q2} \mapsto \{\mathtt{next}{:}a\}_\gamma * \mathsf{ls}(a)_\beta$ in (6) describes a list starting from `q2` of length at least one (because of cell `q2`). The list sub-analysis decides that this length information is not necessary for verifying the memory safety of the program, and it plans to drop the information by replacing the sub-formula by $\mathsf{ls}(\mathtt{q2})$. To do this, the analysis inserts the instruction $\mathtt{moveRgn}(\gamma, \beta)$ for moving all cells in γ to β, and analyzes the inserted instruction:

$$(\exists a.\ \mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{q2} \mapsto \{\mathtt{next}{:}a\}_\beta * \mathsf{ls}(a)_\beta) \ \wedge \ (\exists b.\ \mathsf{tr}(\mathtt{q1t})_\alpha * b \mapsto \{\}_\beta * \mathsf{true}_\beta). \tag{7}$$

The reason for inserting the instruction $\mathtt{moveRgn}(\gamma, \beta)$ is to make sure that the changes in the values of region variables happen consistently for both conjuncts (i.e., both sub-analyses), although the changes are initiated by the need for abstracting a part of the first conjunct. Now, both the head `q2` and the tail a are in the same set β, so the abstraction applies and gives the final result:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathsf{true}_\beta). \tag{8}$$

Here $b \mapsto \{\}_\beta * \mathsf{true}_\beta$ is also abstracted to true_β by the tree sub-analysis. This amounts to forgetting the fact that β contains at least one cell.

Our formalization of the ideas described so far will form the rest of the paper.

3 Formal Setting for Region Variables

Instrumented storage model. We use a storage model where a state consists of three components. The first two are the usual ones, namely, the stack for

$$
\begin{array}{lll}
s,h,\eta \models \varphi_\alpha & \iff & s,h,\eta \models \varphi \text{ and } \mathsf{dom}(h) = \eta(\alpha) \\
s,h,\eta \models e \in \alpha & \iff & [\![e]\!]s \in \eta(\alpha) \\
s,h,\eta \models e \mapsto \{\vec{\mathtt{f}} : \vec{e}\} & \iff & \mathsf{dom}(h) = \{[\![e]\!]s\} \text{ and } h([\![e]\!]s)\mathtt{f}_i = [\![e_i]\!] \text{ for all } 1 \leq i \leq |\vec{\mathtt{f}}| \\
s,h,\eta \models p(\vec{e}) & \iff & h \in [\![p]\!]([\![\vec{e}]\!]s) \\
s,h,\eta \models \mathsf{emp} & \iff & \mathsf{dom}(h) = \emptyset \text{ and } \eta(\alpha) = \emptyset \text{ for all } \alpha \\
s,h,\eta \models P * Q & \iff & \exists h_1,h_2,\eta_1,\eta_2.\ (h_1,\eta_1) \bullet (h_2,\eta_2) = (h,\eta) \\
 & & \qquad\qquad\qquad \text{and } s,h_1,\eta_1 \models P_1 \text{ and } s,h_2,\eta_2 \models P_2
\end{array}
$$

Fig. 2. Semantics of sample assertions. We assume a function $[\![e]\!]$ from Stacks to Vals that defines the meaning of expression e, and a mapping $[\![p]\!]$ from value tuples to heaps that specifies the semantics of primitive predicate p.

program variables and the heap for dynamically allocated cells. The third one is, however, unusual, and it defines the values of region variables.

To give a formal definition of our model, we need four disjoint countable sets: a set Addrs of addresses; a set Vars of normal variables x, y, z; sets Fields and Regions that respectively contain field names $\mathtt{f}, \mathtt{g}$ of heap cells and region variables α, β, γ. We assume that a fixed constant *null* is not in Addrs. The storage model is defined by the following equations:

$$
\begin{gathered}
\mathsf{Vals} = \mathsf{Addrs} \cup \{\mathit{null}\} \quad \mathsf{Stacks} = \mathsf{Vars} \to \mathsf{Vals} \quad \mathsf{Heaps} = \mathsf{Addrs} \rightharpoonup_{\mathit{fin}} (\mathsf{Fields} \rightharpoonup \mathsf{Vals}) \\
\mathsf{Partitions} = \mathsf{Regions} \to \mathcal{P}(\mathsf{Addrs}) \quad \mathsf{States} = \mathsf{Stacks} \times \mathsf{Heaps} \times \mathsf{Partitions}
\end{gathered}
$$

Note that a state has three components $(s, h, \eta) \in \mathsf{States}$, where s defines the values of stack variables, h specifies the contents of allocated cells, and η maps region variables to address sets. We call a pair (h, η) *well-formed* if the mapping η defines a partition of allocated cells, that is, the following holds:

$$
(\mathsf{dom}(h) = \cup_{\alpha \in \mathsf{Regions}} \eta(\alpha)) \ \wedge \ (\forall \alpha, \beta \in \mathsf{Regions}.\ \alpha \not\equiv \beta \implies \eta(\alpha) \cap \eta(\beta) = \emptyset).
$$

A state (s, h, η) is *well-formed* when (h, η) is well-formed. In the rest of this paper, we consider only well-formed states and pairs of heaps and region-maps.

Note that in a well-formed state, every allocated address belongs to a unique region. As a result, a fact about an allocated address l can be approximated by the region variable α containing l. For instance, when the variable x contains the address l of an allocated cell (i.e., $s(x) = l$), we can approximate this information by $s(x) \in \eta(\alpha)$. Our analysis uses this approximation to form the lightweight information to be passed among the sub-analyses.

Assertions. Assertions φ describe properties of states, and are defined as follows:

$$
\begin{array}{lll}
e ::= x \mid \mathtt{null} \qquad & \varphi ::= & \varphi_\alpha \mid e = e \mid e \in \alpha \mid e \mapsto \{\vec{\mathtt{f}} : \vec{e}\} \mid p(\vec{e}) \\
 & \mid & \mathsf{emp} \mid \varphi * \varphi \mid \mathsf{true} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\, \varphi
\end{array}
$$

This is a variant of the assertion language from separation logic [12]. The first φ_α says that the heap satisfies φ and all the allocated addresses in the heap form the set α. This is the most unusual case of our assertion language, and it enables one

$$\begin{aligned}
[\![\texttt{assume}(b)]\!](s,h,\eta) &= \textbf{if } ([\![b]\!]s = \mathit{true}) \textbf{ then } \{(s,h,\eta)\} \textbf{ else } \emptyset \\
[\![x := \texttt{new}_{\alpha,F}()]\!](s,h,\eta) &= \{(s[x \mapsto l], h[l \mapsto v], \eta[\alpha \mapsto \eta(\alpha) \cup \{l\}]) \mid \\
&\qquad l \in \mathsf{Addrs} \setminus \mathsf{dom}(h) \text{ and } v \text{ is a function from } F \text{ to } \mathsf{Vals}\} \\
[\![\texttt{move}(e,\alpha)]\!](s,h,\eta) &= \textbf{if } \neg(\exists\beta.\, [\![e]\!]s \in \eta(\beta)) \textbf{ then } \mathit{err} \\
&\quad \textbf{else } \{(s,h,\eta[\beta \mapsto \eta(\beta) \setminus \{[\![e]\!]s\}, \alpha \mapsto \eta(\alpha) \cup \{[\![e]\!]s\}])\} \\
[\![\texttt{moveRgn}(\alpha,\beta)]\!](s,h,\eta) &= \{(s,h,\eta[\alpha \mapsto \emptyset, \beta \mapsto \eta(\alpha) \cup \eta(\beta)])\}
\end{aligned}$$

Fig. 3. Semantics of sample primitive instructions. We assume a function $[\![b]\!]$ from Stacks to $\{\mathit{true}, \mathit{false}\}$ that defines the meaning of boolean b.

to talk about the values of region variables, the new part of our storage model. The next two are the usual equalities on expressions and the membership of an expression to a region variable. The assertion $x \mapsto \{\vec{\mathtt{f}} \colon \vec{e}\}$ means a heap containing only one cell x that stores $\vec{e}$ in fields $\vec{\mathtt{f}}$. This definition does not require that $\vec{\mathtt{f}}$ be the only fields in cell x. Hence, the cell x can have fields other than $\vec{\mathtt{f}}$. The following case $p(\vec{e})$ is the application of a primitive predicate p, such as the tree or singly-linked list predicates, and it is mainly used to describe a recursive data structure. Our assertion language includes separating connectives—emp for the empty heap and the region variables all having the empty set, and $\varphi * \psi$ for the splitting of both the heap and the region-variable map such that one pair satisfies φ and the other ψ. The remaining cases are the standard connectives from classical logic, and they have the usual meanings. We point out that other standard connectives from classical logic can be defined in a standard way.

The formal semantics is given by a satisfaction relation $\models$ between well-formed states and assertions $(s,h,\eta) \models \varphi$, and sample clauses of the semantics appear in Figure 2. The clause for $\varphi * \psi$ uses the following partial combining operator $(h_1,\eta_1) \bullet (h_2,\eta_2)$ on well-formed pairs of heaps and region-maps:

$$(h,\eta) \bullet (h',\eta') = \begin{cases} (h \uplus h',\ \lambda\beta.\, \eta(\beta) \uplus \eta'(\beta)) & \text{if } \mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

The operator merges two pairs of heaps and region-maps when they do not overlap. The definition of $\varphi * \psi$ uses this operator to express the splitting of the heap and region-map components. Also note that the semantics of emp says that both the heap and the region map are empty.

Syntax and semantics of programs. We consider simple imperative programs specified in terms of standard control flow graphs. These programs are directed graphs (V,E) with two distinguished vertices $\mathsf{entry}, \mathsf{exit} \in V$ and a labeling function L from E to primitive instructions. The vertex entry is required to have no incoming edges and exit no outgoing edges.

The syntax of primitive instructions c are given by the following grammar:

$$\begin{array}{ll}
e ::= x \mid \texttt{null} & c ::= \texttt{assume}(b) \mid x := e \mid x := e.\texttt{f} \mid e.\texttt{f} := e \\
b ::= e = e \mid e \neq e & \quad\ \mid \texttt{free}(e) \mid x := \texttt{new}_{\alpha,F}() \quad (\text{where } F \subseteq \mathsf{Fields}) \\
\quad \mid b \wedge b \mid b \vee b & \quad\ \mid \texttt{move}(e,\alpha) \mid \texttt{moveRgn}(\alpha,\beta)
\end{array}$$

Most cases are standard imperative operations. For instance, $\texttt{assume}(b)$ checks whether the input state satisfies b. If so, it skips. Otherwise, it diverges. The only exceptions are the last three cases. The instruction $x := \mathtt{new}_{\alpha,F}()$ allocates a new cell with fields F, and puts this cell into the region α. The fields of this new cell are uninitialized. The next two $\mathtt{move}(e,\alpha)$ and $\mathtt{moveRgn}(\alpha,\beta)$ are ghost instructions that mainly manipulate the region-map parts of states. When cell e is allocated in the input state, $\mathtt{move}(e,\alpha)$ removes this cell from its current region, and puts it in the region α. The instruction $\mathtt{moveRgn}(\alpha,\beta)$ moves all the cells in the region α to the region β. Hence, at the end of this instruction, α contains no cells, while β contains all cells that used to be in α. The meaning of both instructions is not ambiguous, because we assume that the input states are well-formed and so all allocated addresses belong to only one region variable.

Our analysis uses move and moveRgn to ensure that the region-map part of a state carries useful information about heap data structures. In particular, it aims to put each data structure, such as a list or a tree, in its own partition described by some region variable α, because then knowing $e \in \alpha$ is sufficient to identify the data structure containing e.

The formal meanings of primitive instructions are given in terms of functions from States to $\mathcal{P}(\mathsf{States}) \cup \{\mathit{err}\}$, where *err* models a memory error. Sample cases of the semantics appear in Figure 3.

4 Abstract States

Our abstract domain consists of assertions of the form:

$$(\varphi_{1,1} \vee \ldots \vee \varphi_{1,m_1}) \,\wedge\, (\varphi_{2,1} \vee \ldots \vee \varphi_{2,m_2}) \,\wedge\, \ldots \,\wedge\, (\varphi_{n,1} \vee \ldots \vee \varphi_{n,m_n}). \qquad (9)$$

Each conjunct here records the current analysis result of one sub-analysis. For instance, the first conjunct could express the findings of the list analysis, and say how fields for singly-linked lists are connected in the heap. The second conjunct could, on the other hand, be concerned with the result of the tree analysis, and describe the connection of tree-related fields. Notice that a disjunction appears right under the conjunction. This disjunction is used by a sub-analysis to keep track of various correlations of stack variables and heap data structures explicitly. We point out that this is the only disjunction explicitly appearing in the abstract state; $\varphi_{i,j}$ does not contain any disjuncts inside.

Formally, our domain is parameterized by a finite collection $\mathcal{F} = \{F_i\}_{1 \leq i \leq n}$ of sets of fields and primitive predicates p. The intention is that n specifies the number of sub-analyses, and that each F_i describes the fields and primitive predicates that the sub-analysis i cares about.

Once a parameter $\mathcal{F}$ is given, we can construct our abstract domain $\mathcal{D}(\mathcal{F})$ in three steps. First, we define special forms of assertions, called symbolic heaps:

$$\begin{array}{lcll}
\Pi & ::= & \mathsf{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi & \textit{Pure formulae} \\
\Sigma & ::= & \mathsf{true}_\alpha \mid (e \mapsto \{\vec{\mathtt{f}} : \vec{e}\})_\alpha \mid (p(\vec{e}))_\alpha \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Spatial formulae} \\
H & ::= & \exists \vec{x}.\, \Pi \wedge \Sigma & \textit{Symbolic heaps}
\end{array}$$

The Π part of a symbolic heap describes the information about variables, and the Σ part expresses a property on the heap and region-map components of states.

Note that in a symbolic heap, the region subscript $-_\alpha$ is used only in limited places with three basic predicates. Furthermore, the pure part of a symbolic heap does not contain membership expressions $e \in \alpha$; all memberships are implicitly expressed using the subscript formulae $-_\alpha$. These and other syntactic restrictions in symbolic heaps (such as the absence of disjunction and negation) are imposed so that we can reuse the core components of existing separation-logic based shape analyses, such as abstraction algorithms and transfer functions [6]. We write SH for the set of all symbolic heaps.

Second, we define a set of assertions used by each sub-analysis i. Let SH_i be the set of symbolic heaps H such that all points-to predicates $(e \mapsto \{\vec{\mathtt{f}} : \vec{e}\})_\alpha$ in H mention only fields in F_i (i.e., $\vec{\mathtt{f}} \subseteq F_i$), and all primitive predicates $(p(\vec{e}))_\alpha$ in H belong to F_i (i.e., $p \in F_i$). The domain for the sub-analysis i is $\mathcal{D}_i = \mathcal{P}_{\mathit{fin}}(\mathsf{SH}_i)$. The finite powerset operator is used here to express finite disjunction. For instance, the set $\{H_1, \ldots, H_m\} \in \mathcal{D}_i$ means the disjunction $H_1 \vee \ldots \vee H_m$.

Finally, the abstract domain $\mathcal{D}(\mathcal{F})$ is defined by $\mathcal{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \cup \{\top\}$ for $n = |\mathcal{F}|$. The cartesian product means the conjunction of assertions. For instance, the assertion (9) in the beginning of this section is formally represented by the tuple $(\{\varphi_{1,1}, \ldots, \varphi_{1,m_1}\},\ \{\varphi_{2,1}, \ldots, \varphi_{2,m_2}\},\ \ldots,\ \{\varphi_{n,1}, \ldots, \varphi_{n,m_n}\})$ in this domain. The element $\top$ means the possibility of error. We will use d to denote a non-$\top$ element in $\mathcal{D}$, and d_i to mean the i-th component of d.

The domain $\mathcal{D}(\mathcal{F})$ is a lattice, when $\top$ is considered the largest element and the non-$\top$ elements are ordered pointwise. Then, the lattice operations of $\mathcal{D}(\mathcal{F})$ are obtained by extending corresponding operations on the $\mathcal{D}_i$'s pointwise. For instance, the join $d \sqcup d'$ is given by $(d_1 \sqcup d'_1, \ldots, d_n \sqcup d'_n)$.

Weak reduction operator. One important operator of our domain is a weak reduction operator that transfers information among components of abstract states. The transferred information is about the allocatedness of a cell and a region variable α containing this cell. For instance, consider the abstract state:

$$\big(\mathtt{x} \mapsto \{\mathtt{next}{:}0\}_\alpha \vee (\exists a.\, \mathtt{x} \mapsto \{\mathtt{next}{:}a\}_\alpha * \mathsf{ls}(a)_\alpha)\big) \;\;\wedge\;\; \mathsf{tr}(\mathtt{y})_\alpha$$

where only the first conjunct says that cell $\mathtt{x}$ is allocated and belongs to the set α. Using our reduction operator, we can transfer this information about cell $\mathtt{x}$ from the first to the second conjunct. Given appropriate parameters, the operator transforms this abstract state to the one below:

$$\begin{aligned}&\big(\mathtt{x} \mapsto \{\mathtt{next}{:}0\}_\alpha \vee (\exists a.\, \mathtt{x} \mapsto \{\mathtt{next}{:}a\}_\alpha * \mathsf{ls}(a)_\alpha)\big)\\ &\wedge \exists uvw.\, \mathsf{tseg}(\mathtt{y}, 0, \mathtt{x}, u)_\alpha * \mathtt{x} \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_\alpha * \mathsf{tseg}(v, \mathtt{x}, 0, \_)_\alpha * \mathsf{tseg}(w, \mathtt{x}, 0, \_)_\alpha.\end{aligned}$$

Here the predicate $\mathsf{tseg}(a, b, c, d)$ describes a rooted tree segment with one hole. The root is a and its parent pointer points to b. The hole of the segment is an outgoing pointer from the tree, going from address d to address c. The source d belongs to the segment, but the target c does not. We write _ in the parameter of tseg when we do not want to specify the parameter.[2] Note that the second conjunct now talks about the allocatedness of cell $\mathtt{x}$ and its membership of α.

[2] Formally, $\varphi * \mathsf{tseg}(a, b, c, \_)$ is an abbreviation for $\exists d. \varphi * \mathsf{tseg}(a, b, c, d)$ for a fresh d.

$$\mathsf{get}_{\Pi}(e,\ \mathsf{emp}) = \mathsf{NoInfo}$$

$$\mathsf{get}_{\Pi}(e,\ e'{\mapsto}\{\vec{\mathtt{f}}{:}\vec{e''}\}_{\alpha} * \Sigma) = \mathbf{if}\ (\Pi \vdash e = e')\ \mathbf{then}\ \alpha\ \mathbf{else}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$$

$$\mathsf{get}_{\Pi}(e,\ p(\vec{e'})_{\alpha} * \Sigma) = \mathbf{if}\ (\Pi \wedge p(\vec{e'}) \vdash e{\mapsto}\{\} * \mathsf{true})\ \mathbf{then}\ \alpha\ \mathbf{else}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$$

$$\mathsf{getRegion}(e,\ H) = \mathbf{let}\ (\exists\vec{x}.\, \Pi \wedge \Sigma) = H\ \mathbf{in}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$$

$$\begin{array}{l}\mathsf{caseID}(e, \mathsf{tr}(e_0)_{\alpha}) = \\ \quad \{(uvwxy,\ \mathsf{tseg}(e_0, 0, e, u)_{\alpha} * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_{\alpha} * \mathsf{tseg}(v, e, 0, x)_{\alpha} * \mathsf{tseg}(w, e, 0, y)_{\alpha})\}\end{array}$$

$$\begin{array}{l}\mathsf{caseID}(e, \mathsf{tseg}(e_0, e_1, e_2, e_3)_{\alpha}) = \\ \quad \{(uvwx,\ \mathsf{tseg}(e_0, e_1, e, u)_{\alpha} * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_{\alpha} * \mathsf{tseg}(v, e, e_2, e_3)_{\alpha} * \mathsf{tseg}(w, e, 0, x)_{\alpha}), \\ \quad\ (uvwx,\ \mathsf{tseg}(e_0, e_1, e, u)_{\alpha} * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_{\alpha} * \mathsf{tseg}(v, e, 0, x)_{\alpha} * \mathsf{tseg}(w, e, e_2, e_3)_{\alpha})\}\end{array}$$

$$\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ \mathsf{emp}) = \emptyset$$

$$\begin{array}{l}\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ e'{\mapsto}\{\vec{\mathtt{f}}{:}\vec{e''}\}_{\beta} * \Sigma') = \\ \quad \mathbf{if}\ (\Pi \vdash e \neq e'\ \text{or}\ \alpha \not\equiv \beta)\ \mathbf{then}\ \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * e'{\mapsto}\{\vec{\mathtt{f}}{:}\vec{e''}\}_{\beta},\ \Sigma') \\ \quad \mathbf{else}\ \{([],\ e{=}e',\ \Sigma * e'{\mapsto}\{\vec{\mathtt{f}}{:}\vec{e''}\}_{\beta} * \Sigma')\} \cup \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * e'{\mapsto}\{\vec{\mathtt{f}}{:}\vec{e''}\}_{\beta},\ \Sigma')\end{array}$$

$$\begin{array}{l}\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ p(\vec{e'})_{\beta} * \Sigma') = \\ \quad \mathbf{if}\ (\alpha \not\equiv \beta)\ \mathbf{then}\ \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * p(\vec{e'})_{\beta},\ \Sigma') \\ \quad \mathbf{else}\ \{(\vec{a}, \mathsf{true}, \Sigma * \Sigma'' * \Sigma')\ \mid\ (\vec{a}, \Sigma'') \in \mathsf{caseID}(e, p(\vec{e'}))\} \cup \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * p(\vec{e'})_{\beta}, \Sigma')\end{array}$$

$$\begin{array}{l}\mathsf{caseSH}(e, \alpha, H) = \\ \quad \mathbf{let}\ (\exists\vec{x}.\, \Pi \wedge \Sigma) = H\ \mathbf{in}\ \{\exists\vec{x}\vec{a}.\, (\Pi \wedge \Pi') \wedge \Sigma'\ \mid\ (\vec{a},\ \Pi',\ \Sigma') \in \mathsf{case}_{(e,\alpha,\Pi)}(\mathsf{emp},\ \Sigma)\}\end{array}$$

Fig. 4. Subroutines getRegion and caseSH$_i$. The function caseID below is a parameter provided for each primitive predicate p. In the figure, we give an example of caseID for tr and tseg.

Our operator is defined by lifting a similar reduction operator on symbolic heaps to abstract states. We first describe this original unlifted operator, denoted trans. Let i be a sub-analysis id and e an expression.

$$\begin{array}{l}\mathsf{trans}_i(e)(H : \mathsf{SH}, H' : \mathsf{SH}_i) : \mathcal{D}_i = \\ \quad \mathbf{let}\ R = \mathsf{getRegion}(e, H)\ \mathbf{in}\ \mathbf{if}\ (R = \mathsf{NoInfo})\ \mathbf{then}\ \{H'\}\ \mathbf{else}\ \mathsf{caseSH}(e, R, H').\end{array}$$

The operator $\mathsf{trans}_i(e)(H, H')$ transfers information about cell e from H to H', and the transferred information talks about the allocatedness of e and a region variable that contains e. The operator starts by calling the subroutine $\mathsf{getRegion}(e, H)$, which has two possible outcomes. The first outcome is NoInfo indicating that H does not have any information on cell e. In this case, the input H' gets no information from H, and it becomes the output of trans. The second outcome is a region variable α satisfying the entailment $H \vdash e \in \alpha$, which means that according to H, the region variable α contains cell e. Given this outcome, the operator trans conjoins the membership information $e \in \alpha$ with H', and calls a case-analysis routine that transforms the assertion back into a set of symbolic heaps in SH$_i$, while ensuring the soundness condition expressed below:

$$\mathcal{H} = \mathsf{caseSH}(e, \alpha, H') \implies (e \in \alpha \wedge H') \vdash \bigvee \mathcal{H}$$

One implementation of getRegion and caseSH is given in Figure 4.

For sub-analysis ids i, j and an expression e, we define our weak reduction operator $\mathsf{trans}_{i\to j}(e) : \mathcal{D} \to \mathcal{D}$ by $\mathsf{trans}_{i\to j}(e)(\top) = \top$ and

$$\mathsf{trans}_{i\to j}(e)(d) = \mathbf{let}\ \mathcal{H} = \bigcup\{\mathsf{trans}_j(e)(H, H') \mid (H, H') \in d_i \times d_j\}\ \mathbf{in}\ d[j \mapsto \mathcal{H}].$$

This operator applies trans_j to all possible symbolic-heap combinations from the i and j-th components of d, and uses the result to update the j-th component.

Note that the parameters i, j, e control the transferred information by our reduction operator. It restricts the source to only one component of an abstract state, and does a similar restriction on the target. Furthermore, it transfers information only about cell e, with respect to its membership to one region variable. This fine-grained control is essential for the performance of our analysis. Based on the results of a pre-analysis, our analysis does only necessary information transfer among component sub-analyses, by using our reduction operator with carefully chosen parameters and only at necessary program points.

5 Pre-analysis

The input to our analysis is a control flow graph $G = (V, E, \mathsf{entry}, \mathsf{exit}, L)$ and an initial abstract state $d_{\mathit{init}} \in \mathcal{D} \setminus \{\top\}$. Since G represents a normal C program, its labelling L does not use our ghost instructions or weak reduction operator.

Given this input, the analysis first runs a pre-analysis, which changes G by inserting ghost instructions of the form $\mathtt{move}(e, \alpha)$ or our weak reduction operator $\mathsf{trans}_{i \to j}(e)$. Intuitively, the pre-analysis finds a program point v such that if cell e is allocated at v in the concrete semantics, all sub-analyses are likely to infer the allocatedness of e. Then, it picks a fresh region variable α, and inserts $\mathtt{move}(e, \alpha)$ after v. For $\mathsf{trans}_{i \to j}(e)$, the pre-analysis inserts this instruction before a program point v', if it makes the following three conclusions at v':

1. The cell e will be dereferenced by the sub-analysis j.
2. The sub-analysis is *unlikely* to infer that all cells reachable from e by F_j-fields are allocated or `null`, while this allocation property indeed holds in the concrete semantics.
3. But the sub-analysis i is *likely* to infer the same type of information about cells reachable from e by F_i-fields.

Here $\{F_i\}_{1 \le i \le n}$ is a parameter to our analysis. The first can be detected easily by a simple syntactic check, but the other two require more sophisticated reasoning. In the remainder of this section, we focus on this reasoning as well as that used for inserting $\mathtt{move}(e, \alpha)$.

Both types of reasoning are based on data-flow analyses. Let $\mathsf{Subanalyses}$ be the set of sub-analysis ids $\{1, \ldots, n\}$, and Exp the set of expressions in the input control flow graph G. Define the domain $\mathcal{D}_{\mathit{pre}}$ by $\mathcal{D}_{\mathit{pre}} = \mathcal{P}(\mathsf{Subanalyses} \times \mathsf{Exp})$. Two data-flow analyses compute maps from program points to $\mathcal{D}_{\mathit{pre}}$, denoted R^r and R^p, by repeatedly applying the following equations:

$$\text{for } k \in \{r, p\} \text{ and } v \in V \setminus \{\mathsf{entry}\}, \quad R^k_{n+1}(v) = \bigcap_{(v',v) \in E} (\!|L(v', v)|\!)^\sharp_k (R^k_n(v')).$$

This commonality leaves only $R^r(\mathsf{entry})$, $R^p(\mathsf{entry})$, $(\!|L(v', v)|\!)^\sharp_r$, and $(\!|L(v', v)|\!)^\sharp_p$ unspecified. We give this missing information about the two data-flow analyses in Figure 5.

$R^r_n(\mathsf{entry}) = \{(i,e) \mid e \text{ is } \mathtt{null} \text{ or appears in all } H \text{ in } (d_{init})_i\}$, $\quad R^p_n(\mathsf{entry}) = \emptyset$

For $k \in \{r,p\}$, $(\!|c|\!)^\sharp_k(X) = (X \cup \mathsf{deref}(c)) \setminus \mathsf{kill}(c) \cup \mathsf{gen}_k(c,X)$ with $\mathsf{deref}, \mathsf{kill}, \mathsf{gen}_k$ below:

instr c	$\mathsf{deref}(c)$	$\mathsf{kill}(c)$	$\mathsf{gen}_r(c,X)$	$\mathsf{gen}_p(c,X)$
$\mathtt{assume}(b)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$x := e$	$\emptyset$	$\mathsf{Subanalyses} \times \{x\}$	$\{(i,x) \mid (i,e) \in X\}$	$\{(i,x) \mid (i,e) \in X\}$
$x := e.\mathtt{f}$	$\{(i,e) \mid \mathtt{f} \in F_i\}$	$\mathsf{Subanalyses} \times \{x\}$	$\{(i,x) \mid \mathtt{f} \in F_i\}$	$\emptyset$
$e.\mathtt{f} := e'$	$\{(i,e) \mid \mathtt{f} \in F_i\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{free}(e)$	$\emptyset$	$\mathsf{Subanalyses} \times \{e\}$	$\emptyset$	$\emptyset$
$x := \mathtt{new}_{\alpha,F}()$	$\emptyset$	$\mathsf{Subanalyses} \times \{x\}$	$\mathsf{Subanalyses} \times \{x\}$	$\mathsf{Subanalyses} \times \{x\}$

Fig. 5. Subroutines used by the pre-analysis. The abstract state $d_{init} \in \mathcal{D}$ is the initial abstract state given as the input to the whole analysis.

Our intention is that if $(i,e) \in R^r(v)$, then sub-analysis i likely infers that at program point v, all cells reachable from e by F_i-fields are allocated or `null`, if this property holds in the concrete semantics. Also, by $(i,e) \in R^p(v)$, we intend that the sub-analysis i knows enough to prove the allocatedness of e at program point v, if e is indeed allocated in the concrete semantics. Hence, if our intentions are properly implemented, the reasoning steps necessary for inserting trans and $\mathtt{move}(e,\alpha)$ can be done using R^r and R^p.[3]

The definitions of $(\!|c|\!)^\sharp_r$ and $(\!|c|\!)^\sharp_p$ in Figure 5 follow our intentions. The only exception is in the use of deref and $\mathsf{gen}_r(x{:=}e.\mathtt{f}, X)$. Here the definitions assume that after the dereference operation $e.\mathtt{f}$, all sub-analyses caring about `f` know both types of reachability and allocatedness information regarding e. Intuitively, this assumption amounts to hypothesizing that our pre-analysis inserts the reduction operator trans in all the necessary places, so that by the time that the field `f` is dereferenced, every sub-analysis interested in the field knows the necessary information.

Finally, the definitions of $R^p(\mathsf{entry})$ and $R^r(\mathsf{entry})$ in Figure 5 reflect another assumption of ours: That the initial abstract state d_{init} does not imply allocatedness, but it contains expression e in the i-th conjunct only when the `null` status or the allocatedness of cells reachable from e is known to sub-analysis i.

6 Invariant Inference

Next, our analysis runs its main invariant inference engine, which computes an invariant at each program point. Our invariant inference engine takes an initial abstract state d_{init} and the output of our pre-analysis, which is a control flow graph $G = (V, E, \mathsf{entry}, \mathsf{exit}, L)$ that can include ghost instructions $\mathtt{move}(e,\alpha)$ and the reduction operator $\mathsf{trans}_{i \to j}(e)$ (but not $\mathtt{moveRgn}(\alpha,\beta)$). Given this input, the engine computes two maps I and A from program points, the first I to abstract states and the next A to sets of ghost instructions of the form `moveRgn`:

$$M = \{\mathtt{moveRgn}(\alpha,\beta) \mid \alpha,\beta \in \mathsf{Regions}(d_{init}, L)\}, \quad I : V \to \mathcal{D}, \quad A : V \to \mathcal{P}(M).$$

[3] $\mathtt{move}(e,\alpha)$ is inserted after a program point v if $\mathsf{Subanalyses} \times \{e\} \subseteq R^p(v)$. Our second and third conditions for inserting $\mathsf{trans}_{i \to j}(e)$ are $(j,e) \notin R^r(v)$ and $(i,e) \in R^r(v)$.

```
$[\![c]\!]^\sharp(d) = \textbf{if } (\exists i.\ [\![c]\!]^\sharp_i(d_i) = \top) \textbf{ then } \top \textbf{ else } ([\![c]\!]^\sharp_1(d_1), \ldots, [\![c]\!]^\sharp_n(d_n))$
$\mathsf{abs}(d) = (\mathsf{abs}_1(d_1), ..., \mathsf{abs}_n(d_n))$                    $[\![\mathsf{trans}_{i \to j}(e)]\!]^\sharp(d) = \mathsf{trans}_{i \to j}(e)(d)$
$[\![\mathtt{move}(e, \alpha)]\!]^\sharp(d) =$                                 $[\![\mathtt{moveRgn}(\alpha, \beta)]\!]^\sharp(d) = d[\beta/\alpha]$
  $\textbf{let } \mathsf{check}(i, H) =$
      (1) Find finitely many $H_k$'s in $\mathsf{SH}_i$ such that
        $H \vdash \bigvee_{k \in K} H_k$ and $H_k$ has the form $\exists \vec{x_k}.\Pi_k \wedge e \mapsto \{\vec{\mathtt{f}_k}{:}\vec{e'_k}\}_{\beta_k} * \Sigma_k$.
      (2) If cannot find, return $\{\top\}$. Otherwise, for the found $H_k$'s, rename the
        subscript $\beta_k$ of $e \mapsto \{...\}_{\beta_k}$ by $\alpha$ and return $\{\exists \vec{x_k}.\Pi_k \wedge e \mapsto \{\vec{\mathtt{f}_k}{:}\vec{e'_k}\}_\alpha * \Sigma_k\}_{k \in K}$.
    $\textbf{and } r_i = \bigcup \{\mathsf{check}(i, H) \mid H \in d_i\}$ for all $i$
  $\textbf{in if } (\exists i \text{ s.t. } \top \in r_i) \textbf{ then } \top \textbf{ else } (r_1, \ldots, r_n)$
```

Fig. 6. Abstract transfer functions. The abstract value d below is not $\top$. When $\top$ is the input, $\mathsf{abs}(\top) = [\![c]\!]^\sharp \top = \top$. We assume that $[\![c]\!]^\sharp_i$ and abs_i are given.

Here $\mathsf{Regions}(d_{init}, L)$ is the set of region variables appearing in d_{init} or some instruction in the range of L. Note that since $\mathsf{Regions}(d_{init}, L)$ is finite, so are M, its subsets and the collection $\mathcal{P}(M)$. The first map I is the usual result of a program analysis, and keeps an invariant at each program point. The second map A records the ghost instructions dynamically discovered and then executed during the invariant inference. These instructions move cells from one region variable to another, and they are added and executed so as to maintain the relationship between region variables and data structures in the heap.

Our analysis uses the standard fixpoint algorithm for control flow graphs, with one interesting twist regarding the map A for ghost instructions. Assume that for all normal or ghost instructions or our weak reduction operator, c, we are given the transfer function for c, and the following two functions:

$$[\![c]\!]^\sharp : \mathcal{D} \to \mathcal{D}, \qquad \mathsf{abs} : \mathcal{D} \to \mathcal{D}, \qquad \mathsf{enableAbs} : \mathcal{D} \to \mathcal{P}(M). \tag{10}$$

Here $\mathsf{abs}(d)$ abstracts assertions in d, and $\mathsf{enableAbs}(d)$ returns ghost instructions in M that will enable further abstraction of d. Now, for (finite) subsets M_0 of M, define $[\![M_0]\!]^\sharp(d) = ([\![c_n]\!]^\sharp \circ \ldots \circ [\![c_1]\!]^\sharp)(d)$, where $c_1, \ldots, c_n$ is one enumeration of M_0 according to a fixed scheme. (In our analysis, this choice does not matter, because the transfer functions of any two instructions in M commute.) Using what we have assumed or defined, we define the main fixpoint algorithm below:

$$\begin{array}{ll} I_n(\mathsf{entry}) = d_{init}, & I_{n+1}(v) = \bigsqcup_{(v',v) \in E} (\mathsf{abs} \circ [\![A_n(v)]\!]^\sharp \circ [\![L(v', v)]\!]^\sharp)(I_n(v')), \\ A_n(\mathsf{entry}) = \emptyset, & A_{n+1}(v) = A_n(v) \cup \mathsf{enableAbs}(I_{n+1}(v)). \end{array}$$

Note that since $\mathcal{P}(M)$ is finite, there are only finitely many values for A, and the fixpoint computation of A does not cause non-termination. In practice, we found that the analysis time is dominated by the fixpoint computation for I.

To complete the story, we need to discharge our assumption of the three functions in (10). For normal instructions c, we define $[\![c]\!]^\sharp$ and abs by applying componentwise the sub-analyses' transfer functions $[\![c]\!]^\sharp_i : \mathcal{D}_i \cup \{\top\} \to \mathcal{D}_i \cup \{\top\}$ and abstraction routines $\mathsf{abs}_i : \mathcal{D}_i \to \mathcal{D}_i$. The details are given in Figure 6. The figure also shows that $[\![\mathsf{trans}_{i \to j}(e)]\!]^\sharp$ is implemented by the reduction operator with the same name, $[\![\mathtt{moveRgn}(\alpha,\beta)]\!]^\sharp$ by the substitution of the source region variable α by the target β, and $[\![\mathtt{move}(e,\alpha)]\!]^\sharp$ by the exposure of a points-to fact $e \mapsto \{\ldots\}_{\beta_k}$ from a symbolic heap followed by the renaming of its subscript β_k by α. The remaining operator is $\mathsf{enableAbs}$, which we define by $\mathsf{enableAbs}(d) = \{\mathtt{moveRgn}(\alpha,\beta) \mid \mathsf{abs}([\![\mathtt{moveRgn}(\alpha,\beta)]\!]^\sharp(d)) \neq d\}$. This operator returns all the region movement operations that enable further abstraction of its input d.

7 Experiments

We have implemented an interprocedural version of the analysis (based on the RHS algorithm [11]), and applied it to verify the memory safety of two types of programs. The first are toy examples of modest size and with just enough structure to warrant an overlaid analysis. The second are programs lifted from the Linux 2.6.37 code base. The results of our experiments appear in Figure 7.

The figure also includes the numbers obtained by applying our previous analysis built in 2008 to the same examples. This previous analysis couples the sub-analyses more tightly (using the abstract domain $\mathcal{P}(\mathsf{SH}_1 \times \ldots \times \mathsf{SH}_n) \cup \{\top\}$), it does not use ghost instructions, and it transfers information among sub-analyses more frequently than our current analysis. The figure shows that the current implementation performs better, and the performance gain becomes more significant, when a program becomes bigger or more complicated. There are also a number of programs that cannot be analysed at all by the old analysis.

The right-most column records the number of $\mathsf{trans}_{i\to j}(e)$ inserted by the pre-analysis of our current implementation. It shows that very little communication is happening. We consider this a primary factor of the efficiency of the analysis.

The benchmark set consists of the following programs, and can be found at `https://sites.google.com/site/overlaiddata`.

- `list-dio` is an abstract version of the deadline IO scheduler. It uses two doubly-linked lists instead of list and tree. The `sim` version skips the request-move routine, which cannot be verified by the old analysis.
- `many-keys` has an overlaid data structure of doubly-linked lists that are ordered by different keys. The number of lists is annotated in the filename.
- `many-lists` uses multiple doubly-linked lists implemented by different fields. These lists do not share nodes, so they do not form an overlaid data structure. However, our analysis can analyse each list separately, using a distinct conjunct for each list. The number of lists is annotated.
- `cache` has one doubly-linked list and pointers to cells in the list that were recently accessed. We can separately analyse the list and pointers by using our technique. The number of cache pointers is annotated.
- `block/deadline-iosched.c` has an overlaid data structure of a doubly-linked list and a red-black tree to maintain a list of requests. The original source was modified as follows: irrelevant fields and procedures such as ones for locks, and language constructors such as arrays that our analyser does not support, were removed, and assumptions were inserted to tree operations to compensate for our inaccurate tree abstraction. The `sim`/`sim2` version skips

filename	# of	analysis time (sec)		speedup	# of trans
	lines	(A) old	(B) new	(A/B)	inserted
list-dio-sim.c	110	3.12	1.56	2.0	2
list-dio.c	134	–	3.95	–	4
many-keys-3.c	92	1.65	0.72	2.3	2
many-keys-4.c	98	8.16	1.22	6.7	3
many-lists-3.c	106	1.90	1.37	1.4	3
many-lists-4.c	124	12.53	3.05	4.1	4
cache-1.c	88	1.29	0.97	1.3	9
cache-2.c	93	14.70	1.81	7.8	11
linux/block/deadline-iosched-sim.c	1,941	237.67	32.76	7.3	4
linux/block/deadline-iosched-sim2.c	1,968	5,399.73	100.06	54.0	4
linux/block/deadline-iosched.c	2,131	–	364.45	–	5
linux/fs/afs/server-sim.c	712	705.67	22.61	31.2	9
linux/fs/afs/server.c	1,084	–	1,932.65	–	13

Fig. 7. Experimental result. Used Intel Core i7 2.66GHz with 8GB memory.

procedures that the old analyser cannot verify due to its imprecision, as well as procedures of high analysis cost.

- fs/afs/server.c also has a similar data structure to maintain servers but it has one more component of doubly-linked list for removing servers: Servers to be removed are additionally connected to the graveyard list. So, in this case, the overlaid data structure consists of three components.

8 Conclusion

In this paper, we have presented a static analysis for overlaid data structures, capable of verifying memory safety of real world programs. Our insight is to decompose an overlaid data structure into its components, and to track components using sub-analyses as independently as possible, while allowing communication among them using ghost instructions. Besides the progress in verifying more challenging data structures, we hope that our work has provided further evidence that with proper understanding of more programming patterns in systems code, together with specialized abstractions, one can design effective automatic verifiers for ever-larger classes of real-world systems programs.

References

1. Arnold, G., Manevich, R., Sagiv, M., Shaham, R.: Combining shape analyses by intersecting abstractions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 33–48. Springer, Heidelberg (2005)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, p. 268. Springer, Heidelberg (2001)

3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
4. Cherini, R., Rearte, L., Blanco, J.: A shape analysis for non-linear data structures. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 201–217. Springer, Heidelberg (2010)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)
6. 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)
7. Hawkins, P., Aiken, A., Fisher, K.: Reasoning about shared mutable data structures (2010) (manuscript)
8. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data structure fusion. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 204–221. Springer, Heidelberg (2010)
9. Kreiker, J., Seidl, H., Vojdani, V.: Shape analysis of low-level C with overlapping structures. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 214–230. Springer, Heidelberg (2010)
10. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. In: IEEE TSE (2006)
11. Reps, T., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)
12. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24(3), 217–298 (2002)
14. 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)

KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs

Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan

Fujitsu Labs of America, CA
{gli,ighosh,sree.rajan}@us.fujitsu.com

Abstract. We present the first symbolic execution and automatic test generation tool for C++ programs. First we describe our effort in extending an existing symbolic execution tool for C programs to handle C++ programs. We then show how we made this tool generic, efficient and usable to handle real-life industrial applications. Novel features include extended symbolic virtual machine, library optimization for C and C++, object-level execution and reasoning, interfacing with specific type of efficient solvers, and semi-automatic unit and component testing. This tool is being used to assist the validation and testing of industrial software as well as publicly available programs written using the C++ language.

1 Introduction

With the ubiquitous presence of software programs permeating almost all aspects of daily life, providing robust and reliable software has become a necessity. Traditionally, software quality has been assured through manual testing which is tedious, difficult, and often gives poor coverage of the source code especially when availing of random testing approaches. This has led to much recent work in the formal validation arena [4,1]. One such formal technique is symbolic execution which can be used to automatically generate test inputs with high structural coverage for the program under test. The widely used symbolic execution engines currently are able to handle C or Java programs only. So far there has been no such formal tool designed specifically for the automatic validation and test generation for C++ programs. Currently C++ is the language of choice for most low-level scientific and performance critical applications in academia and industry. This paper describes our efforts in creating the first industrially usable symbolic execution engine for C++ programs.

Symbolic execution [4,1] performs the execution of a program on symbolic (open) inputs. It characterizes each program path it explores with a path condition which denotes a series of branching decisions. The solutions to path conditions are the test inputs that will assure that the program under test runs along a particular concrete path during concrete execution. Typically a decision procedure such as a SMT (Satisfiability Modulo Theory) solver is used to find the solutions and prune out false paths. Exhaustive testing is achieved by exploring all true paths. Some sanity properties can also be checked such as memory out-of-bound access, divide-by-zero, and certain types of user-defined assertions.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 609–615, 2011.

Our tool is built on top of a symbolic execution engine KLEE [4] which is able to handle sequential C programs (mainly Unix utility programs). Our extended tool addresses the following questions and issues:

- How to extend a symbolic executor to handle C++ features?
- What optimizations are necessary to make the tool efficient and scalable?
- Can the tool work well on industrial applications and other important programs and beat manual testing with minimal manual effort?

2 Extended Executor for C++ Programs

As shown in Fig. 1, the tool's flow is similar to KLEE's. A C++ program is compiled into LLVM [7] bytecode, which is interpreted by KLOVER for symbolic execution. To handle the C++ library constructs we use a special C++ library which is described later. After the execution, statistics information and sanity check results are given. The other set of outputs are concrete test cases, which can be replayed in the real setting (*e.g.* compiled by GCC and run in a machine). After the replay, source program coverage is produced by `gcov`.

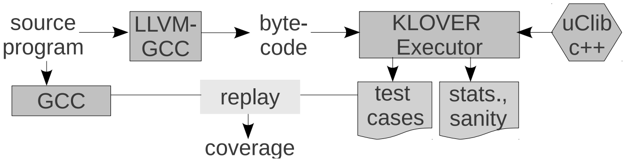


Fig. 1. Overall architecture of KLOVER

Virtual Machine State. A symbolic state in KLOVER models a machine execution state. A register stores a concrete value or a symbolic expression. A memory is organized as components, each of which has a concrete address and an array of bytes recording the value. The fields of a C++ object are allocated consecutive memory blocks. In the following example, the two fields of object 1 (with runtime type t_1) satisfies $\mathrm{m}_{1,2} = \mathrm{m}_{1,1} + \texttt{size}(\mathrm{fd}_{1,1})$. The memory blocks of different objects do not have to be consecutive, which can support automatic dynamic resizing. If a pointer can refer to multiple components, then a new state is generated for each possible reference (determined by SMT solving).

$\mathrm{object}_1 : \mathrm{t}_1$		...	$\mathrm{object}_2 : \mathrm{t}_2$		...
$(\mathrm{m}_{1,1},\mathrm{fd}_{1,1})$	$(\mathrm{m}_{1,2}, \mathrm{fd}_{1,2})$	...	$(\mathrm{m}_{2,1}, \mathrm{fd}_{2,1})$	...	...

C++ Language Features. Most C++ features such as templates and class inheritance are handled by the LLVM-GCC compiler. However, since C++ is far more complicated than C, there may be extra LLVM instructions (mainly intrinsic functions) and external functions which KLEE doesn't handle. Presently KLOVER can handle most of the widely occurring C++ specific LLVM instructions and external functions, and we are extending the tool further to handle the complete set. The new instructions and issues include:

- Advanced Instructions. For example, the llvm.stacksave intrinsic is used to remember the current state of the function stack, which is to be restored by llvm.stackrestore. The implementation of these instructions follows their semantics and is quite straightforward.
- Exceptions. An important feature of C++ is to provide built-in support for exceptions. The several llvm.eh. instructions along with a few external functions need to be interpreted in the right exception semantics, *e.g.* propagate the exceptions up the stack. We introduce a specific data structure to represent exceptions, build the exception table, control the bytecode execution flow of exceptions, and interpret exception instructions.
- C++ RTTI. C++'s Run-time Type Information system keeps information about an objects type at runtime. Besides enabling RTTI in LLVM-GCC, we keep track of the the runtime types of objects of polymorphic classes so as to handle operations such as dynamic_cast. Class hierarchy information is inferred from the type definitions in LLVM and the control flow.
- Memory Model . C++'s memory model involves many atomic operations and synchronization intrinsics. The compilation to specific platforms may also involve them. For example, llvm.memory.barrier guarantees ordering between specific pairs of memory access types, and lvm.atomic .load.add performs the add and store atomically. We do not address concurrency in this paper; while in [5] we describe how to extend KLOVER for GPU programs and compare symbolic execution with other symbolic methods [6] for concurrent programs.

C++ Library. The C++ standard includes a library for all commonly used data structures and algorithms. We choose and optimize the uClibc++ library [8] so as to improve the performance of symbolic execution. We compile this library into LLVM bytecode and load it into the engine dynamically. We maintain two versions of the C++ library: one for symbolic execution, the other one for handling concrete values and the Just-In-Time compilation of external functions.

3 Optimizations

To scale up the tool we adopt a variety of optimizations which are essential to the tool's performance and usability. These optimization are in addition to the ones done in KLEE which KLOVER inherits. The new optimizations can have a huge impact on the quality of results and symbolic execution time, as shown by the following example. We compare the cases without any optimization, with our optimized library, and with our string solver, on the main benchmark program used in [2]. The first two cases could not achieve full branch coverage since the input string is of specific size. In this section we elaborate these optimizations.

No Optimization			+Optimized Lib.			+FLA String Solver		
#tests	bran. cov.	time	#tests	bran. cov	time	#tests	bran. cov.	time
>10,000	67%	>2 hr.	6	67%	6 sec.	9	100%	3 sec.

The standard C++ library is designed for concrete execution. Efficient symbolic execution requires rewriting all the C and C++ class implementations to:

(1) avoid unnecessary conditional statements to reduce the number of generated paths; (2) convert expensive expressions into cheaper ones; and (3) build fast decision procedures into the library implementation. KLOVER has optimized a number of commonly-used classes and algorithms with a similar purpose as [3].

Library Optimization (Operational Approach). The first optimization technique modifies the body of a function, which will be executed directly. For example, the compare method of the String class is as follows. It will produce only one path regardless of the values of the two input strings (of concrete lengths).

```
_UCXXEXPORT int compare(const basic_string& str) const {
  size_type rlen = vector<Ch, A>::elements;
  if (rlen > str.elements) rlen = str.elements;
    int v = 0; // 1, 0 and -1 stand for gt, eq and lt respectively
    for (size_type i = 0; i < rlen; i++)
      v += (~(!v)+1) & ((operator[](i)>str[i]) - (operator[](i)<str[i]));
    v += (~(!v)+1) & ((vector<Ch, A>::elements > str.elements) -
                      (vector<Ch, A>::elements < str.elements));
    return v;
}
```

Library Optimization (Relational Approach). We can build a solver in the source code without "executing" the implementation. For example, the find_last_of method is as follows, where Vr denotes the return value. Function assume informs the executor to record the constraint. This implementation relates the inputs and outputs using logical formulas. It also produces only one path. Building solvers through source code definitions is a core feature of KLOVER.

```
find_last_of (const char c) {
  size_type rlen = vector<Ch, A>::elements;
  assume(Vr >= -1 && Vr < rlen);
  for (size_type i = 0; i != rlen ; i++)
    assume(i <= Vr || operator[](i) != c);
  assume(Vr == -1 || operator[](Vr) == c);
}
```

Object-level Execution and Reasoning. One of the main features of C++ is class and object. KLOVER's intermediate language (IL) is extended to model them directly. During the execution, a method call is not immediately expanded to its implementation when it is first encountered. Instead, a "lazy evaluation" approach is adopted to delay the evaluation until needed. Consider the following code. When the condition is encountered, KLOVER builds the expression str.substr(str.find_last_of('/') + 1) = "EasyChair", which can be simplified to str = s1 + "/EasyChair" for a free string variable s1. KLOVER builds in such simplifications and decision procedures (see next section) for common classes. We may simply use the library definitions of the methods to interpret this expression — now the interpretation is delayed to the condition point. We believe that object level abstractions is crucial for a Object-oriented language like C++.

```
int k = str.find\_last_of(’/’);
string rest = str.substr(k + 1);
if (rest == "EasyChair")  ...
```

Specific Solvers. To further improve the performance of object-level reasoning, we implement off-the-shelf solvers for some common data structures. For instance we've implemented a string solver based on SMT solving which is similar to that in [2]. Consider the above example. Our solver creates the following expression constraining the values and lengths of the string variables. The constraints on the lengths are first extracted and solved to obtain a (minimal) instance for each length. Then the length of each string is set and the string constraint is solved. With such built-in solvers KLOVER not only improves the performance, but also allows more feasible inputs (*e.g.* having variable lengths).

$$
\begin{aligned}
&\land\ (k = -1\ \land\ \forall i \in [0, \mathtt{len}(\mathit{str})) : \mathit{str}[i] \neq \text{'/'}\ \lor \\
&\quad k \in [0, \mathtt{len}(\mathit{str}))\ \land\ \mathit{str}[k] = \text{'/'}\ \land\ \forall i \in [k+1, \mathtt{len}(\mathit{str})) : \mathit{str}[i] \neq \text{'/'}) \\
&\land\ \mathit{rest} = \mathit{str}[k+1, \mathtt{len}(\mathit{str}))\ \land\ \mathtt{len}(\mathit{rest}) = \mathtt{len}(\mathit{str}) - k - 1 \\
&\land\ \mathit{rest} = \text{“EasyChair”}\ \land\ \mathtt{len}(\mathit{rest}) = 9
\end{aligned}
$$

4 Experimental Results

KLOVER requires a driver to invoke a C++ program with symbolic inputs. A user is free to mark any input symbolic, but should ensure that the relationship between the inputs is appropriate. Since C++ program encapsulate the members of a class, the driver calls a class's public methods to make an object "symbolic".

KLOVER supports the declaration of an array to have a symbolic length. When an access to such an array is out of bounds; we increase dynamically the array's size to accommodate this access (up to some ceiling). KLOVER also supports the declaration of a possibly null pointer. Through such extensions, KLOVER is able to reduce the manual testing burden significantly over KLEE.

We run KLOVER on some real-life applications developed in Fujitsu on a laptop with two 1.60GHz processors and 2GB memory. Table 4 compares KLOVER with the manual method for unit testing. The size of these classes are about 5,500 lines of code in total. The (semi-automated) drivers for KLOVER are much more succinct and apprehensive than the manually written ones. For each class, KLOVER achieves much higher coverage by producing only a small number of test cases (<20). KLOVER's unit testing is able to beat manual testing in all cases, both in line coverage and branch coverage. Similar results have been obtained on other industrial instances, *e.g.* a real application with around 130,000 and 51,000 lines in the *.hh and *cc files respectively.

We also run KLOVER on some C++ applications which are publicly available (*e.g.* at www.sourceforge.com). Table 4 shows the results for SHA-1 (a cryptographic hash function), Balancing AVL tree, a Regular Expression package (ported from java.util.regex), and a URI package for analyzing URLs. For these widely-used algorithms, there exist no prior effort on checking and testing their C++ versions using symbolic execution or other formal methods. Reg.Exp. and URI contain intensive string operations. KLOVER reveals several bugs (infinite

Table 1. Experimental results of unit testing an industrial application. We compare manual unit testing with KLOVER in terms of the driver's size and the coverage (of format line coverage / branch coverage).

Class	Driver LOC	Coverage	Driver LOC	Coverage	Exec.
	(Manual)	(Unit, Manual)	(KLOVER)	(Unit, KLOVER)	Time
Class 1	547	50.6%/27.65%	73	96.4%/78.8%	2.4s
Class 2	726	96.21%/59.65%	45	100%/75.9%	0.3s
Class 3	537	93.51%/73.33%	58	97.4%/73.6%	0.5s
Class 4	337	94.16%/86.11%	52	100%/87.2%	0.5s
Class 5	286	87.68%/70.27%	95	100%/77%	2.6s

Table 2. Experimental results on the C++ versions of some publicly available programs. KLOVER checks their sanity and produces test cases.

Prog.	Source	Sanity	Coverage	#tests	Coverage	#tests	Exec.
	LOC		(Manual)	(Manual)	(KLOVER)	(KLOVER)	Time
SHA-1	450	Y	—	—	97.50%/91.67%	22	30s
AVLTree	700	Y	81.17%/42.04%	150	92.86%/41.53%	13	10m
Reg.Exp.	3,100	N*	58.88%/59.12%	12	87.87%/89.19%	999(5*)	37s
URI	2,200	Y	86.5%/62.0%	180	89.8%/64.4%	134	2.5m

loops) in "Reg.Exp." which are missed by the manual testing. The replaying in real settings shows that these bugs are real in 5 test cases.

It is possible that some run-time exception cases (*e.g.* running out of memory) are not covered by KLOVER such that the coverage may not reach 100%. The coverage with KLOVER can be improved further with refined drivers. We intentionally keep the drivers simple such that they can be written quickly by the users knowing little about the applications. Yet such drivers (with appropriate constraints on the symbolic inputs) are able to achieve high coverage.

Concluding Remarks. Our tool is the first symbolic executor and test generation designed and tuned particularly for industrial C++ programs. We plan to further extend the tool and scale it up for larger C++ programs.

References

1. Anand, S., Păsăreanu, C.S., Visser, W.: JPF–SE: A symbolic execution extension to java pathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
2. Bjørner, N., Tillmann, N.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, Springer, Heidelberg (2009)
3. Blanc, N., Groce, A., Kroening, D.: Verifying C++ with STL containers via predicate abstraction. In: Automated Software Engineering, ASE (2007)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI (2008)

5. Li, G., Ghosh, I., Rajan, S. P., and Gopalakrishnan, G. GKLEE: A symbolic execution and automatic test generation tool for GPU programs, Draft (2011)
6. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Foundations of Software Engineering, SIGSOFT FSE (2010)
7. The LLVM compiler infrastructure, `http://www.llvm.org/`
8. uClibc++: An embedded C++ library, `http://cxx.uclibc.org`

Fully Symbolic Model Checking for Timed Automata[⋆]

Georges Morbé, Florian Pigorsch, and Christoph Scholl

Department of Computer Science, University of Freiburg
{morbe,pigorsch,scholl}@informatik.uni-freiburg.de

Abstract. In this paper we introduce a new formal model, called finite state machines with time (FSMT), to represent real-time systems. We present a model checking algorithm for FSMTs, which works on fully symbolic state sets containing both the clock values and the state variables. In order to verify timed automata (TAs) with our model checking algorithm, we present two different methods to convert TAs to FSMTs. In addition to pure interleaving semantics we can convert TAs to FSMTs having a parallelized interleaving behavior which allows parallelism of transitions causing no conflicts. This can dramatically reduce the number of steps during verification. Our experimental results show that our prototype implementation outperforms the state-of-the-art model checkers UPPAAL and RED.

1 Introduction

The application area of real-time systems grows with an enormous speed and along with that grows their complexity as well as the damage caused by their failure. For these reasons verification of such systems becomes more and more important. Timed automata (TAs) [1,2] turned out to be an adequate formalism for modeling and verifying real-time systems. Timed automata generalize finite automata by adding real-valued clock variables. All clock variables evolve over time with the same rate and they can be reset during discrete steps which in turn happen in zero-time. Verifying safety properties of TAs can be reduced to the computation of all states reachable from the initial states and checking whether an unsafe state can be reached (forward model checking). Equivalently the problem can be reduced to the computation of all states from which unsafe states can be reached and checking whether some initial states are included in this set of states (backward model checking).

Model checking approaches for TAs based on reachability analysis can be classified into *fully symbolic* and *semi-symbolic* approaches. Semi-symbolic approaches represent discrete locations of TAs explicitly whereas sets of clock valuations are represented symbolically e.g. by *unions of clock zones*. Clock zones are convex regions which result from an intersection of clock constraints of the form $x_i - x_j \sim d$

[⋆] This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 616–632, 2011.

where $d \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$ and x_i, x_j are clock variables. UPPAAL [13,3], the probably most prominent semi-symbolic approach, represents clock zones by so-called difference bound matrices (DBMs) and provides efficient methods for manipulating DBMs. These techniques are well-suited when the sizes of the discrete state space and the numbers of different clock regions per location remain moderate. CDDs [12] make the attempt to represent unions of clock zones more compactly. CDDs are BDD-like data structures where nodes are labeled by clock differences $x_i - x_j$ and the outgoing edges of nodes are labeled by (disjoint) intervals of rational numbers. CRDs [23] are a variant of CDDs where outgoing edges of nodes are labeled by upper bounds for clock differences instead of disjoint intervals. CRDs were combined with BDDs (leading to CRD+BDDs) to provide a *fully* symbolic representation of the state space in the tool RED [23]. Another fully symbolic representation has been given by difference decision diagrams (DDDs) [18] which are basically BDD representations where the decision variables are boolean abstractions of clock constraints $x_i - x_j \sim d$. Computing all states reachable by evolution of time amounts to the existential quantification of a real-valued variable. Both for CRD+BDDs and DDDs this quantification is performed based on the classical Fourier–Motzkin technique which requires enumerating all paths in the diagram. Restricted to a path representing a conjunction of clock constraints, the Fourier–Motzkin technique is strongly related to quantifier elimination in DBMs by the shortest-path closure [14]. As in DDDs, Seshia and Bryant [22] consider BDD representations using boolean abstractions of clock constraints, however they reduce real-valued quantifier elimination to adding so–called transitivity constraints followed by a series of quantifications for boolean variables. Recently, Clock Matrix Diagrams (CMDs) were introduced [10]. CMDs basically correspond to CRD+BDDs where sequences of edges representing convex constraints are collapsed into single edges labeled by DBMs and boolean variables are restricted to the lowest levels in the variable orders.

In this paper we first introduce a new formal model for real-time systems, called finite state machines with time (FSMT), which is especially suited for symbolic verification algorithms. We present a fully symbolic model checking algorithm for FSMTs. In order to verify TAs (with additional integer variables in the state space) we present a method to convert a TA into an FSMT. In addition to normal interleaving semantics (i.e. asynchronous semantics) of TAs we give a symbolic representation of an FSMT simulating a *'parallelized interleaving'* behavior, which allows parallelism of transitions causing no conflicts. This parallelized interleaving behavior can dramatically reduce the number of steps during verification.

In contrast to partial-order reduction (e.g. [16,19]) which reduces the number of states to be considered during model checking, parallelized interleaving does not avoid certain computation paths or states, but combines their traversal into one *symbolic* step and thus accelerates state space traversal. Consider a TA T composed from n components $TA_1, \ldots, TA_n$ and suppose – for simplicity – that the local discrete transitions of the components are independent, i.e., they are neither related through read or write conflicts nor they synchronize over actions. According to the semantics of the concurrent asynchronous system T a discrete step of T consists in a discrete step of some component TA_i. For the concurrent execution of one discrete step per component, there are $n!$ different sequences and 2^n different states (one state for each subset of executed components). If the specification does not distinguish between these sequences, partial-order reduction can reduce $n!$ sequences to one representative sequence consisting of n transitions. *Symbolic* model checkers without partial-order reduction already compute a symbolic representation of all 2^n states visited on $n!$ sequences by n symbolic steps. Symbolic model checking with *parallelized interleaving* assumes that each component TA_i may or may not take a transition, considers all possible combinations in parallel, and computes a symbolic representation for all these 2^n states by *one single step*. Of course, for the general case we have to analyze which components may run in parallel without changing the semantics.

Path reduction [24] provides an alternative possibility for mitigating negative effects of pure interleaving. Path reduction analyzes components and replaces certain computation paths by single transitions. In that way, computation paths of components are compressed, leading to a reduced number of possible interleavings of different components. Path reduction is orthogonal to our technique, since it preprocesses components, whereas parallelized interleaving improves the parallel execution of *several* components by combining computation paths resulting from different interleavings into one symbolic step.

Our model checking algorithm uses LinAIGs ('And-Inverter-Graphs with linear constraints') [7,21,6] to describe the state space. LinAIGs provide a fully symbolic representation both for the continuous part (i.e. the clock values) and the discrete part (i.e. the state variables). For state space compaction LinAIGs profit to a large extent from the enormous progress made in the area of SAT and SMT (SAT modulo Theories) solving [4,9]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [15] which is especially suitable for LinAIG representations.

First experimental results show that our prototype implementation outperforms UPPAAL and RED in both configurations, for pure interleaving behavior and for parallelized interleaving behavior. The results also indicate that for benchmarks allowing parallelized interleaving behavior this approach has a stunning performance due to reduction of the number of steps during verification.

The paper is organized as follows. In Sect. 2 we give a brief review of the well-known timed automata (TA), then we introduce finite state machines with time (FSMT) in Sect. 3. In Sect. 4 we provide an insight into the functioning of our model checking algorithm. In Sect. 5 we introduce a method to convert a TA into an FSMT using standard interleaving and parallelized interleaving. Sect. 6 is dedicated to the results where we evaluate our approach with both configurations. We conclude the paper in Sect. 7.

2 Preliminaries – Timed Automata

Real-time systems are often represented as timed automata (TAs) [1,2].

TAs use clock variables $X := \{x_1, \ldots, x_n\}$. The set of clock constraints $\mathcal{C}(X)$ contains atomic constraints of the form $(x_i \sim d)$ and $(x_i - x_j \sim d)$ with $d \in \mathbb{Q}$

and $\sim \in \{<, \leq, =, \geq, >\}$. Let $\mathcal{C}_c(X)$ be the set of conjunctions over clock constraints. $c \in \mathcal{C}_c(X)$ describes a subset of $\mathbb{R}^n$, namely the set of all valuations of variables in X which evaluate c to true.

We consider TAs with integer variables. Let $Int := \{int_1, \ldots, int_r\}$ be a set of bounded integer variables. $lb : Int \to \mathbb{Z}$ and $ub : Int \to \mathbb{Z}$ assign lower and upper bounds to $int_i \in Int$ $(lb(int_i) \leq ub(int_i))$. Let $Assign(Int)$ be the set of assignments to integer variables. The right-hand side of an assignment to an integer variable int_i may be an integer arithmetic expression over integer variables and integer constants.

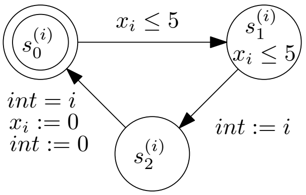


Fig. 1. Example TA_i

Let $Cond(Int)$ be a set of constraints of the form $(int_i \sim d)$ and $(int_i \sim int_j)$ with $d \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $int_i, int_j \in Int$. Let $Cond_c(X, Int)$ be the set of conjunctions over clock constraints and constraints from $Cond(Int)$.

Example 1. The timed automaton TA_i shown in Fig. 1 has only one clock variable, $X = \{x_i\}$. It has three 'locations' $s_0^{(i)}$, $s_1^{(i)}$, and $s_2^{(i)}$. Locations are connected by transitions which may be labeled. The transition $(s_2^{(i)} \to s_0^{(i)})$, e.g., is labeled with the guard $int = i$, with an assignment $int := 0$, and with the clock reset $x_i := 0$. The location $s_1^{(i)}$ is labeled with a clock constraint $x_i \leq 5$ which is a so-called location invariant.

In general, transitions in TAs are labeled with guards, actions, assignments to integers and resets of clocks. Guards are restricted to conjunctions of clock constraints and constraints on integers. Actions from $Act := \{a_1, \ldots, a_k\}$ are used for synchronization between different TAs. For our purposes they do not have a special meaning when considering one timed automaton in isolation. Transitions in different automata labeled with the same actions are taken simultaneously. If a transition in a TA is not labeled by an action, then this transition can only be taken, if all other TAs stay in their current location. Resets are assignments to clock variables of the form $x_i := 0$. Invariants in TAs are conjunctions of clock constraints assigned to locations. A TA may stay in a location as long as the location invariant is not violated. Timed automata are formally defined as follows:

Definition 1 (Timed Automaton). *A timed automaton* (TA) *is a tuple* $\langle L, l_0, X, Act, Int, lb, ub, E, Inv\rangle$ *where* L *is a finite set of locations,* $l_0 \in L$ *is an initial location,* $X = \{x_1, \ldots, x_n\}$ *is a finite set of real-valued clock variables,* Act *is a finite set of actions,* $Int = \{int_1, \ldots, int_r\}$ *is a finite set of integer variables.* $lb : Int \to \mathbb{Z}$ *and* $ub : Int \to \mathbb{Z}$ *assign lower and upper bounds to each* $int_i \in Int$ *with* $lb(int_i) \leq ub(int_i)$ *for* $1 \leq i \leq r$, $E \subseteq L \times Cond_c(X, Int) \times (Act \cup \{\epsilon\}) \times 2^X \times 2^{Assign(Int)} \times L$ *is a set of transitions and the function* $Inv : L \to \mathcal{C}_c(X)$ *assigns a conjunction of clock constraints as invariant to each location. If for* $e = (l, g_e, act, r_e, assign_e, l') \in E$ *it holds that* $act \in Act$*, then we call* e *a transition with a synchronizing action; if* $act = \epsilon$*, then we call* e *a transition without synchronizing action.*

Definition 2 (Semantics of a Timed Automaton). *Let $TA = \langle L, l_0, X, Act, Int, lb, ub, E, Inv\rangle$ be a timed automaton. A state of TA is a combination of a location and a valuation of the clock variables and integer variables.*

- *There is a continuous transition from state $s = (l, x_1^v, \dots, x_n^v, int_1^v, \dots, int_r^v)$ to state $s' = (l, x_1^w, \dots, x_n^w, int_1^v, \dots, int_r^v)$ ($s \rightarrow^c s'$) iff $(x_1^v, \dots, x_n^v)$ and $(x_1^w, \dots, x_n^w)$ fulfill $Inv(l)$, $lb(int_i) \leq int_i^v \leq ub(int_i)$ $\forall 1 \leq i \leq r$, and there is $t \in \mathbb{R}_0^+$ with $\forall 1 \leq j \leq n : x_j^w = x_j^v + t$.*
- *There is a discrete transition from state $s = (l, x_1^v, \dots, x_n^v, int_1^v, \dots, int_r^v)$ to state $s' = (l', x_1^w, \dots, x_n^w, int_1^w, \dots, int_r^w)$ ($s \rightarrow^d s'$) iff $(x_1^v, \dots, x_n^v)$ fulfills $Inv(l)$, $(x_1^w, \dots, x_n^w)$ fulfills $Inv(l')$, $lb(int_i) \leq int_i^v, int_i^w \leq ub(int_i)$, $\forall 1 \leq i \leq r$, and $\exists e = (l, g_e, act, r_e, assign_e, l') \in E$ with $(x_1^v, \dots, x_n^v, int_1^v, \dots, int_r^v)$ fulfills the guard g_e, $x_i^w = 0$ for $x_i \in r_e$, $x_i^w = x_i^v$ for $x_i \notin r_e$, the values $int_1^w, \dots, int_r^w$ result from $int_1^v, \dots, int_r^v$ by applying the assignments in $assign_e$.*
- *$\rightarrow = \rightarrow^d \cup \rightarrow^c$ is the transition relation of a TA. A trajectory of a TA is a finite or infinite sequence of states $(s^j)_{j \geq 0}$ with $s^0 = (l_0, 0, \dots, 0, lb(int_1), \dots, lb(int_r))$ and $s^{j-1} \rightarrow s^j$ for each $j > 0$. A state is reachable, if there is a trajectory ending in that state.*

A timed system is a system of p timed automata $\{TA_1, \dots, TA_p\}$. A timed system has an interleaving semantics, i.e., transitions in different timed automata may not be taken simultaneously unless they synchronize over actions. For simplicity, we assume that only two timed automata are able to synchronize over a binary synchronization channel. As usual, the composition of p timed automata is again a timed automaton.

3 Finite State Machines with Time

Now we present a new formal model to represent real-time systems, the finite state machines with time, which are especially suited for being represented symbolically. A finite state machine with time, to which we will refer as FSMT in this paper, is an extension of finite state machines by real-valued clock variables. Later on, we will present a fully symbolic model checking algorithm for FSMTs and then a translation from TAs into FSMTs.

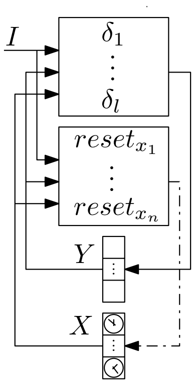


Fig. 2. FSMT

Let $X := \{x_1, \dots, x_n\}$ be the set of real-valued clock variables, $Y := \{y_1, \dots, y_l\}$ a set of (binary) state variables, $I := \{i_1, \dots, i_h\}$ a set of (binary) input variables. Let $\mathcal{C}_b(X)$ be the set of arbitrary boolean combinations of clock constraints and $\mathcal{C}_b(X, Y)$ be the set of arbitrary boolean combinations of clock constraints and state variables (similarly for $\mathcal{C}_b(X, Y, I)$). As usual, $c \in \mathcal{C}_b(X, Y)$ describes a subset of $\mathbb{R}^n \times \{0, 1\}^l$, namely the set of all valuations of variables in X and Y which evaluate c to true. An FSMT is defined as follows (see Fig. 2 for an illustration):

Definition 3 (FSMT). *A finite state machine with time (FSMT) is a tuple* $\langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv\rangle$ *where* $X := \{x_1, \ldots, x_n\}$ *is a set of clock variables,* $Y := \{y_1, \ldots, y_l\}$ *is a set of state variables,* $I := \{i_1, \ldots, i_h\}$ *is a set of input variables,* $init : (\mathbb{R}_0^+)^n \times \{0,1\}^l \rightarrow \{0,1\}$ *is a predicate describing the set of initial states,* $\delta_i : (\mathbb{R}_0^+)^n \times \{0,1\}^l \times \{0,1\}^h \rightarrow \{0,1\}$ $(1 \leq i \leq l)$ *are transition functions,* $reset_{x_j} : (\mathbb{R}_0^+)^n \times \{0,1\}^l \times \{0,1\}^h \rightarrow \{0,1\}$ $(1 \leq j \leq n)$ *are reset functions,* $Inv : (\mathbb{R}_0^+)^n \times \{0,1\}^l \rightarrow \{0,1\}$ *is a predicate describing a state invariant, and* $init \wedge \neg Inv = 0$*. The functions* δ_i *and* $reset_{x_j}$ *can be represented by boolean combinations from* $\mathcal{C}_b(X, Y, I)$*, init and* Inv *can be represented by boolean combinations from* $\mathcal{C}_b(X, Y)$*.*

A state of an FSMT is a valuation $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v) \in (\mathbb{R}_0^+)^n \times \{0,1\}^l$ of the clock variables and the state variables. A valuation $(y_1^v, \ldots, y_l^v)$ is also called a *location* of the FSMT. Trajectories of an FSMT always start in states fulfilling *init* and all states on these trajectories have to fulfill the state invariant Inv. An FSMT may perform discrete steps which are defined by transition functions δ_i based on the valuations of clocks, state variables, and inputs. When performing a discrete step, a clock x_i is reset to 0 iff $reset_{x_i}$ evaluates to 1. Moreover an FSMT may perform continuous steps (or time steps) where it stays in the same location, but lets time pass. This means that all clocks may be increased by the same constant as long as the resulting state stays in the set described by Inv. More formally, the semantics of FMSTs is defined as follows:

Definition 4 (Semantics of an FSMT). *Let* $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv\rangle$ *be an FSMT.*

- *There is a continuous transition from state* $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v)$ *to state* $s' = (x_1^w, \ldots, x_n^w, y_1^v, \ldots, y_l^v)$ $(s \rightarrow^c s')$ *iff* $Inv(s) = Inv(s') = 1$ *and there is* $t \in \mathbb{R}_0^+$ *with* $\forall 1 \leq j \leq n : x_j^w = x_j^n + t$*.*[1]
- *There is a discrete transition from state* $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v)$ *to state* $s' = (x_1^w, \ldots, x_n^w, y_1^w, \ldots, y_l^w)$ $(s \rightarrow^d s')$ *iff* $Inv(s) = Inv(s') = 1$ *and there is* $(i_1^v, \ldots, i_h^v) \in \{0,1\}^h$ *with*
  $\forall 1 \leq i \leq l : y_i^w = \delta_i(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v),$
  $$\forall 1 \leq j \leq n : x_j^w = \begin{cases} x_j^v, & \text{if } reset_{x_j}(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v) = 0 \\ 0, & \text{if } reset_{x_j}(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v) = 1. \end{cases}$$
- $\rightarrow = \rightarrow^d \cup \rightarrow^c$ *is the transition relation of* F*. A trajectory of* F *is a finite or infinite sequence of states* $(s^j)_{j \geq 0}$ *with* $init(s^0) = 1$ *and* $s^{j-1} \rightarrow s^j$ *for each* $j > 0$*. A state is reachable, if there is a trajectory ending in that state.*

We consider systems of FSMTs $\{F_1, \ldots, F_p\}$, where the components are running in parallel. Communication in such a system is realized just as for communicating

[1] Usually we require that Inv fulfills the following property: If we fix variables $y_1, \ldots, y_l$ of Inv to arbitrary constant values 0 or 1, then the resulting predicate shall describe a convex set. If this would not be the case, then there could be a continuous transition from s to s' with time step of length t, but no continuous transition from s to s'' with time step of length $t' < t$, since $Inv(s'') = 0$.

FSMs. FSMTs communicate by reading each other's state variables, clocks, and shared input variables. Thus, composition of FSMTs is done just by replacing input variables of the components by state variables of other components or by inputs of the overall system. The composition of p FSMTs $F_1, \ldots, F_p$ is again an FSMT:

Definition 5. *Let $F_1, \ldots, F_p$ be FSMTs with $F_i = \langle X, Y^{(i)}, I^{(i)}, init^{(i)}, \delta^{(i)}, (reset^{(i)}_{x_1}, \ldots, reset^{(i)}_{x_n}), Inv^{(i)} \rangle$, $Y^{(i)} = \{y^{(i)}_1, \ldots, y^{(i)}_{l_i}\}$, $I^{(i)} = \{i^{(i)}_1, \ldots, i^{(i)}_{h_i}\}$. Let $map : \bigcup_{i=1}^{p} I^{(i)} \rightarrow (I \cup \bigcup_{i=1}^{p} Y^{(i)})$ be a mapping for the inputs of components $F_1, \ldots, F_p$ and let $I = \{i_1, \ldots, i_h\}$ be the set of (global) inputs. Then the composition of $F_1, \ldots, F_p$ wrt. map is an FSMT F with $F = \langle X, \bigcup_{i=1}^{p} Y^{(i)}, I, \bigwedge_{i=1}^{p} init^{(i)}, (\tilde{\delta}^{(1)}, \ldots, \tilde{\delta}^{(p)}), (\vee_{i=1}^{p} reset^{(i)}_{x_1}, \ldots, \vee_{i=1}^{p} reset^{(i)}_{x_n}), \wedge_{i=1}^{p} Inv^{(i)} \rangle$ and $\tilde{\delta}^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, i_1, \ldots, i_h) = \delta^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, map(i^{(i)}_1), \ldots, map(i^{(i)}_{h_i}))$.*

4 Model Checking Algorithm

Algorithm 1. Model checking algorithm

```
$\Phi_0 := \neg safe; \Phi_{collect} := 0; i := 0$
while ($\Phi_i \wedge \neg\Phi_{collect} \neq 0$) do
    if ($\Phi_i \wedge$ init $\neq 0$) then return false
    $\Phi_{collect} := \Phi_{collect} \vee \Phi_i$
    $i := i + 1$
    $\Phi_i := Pre_c(\Phi_{i-1})$
    if ($\Phi_i \wedge$ init $\neq 0$) then return false
    $\Phi_i := Pre_d(\Phi_i)$
return true
```

Algorithm: Our model checking algorithm is a backward model checking algorithm working on an FSMT $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv \rangle$ as defined in Def. 3. It starts with the negation of a safety predicate $safe$ and – step by step – computes sets of states from which $\neg safe$ can be reached. The main loop consists of a continuous step given by $\Phi_i := Pre_c(\Phi_{i-1})$ and a discrete step given by $\Phi_i := Pre_d(\Phi_i)$. The implementation of Pre_c and Pre_d will be shown below. After each of these steps we test whether one of the initial states was reached. The main loop is left when an initial state was reached (which means that the safety property is violated) or when a fixpoint is reached (which means that the safety property holds).

Continuous step: Let $\Phi(x_1, \ldots, x_n, y_1, \ldots, y_l)$ be a state set of our model checking algorithm. Then the state set reachable by a (backward) continuous step (letting time pass) can be described by

$$Pre_c(\Phi)(x_1, \ldots, x_n, y_1, \ldots, y_l) = \exists\lambda\, [(\lambda \geq 0) \wedge \Phi(x_1 + \lambda, \ldots, x_n + \lambda, y_1, \ldots, y_l)] \wedge Inv(x_1, \ldots, x_n, y_1, \ldots, y_l) \tag{1}$$

Discrete step: The resulting state set $Pre_d(\Phi)$ of a discrete step contains all predecessors of Φ from which Φ can be reached by a discrete transition in the FSMT. The first part of the discrete step is a substitution of the state variables and the clock constraints in the current state set representation Φ. (Note that as an invariant of our model checking algorithm all computed state set representations are in $\mathcal{C}_b(X, Y)$, i.e., they are boolean combinations of boolean variables and clock constraints.) Each state variable y_i is substituted with its transition function δ_i:

$$y_i \leftarrow \delta_i\left(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h\right) \tag{2}$$

Consider a clock constraint of the form $(x_i - x_j \sim d)$ with $x_i, x_j \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ and $d \in \mathbb{Q}$. There are only four possible cases how a clock constraint can be changed due to resets executed during a transition: (1) x_i and x_j are reset, (2) only x_i is reset, (3) only x_j is reset or (4) none of the clock variables in the constraint is reset. We use the reset conditions $reset_{x_i}$ to determine when a clock variable x_i is reset. The substitution for each clock constraint of the form $(x_i - x_j \sim d)$ in the state set is then

$$\begin{aligned}(x_i - x_j \sim d) \leftarrow (\; & (\; reset_{x_i} \wedge reset_{x_j} \wedge (0 \sim d)\;) \vee \\ & (\; \overline{reset_{x_i}} \wedge reset_{x_j} \wedge (x_i \sim d)\;) \vee \\ & (\; reset_{x_i} \wedge \overline{reset_{x_j}} \wedge (-x_j \sim d)\;) \vee \\ & (\; \overline{reset_{x_i}} \wedge \overline{reset_{x_j}} \wedge (x_i - x_j \sim d)\;)\;)\end{aligned} \tag{3}$$

(Of course, $(0 \sim d)$ reduces to constant 0 or 1.)

$\Phi'\left(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h\right)$ is obtained from $\Phi\left(x_1, \ldots, x_n, y_1, \ldots, y_l\right)$ by substituting all state variables as shown in Eqn. (2) and all clock constraints as shown in Eqn. (3) simultaneously.

The second part of the discrete step is a quantification of the boolean input variables $i_1, \ldots, i_h$ in Φ' followed by an intersection with the invariant Inv:

$$\begin{aligned}&Pre_d(\Phi)(x_1, \ldots, x_n, y_1, \ldots, y_l) = \\ &[\exists i_1, \ldots, i_h\; \Phi'\left(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h\right)] \wedge Inv\left(x_1, \ldots, x_n, y_1, \ldots, y_l\right)\end{aligned} \tag{4}$$

Implementation based on LinAIGs: We have implemented a prototype of the model checking algorithm using LinAIGs [7,21,6] for representing sets of states. LinAIGs are able to provide a compact representation for arbitrary boolean combinations of linear constraints and boolean variables (which of course include the formulas from $\mathcal{C}_b(X, Y)$). LinAIGs consist of both a boolean and a continuous part. The boolean part of LinAIGs is represented by functionally reduced And-Inverter-Graphs (FRAIGs) [17,20], which basically are boolean circuits consisting only of and gates and inverters. In order to represent the continuous part, LinAIGs use a set of boolean constraint variables Q where each linear constraint is encoded by some $q_l \in Q$. For keeping the overall representation as compact as possible, LinAIGs make heavy use of SMT solvers [4,9]. SMT solvers are used to prove that nodes represent equivalent predicates and thus can be merged. Moreover, they are used to detect and remove 'redundant linear constraints', i.e., constraints which are present in the current LinAIG, but not really needed for describing the represented predicate. This operation fights the increase in the number of linear constraints / boolean constraint variables which was already observed in [22]. Since in our application the linear constraints are restricted to clock constraints, we do not need SMT solvers for full linear arithmetic, but only for difference logic which can be solved much more efficiently.

Apart from boolean operations, LinAIGs support quantification of boolean and real variables and thus fit exactly the technical needs of our implementation. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [15]. (This method can even be used for linear constraints instead of more restricted clock constraints.) If there are k clock constraints with variable x_i, then the existential quantification $\exists x_i \varPhi$ for LinAIG $\varPhi$ can basically be reduced to $O(k)$ substitutions of test points into $\varPhi$ with an overall worst-case increase of the representation by a factor of $O(k)$. The quantifier elimination method from [22] is specialized to difference logic and adds up to $O(n^2)$ transitivity constraints to $\varPhi$ (when n is the number of clock variables), followed by $O(k)$ quantifications of boolean variables. Quantification of $O(k)$ boolean variables may increase the representation including transitivity constraints by a factor of $O(2^k)$ in the worst case. However, since such worst-case considerations do not always reflect the situation in practical applications we plan to implement the quantifier elimination method from [22] in the future as well and compare the results.

5 From Timed Automata to FSMTs

In order to be able to verify systems of TAs using our framework presented so far, we present how to convert a system of TAs into an FSMT.

Components of FSMTs run in parallel, whereas components of TAs run asynchronously (one after the other) according to the interleaving semantics (unless parallelism is enforced by synchronization actions). In our translation we consider two different implementations of the interleaving semantics of TAs. At first, in Sect. 5.2, we show how to transform a TA into an FSMT keeping its *pure interleaving* behavior. Then, in Sect. 5.3, we present how to convert a TA into an FSMT with a *parallelized interleaving* behavior, in which we allow – in addition to single steps of components according to the interleaving semantics – parallelism for transitions causing no conflicts when taken in parallel. The different conflicts possible with parallelized interleaving behavior are also described in Sect. 5.3. The motivation for the parallelized interleaving variant consists in an accelerated state space traversal.

5.1 First Steps of Translation

We consider a system of p timed automata $\{TA_1, \ldots, TA_p\}$. The locations of timed automaton $TA_q = \langle L^{(q)}, l_0^{(q)}, X, Act, Int, lb, ub, E^{(q)}, \; Inv^{(q)} \rangle$ $(1 \leq q \leq p)$ are encoded with boolean variables $y_1^{(q)}, \ldots, y_{l_q}^{(q)}$ (the location bits) for which we use a logarithmic encoding with $l_q = \lceil log\left(L^{(q)}\right) \rceil$. The sets of location bits of two different TAs are disjoint. The integer variable int_i with $(1 \leq i \leq r)$

occurring in the timed system is replaced by a binary encoding of boolean variables $b_1^{(i)}, \ldots, b_{f_i}^{(i)}$ (the integer bits). As $lb\,(int_i)$ and $ub\,(int_i)$ are known for all $1 \leq i \leq r$, the number of integer bits f_i needed to represent int_i is also known. The location bits and the integer bits together form the set of state bits $\{y_1, \ldots, y_l\}$.

The location invariants in a TA can be merged into one condition for the complete automaton of the form $Inv^{(q)}(y_1^{(q)}, \ldots, y_{l_q}^{(q)}, x_1, \ldots, x_n)$ (by a simple conjunction of an implication for each location with the meaning 'if TA_q is in location l, then the location invariant of l holds').

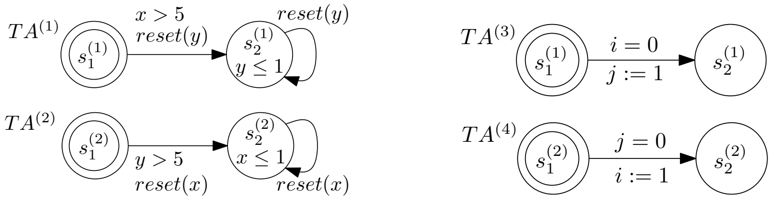
A timed automaton TA_q has a total of $m_q := |E^{(q)}|$ transitions. Assume that transition i in TA_q is a transition with the discrete location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ as source and the discrete location $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$ as destination. Let the transition i be labeled with a guard $g_i^{(q)}$ and a reset set $r_i^{(q)} \in 2^{\{x_1, \ldots, x_n\}}$. In order to make things easier in Sect. 5.2 and Sect. 5.3, the guard $g_i^{(q)}$ is extended by the constraint that the source of its corresponding edge is location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$, i.e., it is changed to the new guard $g_i'^{(q)} := g_i^{(q)} \wedge \left((y_1^{(q)})^{\epsilon_1^{(i,s)}} \wedge \ldots \wedge (y_{l_q}^{(q)})^{\epsilon_{l_q}^{(i,s)}} \right)$.

Moreover, a transition i in TA_q may be labeled with a synchronization action $a_{q,i}$. How to treat these actions is shown in Sect. 5.2 for interleaving behavior and in Sect. 5.3 for parallelized interleaving behavior.

5.2 Modifications for Pure Interleaving Behavior

In order to use the model checking algorithm with pure interleaving behavior, it has to be assured that at any time only one TA may take a transition while the others remain in their current location unless of course two TAs synchronize. (Remember that for simplicity we confine ourselves to binary synchronization.) We have two types of transitions which have to be considered separately:

- For transitions of two different TAs without synchronization actions it has to be ensured that they are not enabled at the same time. For this we use new input variables $\{e_{l-1}, \ldots, e_0\}$, $l = \lceil log(p) \rceil$ in a system of p timed automata and we add different assignments for these new input variables to the guards of such transitions: For each transition i in a timed automaton TA_q which is not labeled with a synchronization action we add these input variables to the guard $g_i'^{(q)}$ and get a new guard $g_i''^{(q)} = g_i'^{(q)} \wedge \left(e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0} \right)$ with $bin\,(q) = (q_{l-1}, \ldots, q_0)$. ($bin\,(q)$ is the binary representation of q.)
- For transitions labeled with a synchronization action we cannot use the previous modification as this would cause the synchronized transitions not be enabled at the same time. Let us assume that transition i in TA_q and transition j in TA_k are labeled with the same action $a_{\{(q,i),(k,j)\}}$. To assure synchronization without the use of actions we extend the guards of the synchronized transitions. The new guard of transition i in TA_q and of transition j in TA_k is $g_i''^{(q)} = g_j''^{(k)} := g_i'^{(q)} \wedge g_j'^{(k)} \wedge \left(\left(e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0} \right) \vee \left(e_{l-1}^{k_{l-1}} \wedge \ldots \wedge e_0^{k_0} \right) \right)$



(a) Read/write problem on clocks

(b) Read/write problem on integers

Fig. 3. Conflicts caused by parallel behavior

with $bin\,(k) = (k_{l-1}, \ldots, k_0)$ and $bin\,(q) = (q_{l-1}, \ldots, q_0)$. This allows us to realize synchronization without using actions simply by the fact that one component may read the state bits and inputs of the other component.

Since for an FSMT we have to define transition *functions*, we have to avoid the case that there is a state where no transition into a successor state is enabled. For this reason we introduce a self loop to every location in each timed automaton TA_q. The self loop of a location l_i gets as guard the conjunction of the negated guards of all outgoing transitions, thus the self loop of a location is enabled whenever no other outgoing transition is enabled.

Moreover, we have to exclude non-deterministic behavior (as allowed for TAs) to arrive at deterministic transition *functions* for FSMTs. When more than one transition is enabled in a TA at the same time it is chosen non-deterministically which one is taken. To establish determinism for FSMTs we use new input variables. For a set of t transitions with the same source we build a graph with one node for each transition and we add an edge between two transitions e_1 and e_2 iff e_1 and e_2 are non-disjoint. Then the question how many additional input variables are needed in order to make guards non-disjoint is reduced to a coloring problem for the resulting graph. If col is the number of colors needed for coloring, then we need $\lceil \log(col) \rceil$ input variables to make the guards disjoint. These input variables can be shared within a TA but must not be shared among different TAs. A timed automaton TA_q requires $t^{(q)} = \lceil \log(col_{max}^{(q)}) \rceil$ input variables to guarantee determinism, where $col_{max}^{(q)}$ is the maximum number of colors occurring for transitions with the same source.

After these transformations we can build the transition functions, reset conditions and invariant to get an FSMT representation of the timed system with pure interleaving behavior. This is shown in Sect. 5.4.

5.3 Modifications for Parallelized Interleaving Behavior

In the previous section we have seen which modifications have to be done to convert a timed system into an FSMT with pure interleaving behavior. In this section we will show the modifications to get an FSMT with parallelized interleaving behavior. To this end several potential conflicts have to be considered.

- In a parallelized interleaving run there may be conflicts caused by resets of clock variables. Consider the timed system shown in Fig. 3(a) which consists of components $TA^{(1)}$ and $TA^{(2)}$. When parallel transitions of two components are allowed, the state $(s_2^{(1)}, s_2^{(2)}, 0, 0)$ is reachable from state $(s_1^{(1)}, s_1^{(2)}, 6, 6)$ by taking the transitions from $s_1^{(1)}$ and $s_1^{(2)}$ in parallel. But according to interleaving semantics this state is unreachable. If w.l.o.g. $TA^{(1)}$ takes the transition leaving its initial state first, then it resets the clock y and y will never be larger than 1. Thus the transition of $TA^{(2)}$ from $s_1^{(2)}$ will never be enabled and $TA^{(2)}$ always stays in its initial location. (A similar observation holds for the case that $TA^{(2)}$ is executed first.)

  To avoid the problem of reaching more states than allowed by the semantics of interleaving, we force the timed system to simulate a pure interleaving behavior in such cases by adding read/write-enable numbers for clock variables. Assume q timed automata $TA_{i_1}, \ldots, TA_{i_q}$ having transitions which *both* read *and* reset a clock variable x_i at the same time. Then we need $\lceil \log(q+2) \rceil$ additional input variables to encode read/write-enable numbers rw^{x_i}. With the following approach these read/write-enable numbers inhibit that transitions reading x_i and transitions resetting x_i are enabled at the same time: Each guard of a transition in TA_{i_k} $(1 \leq k \leq q)$ with transitions reading and resetting x_i is extended by '$rw^{x_i} = bin(k+1)$'. The guard of each transition only reading x_i (only resetting x_i) is extended by '$rw^{x_i} = bin(0)$' ('$rw^{x_i} = bin(1)$'). Note that enabling parallel transitions only reading x_i or enabling parallel transitions only writing x_i does not cause a problem. (All writes set the clock value to the same value 0.)[2]

- Another conflict of the same type may occur with integers. It is obvious that two transitions updating the same integer int_i must not be taken in parallel because of write/write problems. But, just as we have seen for clock variables there may also be read/write conflicts on integer variables. In the timed system consisting of $TA^{(3)}$ and $TA^{(4)}$ shown in Fig. 3(b) the state $(s_2^{(1)}, s_2^{(2)})$ is not reachable according to interleaving semantics. However it is reachable, if transitions can be taken in parallel.

  Just as for the read/write conflict for clock variables we force the timed system to take an interleaving behavior for transitions causing conflicts on integer variables. For each integer int_i we introduce a read/write-enable number rw^{int_i}. The guard of each transition reading the value of integer int_i is extended by '$rw^{int_i} = bin(0)$'. Assume q TAs $TA_{i_1}, \ldots, TA_{i_q}$ updating int_i. Each guard of a transition in TA_{i_k} $(1 \leq k \leq q)$ which updates int_i is extended by '$rw^{int_i} = bin\,(k)$'. This makes it impossible that two TAs write int_i at the same time, since the corresponding guards cannot be enabled at the same time. Equally it is impossible that any integer variable is read and updated in the same discrete transition.

[2] Under certain circumstances the number of needed input variables can be minimized based on the fact that transitions of the *same* component TA_i can not be executed in parallel anyway.

The synchronization is handled in a similar way as we have seen in Sect. 5.2 for pure interleaving behavior. Let us assume that transition i in TA_q and transition j in TA_k are labeled with the same synchronization action $a_{\{(q,i),(k,j)\}}$. Then the guards of both transitions are changed to $g_i^{\prime\prime(q)} = g_j^{\prime\prime(k)} := g_i^{\prime(q)} \wedge g_j^{\prime(k)}$. The action $a_{\{(q,i),(k,j)\}}$ is no longer needed to synchronize the transitions. Both components in the system synchronize by reading each others state bits and inputs.[3]

Parallelized interleaving is introduced to accelerate model checking runs by reaching certain states faster. But of course, we should not lose intermediate states of interleaved executions. For that reason we give each component the non-deterministic choice to stay in its current location during a discrete step. For this we introduce a self loop with guard 'true' to every location in the automaton. By taking this transition the automaton does not leave the current location and does no assignments to clocks or integer variables. Then, to introduce determinism we do the same modifications using input variables as we have done for pure interleaving behavior in Sect. 5.2.

The resulting system is deterministic and has a parallelized interleaving behavior. In the following section we show how to compute transition functions, reset conditions and a global invariant.

5.4 Computation of a Symbolic Representation

Based on the guards $g_i^{\prime\prime(q)}$ for transitions i of TA_q (from $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ to $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$) as computed in Sect. 5.2 or 5.3 it is easy to compute the transition functions for state bits encoding locations of TA_q. We have to consider m'_q transitions for T_q (including new self loops added in Sect. 5.2 or 5.3). The transition function $\delta_j^{(q)}$ computes when the state bit j in the modified automaton TA_q is set to true. (Assume that the set of all input variables we have added according to Sect. 5.2 or 5.3 is $\{i_1, \ldots, i_h\}$.) Then

$$\delta_j^{(q)}(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) = \bigvee_{\substack{1 \le i \le m'_q \\ \epsilon_j^{(i,d)}=1}} g_i^{\prime\prime(q)}(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \quad (5)$$

The transition functions for state bits resulting from encoding of integer variables are derived from location encodings, the guards computed in Sect. 5.2 or 5.3, and right-hand side expressions of assignments.[4] Details are omitted here.

Besides the transition functions we need the reset functions for clocks. The following function indicates when the clock variable x_i is reset in TA_q:

[3] For ease of exposition we omit the special case of concurrent read/write or write/write on synchronizing transitions here.

[4] In our prototype implementation we restrict the right-hand sides of assignments to integer constants, integer variables and additions of two integers.

$$reset^{(q)}_{x_i}\,(x_1,\ldots,x_n,y_1,\ldots,y_l,i_1,\ldots,i_h) = \bigvee_{\substack{1\le i\le m'_q\\ x_i\in r^{(q)}_i}} g''^{(q)}_i\,(x_1,\ldots,x_n,y_1,\ldots,y_l,i_1,\ldots,i_h) \quad (6)$$

The overall reset function for a clock x_i is computed by $reset_{x_i} = \vee^{p}_{q=1} reset^{(q)}_{x_i}$.

As a last component of the FSMT computed from a system of timed automata $TA_1,\ldots,TA_P$, we compute the global invariant Inv simply by conjunction of all local invariants $Inv^{(q)}$.

The transition functions, reset conditions, and the invariant provide a fully symbolic representation of the corresponding FSMT. Our model checking algorithm uses this representation to perform fully symbolic model checking.

6 Experimental Results

Tab. 1 shows the results of our prototype FSMT-MC applied to several benchmarks with safety properties. In principle our LinAIG based implementation would allow model checking for full TCTL, but in our prototype only model checking for safety properties is implemented. We ran FSMT-MC with pure interleaving behavior (FSMT-MC inter) and with parallelized interleaving behavior (FSMT-MC para) and we compare the results to two state-of-the-art model checkers Uppaal v.4 and RED 8. By default Uppaal performs forward model checking and RED performs backward model checking. For Uppaal we tried both breadth first ('bf') and depth first ('df') traversal. We did not perform a comparison with TMV [22]; for safety properties TMV was outperformed by RED in [22]. All benchmarks were originally modeled as timed automata and were automatically translated into FSMTs with pure interleaving and parallelized interleaving behavior. CPU times for our (un-optimized) translator are also given in Tab. 1, column (TA2FSMT). We have conducted all experiments on a 16 core AMD Opteron with 2.3 GHz and 64 GB RAM with a time limit of 3 CPU hours and a memory limit of 2 GB. An entry 'to' in the table shows that the time limit was reached, an entry 'mo' shows that the memory limit was reached. All times in Tab. 1 are given in CPU seconds.

For our experiments we used parameterized benchmarks containing a number n of identical components, since this made it easy for us to generate sets of increasingly complex benchmarks for comparison. Actually we do not consider parameterized benchmarks as the main field of application for our algorithm and thus we did not make use of symmetry reduction [11,5], neither within our tool nor within any competitor. The first column in Tab. 1 gives the number of components for each benchmark instance.

The first benchmark implements our toy example from Fig. 1. It consists of n TAs $TA_1,\ldots,TA_n$ as shown in Fig. 1 with $\neg safe := \bigwedge^{n}_{i=1} s^{(i)}_1$ (which is reachable from the initial states). Comparing pure interleaving and parallelized interleaving, we can observe an enormous performance gain for parallelized interleaving due to a reduction of the number of steps in state space traversal. Our algorithm with parallelized interleaving behavior can finish state space traversal just after one step by taking the transition $(s_0^{(i)} \rightarrow s_1^{(i)})$ for all TA_i in parallel. Our algorithm with pure interleaving behavior computes in one step for each state reached so far all the predecessors reachable by one backward step of an arbitrary automaton. Thus in this simple example it needs n steps for a system with n processes to arrive at the initial state. Uppaal performs much worse on this example, since it works on an explicit representation of locations and it computes all possible permutations of enabled transitions step by step. Our approach clearly outperforms RED as well which is based on a different fully symbolic representation and performs only pure interleaving.

The second benchmark is Fischer's well known mutual exclusion protocol. As property we verify if it is possible that all components are in the critical region at the same time. As we can see in Tab. 1 the results of our algorithm with a pure interleaving behavior are better than the results with a parallelized interleaving behavior. This is caused by the fact that the Fischer protocol does not allow parallel behavior. Even if we run our model with a parallelized interleaving behavior, a pure interleaving behavior is simulated due to the read/write-enable numbers for the integer variable used in the benchmark. The additional inputs for the read/write-enable numbers which have to be quantified in the discrete step are responsible for the loss of performance. But in both configurations for pure interleaving and for parallelized interleaving behavior our symbolic model checking algorithm can solve systems with a lot more processes than Uppaal and RED.

The third benchmark 'critical region' models a system with n processes and a distributed arbiter which controls access to a critical region.[5] As we can see

Table 1. Experimental results

		UPPAAL		RED	FSMT-MC		TA2FSMT	
		bf	df		inter	para	inter	para
toy ex.	8	0.1	0.2	20	4	0.2	5	4
	9	0.3	0.9	mo	5	0.2	6	5
	14	360	777	mo	36	0.5	9	9
	15	mo	mo	mo	58	0.5	9	10
	22	mo	mo	mo	3308	1.4	41	21
	23	mo	mo	mo	to	1.5	42	24
	100	mo	mo	mo	to	73	170	936
Fischer	7	0.2	0.3	33	6	10	3	2
	8	1	1.6	mo	19	24	4	3
	11	77	308	mo	151	101	8	4
	12	305	1686	mo	256	580	9	5
	13	1190	9046	mo	517	1052	10	5
	14	mo	to	mo	1259	1677	11	5
	18	mo	to	mo	2515	4267	57	8
	19	mo	to	mo	3218	to	59	9
	21	mo	to	mo	8918	to	68	10

		UPPAAL		RED	FSMT-MC		TA2FSMT	
		bf	df		inter	para	inter	para
crit. region	4	0.5	0.9	3	7	4	3	3
	5	17	51	27	17	8	4	4
	6	860	5294	mo	31	17	6	6
	7	to	to	mo	71	50	7	8
	13	to	to	mo	3869	1113	22	23
	15	to	to	mo	8423	3627	28	32
	16	to	to	mo	to	2776	39	35
	17	to	to	mo	to	8762	46	42
FDDI	9	0.1	3	76	4	4	22	40
	10	0.2	13	mo	5	5	26	49
	14	1	5445	mo	19	24	50	100
	15	2	to	mo	29	39	57	117
	39	8081	to	mo	2865	509	556	2590
	40	to	to	mo	479	4852	591	1828
	46	to	to	mo	4486	449	841	2070
	47	to	to	mo	to	3086	892	1652

[5] A detailed description of this benchmark and models for all benchmarks used in this paper can be found at http://www.informatik.uni-freiburg.de/~morbe/fsmt/.

in Tab. 1 our model checking algorithm is able to handle much more processes than Uppaal and RED. As this benchmark allows parallel behavior our model checking algorithm with parallelized interleaving performs best and it can solve up to 17 processes whereas Uppaal runs into a timeout already for 7 processes and RED exceeds the memory limit already for 6 processes.

Finally, our last benchmark 'FDDI' models a fiber-optic token ring local area network [8] where we check that the token is always at exactly one station. Uppaal is able to solve instances up to 39 stations, RED up to 9 stations. For the FDDI model parallelized interleaving performs only slightly better than pure interleaving. However, both variants are superior to Uppaal and RED. The version with pure interleaving arrives at 46 instances, the version with parallelized interleaving at 47 instances.

7 Conclusions

We presented a new formal model to represent real-time systems, the finite state machine with time, which is well-suited for fully symbolic verification algorithms. We presented a backward model checking algorithm to verify FSMTs. In order to verify TAs with our algorithm we presented two different methods to convert TAs into FSMTs. The resulting FSMT has either a pure interleaving behavior or a parallelized interleaving behavior, which can dramatically reduce the number of verification steps and brings an enormous gain of performance for certain benchmark classes. Even for other benchmarks like the well-known Fischer protocol which do not profit from parallelized interleaving, our model checker outperforms other state-of-the-art model checkers due to its fully symbolic data structure building upon the success of modern SMT solvers. Based on the same algorithmic framework we plan to develop a model checker supporting forward or combined forward / backward model checking as well.

References

1. Alur, R.: Timed automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., van Rossum, P., Schulz, S., Sebastiani, R.: MathSAT: Tight integration of SAT and mathematical decision procedures. J. Autom. Reasoning 35(1-3), 265–293 (2005)
5. Clarke, E., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
6. Damm, W., Dierks, H., Disch, S., Hagemann, W., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. Science of Computer Programming (to appear, 2011)

7. Damm, W., Disch, S., Hungar, H., Jacobs, S., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact state set representations in the verification of linear hybrid systems with large discrete state space. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 425–440. Springer, Heidelberg (2007)
8. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
9. 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)
10. Ehlers, R., Fass, D., Gerke, M., Peter, H.J.: Fully symbolic timed model checking using constraint matrix diagrams. In: RTSS, pp. 360–371 (2010)
11. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
12. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nordic J. of Computing 6, 271–298 (1999)
13. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
14. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: RTSS, pp. 14–24. IEEE Computer Society, Los Alamitos (1997)
15. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. 36(5), 450–462 (1993)
16. Mazurkiewicz, A.W.: Basic notions of trace theory. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 285–363. Springer, Heidelberg (1989)
17. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: A unifying representation for logic synthesis and verification. Tech. rep., EECS Dept. UC Berkeley (2005)
18. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 111–125. Springer, Heidelberg (1999)
19. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
20. Pigorsch, F., Scholl, C., Disch, S.: Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. In: FMCAD, pp. 89–96. IEEE Computer Society, Los Alamitos (2006)
21. Scholl, C., Disch, S., Pigorsch, F., Kupferschmid, S.: Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 383–397. Springer, Heidelberg (2009)
22. Seshia, S.A., Bryant, R.E.: Unbounded, fully symbolic model checking of timed automata using boolean methods. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 154–166. Springer, Heidelberg (2003)
23. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Int. J. Softw. Tools Technol. Transf. 6, 77–97 (2004)
24. Yorav, K., Grumberg, O.: Static analysis for state-space reductions preserving temporal logics. Formal Methods in System Design 25, 67–96 (2004)

Complete Formal Hardware Verification of Interfaces for a FlexRay-Like Bus

Christian Müller* and Wolfgang Paul

Saarland University, Computer Science Department,
66123 Saarbrücken, Germany
{cm,wjp}@cs.uni-saarland.de

Abstract. We report the first complete formal verification of a time-triggered bus interface at the gate and register level. We discuss hardware models for multiple clock domains and we review known results and proof techniques about the essential components of such bus interfaces: among others serial interfaces, clock synchronization and bus control. Combining such results into a single proof leads to an amazingly subtle theory about the realization of direct connections between units (as assumed in existing correctness proofs for components of interfaces) by properly controlled time-triggered buses. It also requires an induction arguing simultaneously about bit transmission across clock domains, clock synchronization and bus control.

1 Introduction

Clean formal definitions of time-triggered systems have been given in [11] and [9]. The simple time-triggered systems whose hardware realization is studied in this paper are inspired by the FlexRay standard [5]. They are constructed by coupling several processors by means of bus interfaces to a single real time bus as shown in Figure 1.

A processor together with its bus interface is called an electronic control unit (ECU). We denote by ECU_i the i'th ECU. The hardware of each ECU is clocked by its own oscillator. Oscillators of different ECU's have almost but not exactly identical clock periods τ_i (corresponds to ECU_i). As a consequence of this, timers on different ECU's tend to drift apart and need to be synchronized periodically.

On the bus interface of each ECU_i pairs of send and receive buffers $ECU_i.sb(j)$ and $ECU_i.rb(j)$ with $j \in \{0, 1\}$ serve both for the local communication between processors and their local bus interface and for communication between bus interfaces over the bus.

Time-triggered systems work in rounds r each consisting of a fixed even number ns of slots $s \in \{0, \ldots, ns-1\}$ according to a fixed schedule which is identical for each round. The work consists of:

* The author was supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 633–648, 2011.

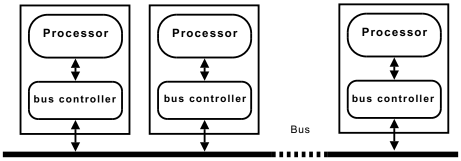


Fig. 1. ECU's Interconnected by a Communication Bus

- Local computation. During slot s each processor can access buffers $sb((s+1) \bmod 2)$ and $rb((s+1) \bmod 2)$ of its bus interface via memory mapped I/O for local computation. We will not consider local computation in this paper.
- Message broadcast. A fixed scheduling function $send$ specifies for each slot s the ECU $ECU_{send(s)}$ whose bus facing send buffer $sb(s \bmod 2)$ will broadcast to the bus facing receive buffers $rb(s \bmod 2)$. Observe that in the following slot $s+1 \bmod ns$ these receive buffers face the processors. This allows to overlap local computation with message broadcast. It is a special case of pipelining in the sense of [10].

We refer to the slot s of round r by (r,s). We denote the first local cycle of slot (r,s) on ECU_u by $\alpha_u(r,s)$ and the last cycle by $\omega_u(r,s)$. The values $\alpha_u(r,0)$ and $\omega_u(r,ns-1)$ at boundaries of rounds are determined in a non-trivial way by clock synchronization. The standard local length of a slot is cs cycles. How many cycles of a slot have locally passed is tracked by local counters. An 'interior' slot boundary $\alpha_u(r,s) = \omega_u(r,s-1)$ occurs, if the local counter is 0 mod cs.

We denote by ECU_u^j the state of ECU_u at hardware cycle j. The essential correctness statement of the bus interfaces whose formal proof can be found at [1] is then a very simple and clean statement about message transfer:

Theorem 1. $\forall\, u,r,s:\ ECU_{send(s)}^{\alpha_{send(s)}(r,s)}.sb(s \bmod 2) = ECU_u^{\omega_u(r,s)}.rb(s \bmod 2)$

A hardware realization of such a bus interface has obviously to deal with the following 5 problems:

1. Definition of a hardware model with multiple clock domains.
2. Bit transfer across clock domains. As setup and hold times for registers cannot be guaranteed across different clock domains, i.e., here over the bus, the sender puts each message bit for $n > 1$ cycles on the bus. The receiver will try to sample $m \leq n$ of these 'hardware' bits roughly in the middle of the n bits.[1] Situations where hardware bits are incorrectly sampled from the bus due to missed set up or hold times have to be dealt with. Note that in such situations the receiving registers do not necessarily sample bits in an unpredictable way. They can also become metastable.[2] Registers that are not clocked can stay metastable for very many cycles.

[1] The FlexRay standard requires $n = 8$, $m = 5$ and a majority vote on the sampled bits. This allows the correction of certain bit errors on the bus.

[2] They hang at the voltage between the thresholds recognized as 0 and 1.

3. Message transfer across clock domains. If a message $m[0 : \ell - 1]$ consisting of ℓ bytes is transmitted, then the sender inserts at the start of each byte $m[i]$ so called *sync edges* SE into the message. It also inserts a *message start sequence* MS and a *message end sequence* ME.[3] Thus, message m is transformed by the sender into:

   $$f(m) = MS \circ SE \circ m[0] \ldots SE \circ m[\ell - 1] \circ ME$$

   and then each bit of $f(m)$ is put n times on the bus. The purpose of the sync edges is to permit the receiver a *low-level clock synchronisation.* Because sync edges occur at regular intervals, the receiver knows when to expect them in the absence of clock drift. If a sync edge before byte $m[i]$ occurs 1 receiver cycle earlier/later than without clock drift, then the receiver knows that his clock has slipped/advanced against the sender clock and adjusts the cycles when it samples the m hardware bits belonging to the same n copies of a message bit accordingly.

   The final Theorem 1 we aim at is clearly a theorem about message transfer using such a mechanism. Hardware devices performing such a transfer are called *serial interfaces.* Correctness theorems about serial interfaces assume a single sender in one clock domain directly connected by a wire to a single receiver in a second clock domain.
4. Control of bus contention. During each slot s there must be a *transmission window* where for all ECU's the local timers indicate that they are in slot s. During this window $ECU_{send(s)}$ broadcasts a send buffer content and all other ECU's stay off the bus. This is achieved in the following way: (i) Let $D = Q + d + 1$ where Q is the maximum difference of local cycle counts between clock synchronizations, d is the pipeline depth of the transmission pipeline and the 1 accounts for effects of cross domain clocking (see Schedule Constraint 2 of [10]). Then the sender $ECU_{send(s)}$ begins transmission only D cycles after the local start of slot s and finishes transmission D cycles before the start of the next slot. (ii) All other ECU's stay off the bus during the complete slot s.

   Note that during the transmission window of slot s the bus acts for each ECU_i like a direct wire between $ECU_{send(s)}$ and ECU_i. Thus, correctness of serial interfaces can be applied during this window. Note however that bus contention control hinges on clock synchronization.
5. Clock synchronization. At the start of each round ECU's exchange synchronization messages in order to synchronize clocks according to some protocol. Non-trivial protocols based on Byzantine agreement are used if one wants to provide fault tolerance against the failure of some ECU's. Without fault tolerance a single sync message broadcast from a master ECU suffices at the start of each round. Note however that sync messages need to be transferred. The natural vehicle for this transfer is the bus. Thus, clock synchronization hinges on message transfer (at least for the synchronization messages),

[3] Along the lines of the FlexRay protocol for instance, we use falling sync edges 10 before each byte, 01 as start sequence and end sequence.

> message transfer on bus contention control and bus contention control on clock synchronization. Any theorem stating in isolated form the correctness of clock synchronization, bus contention control or message transfer alone must use hypotheses which break this cycle in one way or the other. If the theorem is to be used as part of an overall correctness proof, then one must be able to discharge these hypotheses in the induction step of a proof arguing simultaneously about clock synchronization, bus contention and message transfer. A paper and pencil proof of this nature can be found in [7].

The remainder of this paper is organized as follows. In Section 2 we discuss hardware models with multiple clock domains and develop some basic theory about the operation of buses spanning multiple clock domains. In Section 2.3 and 3 we review related work about the verification of serial interfaces and clock synchronization algorithms. In our presentation we highlight situations, where existing proofs assume direct connections between units that need to be realized by properly controlled buses in the overall proof. Section 4 outlines the overall correctness proof. It contains a quite involved induction hypothesis (Theorem 3) and a quite subtle argument that ‘nothing happens’ between the end of the message window of the last slot of round $r-1$ and the transmission of the synchronization message of round r. We also present some details about the mechanization of the proof. In Section 5 we conclude and discuss future work.

2 Models for Hardware with Multiple Clock Domains

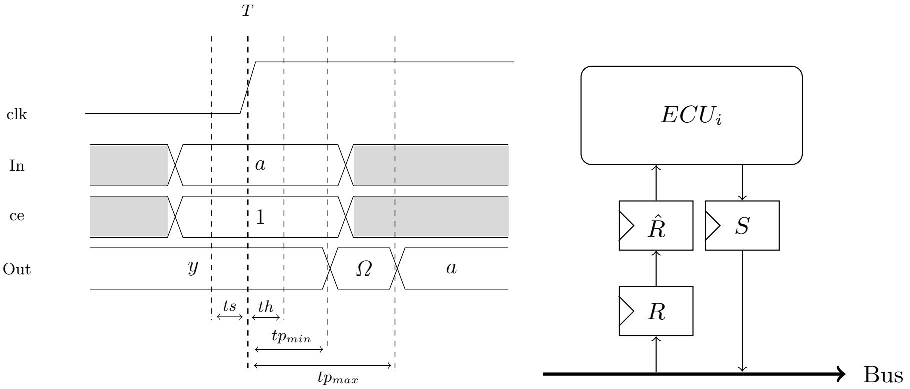
2.1 The Detailed Model

The obvious *detailed* model for this purpose is obtained by formalization of data sheets for hardware components as in [12]. The logic has three values 0, 1 and Ω. The latter value models any voltage between the thresholds recognized as 0 and 1. Signals are mappings from real time $\mathbb{R}$ to $\{0, 1, \Omega\}$; thus, $I(t)$ denotes the value of signal I at real valued time t. This allows to give algebraic definitions for detailed timing diagrams of circuits consisting of gates, registers and memories. The part of this model relevant for 1-bit registers with data input signal In, data output signal Out and clock enable signal ce has as parameters setup time ts, hold time th, as well as minimum and maximum propagation delays tp_{min} and tp_{max}. It is easy to specify that setup and hold times, e.g., for the data input are met for a clock edge at time T:

$$\exists\ a \in \{0, 1\} : \forall t \in [T - ts, T + th] : In(t) = a$$

If we also assume that setup and hold times are met for the clock enable signal ce at time T and $\forall t \in [T - ts, T + th] : ce(t) = 1$, then the output of the modeled register i) does not change before $T + tp_{min}$, ii) becomes undefined in $(T + tp_{min}, T + tp_{max})$ and iii) assumes the new value $In(T)$ from $T + tp_{max}$ until the next clock edge say at time $T + \tau_i$:

$$Out(t) = \begin{cases} Out(T) & : t \in (T, T + tp_{min}] \\ \Omega & : t \in (T + tp_{min}, T + tp_{max}) \\ In(T) & : t \in [T + tp_{max}, T + \tau_i] \end{cases}$$



(a) Update of an Analog Register

(b) Connection of an ECU to Bus

Fig. 2.

This behaviour is also illustrated in Figure 2(a).

This simple part of the definition which models completely regular clocking has an important consequence. Imagine we have $In(T) = Out(T)$, i.e., we clock the old value again into the register. Then in the digital abstraction the output is constant in two consecutive cycles. In the detailed hardware model however (and in reality) we have a possible spike $Out(t) = \Omega$ for $t \in (T + tp_{min}, T + tp_{max})$. If we want to guarantee, that the output really stays stable ($Out(t) = Out(T)$ for $t \in (T, T + \tau_i)$) , we need to disable clocking at edge T: $ce(t) = 0$ for $t \in [T - ts, T + ts]$. If clocking is properly enabled but setup or hold time is violated or the input is undefined, then after the maximum propagation delay the output stays 0, 1 or undefined (the latter case models metastabilty):

$$\exists a \in \{0, 1, \Omega\} : \forall t \in [T + tp_{max}, T + \tau_i] : Out(t) = a$$

Metastabilty is highly unlikely to occur. That a metastable value in one register R which is clocked into a second register $\hat{R}$ results in the metastability of $\hat{R}$ too is so unlikely that one models it as impossible.[4] The result is an unpredictable digital value:

$$\exists a \in \{0, 1\} : \forall t \in [T + tp_{max}, T + \tau_i] : \hat{R}(t) = a$$

Using minimal and maximal propagation delays of gates and access times of memories one can complete this model in a straightforward way. Correctness theorems about any hardware - with multiple clock domains or not - should in the end hold in this one detailed model.

We will denote by single capital letters, e.g., R_u digital registers of clock domain u; we denote by $In_{R,u}$ their analog input signals and by $Out_{R,u}$ their analog output signals in the detailed model.

[4] Using two subsequent registers to avoid metastability is a common technique in hardware design.

2.2 The Hybrid Model and the Bus

Fortunately, in our case we can restrict the use of the detailed model to the part of the hardware, where clock domain crossing occurs. This portion consists of the bus, the send register S and the receiver register R as depicted in Figure 2(b). The remaining hardware is partitioned into clock domains, one for each ECU. In each clock domain u (clock domain of ECU_u) we can abstract from the detailed model local digital hardware configurations h_u with, e.g., register components $h_u.R \in \{0,1\}$, hardware cycles $i \in \mathbb{N}$, configuration h_u^i during cycle i in the following way. We couple local cycle numbers i on ECU_u (hence, in clock domain u) with the real time $e_u(i)$ of the i'th local clock edge on ECU_u by setting for some constant c_u: $e_u(i) = c_u + i \cdot \tau_u$. This edge starts local cycle i on ECU_u. For $t \in [e_u(i) + tp_{max}, e_u(i) + \tau_u]$, i.e., after the propagation delay the output $Out_{R,u}(t)$ of register R on ECU_u is stable for the rest of local cycle i of ECU_u. We abstract this value to the digital register value:

$$h_u^i.R = dig(Out_{R,u}, e_u(i+1))$$

The function $dig(s,t)$ digitizes the output of signal s at real valued time t: $dig(s,t) :=$ **if** $s(t) \neq \Omega$ **then** $s(t)$ **else** x, for some $x \in \{0,1\}$.

If we now define a hypothesis 'Correct detailed timing analysis' stating that for all local cycles setup and hold times for register inputs are met[5], then one can show that the hardware configurations h_u^i we just abstracted are exactly the configurations one would get by applying the transition function δ_H of ordinary digital hardware models

Theorem 2. *Assume correct detailed timing analysis. Then:* $\forall i : h_u^{i+1} = \delta_H(h_u^i)$.

This justifies the use of ordinary digital logic within clock domains and restricts the use of the detailed model to the boundaries between the domains, in our case the bus.

2.3 Transmitting Bits across the Bus

We use two lemmas from Schmaltz [12] dealing with bit transmission across clock domains in the detailed register model. Schmaltz has formally verified the bit transmission correctness between two directly linked digital registers with different clocks.

To introduce these lemmas formally we need one more definition. Let R and S be two registers from two different clock domains u and v, respectively; they are interpreted using the detailed register model. Assume the clock enable signal of the send register S is active in cycle c. By the definition of the detailed register model, the output of S will change right after $e_v(c) + tp_{min}$. We want to know at which minimal cycle the receive register R will 'notice' the change of its input signal changing its own content. We call such cycle the *next affected cycle* and define it as: $cy_{u,v}(c) = min\{\ x \mid e_v(c) + tp_{min} < e_u(x) - ts\}$. Thus, the receiver's

[5] Summing propagation delays along appropriate paths.

next affected cycle of sender's cycle c is the first cycle, whose set up time lies right after the output change of S.

Lemma 1 (Low-Level Bit Transmission Correctness). *Assume that the clock enable signal of register R is always active, and the analog input of R is the analog output of S during n sender cycles starting from some cycle c:*

$$\forall i \in [e_v(c) + tp_{min}, e_v(c+n)] : In_{R,u}(t) = S_{Out,v}(t)$$

If S samples a stable input value $\mathcal{V}$ at cycle c and then clocking of S is disabled for n subsequent cycles, then R will sample the new output of S for at least $m+\sigma$ cycles starting from the next affected cycle $\xi = cy_{u,v}(c)$:

$$\forall i \in [0, m-1] : R^{\xi+i+\sigma} = \mathcal{V}, \textit{ for } \sigma \in \{0,1\}$$

Note that $m \leq n$ due to possible clock drifts of the register clocks. The additional cycle σ (delay) may arise if set up or hold times are violated.[6] The direct linking of registers should be considered as an abstraction of the bus, when there is no bus contention. Moreover, Schmaltz extended Lemma 1 to a high-level message transmission correctness for a concrete send and receive units obeying the message protocol mentioned in Section 1.

Lemma 2 (High-Level Message Transmission Correctness). *Let su be a send unit of ECU_v; let ru be a receive unit of ECU_u. Let R be the input register of ru and S the output register of su. Let L be the length of a transmitted message in bytes. Let ts be the length of a message transmission in local cycles. Let $bc_v(i)$ be the cycle, when the send unit su starts the broadcast of the i'th byte. Let $cy_{u,v}(c) = \xi$ for a receiver cycle ξ. Assume that:*

1. *su starts a message broadcast in cycle c;*
2. *ru is idle in cycle ξ.*
3. *send and receive registers are directly connected for ts cycles:* $\forall t \in [e_v(c) + tp_{min}, e_v(c+ts-1)] : In_{R,u}(t) = S_{Out,v}(t)$

Then, every byte of the transmitted message is transmitted correctly from the send buffer $su.sb$ to an intermediate byte buffer $ru.rByte$:

$$\forall\ i \leq L\ :\ su^{bc_v(i)}.sb[i] = ru^{cy_{u,v}(bc_v(i))+80+\gamma}.rByte,\ \ \textit{with } \gamma \in \{-1,0,1\}$$

On the left side, $su^{bc_v(i)}.sb[i]$ denotes the content of the i'th byte in send buffer $su.sb$ (slot is not fixed, so sb can be instantiated by any of two buffers) of send unit su at cycle $bc_v(i)$. On the right side, we have the content of a buffer $ru.rByte$ of the receive unit ru at the next cycle $cy_{u,v}(bc_v(i))$ affected by the transmission of the i'th byte, plus $80+\gamma$. That is, due to clock drift, the entire transmitted byte will be successfully sampled either in 79 cycles after the next affected cycle, or in 80/81 cycles. This Lemma was proven by model checking of control automata of su and ru, and by interactive combining of these results using Lemma 1.

[6] This happens, if $e_v(c) + tp_{min} < e_u(\xi) - ts < e_v(c) + tp_{max}$.

2.4 Modeling the Bus

We model the outputs of send registers S_v facing the bus as open collector output. In this case the bus computes the logical 'and' of the signals put on the bus:

$$\forall t : bus(t) = \bigwedge_v Out_{S,v}(t)$$

The intended use of the bus is to simulate for all slots s the direct connection of a sender register $S_{send(s)}$ to the receiver registers R_u on the bus during the transmission window $W(s) \subset [e_u(\alpha(r,s)), e_u(\omega(r,s))]$[7] of slot s, because this permits to apply results about clock domain crossing bit transmission for pairs of directly connected senders and receivers. Obviously, this is achieved by keeping $Out_{S,u}(t) = 1$ for all t in the window and all $u \neq send(s)$.

Lemma 3 (Absence of Bus Contention). *If for all $u \neq send(s)$ and for all $t \in W(s)$ we have $Out_{S,u}(t) = 1$, then $\forall t \in W(s) : bus(t) = Out_{S,send(s)}$*

While this lemma is trivial, showing the hypothesis requires not only to show that the S_u have constant value 1 in the digital model for $u \neq send(s)$. One has also to establish the absence of spikes by showing (in the digital model) that clocking of these registers is disabled in the transmission window. This depends on correct schedule execution, which, in turn, depends on the absence of bus contention at the previous synchronizations, etc.

2.5 An Alternative Model

A model more suited for model checkers than the hybrid model above has been proposed and used in [4]. This clock model is based on so-called *timeout automata*. The progress of global time is enforced cooperatively by sender and receiver clocks. This model deals with metastability but does not model the set up and hold times explicitly. The clock modeling is partially protocol-dependent. Since the sender's clock progress depends on the receiver's clock progress, it is not fully clear how to extend this model to several receiver clocks. Unfortunately, no discussion was provided about the gap between the given stack of abstractions and modeling of actual hardware (counterpart to our Theorem 2).

3 Related Work and Results Used

In [9], Pike has presented several results. One part of his work related to this paper was a corrected and significantly extended version of Rushby's formalism [11], which allows to verify time-triggered systems by abstracting them to synchronous protocols. He specifies timing constraints, which a schedule of a time-triggered protocol has to fulfill, to form in its system run so-called 'cuts'. These cuts are

[7] $W(s)$ is the time segment in global time where all ECU's are locally in slot s; it should be long enough to transmit a message.

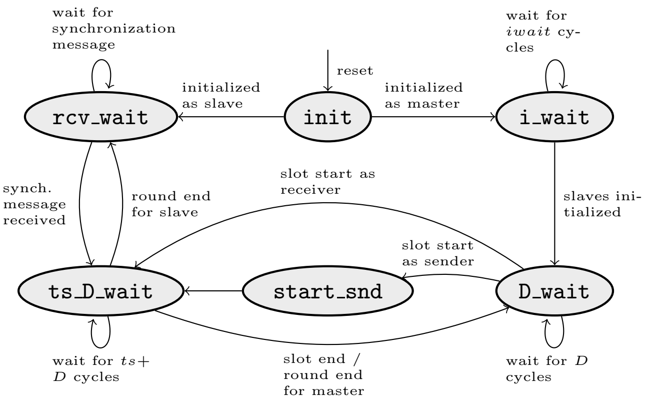


Fig. 3. Scheduler Automata

time segments, where the state of the time-triggered system can be related to a state of its synchronous counterpart. The correctness of the time-triggered schedule would then follow from the correctness of the synchronous protocol. He also applies this approach to verify the schedule timings of two protocols of the SPIDER bus architecture: Clock Synchronization and Distributed Diagnosis.[8] However, it remains unclear how exactly the timing properties of hardware implementations of these protocols were derived and mapped to the formal model. Moreover, the proposed technique assumes even in the proof of timing properties of a Clock Synchronization protocol already synchronized clocks initially. That is, to extend the proof to *all round correctness* one needs to use the proposed proof as an induction step. However, to show the initial clock synchronization the startup correctness might become necessary.

In [8], Pfeifer formally analyzes two fault-tolerant algorithms implemented in the Time-Triggered Protocol TTP/C: Group Membership and Clock Synchronization. Both algorithms were analyzed, using a hand-derived mathematical specification of the TTP/C protocol. Although, the algorithms have a circular dependency on each other, Pfeifer has analyzed them in isolated form and on different levels of abstraction. He also introduces an abstract principle how to combine both proofs resolving the dependency.

Böhm has used Lemma 1 to verify the correct schedule execution for directly wired bus controllers [3]: one master and n slaves. The master ECU always sends a message in slot 0. The first bit of this message serves as a synchronization signal. The scheduler used in the connected ECU's is the main control block of the bus controller. It computes the internal state of the controller and counts passed slots. The schematic representation of the scheduler state automaton is depicted in Figure 3. After the initial reset signal, all ECU's are in state `init`. After all configuration registers are written, the operating system running on top of the processor initiates a special signal, which forces all ECU's to change

[8] Note that no correctness of the synchronous version of these protocols was provided.

either to state `i_wait` if the ECU was configured as a master ECU, or to state `rcv_wait` otherwise. The master waits in state `i_wait` for a certain amount of cycles *iwait*. Assuming that all ECU's are started roughly at the same time, it should be guaranteed that after the master's initialization and additional *iwait* cycles all slaves are initialized and are in state `rcv_wait`. The size of *iwait* can be estimated by industrial worst case execution time analyzers [7]. In state `rcv_wait` slaves are waiting for a synchronization message after the initialization and after the end of each round. In state `ts_D_wait` each ECU is waiting for $ts + D$ cycles, which is a sum of transmission length (ts) and offset D before the end of a slot. In this state an ECU is sending or receiving, depending on its role in the current slot. Then a slave ECU either switches back to `rcv_wait` if the maximal number of slots in a round is reached, or to state `D_wait`. In `D_wait` all ECU's are waiting for D hardware cycles. This is a waiting time (cf. Section 1) before each new slot, which is necessary due to clock drift to guarantee that all ECU's have finished the previous slot. Afterwards, each ECU enters a new slot. If it enters a slot as a sender, the scheduler sends a broadcast signal to the send unit going through state `start_snd`. Otherwise, it switches directly to state `ts_D_wait`. The master ECU acts as a sender in the first slot of each round. It goes through state `start_snd` sending a new synchronization message to the bus. Afterwards, it switches as a receiver between states `D_wait` and `ts_D_wait` during $ns - 1$ slots.

We say synchronization happens i) on the master at the state transition from `D_wait` to `start_send`; at this point the local timer of master has value D. ii) on a slave at the state transition from `rcv_wait` to `ts_D_wait` which is triggered by the synchronization message sent by the master. On this transition the slave sets its local timer to D.

W.l.o.g., we assign to the master ECU the index 0: ECU_0. We can now define the missing slot boundaries. On the master all slots have length cs: $\alpha_0(r, s) = \omega_0(r, s - 1) = \alpha_0(r, s - 1) + cs$. The master executes its schedule statically. On slaves u slot $(r - 1, ns - 1)$ ends and slot $(r, 0)$ starts when the synchronization message is received: $\alpha_u(r, 0) = \omega_u(r - 1, ns - 1) = cy_{u,0}(\alpha_0(r, 0) + D)$.

We define that the system is ready for synchronization at cycle c of the master, when:

- in local cycle c the master is in state `D_wait`, its local timer is $D - 1$ and its serial interface is idle and, hence, ready for data transmission;
- for $u \neq 0$ in local cycle $cy_{u,0}(c)$, the receiving ECU_u is in state `rcv_wait`, its serial interface is idle, and, thus, ready for data sampling and the internal pipeline of the receive units has sampled Ones from the bus in the last e local cycles for some fixed constant $e < d$ (thus no spurious sync messages are already in the pipe).

By assuming that synchronization message is transmitted via *dedicated* direct connections between the master and the slaves, Böhm does not need to assume bus contention control for the transmission of the synchronization signals and, thus, he is in a position to do an induction over all slots s of all rounds r. However in his main result - that we use - he assumes that at the start of some round r ECU's are ready for synchronization and then he argues only about the slots of this round r. He does not argue that at the end of the round the system is ready for synchronization again. We use the following result from [3].

Lemma 4. *Assume: 1. the system is ready for synchronization and the master is in cycle c; 2. the output of master's register S is the input of all* R_u *registers during 8 master cycles:* $\forall\ t \in [e_v(c), e_v(c+7)] : In_{R,u}(t) = Out_{S,v}(t)$.

Then, the master initiates the synchronization message, and all directly connected slaves and the master itself will start the execution of the fixed schedule consisting of ns slots. Each slot on each ECU starts before and ends after the message transmission window defined as transmission time of the sender. After cs local cycles of slot $ns-1$ *the master is in state* `D_wait` *and the slaves are in state* `rcv_wait`*. Formally:*

$\forall$ *slot s of some round r* : *u is receiver in slot s* $\rightarrow$

1. *synchronization happens at round r –* *slave u receives the synchronization message at cycle* $cy_{u,0}(c) + \sigma + d$
2. $W(s) \subset [e_u(\alpha_u(r,s)), e_u(\omega_u(r,s))]$ *where* $W(s) = [e_{send(s)}(\alpha_{send(s)}(r,s) + D), e_{send(s)}(\omega_{send(s)}(r,s) - D)]$
3. ECU_u *with* $u \neq send(s)$ *is in state* `rcv_wait` *in cycle* $\alpha_u(r, ns-1) + cs$*, and* $ECU_{send(s)}$ *is in state* `D_wait` *in cycle* $\alpha_{send(s)}(r, ns-1) + cs$.

Remember that d is the depth of transmission pipeline, i.e., the delay of the path of the signal from the scheduler of the sender to the scheduler of the receiver. That the synchronization message is received by the slaves was proven by applying Lemma 1. After synchronization is achieved the proof of the lemma boils down to a statement about the values of the local counters and can be derived with a very high degree of automation. The timing properties of the schedule hardware were proven interactively by multiplying local times with the corresponding τ.

However, we can only apply Lemmas 1, 2 and 4 in our generalized semantic, consisting of n ECU's interconnected by a bus, if we argue about the absence of bus contention during all signal transmissions on the bus. We have to justify the bus abstraction, which will be assumed using the directly wired send and receive registers. This abstraction will be discharged in the next section.

4 Overall Proof

The correctness of message transmission between all ECU's of our bus system can be roughly split in two parts: (1) the bus value correctness during a message transmission (absence of bus contention), and (2) the high-level message transmission. The first part depends only on the correctness of the scheduler unit and low-level bit transmission between ECU's over the bus. The high-level message transmission relies on the bus value correctness, low-level bit transmission correctness and the correctness of send and receive units executing the message protocol. Thus, formally verifying the first property, we provide a bus architecture, which can be instantiated with an arbitrary message protocol. In

Section 4.2 we present a proof sketch of the first property. In Section 4.3 we outline the correctness proof sketch of the high-level message transmission, consolidating the results of Section 4.2 with results achieved in previous verification efforts explained in Subsection 2.3 and Section 3.

4.1 Global Assumptions

To verify the automotive bus controller, the following axioms were assumed and are implicitly used in every theorem below:

1. Clock periods deviate by at most 0.38%;[9]
2. Slot 0 starts on the master immediately after leaving the `i_wait` state;
3. When master enters slot 0, all slaves are initialized and are waiting for the synchronization;
4. ECU's are configured in such way, that in every slot exactly one ECU is a sender.

Note that we omit assumptions of technical nature here, which are not relevant for the communication protocol itself, like assumption that the processor doesn't change it's configuration registers during a system run, etc.

4.2 Proof Sketch of the Bus Value Correctness

While previous lemmas have assumed in slot s a direct connection between a slave u and the master ($In_{R,u}(t) = Out_{S,send(s)}$), the generalized model extended by the bus, models the analog input of every R_u register as output of the bus:

$$\forall \text{ ECU } u : \forall t : In_{R,u}(t) = bus(t) \tag{1}$$

To use previous results for an overall message transmission correctness, we have to show that $bus(t)$ can be substituted by $Out_{S,send(s)}$ during the transmission time $W(s)$ of every slot s.

Theorem 3 (Bus Value Correctness). *For all rounds r holds:*

1. *the system is ready for synchronization at cycle $\alpha_0(r, 0) + D$ of the master;*
2. *during the entire message transmission time in every slot s of round r the bus value is equal to the analog output of the send register of the sender:*

$$\forall t \in W(s) : bus(t) = Out_{S,send(s)}(t)$$

Before we sketch the proof of Theorem 3, we list three helper lemmas, which do not depend on the cyclic argumentation necessary for the proof of Theorem 3.

Lemma 5. *After the initialization the system is ready for synchronization and the master begins a message broadcast in cycle $\alpha_u(0, 0) + D$.*

[9] This constant is derived from the concrete parameters of our implementation. Note that FlexRay standard assumes maximal deviation of size 0.15%.

This lemma is proven by construction of the scheduler and the control logic of configuration registers. Using mostly a model checker, we have formally verified that after a reset each ECU reaches state `rcv_wait` or `i_wait` according to its status (master/slave). Then we *assume* (cf. Global Assumption 3) that if the master leaves the `i_wait` state, all slaves are in the state `rcv_wait`.

Lemma 6. *Let the send unit of the serial interface of ECU_u be idle during time interval I. Then during I the analog output of the send register of ECU_u is 1:*

$$\forall t \in I : Out_{S,u}(t) = 1$$

This lemma was proven by showing that the send register S is only clocked if the send unit is not idle. We define the time interval of local slot (r, s) of ECU_u as: $slot_u(r, s) = [e_u(\alpha_u(r, s), \omega_u(r, s)]$ and show the next lemma.

Lemma 7. *Let $u \neq send(s)$ and let r be any round. Then the send unit of ECU_u is idle during $slot_u(r, s)$.*

A simple induction about the scheduler automaton shows, that at the start of a slot the send unit is idle. The send unit is only started in state `start_send`. But if ECU_u is not the sender, then the automaton is during the slot only in states $\neq$ `start_snd`. We are finally ready to argue by induction simultaneously about synchronization and bus contention control.

Proof (of Theorem 3). By induction on rounds r. *Base case,* $r = 0$. We first show Claim 1 for $r = 0$. After startup the send units of all slaves are idle and the system is ready for synchronization by Lemma 5.

Next we show Claim 2 for round $r = 0$. We first discharge hypothesis 2 of Lemma 4 for $c = \alpha_0(r, 0) + D$ stating that the bus behaves during the transmission of the sync message like a direct connection between the master and the slaves. A slave ready for synchronization is in state `rcv_wait` and the send unit of its serial interface is idle. In 9 cycles it cannot reach state `start_send`. Using Lemma 6 with $I = [e_0(c), e_0(c + 7)]$ we conclude that ECU_u transmits Ones on the bus during I. By Lemma 3 we get $In_{R,u}(t) = Out_{S,0}(t)$ for all $t \in I$. We are ready to apply Lemma 4. By Claim 1 of Lemma 4 we can conclude, that synchronization takes place and thus slot 0 of round 0 starts on all ECUs. Thus from now on we can use lemmas that argue about slots in round 0.

Consider any slot s of round 0 and any $u \neq send(s)$. By Lemma 7 ECU_u stays off the bus during $slot_u(0, s)$. By Claim 2 of Lemma 4 we have $W(s) \subset slot_u(0, s)$. By Lemma 3 the bus acts like a direct connection between $Out_{S,send(s)}$ and $In_{R,u}$ during $W(s)$ and we have shown Claim 2 of Theorem 3 for round $r = 0$.

Induction step, $r - 1 \rightarrow r$. We assume the theorem holds for round $r - 1$ and show that it holds for round r. We start with Claim 1 for round r. By Claim 1 of the induction hypothesis the system is ready for synchronization at cycle $\alpha_0(r - 1, 0) + D$ of the master. By Claims 1 and 2 of the induction hypothesis for slot 0 of round $r - 1$ Hypothesis 1 and 2 of Lemma 4 hold. By Claim 3 of Lemma 4 each slave u is in state `rcv_wait` in local cycle $\alpha_u(r - 1, ns - 1) + cs$ and the master is in state `D_wait` in local cycle $\alpha_0(r - 1, ns - 1) + cs$.

We show a subtle technical lemma about the time when slaves are waiting for the synchronization signal of the master at the end of rounds. Note that for slaves u the start cycles $\alpha_u(r, 0)$ are only defined after we have shown that synchronization has occurred, thus we cannot use them yet. Fortunately for the master $\alpha_0(r, s)$ is always defined.

Lemma 8. *Let $u \neq 0$ be the index of any slave. Let*

$$t \in [e_u(\alpha_u(r-1, ns-1) + cs, e_u(cy_{u,0}(\alpha_0(r, 0) + D))$$

be a time when slave u waits for the sync signal from the master. Then the slave observes an idle bus at time t: $bus(t) = 1$.

This lemma is proven by contradiction. We fix the first time point t between the last slot of a round $e_u(\alpha_u(r-1, ns-1) + cs)$ and the start of the next round $e_u(cy_{u,0}(\alpha_0(r, 0) + D))$, where the bus value is not equal '1'. By bus construction there is at least one ECU producing bus activity at t. Since t lies *before* the start of a new round, the disturbing ECU cannot be master, since there was no synchronization message yet. If the disturbing ECU is a slave, we show that it should be in state `rcv_wait`, because there was no bus activity before t, hence no synchronization message was received by any slave. Since no one is sending, we use Lemma 6 and the bus construction to show that its value is the idle value '1' which contradicts to our assumption.

Lemma 8 implies that all slaves stay in state `rcv_wait` until local cycle $cy_{u,0}(\alpha_0(r, 0) + D))$. By construction of the schedule automaton the master stays in state `D_wait` until local cycle $\alpha_0(r, 0) + D$. By construction of the serial interfaces the interface of an ECU in state `rcv_wait` is idle. This applies here for the slaves. By the construction of the hardware, non-idle send units of serial interfaces occur only on ECUs in state `ts_D_wait`. Thus, at cycle $\alpha_0(r, 0) + D$ of the master the system is ready for synchronization, Claim 1 of Theorem 3 is shown for round r and we can argue about the slots in round r also for the slaves. The proof of Claim 2 for round r now proceeds along the lines of the proof for round 0. It is somewhat simpler because Hypothesis 2 of Lemma 4 follows directly from Lemma 7. □

4.3 Message Transmission Correctness

After we have proven the correctness of Theorem 3, we can use it to ensure the absence of bus contention during all transmissions of all rounds. We discharge Hypothesis 1 and 2 of Lemma 2 for cycle $c = \alpha_{send(s)}(r, s) + D$ (similarly to the argumentation in Theorem 3) by showing that if a sender ECU_i starts a message transmission at a cycle (r, s), then the receive unit of every receiver will be idle at the corresponding next affected cycle. Hypothesis 3 of Lemma 2 follows from Theorem 3. We also show that all bytes written to the intermediate byte buffer *rByte* of the receive unit will be correctly written to the receive buffer.

Lemma 9. *All bytes written to the intermediate buffer $ru_u.rByte$ in slot s will eventually be written to the receive buffer $ru_u.rb$, and will stay there unchanged until the end of a slot:*

$$\forall\ i \leq L\ : ru_u^{cy_{u,v}(bc_v(i))+80+\gamma}.rByte = ru_u^{\omega_u(r,s)}.rb(s \bmod 2)[i]$$

We also show stability of the active send buffer content. This is possible due to double buffer construction and restricting of the processor access to the buffer $sb(s \bmod 2)$ during a message transmission.

Lemma 10. *The content of the send buffer remains stable during slot s:*

$$\forall\ i \leq L\ : su_v^{\alpha(r,s)}.sb(s \bmod 2)[i] = su_v^{bc_v(i)}.sb(s \bmod 2)[i]$$

Finally, using Lemmas 2, 9 and 10 we can show the main message transmission correctness during all slots of all rounds stated in Theorem 1.

5 Conclusion and Future Work

We presented recent results on verification of an automotive time-triggered real-time bus system, which was inspired by the FlexRay standard. The network consists of electronic control units interconnected by a bus. We have verified the startup routine, the clock synchronization, the correct TDMA scheduling and a low-level bit transmission. Finally, consolidating these results, we have shown the correctness of recurrent message broadcasts during a system run. In contrast to most of previous research, we not only show the algorithmic correctness of protocols, but we also provide justified models to link these protocols to a concrete gate-level hardware.

The verification was carried out on several abstraction layers using a combination of an interactive theorem prover Isabelle/HOL supported by the model checking technique [14]. The ECU implementation has been automatically translated to Verilog [13] directly from formal Isabelle/HOL hardware descriptions and has been synthesized with the Xilinx ISE software. The implementation which consists of several FPGA boards interconnected into a prototype of a distributed hardware network was reported in [6].

During the presented verification work, several bug were discovered in the hardware implementation, as well as in previous verifications. For example, Lemma 1 has originally assumed a permanent connection between the send and receive registers. This abstraction cannot be justified in a bus architecture with multiple senders. Moreover, Lemmas 4 and 2 have assumed that the second receive register $\hat{R}$ contains value '1' in cycle $cy_{u,v}(c) + 1$. This cannot be shown, in the case when the timing constraints of the first receive register R are not met and the metastable value sampled in cycle $cy_{u,v}(c)$ flips to 0 in the next cycle. Furthermore, in the original proof of Lemma 2 used an assumption like: $cy_{u,v}(k + 80 \cdot i) = cy_{u,v}(k) + 80 \cdot i$ which is in general not true.[10]

As part of the future work we see an extension of the presented controller by fault-tolerance features [2]. For example, supplying each ECU with a bus guardians should be easy, by taking the same scheduler with independent clock and slightly modified timing parameters. Moreover, due to redundancy in the

[10] The proof was fixed by Schmaltz on our demand.

message protocol, the bus controller is already fault-tolerant against signal jitter. The computing and verifying of its maximal fault assumptions remains as future work.

References

1. Formal proof of a FlexRay-like bus interface, `http://www-wjp.cs.uni-sb.de/~cm/abc.html`
2. Alkassar, E., Boehm, P., Knapp, S.: Correctness of a fault-tolerant real-time scheduler algorithm and its hardware implementation. In: MEMOCODE 2008, pp. 175–186. IEEE Computer Society Press, Los Alamitos (2008)
3. Böhm, P.: Formal Verification of a Clock Synchronization Method in a Distributed Automotive System. Master's thesis, Saarland University (2007)
4. Brown, G., Pike, L.: Automated verification and refinement for physical-layer protocols. Formal Aspects of Computing, 1–24 (2010)
5. FlexRay Consortium. FlexRay – the communication system for advanced automotive control applications (2006),`http://www.flexray.com/`
6. Endres, E.: FlexRay ähnliche Kommunikation zwischen FPGA-Boards. Master's thesis, Wissenschaftliche Arbeit, Saarland University, Saarbrücken (2009)
7. Knapp, S., Paul, W.: Realistic worst-case execution time analysis in the context of pervasive system verification. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 53–81. Springer, Heidelberg (2007)
8. Pfeifer, H.: Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture. PhD thesis, Universität Ulm, Germany (2003)
9. Pike, L.: Formal Verification of Time-Triggered Systems. PhD thesis, Indiana University (2006), `http://www.cs.indiana.edu/~lepike/phd.html`
10. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: FMCAD 2007, pp. 231–238. IEEE, Los Alamitos (2007)
11. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering 25(5), 651–660 (1999)
12. Schmaltz, J.: A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In: FMCAD 2007 (2007)
13. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In: LFM 2008, NASA (2008)
14. Tverdyshev, S., Alkassar, E.: Efficient bit-level model reductions for automated hardware verification. In: TIME 2008, pp. 164–172. IEEE, Los Alamitos (2008)

Synthia: Verification and Synthesis for Timed Automata*

Hans-Jörg Peter[1], Rüdiger Ehlers[1], and Robert Mattmüller[2]

[1] Reactive Systems Group,
Saarland University, Germany
[2] Foundations of Artificial Intelligence Group,
Freiburg University, Germany

Abstract. We present SYNTHIA, a new tool for the verification and synthesis of open real-time systems modeled as timed automata. The key novelty of SYNTHIA is the underlying abstraction refinement approach [5] that combines the efficient symbolic treatment of timing information by difference bound matrices (DBMs) with the usage of binary decision diagrams (BDDs) for the discrete parts of the system description. Our experiments show that the analysis of both closed and open systems greatly benefits from identifying large relevant and irrelevant system parts on coarse abstractions early in the solution process. SYNTHIA is licensed under the GNU GPL and available from our website.

1 Introduction

A crucial factor for the acceptance of automatic system analysis techniques is how well they scale when the models become more complex. A powerful concept aiming at increasing the scalability is *automatic abstraction refinement*, where, beginning with a coarse abstraction of the original system, only those parts are incrementally refined that are necessary for proving a certain property.

In this paper, we report on SYNTHIA, a new tool that makes abstraction refinement available for the analysis of open real-time systems modeled in a syntactically enhanced variant of the popular timed automata formalism by Alur and Dill [1]. An open system distinguishes between external and internal nondeterminism, of which one type represents an unpredictable environment and the other type represents a partial implementation. SYNTHIA checks if the system is realizable (i.e., whether there exists a full implementation) such that, independent of the environment, some safety requirements are satisfied. SYNTHIA can also certify the (un)realizability by generating a controller that represents safe (violating) implementations (environments). The verification of closed systems, where the implementation is deterministic and complete, is a special case and can equally well be handled by SYNTHIA.

* This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 649–655, 2011.

2 Underlying Approach

Computational Model. We interpret a given open real-time system as a two-player game played on a timed automaton, in which player Adam controls the choices of the partial implementation and player Eve controls the nondeterminism of the unpredictable environment. A specified safety property defines the dual winning conditions for the players: while Eve wins whenever the property is eventually violated, Adam wins when the property is always satisfied.

As fundamental computation model, we consider timed game automata [6], an alternating extension of timed automata [1], where the controllability of each transition is assigned to a particular player. We always assume our timed game automata to be strongly nonzeno. Synthia also supports additional syntactic features such as parallel composition or arithmetic expressions over bounded integer variables. In the following, we use the term location to refer to any pure discrete state information, including integer variable valuations. We consider an asymmetric semantics, where Eve is prioritized in situations in which both players can play an active move, which leads to determinacy of all games.

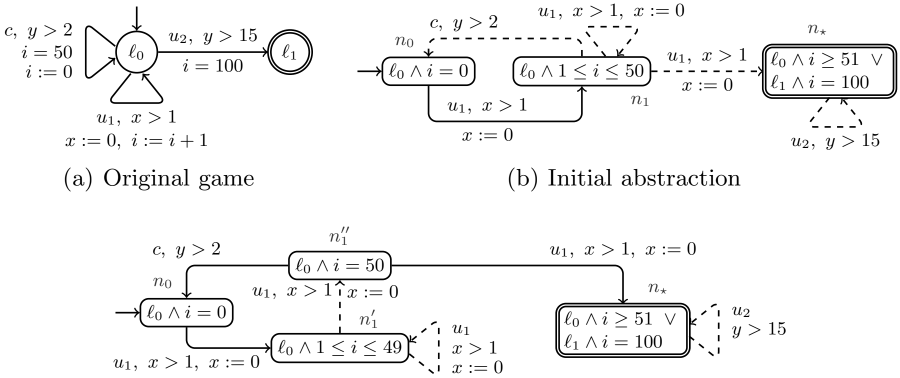
Synthesizing a safety controller corresponds to computing a winning strategy for Adam. For obtaining such a strategy, we compute the set of states from which the players can enforce their respective winning objectives. For Eve, this is done by taking the so-called *attractor* of the set of bad (i.e., requirement-violating) states. Any strategy that enforces staying in the complement of this set is then winning for Adam. In case of a closed system, there are only Eve moves, in which case game solving naturally boils down to checking reachability.

Abstraction Refinement. Synthia's main analysis algorithm is an efficient implementation of the approach introduced in [5]. We collapse sets of concrete locations of the original timed game automaton into single abstract locations. In these so-obtained *syntactic* abstractions, we distinguish between *may* and *must* transitions: between two abstract locations n and n', for a player p,

- there is a *may* transition for p, if there is some concrete location subsumed by n having a p-transition to some concrete location subsumed by n';
- there is a *must* transition for p, if all concrete locations subsumed by n have a p-transition to some concrete location subsumed by n'.

We obtain an abstract game by letting Eve play on her must transitions and Adam play on his may transitions. Clearly, in our abstractions Eve is weakened and Adam is strengthened, compared to the original game. Thus, computing the attractor of the bad states in the abstract game yields an under-approximation of the attractor in the original game.

The abstraction refinement procedure begins with the trivial abstraction that comprises (at most) four abstract locations: (1) one subsuming all initial locations; (2) one subsuming all bad locations; (3) one subsuming all safe locations (from which no bad location is reachable); (4) one subsuming all other locations which are not in (1)-(3). Then, in each iteration of the following refinement loop,



(a) Original game

(b) Initial abstraction

(c) Final abstraction

Fig. 1. Example timed game and its abstractions with transitions surely available (solid lines) and potentially available (dashed lines)

we compute the attractor of the bad states in the respective current abstraction;[1] if it contains a concrete initial state, we can stop the iteration as we can surely deduce a concrete winning strategy for Eve. On the other hand, if the abstract game is safe, we refine the abstraction such that the attractor is extended in the refined game. If no such refinement is found, we can deduce that Eve loses and Adam wins.

Optimizations. The symbolic treatment of the locations allows us to apply computationally cheap but effective optimizations based on pure discrete analyses of the control structure of the original game. For instance, in the initial abstraction, Synthia only considers those locations which are both forward reachable from the initial locations and backward reachable from the bad locations. Also, before constructing the initial abstraction, Synthia enlarges the set of bad locations by those locations from which Eve can force the system into a bad location. Furthermore, the selection of possible refinements can be restricted to (an over-approximation of) the forward reachable states.

Example. Consider the timed game depicted in Fig. 1(a) comprising the clocks x and y, as well as the integer variable i ranging from 0 to 100. Adam controls the transition c and Eve controls the transitions u_1 and u_2. Eve wins when ℓ_1 is eventually visited. It is easy to see that Adam has a winning strategy by playing c whenever it is available (i.e., when $i = 50$). The initial abstraction (Fig. 1(b)) is based on all locations that are reachable in a purely discrete manner. Here, we note that the abstract bad location $n_\star$ subsumes the concrete location $\ell_1 \wedge i = 100$ as well as $\ell_0 \wedge i \geq 51$, which is the result of the bad-location enlargement. As this abstract game is safe, the refinement heuristic determines to split n_1 such that u_1 becomes available for Eve, which leads to an enlargement of the attractor.

[1] In fact, Synthia incrementally *updates* an attractor under-approximation.

The resulting game (Fig. 1(c)) is still safe as the attractor can only be updated for n_1'' to $y - 1 \leq x \vee 1 < x$. Now, any further refinement would not enlarge the attractor (e.g., making the u_1-transition between n_1' and n_1'' available to Eve is useless since then, n_1'' is only entered when $x = 0$ and $y > 2$, in which case only c is available but not u_1).

3 The Tool Synthia

Availability and Usage. Synthia is licensed under the GNU General Public License and available at

http://react.cs.uni-saarland.de/tools/synthia.

Due to the lack of space, it is impossible to explain all of Synthia's features in this paper. Instead, in this section we present some standard usage scenarios. A detailed description of the command line parameters, the file format, as well as a step-by-step tutorial can be found on the tool's website.

A specification is given in form of an XML file and essentially comprises a plant model with requirements. Assuming that `robot.xml` represents a specification, then the simplest way to execute Synthia is the following:

```
$ synthia  robot.xml
```

This lets Synthia check whether there exists a controller influencing the plant such that regardless of the uncontrollable behavior, the requirements are always satisfied. Specifications can have parameters with default values which can be overridden using the command line argument `-D`:

```
-Dprocesses:2  -Ddelay:23  -Dtimeout:42
```

Requirements are given as conjunctions of assumptions and guarantees. A system does not satisfy its requirements if (1) there is a trace that eventually violates the guarantees, and (2) each prefix of that trace satisfies the assumptions. For example, the following lines of an XML specification file encode a requirement describing a location invariant and a bounded reachability guarantee:

```
<assume>in(loc) imply (x <= {delay})</assume>
<guarantee>(not in(goal)) imply (y <= {timeout})</guarantee>
```

To let Synthia synthesize a controller in addition to checking realizability, the following command line parameters can be used:

```
$ synthia  robot.xml  --synth-cont controller.xml
$ synthia  robot.xml  --synth-cont-plant controlled_plant.xml
```

The former call generates a model (in the Synthia file format) that only comprises the controller, while the latter generates a model where the synthesized controller is embedded into the original plant.

Implementation Details. Synthia is written in C^{++} and uses, besides some standard Boost libraries, the CUDD BDD library [7] for representing transition relations and sets of locations, as well as the Uppaal DBM library [4] for representing and manipulating clock zones.

After parsing the specification, as explained in [5], Synthia constructs a BDD-based representation of the control structure and sets up the initial abstraction. As an extension to [5], Synthia also considers abstractions of the guards of abstract transitions. Hence, a refinement either consists of splitting an abstract location or of making a guard of an abstract transition precise.

The actual game solving procedure that runs on the abstract games and updates the attractor under-approximation is implemented as a pure backward solving algorithm. The selection of refinements is carried out in form of a forward zone-based reachability analysis: only those abstract locations and transitions are considered which appear in this analysis.

As a further optimization, additionally to under-approximating the attractor of the requirement-violating states, Synthia also computes an under-approximation of the attractor of the safe states. This is done by a concurrent game solving procedure, in which Adam is weakened and Eve is strengthened.

4 Experimental Results

Table 1 shows a comparison of Synthia 1.2.0 with the game solver Uppaal-Tiga 0.16 [2]. We note that the latter subsumes the basic model checking engine of Uppaal 4.1.4 [3], which is automatically applied in case of closed-system properties.

From left to right, the columns show the benchmark instance, whether it is a safe/realizable instance, the number of refinement steps, the number of abstract locations in the final abstraction, Synthia's running time and memory consumption, the parameters for which Uppaal-Tiga showed the best results, Uppaal-Tiga's number of explored states, running time and memory consumption. Running times are given in seconds, memory consumption in MB, the time limit was set to 4 hours, and the memory limit was set to 4 GB. All experiments were conducted on a 2.6 GHz AMD Opteron computer running Ubuntu 10.04. The model files used for the benchmarks can be downloaded along with the tool.

Fischer and *CSMA/CD* are standard benchmarks from the closed-system verification domain. The instances are parametrized in the number of components. The benchmark *Robot* is to decide whether a robot has a strategy to quickly traverse a square-shaped grid with a wall in the middle that has two gates through which the robot can pass. Up to a certain amount of time, nondeterministically, one of the gates can close upon which the robot has to react. The instances are parametrized in the edge length of the grid. The benchmark *Tank* asks for the existence of a controller that controls the inflow to a water tank such that a desired fill level is reached within a given amount of time. It is parametrized by the precision in which the continuous flow is digitized.

Except for unsafe Fischer, where a depth-first-search on the precise system quickly detects the error, Synthia's abstraction refinement approach always

Table 1. Performance comparison of Synthia with Uppaal-Tiga

	Safe /	Synthia				Uppaal-Tiga			
Benchmark	Realizable	Steps	Abs	Time	Mem	Params	States	Time	Mem
Fischer 60	No	116	119	2957	1321	-o1	520	5	39
Fischer 65	No	126	129	2247	935	-o1	6	1	26
Fischer 70	No	TIMEOUT				-o1	256	4	41
Fischer 13	Yes	169	172	4	86	-C -S2	29122758	1324	1127
Fischer 14	Yes	196	199	5	88	-C -S2	93835680	4661	3501
Fischer 15	Yes	225	228	6	90		MEMOUT		
Fischer 30	Yes	900	903	355	228		MEMOUT		
Fischer 40	Yes	1600	1603	2628	453		MEMOUT		
Fischer 51	Yes	2601	2604	14262	1405		MEMOUT		
Fischer 52	Yes	TIMEOUT					MEMOUT		
CSMA/CD 15	Yes	4	5	6	118	-C	11681796	442	2639
CSMA/CD 16	Yes	4	5	8	158	-S2	27901956	1302	3072
CSMA/CD 17	Yes	4	5	16	252		MEMOUT		
CSMA/CD 21	Yes	4	5	1474	3804		MEMOUT		
CSMA/CD 22	Yes	MEMOUT					MEMOUT		
Robot 300	No	356	360	10	87		43147361	486	120
Robot 500	No	596	600	58	98		199601281	2322	233
Robot 1000	No	1196	1200	182	145		TIMEOUT		
Robot 2000	No	2396	2400	1153	365		TIMEOUT		
Robot 3000	No	TIMEOUT					MEMOUT		
Robot 300	Yes	376	380	18	97		92655832	1067	165
Robot 500	Yes	626	630	132	129		429037638	4965	356
Robot 1000	Yes	1251	1255	401	256		TIMEOUT		
Robot 2000	Yes	2501	2505	8011	1061		MEMOUT		
Robot 3000	Yes	TIMEOUT					MEMOUT		
Tank 100	No	28	19	3	91		85852	183	416
Tank 300	No	28	19	14	125	-F1	512034	1532	1309
Tank 500	No	28	19	32	176	-F1	1993710	13991	3882
Tank 1000	No	28	19	50	280		MEMOUT		
Tank 5000	No	28	19	1724	996		MEMOUT		
Tank 10000	No	28	19	8077	1923		MEMOUT		
Tank 10	Yes	54	35	1	80		482227	55	280
Tank 20	Yes	42	29	2	82		1978965	449	1668
Tank 30	Yes	45	31	2	84		MEMOUT		
Tank 100	Yes	55	36	8	113		TIMEOUT		
Tank 500	Yes	44	31	51	205		MEMOUT		
Tank 1000	Yes	53	35	107	359		MEMOUT		
Tank 5000	Yes	45	32	1865	1354		MEMOUT		
Tank 10000	Yes	53	35	8349	3808		MEMOUT		

clearly outperforms Uppaal-Tiga. Interestingly, while Uppaal-Tiga suffers from an exponential blow-up for increasing instance sizes, the final abstractions found by Synthia are several orders of magnitude smaller than the original systems: quadratic in the number of components for Fischer, linear in the edge length of the grid for Robot, or even of constant size for CSMA/CD and Tank.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theo. Comp. Sci. 126(2) (1994)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
3. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing uppaal over 15 years. Softw. Pract. Exper. 41(2), 133–142 (2011)

4. David, A.: UPPAAL DBM Library release 2.0.8 (2011)
5. Ehlers, R., Mattmüller, R., Peter, H.-J.: Combining symbolic representations for solving timed games. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 107–121. Springer, Heidelberg (2010)
6. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, Springer, Heidelberg (1995)
7. Somenzi, F.: CUDD: CU Decision Diagram package release 2.4.2 (2009)

FixBag[*]: A Fixpoint Calculator for Quantified Bag Constraints

Tuan-Hung Pham[1], Minh-Thai Trinh[2],
Anh-Hoang Truong[2], and Wei-Ngan Chin[3]

[1] University of Minnesota, Twin Cities,
hung@cs.umn.edu
[2] Vietnam National University, Hanoi,
{thaitm_52,hoangta}@vnu.edu.vn
[3] National University of Singapore,
chinwn@comp.nus.edu.sg

Abstract. Abstract interpretation techniques have played a major role in advancing the state-of-the-art in program analysis. Traditionally, stand-alone tools for these techniques have been developed for the numerical domains which may be sufficient for lower levels of program correctness. To analyze a wider range of programs, we have developed a tool to compute symbolic fixpoints for *quantified bag domain*. This domain is useful for programs that deal with collections of values. Our tool is able to derive both *loop invariants* and *method pre/post conditions* via fixpoint analysis of recursive bag constraints. To support better precision, we have allowed disjunctive formulae to be inferred, where appropriate. As a stand-alone tool, we have tested it on a range of small but challenging examples with acceptable precision and performance.

1 Introduction

Abstract interpretation [2] is a technique to infer program's properties. It requires the least fixed point of a monotone function over an abstract domain of the program's semantics to be computed. Let $\langle L, \prec \rangle$ be a complete lattice and $\bot$ be its least element. A function $f : L \rightarrow L$ is monotone if $f(u) \prec f(v)$ when $u \prec v$ with all u, v in L. One classical method to find the fixed point is Kleene iteration, which computes the ascending chain $f_0 = \bot$, $f_{i+1} = f(f_i)$ with $i > 0$ until we find i^* satisfying $f_{i^*+1} = f_{i^*}$. Widening operator [3] is used to guarantee that the ascending chain is finite.

Traditionally, stand-alone abstract interpretation (AI) tools have been developed for the numerical domains (such as Octagon [8] and Polyhedra [4]). Little attention has been paid to building such tools for richer pure domains, such as bags, maps and sequences. Stand-alone AI tools focus primarily on the logics of the abstract domains and use sound mechanisms for approximating recursion via fixed point computation. They have been widely adopted by program analysis systems that are customized to analyze properties from programs (e.g. [1,9]).

[*] The tool is available at http://loris-7.ddns.comp.nus.edu.sg/~project/fixbag/.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 656–662, 2011.

Some recent works [10,11] have proposed methods to automatically infer disjunctive numerical invariants for added precision. However, numerical invariants are often insufficient for higher levels of program correctness. For example, many programs are constructed to compute a collection of values whose correctness cannot be captured using only numerical properties. Instead, we require a *quantified bag domain* to provide more precise program analyzers for such programs. To the best of our knowledge, there is no current published tool that can discover quantified bag invariants. We present FIXBAG, a stand-alone fixpoint calculator for quantified bag constraints. The tool has the following characteristics:

- FIXBAG can infer disjunctive fixed points of formulae with bag constraints. The maximum number of disjuncts is provided by end-users. Supported bag's operators are union ($\cup$), intersection ($\cap$), and subset ($S_1 \subseteq S_2$), where S_1 and S_2 denote bags.
- FIXBAG can find fixed points with quantified constraints. Specifically, the system supports the universal quantifier of the form $\forall x \in S : P(x)$ and the existential quantifier of the form $\exists x \in S : P(x)$, where x, S, and $P(x)$ are a variable, a bag, and a predicate concerning x, respectively.
- FIXBAG partially supports arithmetic constraints on size properties over bags. Currently, FIXBAG allows the following types of size properties to be inferred: $|S_1| = m \times |S_2|$, $m \leq |S|$, or $|S| \leq m$, where m denotes an integer.

Section 2 gives an overview via examples. Section 3 introduces the algorithm to infer fixed points, as used in our tool. Section 4 summarizes our experimental results. Section 5 concludes with a short discussion on related works.

2 Motivating Examples

Our tool is able to compute disjunctive fixpoints for constraint abstractions over the bag domain. To illustrate its capability, we shall analyze two list functions that are commonly used in functional languages by initially showing their respective constraint abstractions prior to fixpoint analysis. We stress that our tool is language-independent, as its inputs are logical formulae (with bag and size constraints) that may be applied to similar abstractions for programs from other programming languages too.

Our first example is a well-known `filter` function, which selects elements from a list that satisfy a predicate, `p`, as given below in Caml syntax:

```
filter p xs = match xs with
    | [] → []
    | x : xs → if (p x) then x : (filter p xs) else (filter p xs)
```

The corresponding constraint abstraction, named *filterB*, for this function has three parameters: p to denote the predicate `p` of `filter`, S to capture the elements of input list `xs`, and R to capture the elements of the method's output.

$$\begin{array}{rcl} \mathit{filterB}(p,S,R) & \equiv & S{=}\{\} \wedge R{=}\{\} \ \vee \ \exists x, S_1, R_1 \cdot S{=}\{x\}{\cup}S_1 \wedge \\ & & \quad (p(x) \wedge R{=}\{x\}{\cup}R_1 \wedge \mathit{filterB}(p,S_1,R_1) \ \vee \\ & & \quad \neg p(x) \wedge R{=}R_1 \wedge \mathit{filterB}(p,S_1,R_1)) \end{array}$$

When this constraint abstraction is passed to our tool, we could infer the following fairly precise closed-form formula using universally quantified bags:

$$\mathit{filterB}(p, S, R) \;\equiv\; (\forall x \in R\!: p(x)) \;\wedge\; (\forall x \in S - R\!: \neg p(x)) \;\wedge\; R \subseteq S$$

Our next example is a membership function that determines if an element exists within an input list or not. Its Caml code is given below:

```
mem v xs  =  match xs with | [] → false
                           | x : xs → if x = v then true else (mem v xs)
```

The corresponding constraint abstraction has three parameters, as shown:

$$\mathit{memB}(v, S, r) \equiv S{=}\{\} \wedge \neg r \;\vee\; \exists x, S_1 \cdot S{=}\{x\} \cup S_1 \wedge (x{=}v \wedge r \;\vee\; x{\neq}v \wedge \mathit{memB}(v, S_1, r))$$

A precise closed-form formula for this function requires both disjunction and quantified bag formulae, as shown below, which our tool can derive:

$$\mathit{memB}(v, S, r) \;\equiv\; (\forall x \in S : x{\neq}v) \wedge \neg r \;\vee\; (\exists x \in S : x{=}v) \wedge r$$

These two examples show that a good treatment of quantified formulae and disjunctions is needed to support more precise analysis.

3 Algorithm

This section presents the algorithm behind FixBag. One of the necessary operators used in fixpoint analysis is hulling, which is well-developed for numerical domains. However, to the best of our knowledge, there is no work that calculates the hulling operations on bag/set domain to date. To realize this, we propose a rule-based approach that uses propagation and simplification rules to attain the hulling of formulae for the bag domain. Similar to CHR [5], our propagation rules $\mathcal{R}_p$ add new implied constraints to a formula while simplification rules $\mathcal{R}_s$ reduce the size of a formula by removing redundant constraints from it. Although these rules [1], by themselves, simply preserve the logical equivalences of the original formula, they can help create intermediate results that would play important roles in other operations. Let $\phi_1 \Cap \phi_2$ be $\bigwedge_{i=1}^{k} d_i$ where $d_1, ..., d_k$ are all shared conjuncts of two conjunctive formulae ϕ_1 and ϕ_2. Definition 1 shows how to find the hulling result of ϕ_1 and ϕ_2.

Definition 1 (Hulling). *Given two conjunctive formulae ϕ_1 and ϕ_2, we divide each of them into two parts: the first one (Γ) contains all conjuncts of the form $m_1 \leq |S| \leq m_2$ and the other one (Δ) has the remaining conjuncts. Thus, the two original formulae are represented as $\phi_1 = \Gamma_1 \wedge \Delta_1$ and $\phi_2 = \Gamma_2 \wedge \Delta_2$. The hulling operation is defined as $\phi_1 \boxtimes \phi_2 = \Gamma_1 \boxtimes \Gamma_2 \wedge \Delta_1 \boxtimes \Delta_2$ where*

- $\Gamma_1 \boxtimes \Gamma_2 =$

$$\underbrace{(\bigwedge m_{i1} \leq |S_i| \leq m_{i2})}_{(1)} \wedge \underbrace{(\bigwedge m_{j1} \leq |S_j| \leq m_{j2})}_{(2)} \wedge \underbrace{(\bigwedge min(m_{i'1}, m_{j'1}) \leq |S_{i'j'}| \leq max(m_{i'2}, m_{j'2}))}_{(3)}$$

where (1) contains all $(m_{i1} \leq |S_i| \leq m_{i2}) \in \Gamma_1$ that S_i is not in in Γ_2, (2) consists of all $(m_{j1} \leq |S_j| \leq m_{j2}) \in \Gamma_2$ that S_j is not in Γ_1, and (3) has all $S_{i'j'}$ that has $(m_{i'1} \leq |S_{i'j'}| \leq m_{i'2}) \in \Gamma_1$ and $(m_{j'1} \leq |S_{i'j'}| \leq m_{j'2}) \in \Gamma_2$.

[1] $\mathcal{R}_p$ and $\mathcal{R}_s$ are available at the tool's website.

- $\Delta_1 \boxtimes \Delta_2 = \underset{\mathcal{R}_s}{\texttt{simplify}}(\Delta_1 \widehat{\boxtimes} \Delta_2)$ *where* $\Delta_1 \widehat{\boxtimes} \Delta_2 = \underset{\mathcal{R}_p}{\texttt{propagate}}\Delta_1 \Cap \underset{\mathcal{R}_p}{\texttt{propagate}}\Delta_2$.

The latter is the harder operation. Intuitively, we find two closures (corresponding to Δ_1 and Δ_2) of conjunctive constraints that are closed under the set of propagation rules $\mathcal{R}_p$, compute the intersection of the closures, and simplify the result by the collection of simplification rules $\mathcal{R}_s$. We also define a version of the hulling operation without simplification as $\phi_1 \widehat{\boxtimes} \phi_2 = \Gamma_1 \boxtimes \Gamma_2 \wedge \Delta_1 \widehat{\boxtimes} \Delta_2$. It is needed when we want to check whether a particular conjunct contributes to the hulling result or not. This notion is used to measure the closeness of two conjunctive formulae in Definition 2. Given two conjunctive formula ϕ_1 and ϕ_2, $\phi_1 \oslash \phi_2$ quantifies their closeness as a rational number[2] in the range of 0..1. The larger the number is, the closer they are to each other. We denote $\lceil \phi \rceil$ ($\lfloor \phi \rfloor$) the number of conjuncts (disjuncts) in a conjunctive (disjunctive) formula ϕ.

Definition 2 (Affinity Measure). *Given two conjunctive formulae ϕ_1 and ϕ_2, the affinity measure $\oslash$ is defined as follows:* $\phi_1 \oslash \phi_2 = \frac{\lceil (\phi_1 \wedge \phi_2) \Cap (\phi_1 \widehat{\boxtimes} \phi_2) \rceil}{\lceil \phi_1 \wedge \phi_2 \rceil}$

While our hulling only works with conjunctive formulae, selective hulling [10] can deal with disjunctive ones. Our tool can increase the precision of the output fixpoints by allowing up to μ disjuncts to be present during the analysis process. The main idea is to repeatedly call hulling with a closest pair of disjuncts taken from the latest formula until there are at most μ disjuncts remaining. The function `normalize` converts a given formula into the disjunctive normal form.

Definition 3 (Selective Hulling). *Given a disjunctive formula $\phi = \bigvee_{i=1}^{k} d_i$ and a maximum number of disjuncts μ, the selective hulling operation $\oplus_\mu$ is defined as $\oplus_\mu \phi = \texttt{normalize}(\widehat{\oplus}_\mu \phi)$ where $\widehat{\oplus}_\mu \phi$ is recursively defined as follows:*

$$\widehat{\oplus}_\mu \phi = \begin{cases} \phi \ \textit{if}\ k \leq \mu \\ \widehat{\oplus}_\mu \big((d_{i'} \boxtimes d_{i''}) \vee \bigvee_{i \in \{1..k\} \setminus \{i', i''\}} d_i \big) \ \textit{if}\ k > \mu \\ \qquad \textit{where}\ (i', i'') = \texttt{argmax}(d_{j'} \oslash d_{j''}) \ \textit{with}\ 1 \leq j', j'' \leq k \ \textit{and}\ j' \neq j'' \end{cases}$$

Widening [3] is used to ensure that the fixpoint analysis terminates. To maintain disjunctions in the widening process, selective widening [10] is required.

Definition 4 (Selective Widening). *Given two disjunctive formulae $\phi_1 = \bigvee_{i=1}^{k} d_i$ and $\phi_2 = \bigvee_{j=1}^{k} e_j$, the selective widening operation $\phi_1 \nabla \phi_2$ is defined as $\phi_1 \nabla \phi_2 = \texttt{normalize}(\phi_1 \widehat{\nabla} \phi_2)$ where $\phi_1 \widehat{\nabla} \phi_2$ is recursively defined as follows:*

$$\phi_1 \widehat{\nabla} \phi_2 = \begin{cases} \phi_1 \boxtimes \phi_2 \ \textit{if}\ k = 1 \\ (d_{i'} \boxtimes e_{j'}) \vee \big(\bigvee_{i \in \{1..k\} \setminus \{i'\}} d_i \widehat{\nabla} \bigvee_{j \in \{1..k\} \setminus \{j'\}} e_j \big) \ \textit{if}\ k > 1 \\ \qquad \textit{where}\ (i', j') = \texttt{argmax}(d_{i''} \oslash e_{j''}) \ \textit{with}\ 1 \leq i'', j'' \leq k \end{cases}$$

[2] For simplicity, we may also use an integer-based affinity measure [10] defined as $\phi_1 \oslash \phi_2 = \texttt{round}(\frac{\lceil (\phi_1 \wedge \phi_2) \Cap (\phi_1 \widehat{\boxtimes} \phi_2) \rceil}{\lceil \phi_1 \wedge \phi_2 \rceil} \times 98) + 1$.

The algorithm to find the fixpoint for formulae with bag constraints shares the same ideas with the one used in [10] to infer disjunctive postconditions. The main challenges here are how we support affinity measure, selective widening, and selective hulling to work with the bag domain. Given a recursive function f, we start with $f_0 = \bot$ (which is `false` in the bag domain) and then compute an ascending chain $f_1, f_2, ...,$ until we find two equal consecutive elements $f_{i^*+1} = f_{i^*}$. If the current value in the chain is f_i, the next item f_{i+1} will be calculated as $\mathit{bottomup}(f, f_i, \mu)$, which is either $f(f_i)$ or its sound approximation with the help of selective hulling and widening operations. The first kind of operations helps us achieve a disjunctive fixpoint while the second one ensures the chain will converge. Algorithm 1 shows how we can obtain f_{i+1} from f, f_i, and μ.

Algorithm 1. Calculating $f_{i+1} = \mathit{bottomup}(f, f_i, \mu)$

```
$f_{f_i} \leftarrow \texttt{normalize}\big(f(f_i)\big)$
if $\lfloor f_{f_i} \rfloor < \mu$ then
    return $f_{f_i}$
else if $f_i = \bot$ or $\lfloor \oplus_\mu f_{f_i} \rfloor < \mu$ or $\lfloor f_i \rfloor < \mu$ then
    return $\oplus_\mu f_{f_i}$
else
    return $f_i \nabla (\oplus_\mu f_{f_i})$
```

First, we normalize the result of $f(f_i)$ to achieve f_{f_i}, which is a candidate for f_{i+1}. If $\lfloor f_{f_i} \rfloor < \mu$, we return f_{f_i}, otherwise we need to find a suitable approximation of f_{f_i} that has no more than μ disjuncts. If $f_i = \bot$ or $\lfloor \oplus_\mu f_{f_i} \rfloor < \mu$, the approximation is $\oplus_\mu f_{f_i}$. If the two previous conditions fail, we have $f_i \neq \bot$ and $\lfloor \oplus_\mu f_{f_i} \rfloor = \mu$. At this point, the approximation will depend on $\lfloor f_i \rfloor$ because selective widening only works with two formulae that have the same number of disjuncts. Therefore, if $\lfloor f_i \rfloor < \mu$, we still return $\oplus_\mu f_{f_i}$. Finally, when $\lfloor \oplus_\mu f_{f_i} \rfloor = \lfloor f_i \rfloor = \mu$ holds, the best approximation is $f_i \nabla (\oplus_\mu f_{f_i})$, which not only maintains up to μ disjuncts but also contributes to the convergence of the chain.

Soundness. The soundness of our tool is critically dependent on the soundness of the closure process used in the hulling operation. This process is guaranteed to be sound, as long as each simplification and propagation rule (from $\mathcal{R}_s$ and $\mathcal{R}_p$, respectively) preserves logical implication. This property helps to ensure that selective widening and selective hulling will always generate a sound over-approximation of f_{f_i}.

4 Experimental Results

We tested our tool on a set of methods from the OCaml's List library. To support higher-order functions, FixBag handles uninterpreted functions and multi-argument predicates. It takes the abstraction of each function and a number μ, denoting the maximal number of disjuncts allowed during the fixpoint inference, as its arguments and returns two closed-form (fixpoint) formulae, one to denote the function's post-condition and the other to derive its pre-condition.

We also measured the running-time of the analysis process. These results and the inference rules are detailed at our tool's website.

We have encountered several examples where disjunctive analysis can obtain more precise fixpoints than conjunctive analysis. Conjunctive analysis can be simulated using $\mu = 1$. In general, each analyzed function has an upper bound of μ; increasing μ over this bound does not help achieve more precise fixpoints and does not affect the analysis time.

5 Related Works and Conclusion

Libraries to support abstract interpretation are popular for program analysis systems, but they are focused mostly on the numeric domains [8,1]. In the non-numeric domains, abstract interpretation tools have been developed for shape analysis [7,12] and for constraint-based analysis [6]. The former is for discovering data shapes of heap-manipulating programs rather than their pure properties; and are thus focused on program codes rather than logical formulae. The latter is meant as a scalable tool for flow-based constraints, rather than for analyzing collections. Both systems do not automatically handle quantified formulae and have restricted use of disjunctive formulae.

We have built a stand-alone abstract interpretation tool for quantified bag domain. Our use of simplification and propagation techniques is inspired from CHR [5], while the use of affinity-based hulling and widening is targeted at more precise disjunctive fixpoints. Our experiments have shown that our tool is capable of efficiently analyzing the collection properties for non-trivial functions.

Acknowledgement. We thank the reviewers for insightful feedback, and gratefully acknowledge the support of MoE research grant R-252-000-411-112.

References

1. 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. SCP 72, 3–21 (2008)
2. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977, pp. 238–252 (1977)
3. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: PLILP, pp. 269–295 (1992)
4. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL 1978, pp. 84–96 (1978)
5. Frühwirth, T.W.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming 37(1-3), 95–138 (1998)
6. Kodumal, J., Aiken, A.: Banshee: A Scalable Constraint-Based Analysis Toolkit. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 218–234. Springer, Heidelberg (2005)
7. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)

8. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
9. 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)
10. 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)
11. 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)
12. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)

Analyzing Unsynthesizable Specifications for High-Level Robot Behavior Using LTLMoP*

Vasumathi Raman and Hadas Kress-Gazit

Cornell University,
Ithaca, NY, USA 14853
vraman@cs.cornell.edu, hadaskg@cornell.edu

Abstract. Recent work in robotics has applied formal verification tools to automatically generate correct-by-construction controllers for autonomous robots. However, when it is not possible to create such a controller, these approaches do not provide the user with feedback on the source of failure, making the experience of debugging a specification somewhat ad hoc and unstructured, and a source of frustration for the user. This paper describes an extension to the LTLMoP toolkit for robot mission planning that encloses the control-generation process in a layer of automated reasoning to identify the cause of failure, and targets the users attention to flawed portions of the specification.

Keywords: synthesis, LTL, GR(1), unrealizability, unsatisfiability, robot control.

1 Introduction

High-level robot control is a topic of current research in robotics. The goal is to automatically generate controllers for autonomous robots to achieve desired high-level behavior involving a non-trivial sequence of actions, such as, “collect all my socks from the apartment floor and put them in the laundry bag”. Recent work in robotics [11, 15] has applied efficient synthesis techniques [12] to automatically generate provably correct, closed loop, low-level robot controllers that satisfy high-level behaviors specified in temporal logic. A discrete abstraction of the workspace is used, and the robot goals and environment assumptions are described using Linear Temporal Logic, which can express a rich set of infinite behaviors. The generated continuous robot controllers are provably correct in that the closed loop system they form is guaranteed, by construction, to satisfy the desired specification when the robot operates in an environment that satisfies the modeled assumptions.

However, such synthesis-based approaches present the user with no feedback when synthesis is impossible, i.e., when there exists an environment in which the robot fails to achieve the desired behavior – we call such a specification *unsynthesizable*. An unsynthesizable specification is either *unsatisfiable*, in which case the robot cannot achieve the desired behavior in any environment, or *unrealizable*, in which case there exist environments that can thwart the robot. When the specification is unsynthesizable, the

* This work was supported by NSF CAREER CNS-0953365 and ARO MURI (SUBTLE) W911NF-07-1-0216.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 663–668, 2011.

above approaches fail to produce the desired behavior, but do not provide the user with a source of failure, or suggest changes that would allow synthesis of a controller. In addition, even when synthesis is possible, the generated automaton may fail to produce the desired behavior for reasons that involve unsatisfiability or unrealizability of the environment assumptions. This can make the experience of debugging a specification somewhat ad hoc and unstructured, and a source of frustration. This paper describes a procedure for enclosing the control-generation process in a layer of automated reasoning that focuses the cause of failure and targets the users attention to relevant portions of the specification. We present a method of narrowing down the source of unsynthesizability in specifications that can be represented as GR(1) formulas in LTL.

The debugging procedure described in this paper is implemented within Linear Temporal Logic MissiOn Planning (LTLMoP)[8], an open source, modular, Python-based toolkit that allows users to input structured English specifications describing high-level robot behavior, and automatically generates and implements the relevant hybrid controllers using the approach of [11]; the synthesized controllers can be embedded within a simulator or used with physical robots. The most recent version of LTLMoP can be downloaded online[1]. There has been considerable previous work on analyzing unsatisfiable and unrealizable LTL formulas [2, 4–6, 14]. Our work is most closely related to that of [9], who present an interactive tool, RATSY [3] for demonstrating specification unrealizability. The debugging procedure we implement for LTLMoP differs from RATSY in that rather than allowing the user to explore counterexamples, it provides explicit information about specification components in the context of the robot control problem. We highlight flawed portions of the user-defined specification (either desired system behavior or environment assumptions), and identify cases of unexpected behavior that specifically affect this domain, such as trivial solutions.

2 Technical Overview

We first review some preliminaries relating to the application of formal methods to high-level robot control, and outline LTLMoP's controller-synthesis procedure (depicted in Fig. 1). We consider a robot functioning in a continuous environment. The robot reacts to the environment as perceived through its sensor inputs, and chooses from a set of actions including moving between adjacent locations. The tasks themselves include infinite behaviors such as visiting locations or performing actions infinitely often.

Applying formal methods techniques such as model checking and synthesis to continuous settings in robotics requires a discrete abstraction of problems to enable description with a formal language. As mentioned earlier, the underlying formal language used to define high-level specifications in this work is Linear Temporal Logic (LTL) (cf. [7]). LTLMoP includes a parser that automatically translates English sentences from a defined grammar [10] into LTL formulas. This allows users to define desired robot behaviors (including reactive behaviors) and specify assumptions on the environment's behavior using an intuitive descriptive language rather than the underlying formalism.

As shown in Fig. 1, LTLMoP takes as input a user-defined specification, a map of the environment and a description of the robot sensors and actuators. The specification

[1] `http://ltlmop.github.com`

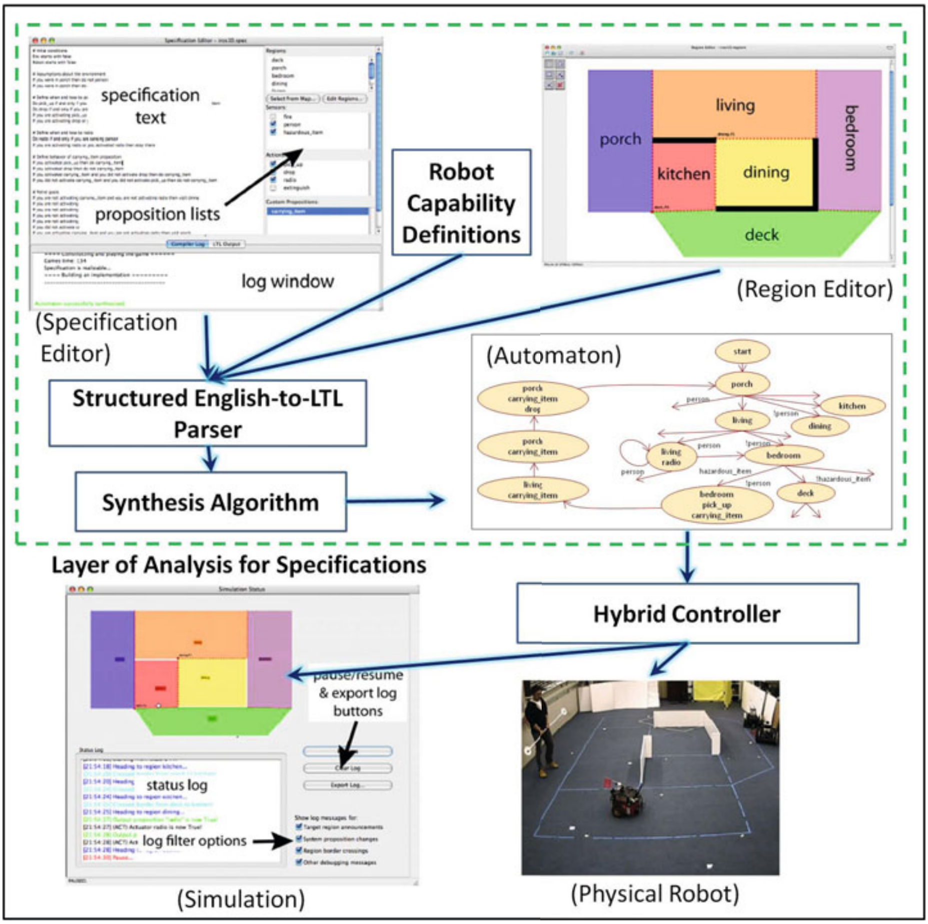


Fig. 1. Overview of LTLMoP architecture

is parsed into a formula of the form $\varphi = (\varphi_e \Rightarrow \varphi_s)$, where φ_e includes assumptions about the sensor propositions, and thus about the behavior of the environment, and φ_s represents the desired robot behavior. φ is in the subclass of LTL described in [12], and the efficient algorithm introduced therein is used to synthesize an automaton that implements the input specification with the described robot in the given environment. The synthesis algorithm is implemented in the JTLV framework [13], with formulas for the system and environment initial conditions, transitions and goals represented as Binary Decision Diagrams (BDDs).

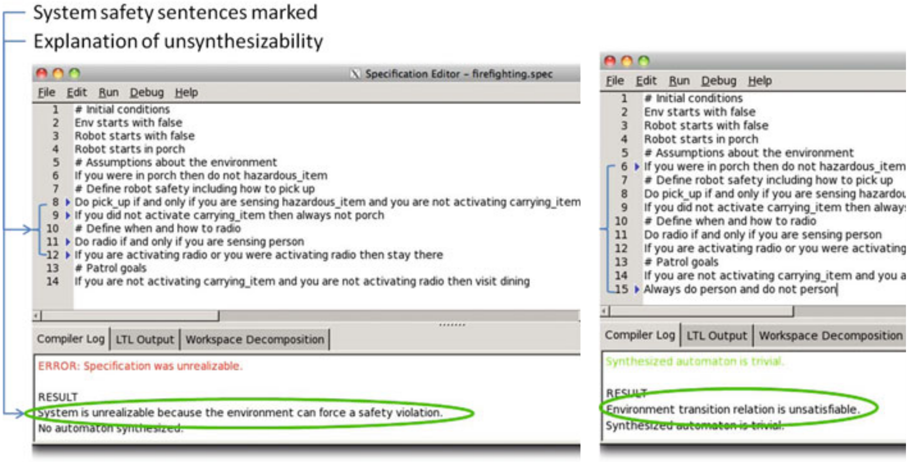
The created automaton is correct-by-construction: if a behavior can be achieved in all environments satisfying the given assumptions, then the LTL formula describing the behavior holds for every possible execution of the automaton. This implementing discrete automaton is then viewed as a hybrid controller, wherein a transition between states corresponds to the activation of one or more atomic continuous controllers that satisfy the bisimulation property [1] (e.g., the motion controllers are guaranteed to drive the robot from one region to another regardless of the initial state within the region).

We refer the reader to [11] for a complete discussion of the hybrid controller, and to [8] for a description of how atomic controllers are incorporated into the hybrid controller in LTLMoP. Finally, the synthesized hybrid controller can be embedded within a simulator or used with physical robots (such as the Pioneer 3-DX depicted in Fig. 1).

3 Unsynthesizable Specifications and Undesirable Behavior

A specification $\varphi_e \implies \varphi_s$ that does not produce a controller is either *unrealizable* or *unsatisfiable*, and there are several possible reasons for either. In addition, if φ_e is unsatisfiable, then all initial states are winning for the system, and so we do get an automaton, but a trivial one consisting of all the initial states but no transitions. We would like to identify this case, since the resulting behavior is probably not as intended.

With regards to unrealizability, we can just as well consider winning system strategies that prevent the environment from satisfying the formula φ_e. Overloading terminology, we say that the environment is unrealizable in this case. Note that if the environment is unrealizable, an otherwise unrealizable robot specification may be synthesizable if the robot can win by preventing the environment from upholding its assumptions. In fact, if the environment is unsatisfiable, every robot specification (even an unsatisfiable one) is synthesizable. In the robotics domain, we would like to flag this case, since we would like the robot to fulfill its goals rather than thwart the given environment.



```
# Initial conditions
Env starts with false
Robot starts with false
Robot starts in porch
# Assumptions about the environment
If you were in porch then do not hazardous_item
# Define robot safety including how to pick up
Do pick_up if and only if you are sensing hazardous_item and you are not activating carrying_item
If you did not activate carrying_item then always not porch
# Define when and how to radio
Do radio if and only if you are sensing person
If you are activating radio or you were activating radio then stay there
# Patrol goals
If you are not activating carrying_item and you are not activating radio then visit dining
```

```
ERROR: Specification was unrealizable.

RESULT
System is unrealizable because the environment can force a safety violation.
No automaton synthesized.
```

(a) Analyzing an unsynthesizable specification

```
# Initial conditions
Env starts with false
Robot starts with false
Robot starts in porch
# Assumptions about the environment
If you were in porch then do not hazardous_item
# Define robot safety including how to pick up
Do pick_up if and only if you are sensing hazardou
If you did not activate carrying_item then alway:
# Define when and how to radio
Do radio if and only if you are sensing person
If you are activating radio or you were activating
# Patrol goals
If you are not activating carrying_item and you a
Always do person and do not person
```

```
Synthesized automaton is trivial.

RESULT
Environment transition relation is unsatisfiable.
Synthesized automaton is trivial.
```

(b) Trivially synthesizable spec.

Fig. 2.

Listing 1. Excerpt of a fire-fighting scenario specification with corresponding LTL

1	Env starts with false	$\neg\pi_{person} \wedge \neg\pi_{hazardous\_item}$
2	Robot starts with false	$\neg\pi_{pick\_up} \wedge \neg\pi_{drop} \wedge \neg\pi_{carrying\_item}$
3	Robot starts in **porch**	φ_{porch}
4	If you were in **porch** then do not **person**	$\Box(\varphi_{porch} \Rightarrow \neg \bigcirc \pi_{person})$
5	If you were in **porch** then do not **hazardous_item**	$\Box(\varphi_{porch} \rightarrow \neg \bigcirc \pi_{hazardous\_item})$
6	Do **pick_ up** if and only if you are sensing **hazardous_ item** and you are not activating **carrying_ item**	$\Box(\bigcirc \pi_{pick\_up} \Leftrightarrow (\bigcirc \pi_{hazardous\_item} \wedge \neg \bigcirc \pi_{carrying\_item}))$
7	If you did not activate **carrying_ item** then always not **porch**	$\Box(\neg\pi_{carrying\_item} \rightarrow \neg \bigcirc \varphi_{porch})$
8	Do **radio** if and only if you are sensing **person**	$\Box(\bigcirc \pi_{radio} \Leftrightarrow \bigcirc \pi_{person})$
9	If you are activating **radio** or you were activating **radio** then stay there	$\Box((\bigcirc \pi_{radio} \vee \pi_{radio}) \rightarrow \bigwedge_{l}(\varphi_l \Leftrightarrow \bigcirc \varphi_l))$
10	If you are not activating **carrying_ item** and you are not activating **radio** then visit **dining**	$\Box \Diamond((\neg\pi_{carrying\_item} \wedge \neg\pi_{radio}) \rightarrow \varphi_{dining})$ …

4 Analyzing a Specification in LTLMoP

Given an unsynthesizable specification in LTLMoP, we apply a series of simple checks to determine which components of the corresponding LTL formula are flawed, trace them back to their structured English counterparts, and highlight these in the Specification Editor. We identify unsatisfiability of the system and environment initial conditions, single-step transitions, goals, and the conjunction of transitions and goals using boolean satisfiability tests, without checking the LTL specification as a whole. As a result, some unsatisfiable safety conditions are identified as unrealizable instead.

Consider the specification in Listing 1 from the fire-fighting scenario described in [8]. Removing the environment safety requirement in line 4 makes the specification unrealizable, because the environment can force the robot into a safety violation by setting π_{person} to true in the porch. By line 8, this causes the robot to set π_{radio} to true in the next time step; line 9 then requires it to stay where it is (i.e., $\bigcirc \varphi_{porch}$), but line 7 requires $\neg \bigcirc \varphi_{porch}$. The robot thus has no legal next state. Our analysis determines that the system (robot) is unrealizable because the environment can force a safety violation, and marks all safety sentences as in Fig. 2(a). Consider the same specification, augmented with the (clearly unsatisfiable) environment safety condition, Always person and not person ($\Box(\pi_{person} \wedge \neg\pi_{person})$). Synthesis succeeds, but as noted in Fig. 2(b), the environment liveness is unsatisfiable and the generated automaton is trivial.

5 Conclusions and Future Work

We have described a method for systematically analyzing the environment and system components of autonomous robot control specifications. By exploiting the structure of the specification, we identify possible reasons for failure to create an implementing robot controller. Our approach is implemented as part of the open source LTLMoP toolkit. We enclose the synthesis in a layer of reasoning that identifies the cause of failure, enabling the user to target their attention to the relevant portions of the specification. Once we identify a specification (or part thereof) as unsatisfiable or unrealizable, there is still potential for further analysis. Future work will leverage existing techniques [2, 5, 6, 9] to isolate the source of failure and provide the user with comprehensive feedback, including modifications to the input that would result in an implementing automaton.

References

1. Alur, R., Henzinger, T.A., Lafferriere, G.: George, and G. J. Pappas. Discrete abstractions of hybrid systems. Proceedings of the IEEE, 971–984 (2000)
2. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
3. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A New Requirements Analysis Tool with Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
4. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008)
5. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 52–67. Springer, Heidelberg (2008)
6. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean abstraction for temporal logic satisfiability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 532–546. Springer, Heidelberg (2007)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
8. Finucane, C., Jing, G., Kress-Gazit, H.: LTLMoP: Experimenting with language, temporal logic and robot control. In: IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems, pp. 1988–1993 (2010)
9. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications using simple counterstrategies. In: Formal Methods in Computer-Aided Design, pp. 152–159 (2009)
10. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Translating structured english to robot controllers. Advanced Robotics 22(12), 1343–1359 (2008)
11. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal-logic-based reactive mission and motion planning. IEEE Transactions on Robotics 25(6), 1370–1381 (2009)
12. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
13. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A framework for developing verification algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 171–174. Springer, Heidelberg (2010)
14. Schuppan, V.: Towards a notion of unsatisfiable cores for LTL. In: Fundamentals of Software Engineering, Third IPM International Conference, pp. 129–145 (2009)
15. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon control for temporal logic specifications. In: Hybrid Systems, pp. 101–110 (2010)

Practical, Low-Effort Equivalence Verification of Real Code

David A. Ramos and Dawson R. Engler

Stanford University
ramos@cs.stanford.edu, engler@csl.stanford.edu

Abstract. Verifying code equivalence is useful in many situations, such as checking: yesterday's code against today's, different implementations of the same (standardized) interface, or an optimized routine against a reference implementation. We present a tool designed to easily check the equivalence of two arbitrary C functions. The tool provides guarantees far beyond those possible with testing, yet it often requires less work than writing even a single test case. It automatically synthesizes inputs to the routines and uses bit-accurate, sound symbolic execution to verify that they produce equivalent outputs on a finite number of paths, even for rich, nested data structures. We show that the approach works well, even on heavily-tested code, where it finds interesting errors and gets high statement coverage, often exhausting all feasible paths for a given input size. We also show how the simple trick of checking equivalence of *identical* code turns the verification tool chain against itself, finding errors in the underlying compiler and verification tool.

1 Introduction

Historically, code verification has been hard. Thus, implementors rarely make any effort to do it. We present UC-KLEE, a modified version of the KLEE [2] tool designed to make it easy to verify that two routines are equivalent. This ability is useful in many situations, such as checking: different implementations of the same (standardized) interface, different versions of the same implementation, optimized routines against a reference implementation, and finding compiler bugs by comparing code compiled with and without optimization. Comparing identical code against itself finds bugs in our own tool.

Previously, cross checking code that takes inputs with complex invariants or complicated data structures required tediously constructing these inputs by hand. From experience, the non-trivial amount of code needed to do so can easily dwarf the size of the checked code (e.g., as happens when checking small library routines). Manual construction also leads to missed errors caused by over-specificity. For example, when manually building a linked list containing symbolic data, should it have one entry? Two? A hash table should have how many collisions and in which buckets? Creating all possible instances is usually difficult or even impossible. Further, manually specifying pointers (by assigning the concrete address returned by `malloc`) can limit paths that check relationships

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 669–685, 2011.

on them, such as when an if-statement checks whether one pointer is less than another. In general, if input has many constraints, a human tester will miss one.

In contrast, using our tool is easy: rather than requiring users to manually construct inputs or write a specification to check code against, they simply give our tool two routines (written in raw, unannotated C) to cross check. The tool automatically synthesizes the routines' inputs (even for rich, nested data structures) and systematically explores a finite number of their paths using sound, bit-accurate symbolic execution. It verifies that the routines produce identical results when fed identical inputs on these explored paths by checking that they either (1) write the same values to all escaping memory locations or (2) terminate with the same errors. If one path is correct, then verifying equivalence proves the other is as well. If the tool terminates, then with some caveats (discussed in § 3.4), it has verified equivalence up to a given input size.

Because UC-KLEE leverages the underlying KLEE system to automatically explore paths and reason about all values feasible on each path, it gives guarantees far beyond those of traditional testing, yet it often requires less work than writing even a single test case. We show that the approach works well even on heavily-tested code, by using it to cross check hundreds of routines in two mature, widely-used open source `libc` implementations, where it:

1. Found numerous interesting errors.
2. Verified the equivalence of 300 routines (150 distinct pairs) by exhausting all their paths up to a fixed input size (8 bytes).
3. Got high statement coverage — the lowest median coverage for any experiment was 90% and the rest were 100%.

A final contribution is a simple, novel trick for finding bugs in the compiler and checking tool by turning the technique on itself, which we used to detect a serious LLVM optimizer bug and numerous errors in UC-KLEE.

2 Overview

Cross checking implementations simplifies finding correctness violations because, rather than requiring that users write a functional specification, it lets the tool use a second implementation as a reference — functional differences will show up as mismatches. A natural concern is what happens on invalid inputs. In our experience, real code often shows *error equivalence*, where an illegal input causes the same behavior in both (e.g., when given a null pointer, both cross checked routines crash). Our tool exploits this fact and treats equivalent crashes as equivalent behavior, but flags when one implementation crashes and the other does not. (In general, cross checking cannot detect when two routines make equivalent mistakes.) This finesse works well in practice. In the rare cases where inputs are allowed to produce differing results, it is easy for simple, user-written C code to filter these inputs (discussed further in § 2.1).

We show how UC-KLEE works by walking through the simple but complete example in Figure 1, which gives two trivial routines intended to add a value to a structure field and the cross checking harness that UC-KLEE provides to compare them. The user compiles the routines using UC-KLEE's compiler (LLVM) and gives the resultant bitcode modules and two routine names to UC-KLEE, which links the code against a checking harness and runs the result. At a high level, the cross checking harness executes as follows:

1. It marks all function parameters as containing unconstrained symbolic data (i.e., they can contain any value representable given their size). UC-KLEE will lazily allocate memory on demand if this symbolic data is used as a pointer and dereferenced (discussed below).
2. It uses symbolic execution to explore (ideally all) paths in the two implementations, checking that they produce identical effects when run on the same values.
3. If a path's constraints permit a value that causes an error (such as a division by zero, null pointer dereference, or assertion failure), UC-KLEE verifies that the other routine terminates with the same error when run with the same input values. UC-KLEE also forks execution and explores a path on which the error does not occur so that it can cross-check the routine on the remaining values.
4. At the end of each path, UC-KLEE traverses all reachable memory, and uses its constraint solver to prove this memory has equivalent contents at the end of both paths. If this check fails, it generates a concrete input to demonstrate the difference. If the check succeeds, then with some caveats (see § 3.4) UC-KLEE has verified the two routines as equivalent since the constraints it tracks are accurate and exact (down to the level of a single bit). Thus, if one path is correct, UC-KLEE has verified that the other path is correct as well.

Note that UC-KLEE's equivalence guarantee only holds on the finite set of paths that it explores. Like traditional testing, it cannot make statements about paths it misses. However, in many cases, even if there are too many paths, KLEE can at least show total equivalence up to a given input size.

At a more detailed level, the code in Figure 1 works as follows:

Lines 14–18: stack allocates two variables to pass as the routine's parameters (`f` and `v`) and marks them as symbolic.

Line 21: creates a copy of the current address space, which it will restore later so that `add_bad` runs on identical values.

Line 22: uses `klee_eval` to run `add`. This call returns once for each path explored in `add`. If `add` terminates with an error, the error is stored in `e1`.

Line 3: at the first dereference `f→val`, UC-KLEE checks if `f` can be null. Since `f` has no constraints on its value, it can, so UC-KLEE forks execution and continues as follows:

Error path: adds the constraint that `f` is null, records in `e1` that a null dereference error occurred, and returns from `klee_eval`.

Non-error path: adds the constraint that `f` is not null and attempts to resolve the dereference. It determines that `f` is an unbound symbolic pointer, so it allocates memory (of size `foo`), marks it as symbolic, binds it to `f`, and continues executing until the path completes. It then returns from `klee_eval`.

```
1 : // two routines to cross-check.
2 : int add(foo *f, int v) {
3 :   f->val = f->val + v;
4 :   return f->val;
5 : }
6 : int add_bad(foo *f, int v) {
7 :   f->val = f->val + 1;
8 :   return f->val;
9 : }
10:
11: // harness provided by uc-klee
12: main() {
13:   klee_err e1,e2;
14:   int retv, v;
15:   foo *f;
16:
17:   klee_make_symbolic(&f);
18:   klee_make_symbolic(&v);
19:
20:   // record memory state "add" runs on.
21:   int s0 = klee_snapshot();
22:   klee_eval(retv = add(f,v), &e1);
23:   int s1 = klee_snapshot();
24:
25:   // discard writes, keep path constraints
26:   klee_restore(s0);
27:   klee_eval(retv = add_bad(f,v), &e2);
28:   int s2 = klee_snapshot();
29:
30:   // compare results.
31:   if (!klee_compare_errors(&e1,&e2)
32:       || !klee_compare(s1, s2, &f)
33:       || !klee_compare(s1, s2, &v)
34:       || !klee_compare(s1, s2, &retv))
35:     klee_error("Mismatch!\n");
36: }
```

Fig. 1. Trivial but complete checking example

Line 22 (after `klee_eval`): the two paths execute independently through the remaining code.

Line 23: records the memory state produced by running `add`, which it later compares against the memory state produced by running `add_bad`.

Line 26: restores the values of `f` and `v` that the current path ran `add` on so that `add_bad` runs on identical values. It discards all writes `add` did (otherwise `add_bad` would run with a modified value for `f`→`val`), but preserves all constraints, including any pointers it lazily bound (i.e., the dereference of `f` on line 3).

Line 27: evaluates `add_bad` using `klee_eval`. The error path also returns with a null pointer error (since `f` is constrained to null and line 7 dereferences it). The non-error path executes without error; the dereferences of pointer `f` (lines 7,8) resolve to the same object lazily allocated at line 3.

Line 31: checks that both paths returned with the same error state (they did).

Lines 32–34: checks that the values transitively reachable from the routines' outputs in each memory state are equivalent (§ 3.2 describes this analysis in more detail). On the non-error path, the check for `f` (line 32) fails and produces a test case with `v` equal to some value other than 1 (the single value for which both routines return identical results).

Notes. While the example declares the input variables `f` and `v` with their static types for readability, as far as UC-KLEE is concerned they could have been untyped byte arrays (which is how UC-KLEE treats them in any case) since our implementation correctly handles casting between pointers and integers.

Although this example does not access environment variables, UC-KLEE additionally marks the global `environ` pointer as unbound in order to explore paths where its values are read. Section 4 gives an example difference found as a result.

For performance, UC-KLEE does not explore paths that have identical LLVM instructions since they must produce the same results. UC-KLEE silently prunes a path when it satisfies both of the following conditions: (1) all previously executed basic blocks are identical and (2) all reachable basic blocks are identical.

2.1 Handling Uninteresting Mismatches

A tight specification that maps each input value to a single output value provides the simplest use case for UC-KLEE since *any* difference between implementations constitutes a bug. For looser specifications that include "don't cares," user effort may be needed to suppress uninteresting differences that UC-KLEE would otherwise report. Examples include permitting code to do anything when fed illegal input values or representing "success" by any non-zero integer rather than a single, specific value (e.g., `1`). Note that the problems caused by permitting flexibility are not specific to UC-KLEE — any method (such as randomized or specification-based testing) that checks output values or behavior has to deal with them.

UC-KLEE provides a simple yet general mechanism for eliminating uninteresting mismatches. Instead of invoking checked code directly (lines 22 and 27 in Figure 1), it passes the checked routine and its arguments to a user-supplied function, which calls the checked routine after filtering its input (e.g., by using an if-statement to skip illegal values) and then returns the (possibly canonicalized) return value.

Figure 2 shows an example filter for the `isdigit` function in the C library, specified to return non-zero if its input represents a digit in ASCII and 0 otherwise. The filter first rejects input values that fall outside the range specified in the C standard (line 2). It then invokes the passed-in `isdigit` function (line 4) and canonicalizes all non-zero return values to 1. In our experiments, this routine eliminated all mismatches for `isdigit` and 11 analogous routines.

In practice, even if a specification permits variable behavior, code tends to behave similarly. In fact, the most wide-spread use for UC-KLEE we envision — checking new versions of code against old versions — suffers from this problem the least since such decisions are consistent across revisions. Even where we would expect the most variance in behavior — independently-developed code bases fed error inputs — implementations tend to behave similarly. For example, in our experiments, checked routines typically crashed on illegal pointer inputs rather than returning differing values.

```
1 : int isdigit_f(int (*f)(int), int c){
2 :   if (c < EOF || c > 255)
3 :     return 0;
4 :   return ((*f)(c) != 0);
5 : }
```

Fig. 2. Simple filter routine

Many of the differences UC-KLEE found illustrated needless ambiguities in the underlying standard, which permitted divergent behavior without a subsequent gain in speed, power, or simplicity. In future work, we plan to explore the use of UC-KLEE as an automatic tool for finding such specification imprecisions.

In a sense, UC-KLEE inverts the typical work factor for checking code: a traditional specification-based approach requires specifying what behavior the user cares about (i.e., the functionality the code should implement), whereas UC-KLEE infers this information "for free" by cross checking implementations. On the other hand, UC-KLEE (may) require specifying the "don't care" behaviors (when code is allowed to differ), which typically takes orders of magnitude less effort than specifying functionality. Further, users only need to specify these details on demand, after UC-KLEE detects an uninteresting mismatch. In contrast, specification verification requires non-trivial work before doing any checking.

3 Implementation

In this section we discuss the trickiest part of implementing UC-KLEE: tracking whether a piece of memory contains a pointer and if so, to which memory object. We then describe how it uses this ability both to compare the results of two procedure invocations (§ 3.2) and to lazily allocate memory when an unbound pointer is dereferenced (§ 3.3). We then discuss limitations.

3.1 Referents: Tracking Which Memory Contains Pointers

UC-KLEE tracks pointers using a *referent*-based approach similar to [15,21], but modified to support symbolic execution. Each register, stack location, global, and heap object has a corresponding piece of shadow memory. Whenever code writes a pointer to memory or a register, UC-KLEE writes the starting address of the pointed-to object (its referent) to this shadow memory at the same offset. When code writes a non-pointer value, UC-KLEE clears these shadow values, indicating that the memory does not contain a pointer. The key advantage this approach is that we can determine what object a pointer was *intended* to point to even if the pointer's *actual* value refers to an address outside the bounds of the object (and potentially inside a different, allocated object).

For space reasons we elide most details of this tracking. It roughly mirrors that of a concrete tool, with the one difference that we must reason about reads and writes at symbolic locations — i.e., those calculated using symbolic expressions and hence representing a set of values rather than a single concrete value (a constant). For example, suppose variable `i` has the constraint $0 <= i < n$ where n is the size of array `a`. Then the write `a[i] = &p` conceptually creates a set of n possible (exclusive) arrays since the write of address `&p` could be to `a[0]` or `a[1]` or ... `a[n]` depending on the value of `i`. Subsequent stores using symbolic indices create more possibilities; reads cause similar explosions. Thus, we cannot just trivially calculate which shadow location to propagate this information to or read it from. Fortunately, the solution is easier to describe than the problem: KLEE already has the ability to reason about the reads and writes checked code performs at symbolic locations; by handling and representing shadow memory in the same way as arrays in checked code, we can reuse this same machinery.

Low-level C code can egregiously violate any sensible notion of typing. To handle such code we took the extreme position of completely ignoring static types and treating all memory as potentially containing a pointer. (We may revisit this decision in the future.) While there are a variety of details to get right, the most common problem is that given a dereference `*(p+q)`, we do not robustly know which value (`p` or `q`) holds the address, and which the offset. We determine this information based on usage rather than type, and mark any location as not holding a pointer if the code performs operations such as: bitwise operations that lose upper bits (losing the lower few bits is okay), negation, left shift, multiplication, division, modulus, and subtracting two pointer values. As a result, we can reliably check code that does type debasements far beyond merely casting between pointers and integers.

3.2 Object Comparison

We define two routines as being equivalent on a path if *they write identical values to all memory transitively reachable from their return value and each of their formal arguments.* That is, pointer values (addresses) can differ as long as (1) the objects they point to do not, and (2) the pointer is to the same offset within the object. UC-KLEE checks this property by doing a mark and sweep of all reachable memory and using the constraint solver to prove that all non-pointer bytes are equal. In the concrete case, comparisons reduce to constants, avoiding expensive satisfiability queries. For symbolic bytes that neither routine modifies, the values in each address space snapshot contain identical symbolic expressions, which are trivially equivalent. If UC-KLEE detects a pointer, it adds the referenced objects (from each snapshot) to a queue for later traversal, rather than comparing the objects' addresses, which may differ between the two procedures. In the case of pointers stored into memory at symbolic offsets, it is possible for a particular value to resolve to multiple objects. In this case, UC-KLEE examines every pair of objects to which the two pointers could resolve. If a single pair of objects differs, UC-KLEE flags the error. Note that unless a function's return value or one of its formal arguments contains a pointer to a global variable, UC-KLEE does not automatically compare global variables because multiple implementations typically utilize an incompatible set of globals.

3.3 Lazy Initialization

UC-KLEE uses a variation on *lazy initialization* [16] to dynamically allocate objects on an as-needed basis. This prior work was in the context of checking a single class method in type-safe Java; our implementation aims at cross checking non-type-safe C functions.

When the test harness marks function arguments as symbolic (the call to `klee_make_symbolic` on lines 17–18 of Figure 1), UC-KLEE also sets an "unbound" bit in shadow memory. At each pointer dereference `*p`, UC-KLEE examines this shadow memory to check whether `p` is an unbound pointer and hence must be lazily allocated. If a dereferenced pointer `p` is unbound, UC-KLEE:

1. Allocates an object of size n (discussed below) and marks it as containing unbound, symbolic bytes. Any dereference of this memory will similarly (recursively) allocate an object.

2. Constrains `p`'s referent to equal the address m of the allocated object ($p_{base} = m$), ensuring future dereferences map to the same object.
3. Constrains `p`'s "unbound" bit to *false* (bound). This prevents UC-KLEE from unintentionally allocating a new object during a subsequent dereference of the same pointer. Note that once a referent is bound, it can never become unbound, even if the object is explicitly `free`'ed by the procedure.
4. Constrains `p` to point within the allocated object: $m <= p < m + n$.

Each lazy allocation creates a unique memory object. A different approach would allow unbound pointers to resolve to existing objects of the same type. We did not use this method since it significantly increases the state space and can lead to many false positives. The main drawback of our current approach is that UC-KLEE cannot satisfy address constraints of code that specifically targets pointers to overlapping memory blocks, which limits coverage. We plan to revisit this decision as we encounter more real examples that require it.

Address constraints. Additional complexity arises when code compares unbound pointers. Suppose we take the true path of an equality comparison `x == y` and both `y` and `x` are unbound. If `x` is subsequently dereferenced and bound to a new object, we want `y` to be bound to the same object (and vice versa). We initially thought doing so would require constructing dynamic dependency graphs to determine where a pointer comes from (difficult in the general case given complex symbolic expressions). Fortunately, our shadow memory scheme makes the solution simple: merely constrain `x`'s shadow memory to equal `y`'s ($x_{base} = y_{base}$ and $x_{unbound} = y_{unbound}$). A subsequent binding of one will make the other bound as well. If only one pointer is unbound, we do the same thing, with the same effect. Note that, on the false path (where $x \neq y$), we do *not* add a constraint that their shadow memory differs because `x` and `y` may point to different bytes in the same object (and thus may share a referent).

Code sometimes compares two pointers to different objects using greater-than or less-than conditions (such as a binary tree sorted by address). While not strictly legal, UC-KLEE must nonetheless support such comparisons in order to be effective. Unlike equality, these comparisons add no additional constraints on referents. Instead, when a subsequent dereference causes UC-KLEE to lazily allocate an object, the object's address must satisfy all existing path constraints. To accomplish this, UC-KLEE allocates two large (e.g., 32 megabyte) memory pools on startup at high and low address ranges. Each allocation searches both of these pools for a block whose address does not violate the path constraints. If it cannot find one, the path terminates and UC-KLEE reports a runtime error. The most common cause we observed was code that specifically checks whether two pointers overlap. As we mentioned above, UC-KLEE does not allocate overlapping objects; thus, such constraints cannot be satisfied.

Allocation size. When UC-KLEE lazily allocates an object, it must choose a fixed size for that object. When an unbound pointer references a type of known size (e.g., an `int` or a `struct`), we simply allocate the exact size necessary to store that type. However, we cannot do the same when the pointer might refer to an array (since C arrays do not have statically known sizes). For example, when a '`char *`' (string) is dereferenced, we do not know the size of the string. Unfortunately, making it too small will limit statement coverage, since many (or all) paths would terminate with out-of-bounds dereferences. Making it too large will disguise legitimate out-of-bounds errors in the code. Our current solution is a hack, but it seems to work well enough in practice. We first consider allocating an object of a user-specified minimum size (for our experiments, we found that a minimum size of 8 bytes works in most cases). UC-KLEE queries the constraint solver to test whether this allocation size would satisfy the current memory operation. If not, UC-KLEE iteratively doubles the guess. Within logarithmic time, UC-KLEE either finds a satisfying allocation size or reaches a user-specified maximum (e.g., 2 kilobytes). If the maximum fails, UC-KLEE terminates the path and reports a runtime error to the user.

Depth limits. Lazy initialization allows UC-KLEE to dynamically support hierarchical data structures such as linked lists and trees. To control the resulting path explosion, our tool limits this type of allocation to a user-specified depth limit, incrementing a counter on each nested allocation. If the depth counter exceeds the limit, our tool terminates the path and outputs a warning. Without a depth limit, the path space would quickly become unmanageably large.

3.4 Limitations

This section enumerates the known limitations of UC-KLEE. We are of course vulnerable to bugs in UC-KLEE or its environment modeling code. We check at the implementation level, which makes it easy to work with real code. However, it makes any verification claims true only for the specific compiler and architecture we used for our experiments. These guarantees may not hold on code where the compiler must make a choice among several unspecified behaviors (such as function argument evaluation order) or when running on a machine that differs in some observable way (such as word size or endianness).

During cross checking we only invoke a routine a single time and check it in isolation, missing behaviors that require multiple invocations or coordination across routines. In general, we may miss behaviors for code that depends on the values of addresses (e.g., greater-than or less-than relationships among objects allocated by `malloc`). More specifically, as discussed § 3.3, there are several limitations in our approach to lazy initialization. We assume that lazily allocated objects cannot alias existing objects, so we will not exercise code that checks for overlap (such as `memmem`) or expects an unbound pointer to point to the middle of an existing object (e.g., a circular buffer). We will also miss paths that need objects larger than our maximum size, since these will always terminate with an error. Indirect calls to unbound function pointers are unsupported at this time. On paths that have errors, UC-KLEE is unable to identify the root cause. Thus, it will not detect when two checked routines terminate with the same error type, but from different causes.

The underlying system, KLEE, must replace a symbolic value with a single concrete example when used as an allocation size or in a floating point operation.

Thus, our tool may miss bugs exposed by different concrete values later in the execution path. In addition, KLEE cannot handle routines that return structures, contain inline assembly, call unresolved external routines, or call external routines using symbolic arguments. When a path encounters one of these, UC-KLEE flags the routine as unverified.

4 Evaluation

This section shows that UC-KLEE works well at verifying equivalence by cross checking recent versions of two heavily-tested open source C libraries: UCLIBC, an implementation of the C standard library targeted at embedded devices, and NEWLIB, an embedded `libc` implementation by Red Hat used by Cygwin and Google Native Client. We demonstrate its effectiveness on three common use cases, cross checking: different implementations of the same interface, different versions of the same code, and identical code to find errors in the verification tool chain (in our case: the LLVM compiler and UC-KLEE itself).

We measure the quality of cross checking in two ways: (1) crudely, by the statement coverage it achieves, and (2) by whether checking exhausts all paths and terminates, since that verifies that the routines are equivalent up to a fixed input size when invoked a single time (modulo the limitations discussed in § 3.4).

It is a bit tricky to measure statement coverage for library code. We compute coverage of a cross checked routine as a percentage of the total number of LLVM instructions reachable from it, with the exception that when routine *a* calls another exported routine *b* that we will also cross check, we exclude *b*'s instructions from *a*'s coverage statistics. Usually, such calls can only exercise a small fraction of *b*'s code (e.g., when *a* calls `printf` with a format string that just contains "`hello world`"). On the other hand, if *a* calls *c* and we *do not* generate a test harness for *c*, we *do* count its instructions since we conservatively assume it is an internal helper function that *a* should thoroughly exercise. Note: every instruction is included in the coverage statistics for at least one procedure.

For all experiments, we ran UC-KLEE on each procedure for up to 10 minutes, and allowed each procedure to read from up to 2 symbolic files of 10 bytes each (KLEE argument `--sym-files 2 10`). This was in addition to the symbolic arguments and environment generated by the cross checker. Our machine was a quad-core 2.8 GHz Intel i7 machine with 12GB of RAM running Fedora Linux 12.

4.1 Different Implementations: NEWLIB vs. UCLIBC

Our first experiment cross checks NEWLIB's source repository from July 2010 against UCLIBC version 0.9.31. We modified both libraries to use UC-KLEE's memory allocator. We also disabled several UCLIBC internal startup and shutdown tasks that interfered with UC-KLEE. Finally, to keep the experiments manageable, we disabled optional features, such as wide character and locale support.

We automatically generated a test harness for each routine that both libraries implemented with the exception of variadic routines or those whose prototypes

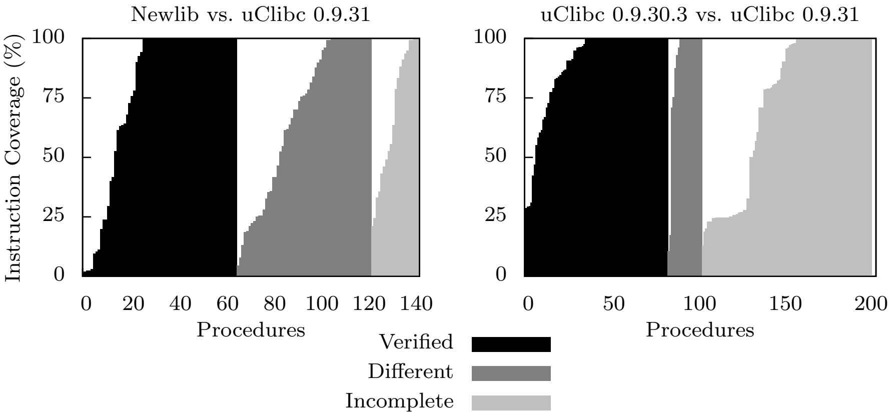


Fig. 3. Instruction coverage reported by our cross checking experiments. Each vertical bar represents a single procedure, sorted by coverage. The "incomplete" category includes routines whose analysis did not complete within 10 minutes or hit a limitation in KLEE or UC-KLEE. The median statement coverage for the left graph was over 90% (59 routines had 100%) and for the right was 100% (105 had 100%).

differed. We could extend our system to support the former by generating multiple test cases for different numbers of arguments. Our experiment tested all other exported procedures, even those that demonstrate weaknesses in our tool.

Figure 3 (left) shows the coverage reported by UC-KLEE. In the routines where UC-KLEE found no differences, it checked 66 to termination (versus 15 where it exceeded the time limit), thereby verifying equivalence for the given input size, despite many having entirely different structure and overall appearance to the human eye. The two implementations of `ffs` ("find first bit set") in Figure 4 are a good example: UC-KLEE exhausted all 33 paths in the test harness and terminated after 6.8 seconds, reaching 100% statement coverage.

```
int ffs (int word) {
  int i=0;
  if (!word)
    return 0;
  for (;;)
    if (((1 << i++)&word) != 0)
      return i;
}
```

(a) NEWLIB

```
int ffs(int i) {
  char n = 1;
  if (!(i & 0xffff)) { n += 16; i >>= 16; }
  if (!(i & 0xff))   { n += 8; i >>= 8; }
  if (!(i & 0x0f))   { n += 4; i >>= 4; }
  if (!(i & 0x03))   { n += 2; i >>= 2; }
  return (i) ? (n+((i+1) & 0x01)) : 0;
}
```

(b) UCLIBC

Fig. 4. Two implementations of `ffs` ("find first set bit") UC-KLEE verifies as equivalent despite their difference in appearance and approach

UC-KLEE found differences in 57 of the 143 functions checked, at least 7 of which were real bugs—despite the code being heavily tested, actively used, and designed to do well-understood tasks. One interesting example was an error in Newlib's implementation of `remove` (Figure 5), which the POSIX standard mandates should work for both files and directories. uc-klee detects that Newlib returns `-1` (error) while uClibc returned `0` (success) when the symbolic input `filename` could refer to a directory. This error would be difficult to detect statically.

```
int _remove_r(struct _reent *ptr,
              const char *filename) {
  if (_unlink_r (ptr, path) == −1)
    return −1;
  return 0;
}

int remove(const char *filename) {
  return _remove_r(_REENT,
                   filename);
}
```

(a) Newlib

```
int remove(const char *filename)
{
  int saved_errno = errno;
  int rv;
  rv = rmdir(filename);
  if ((rv < 0) && (errno == ENOTDIR)) {
    _set_errno(saved_errno);
    rv = unlink(filename);
  }
  return rv;
}
```

(b) uClibc

Fig. 5. uc-klee detects that Newlib does not handle directory removal correctly

We achieved high statement coverage in most but not all procedures. One common cause of low coverage is that we only cross check code using a single invocation. In certain cases, multiple invocations of a routine are required in order to reach additional code. In other cases, one routine may write values to globals or `statics` that are read by another. A good example of both is `atexit`, which registers routines to be run on program termination by `exit`. A simple extension would allow uc-klee to handle such cases.

4.2 Different Versions of the Same Implementation: uClibc

To measure uc-klee's effectiveness at cross checking different versions of the same code, we used it to compare all functions that appeared in both uClibc 0.9.30.3 (March 2010) and uClibc 0.9.31 (April 2010) that were not byte-code identical. This selection yielded 203 routines (out of 399 possible), each of which uc-klee analyzed for up to 10 minutes.

Figure 3 (right) plots the instruction coverage. uc-klee revealed 2 previously unknown bugs and also detected 5 instances of bugs that were patched in the newer release. We elide a detailed discussion for space reasons and instead provide a brief example for each.

The newer version of uClibc introduced a bug in `ctime` (used to convert a time record to a string). The older version used a persistent internal structure (i.e., `static`) for storage that lacked thread safety. The newer version instead used a stack-allocated buffer that it never initialized. A sufficiently large input value caused the returned string to differ, which uc-klee detected.

uc-klee confirmed that a number of bugs present in uClibc 0.9.30.3 were corrected in version 0.9.31. One example is `unsetenv` (below). The old code (left) terminated with an out-of-bounds read when __environ is NULL (e.g., after a call to the function clearenv), while the new code (right) exited gracefully:

```
char **ep = _environ;
while (*ep != NULL) { ... }
```

(a) unsetenv: uClibc 0.9.30.3

```
char **ep = _environ;
if (ep) while (*ep != NULL) { ... }
```

(b) unsetenv: uClibc 0.9.31

4.3 Checking the Checker: Finding Bugs in UC-KLEE and LLVM

A standard caveat in verification papers is that their claims are contingent on the correctness of the verifier and underlying compiler. One of our contributions is the realization that one can detect errors in both by simply attempting to prove the equivalence of identical code, thus turning the verification system on itself.

Finding compiler optimizer bugs. We check that an optimizer has correctly transformed a program path by compiling the same routine both with and without optimization and cross checking the results. With the usual caveats, if any possible value exists that would cause the path to give different results, UC-KLEE will detect it. If there is no such value, it has verified that the optimizer worked correctly on the checked path. If it terminates, it has shown that the optimizer transformed the entire routine correctly. Any discrepancies it finds are due to either compiler bugs or the routine depending on unspecified behavior (e.g., function evaluation order between sequence points). Because the library code we checked intends to be portable, even use of unspecified compiler behavior almost certainly constitutes an error.

We compared all 622 procedures in uClibc 0.9.31, compiled with no optimization (-O0) versus high optimization (-O3). This check uncovered at least one bug in LLVM 2.6's optimizer but did not expose its root cause. For memmem, UC-KLEE reported a set of concrete inputs where the unoptimized code returned an offset within haystack (the correct result), while the optimized code returned NULL, indicating that needle was not found in haystack. We confirmed this bug with a small program. Since LLVM is a mature, production compiler, the fact that we immediately found bugs in this simple way is a strong result. We found a total of 70 differences, but because of time constraints could not determine whether they were due to this bug or others. Future work will be necessary to test optimization levels between these two extremes and attempt to automatically find a minimal set of optimization passes that yield an observable difference.

Finding UC-KLEE bugs. In general, tool developers can detect verifier bugs by simply cross checking a routine against another identical copy of itself (i.e., compiled at the same optimization level). This check has been a cornerstone of debugging UC-KLEE—it often turned up tricky errors after development pushes.

The UC-KLEE bugs we found fell into two main categories: (1) unwanted nondeterminism in UC-KLEE and its environmental models, which makes it hard to replay paths or get consistent results, and (2) bugs in our initial pointer tracking approach. In fact, as a direct result of the tricky cases cross checking exposed in this pointer tracking implementation, we threw it away and instead designed the much simpler and robust method in Section 3.1.

4.4 Results summary

Figure 6 summarizes the results presented in this section. The "KLEE Limitations" row describes procedures that resulted in incomplete testing due to limitations in the underlying KLEE tool: inline assembly (141 procedures), external calls with symbolic arguments (206), and unresolved external calls (17). "UC-KLEE Limitations" are cases where the tool failed to lazily allocate objects either because the required size of the object exceeded our specified maximum of 2KB (20 procedures) or UC-KLEE was unable to allocate an object whose address satisfied the path constraints (117 procedures). Note that limitations resulted in individual paths being terminated. As a result, certain procedures encountered a variety of limitations on different paths. In particular, a procedure deemed "KLEE limited" may have also encountered UC-KLEE limitations, although the converse is not true.

	Newlib/ uClibc	uClibc Versions	LLVM Optimizer	UC-KLEE Self-check
Procedures Checked	143	203	622	622
Procedures Verified	66	84	335	335
Differences Detected	57	20	70	12
No Differences (timeout)	15	30	85	91
KLEE Limitations	4	56	94	147
UC-KLEE Limitations	1	13	38	37
100% Coverage	59	105	367	375
Mean Coverage	72.2%	80.7%	85.6%	85.6%
Median Coverage	90.1%	100.0%	100.0%	100.0%

Fig. 6. Breakdown of procedures checked in each experiment

5 Related Work

This paper builds on the many recent research projects in symbolic execution, such as [2,3,12,16,22], as well as several pieces of our past work. About a decade ago, we showed how to avoid the need for manual specification by cross checking multiple implementations in the context of static bug finding [9], an idea we later used with symbolic execution [2,3]. This latter work only handled complete applications or routines run on manually constructed symbolic input; this paper shows how to easily check code fragments with unbound inputs and how to use cross checking to find bugs in the checking infrastructure itself. This paper is related to under-constrained execution [8], but modified to support the cross checking context, where many of the tricky issues are elided.

Many previous approaches to software checking have been specification based, requiring extensive work on the part of the user. One example is the use of model checking to find bugs in both the design and the implementation of software [1,4,5,11,13,14], which requires manually building test harnesses. A second example is recent verification work that checks code manipulating complex data structures against manually constructed specifications [6,7,10,18]. While both can exploit their specifications to reduce the state space, they require far more user effort than UC-KLEE.

Similar work has attempted to cross check largely identical code by using over-approximation to filter out unchanged portions of Java code [20]. While their technique is sound with respect to verification, a consequence of over-approximation is that reported differences may not concretely affect the output. In contrast, UC-KLEE generates test cases that supply concrete inputs to expose behavioral differences in the code.

Earlier work in cross checking has focused on combinational circuits in hardware [4,17,19]. While an important milestone, hardware verification is simpler than general purpose software equivalence checking, which includes loops, complex pointer relationships, and other difficult constructs.

Smith and Dill [23] recently verified the correctness of real-world block cipher implementations. Their work exploits the key properties that block ciphers have fixed input sizes and loop iterations, enabling full loop unrolling. They developed several constraint optimizations that we hope to adapt for cross-checking general-purpose code.

6 Conclusion

We have presented UC-KLEE, a tool that often makes cross checking two implementations of the same interface easier than writing even a single test case. The preliminary results demonstrate the usefulness of our approach, which often exhaustively explores all paths and verifies two procedures as equivalent up to a given input size.

We are currently building an improved version of UC-KLEE that is capable of cross checking individual code patches rather than complete routines, thereby reducing the problems of path explosion. Further, by jumping to the start of a patch, it will more robustly support code not easily checked by dynamic tools (such as device driver code). We plan to use this ability to check that kernel patches only remove errors or refactor code (for simplicity or performance) but do not otherwise change existing functionality.

Acknowledgements

The authors would like to thank Philip Guo, Diego Ongaro, and Amanda Ramos for their valuable feedback. This work is supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0645962. The paper's words only represent the views of the authors.

References

1. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering, ASE 2000 (2000)
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the Eighth Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 209–224 (December 2008)

3. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 2–23. Springer, Heidelberg (2005)
4. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proc. of the Asia and South Pacific Design Automation Conference, ASP-DAC 2003 (2003)
5. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S.: Bandera: Extracting finite-state models from Java source code. In: Proc. of the 22nd International Conference on Software Engineering (ICSE 2000), pp. 439–448 (June 2000)
6. Deng, X., Lee, J.: Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proc. of the 21st IEEE International Conference on Automated Software Engineering, pp. 157–166 (2006)
7. Elkarablieh, B., Marinov, D., Khurshid, S.: Efficient solving of structural constraints. In: Proc. of the International Symposium on Software Testing and Analysis, pp. 39–50 (2008)
8. Engler, D., Dunbar, D.: Under-constrained execution: making automatic code destruction easy and scalable. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA (July 2007)
9. Engler, D., Yu Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proc. of the 18th ACM Symposium on Operating Systems Principles, SOSP 2001 (2001)
10. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Proc. of the 32nd International Conference on Software Engineering (ICSE 2010), pp. 225–234 (2010)
11. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proc. of the 24th Annual Symposium on Principles of Programming Languages (POPL 1997), pp. 174–186 (January 1997)
12. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005), pp. 213–223 (June 2005)
13. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
14. Holzmann, G.J.: From code to models. In: Proc. of the Second International Conference on Applications of Concurrency to System Design, ACSD 2001 (June 2001)
15. Jones, R., Kelly, P.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proc. of the International Workshop on Automatic Debugging (1997)
16. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proc. of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2003)
17. Kuehlmann, A., Krohm, F.: Equivalence checking using cuts and heaps. In: Proc. of the 34th annual Design Automation Conference, DAC 1997, pp. 263–268 (1997)
18. Marinov, D., Andoni, A., Daniliuc, D., Khurshid, S., and Rinard, M. An evaluation of exhaustive testing for data structures. Tech. rep., MIT Computer Science and Artificial Intelligence Laboratory Report MIT-LCS-TR-921 (2003)
19. Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinational equivalence checking. In: Proc.of the 2006 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2006, pp. 836–843 (2006)

20. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16), pp. 226–237 (2008)
21. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proc. of the 11th Annual Network and Distributed System Security Symposium, pp. 159–169 (2004)
22. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13), pp. 263–272 (September 2005)
23. Smith, E.W., Dill, D.L.: Automatic formal verification of block cipher implementations. In: Proc. of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008), pp. 1–7 (2008)

Relational Abstractions for Continuous and Hybrid Systems*

Sriram Sankaranarayanan[1] and Ashish Tiwari[2]

[1] University of Colorado, Boulder, CO
sriramss@colorado.edu
[2] SRI International, Menlo Park, CA
ashish.tiwari@sri.com

Abstract. In this paper, we define relational abstractions of hybrid systems. A relational abstraction is obtained by replacing the continuous dynamics in each mode by a binary transition relation that relates a state of the system to any state that can potentially be reached at some future time instant using the continuous dynamics. We construct relational abstractions by reusing template-based invariant generation techniques for continuous systems described by Ordinary Differential Equations (ODE). As a result, we abstract a given hybrid system as a purely discrete, infinite-state system. We apply k-induction to this abstraction to prove safety properties, and use bounded model-checking to find potential falsifications. We present the basic underpinnings of our approach and demonstrate its use on many benchmark systems to derive simple and usable abstractions.

1 Introduction

In this paper, we present *relational abstractions* of hybrid systems. A relational abstraction transforms a given hybrid system into a purely discrete transition system by *summarizing* the effect of the continuous evolution of states over time using relations. The abstract discrete system is an infinite-state system that can be analyzed using standard techniques for verifying systems such as k-induction and bounded model checking.

Relational abstractions preserve the discrete behavior of the hybrid system and abstract only its continuous behavior. They work by replacing the continuous dynamics in each mode by means of a relation $R(\boldsymbol{x}_0, \boldsymbol{x})$. The relation R relates a continuous state $\boldsymbol{x}_0$ with a state $\boldsymbol{x}$ that can be potentially reached at some future time instant, through some time trajectory of the system starting from $\boldsymbol{x}_0$. Such a relation R can be interpreted in two ways: **(a)** as a positive invariant set for an associated dynamical system over $\boldsymbol{x}_0, \boldsymbol{x}$, and **(b)** as a discrete transition relation that abstracts the evolution of the continuous states over time.

* Sankaranarayanan's work has been supported by NSF Career grant CNS-0953941. Tiwari's work supported in part by DARPA under Contract No. FA8650-10-C-7078, NSF grants CSR-0917398 and SHF:CSR-1017483.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 686–702, 2011.

The two views above provide two key advantages of the relational abstraction approach. As a consequence of the first view, we can use techniques for generating invariants for continuous systems to generate a relational abstraction. We propose simple extensions of template-based invariant generation techniques, which can abstract systems with linear as well as nonlinear dynamics, to construct relational abstractions. Template-based techniques allow us to specify the form of the relational abstraction [9,19]. Therefore, our technique can be used to obtain *linear* arithmetic relations for systems with *nonlinear* dynamics.

As a consequence of the second view, we obtain discrete infinite-state abstractions of hybrid systems. This enables us to use techniques such as k-induction using decision procedures [37], abstract interpretation [11,21], or virtually any technique for discrete systems, to analyze hybrid systems.

It is well-known that the problem of verifying hybrid systems is quite hard, both in theory and in practice. Recently, many advances have yielded remarkably efficient tools for integrating affine ODEs over sets and that work over large state spaces [24,5,39,16,35,17]. However, we have observed that a significant gap in performance remains when these techniques are used to perform symbolic model checking, along the lines of tools such as HyTech and PHAVer [23,15]. In our experience, this gap stems from the need to handle the dynamics repeatedly for the same mode, often with small variations between subproblems. In this paper, we hypothesize that the situation with continuous dynamics is analogous to that of function calls encountered during program analysis. During inter-procedural program analysis, it is often observed that the analysis of each function call, given the state at the entry to the call, is quite efficient. However, the overall inter-procedural analysis is often not scalable due to repeated analysis of the same function with different actual parameters. Therefore, as in the case of function calls in program analysis, we propose *summarization techniques* that abstract the effect of the dynamics in each mode by a discrete transition [27]. As a result, our technique can efficiently handle continuous dynamics. However, on the flip side, our approach may lose precision if the relational abstraction is too conservative. Furthermore, the computation of relational invariants implicitly doubles the number of state variables.

Our approach is able to prove safety properties of hybrid systems using techniques such as k-induction, as well as to discover potential violations through bounded model checking. To evaluate the idea of using relational abstractions of hybrid systems, we generate relational abstractions of some standard benchmarks and model check these abstractions. We generate relational abstractions using a combination of quantifier elimination tools (REDLOG, QEPCAD) to search for templatized invariants [42,8,38], and polyhedral analysis of ODEs using fixed point iteration over cones [36]. We analyze the resulting relational abstractions using the SAL framework from SRI [34,40]. Our preliminary experiments are quite promising: our approach has the ability to prove properties of hybrid systems that are known to be complex, while at the same performing much more efficiently than symbolic model checkers. The data from our experiments along with an extended version of this paper with proofs will be made available online[1]. We now discuss other related ideas in the literature.

Transition Invariants and Variance Analysis: The idea of defining "progress" invariant predicates over pairs of states, $\mathtt{x}, \overline{\mathtt{x}}$, is well-known in the field of program analysis. A lot of work has been done on verifying *liveness* properties using ranking functions, transition invariants, and progress invariants [4,30,10,18]. However, there are some important distinctions between these various forms of relational invariants. Transition invariants [30] capture the relationship between the current state and *any* previous state (at a particular program location). Transition invariants were used to prove termination. Progress invariants capture the relationship between the current state and the *immediately* previous state (at a particular program location) [18]. Progress invariants were used to compute complexity bounds of programs. The relational abstractions presented here have a subtle difference: they capture the relationship between the current state and *all* previous states *after* the current mode was entered. When combined with the entry and exit conditions of a mode, relational invariants are exactly *summaries* of that mode. We use relational invariants to create abstractions of hybrid systems that can be used, for instance, to verify *safety* properties.

Podelski and Wagner provide a verification procedure for (region) stability properties of hybrid systems [31], where they derive *binary reachability relations* over trajectories of a hybrid system, similar to what is being proposed here. However, there are two key differences in our methodology: (a) Our approach deals with the dynamics at each mode up front, deriving relational abstractions. On the other hand, the technique of Podelski et al. transforms the entire hybrid system, relying on safety verification built into a tool such as Phaver to derive the relations [15]. Our goal in this paper is to make the process more efficient using constraint-based approaches and improve hybrid system safety verification in the first place. (b) Second, our ultimate goal is to verify safety properties efficiently as opposed to verifying stability.

Abstractions of Hybrid Systems: Many different types of *discrete abstractions* have been studied for hybrid systems including predicate abstraction [2,41] and abstractions based on invariants [28]. The use of counterexample-guided abstraction refinement for iterative refinement has also been investigated in the past (Cf. Alur et al. [2] and Clarke et al. [7], for example). In this paper, the proposed abstraction yields a discrete but infinite state system.

Hybridization is a technique for converting nonlinear systems into affine systems by subdividing the invariant region into numerous subregions and approximating the dynamics as a hybrid system by means of a linear differential inclusion in each region [23,3,12]. However, such a subdivision is expensive as the number of dimensions increases and often infeasible if the invariant region is unbounded.

Reachability Analysis: Reasoning about the reachable set of states for flows of nonlinear systems is an important primitive that is used repeatedly in the

[1] Cf. `http://www.csl.sri.com/~tiwari/relational-abstraction/`

analysis of nonlinear hybrid systems. This has been addressed using a wide variety of techniques in the past, including algebraic techniques, interval analysis, constraint propagation, and Bernstein polynomials [32,26,29,33,13].

2 Preliminaries

We present the basic definitions and properties of continuous systems defined by Ordinary Differential Equations (ODE). Let $\mathbb{R}$ denote the set of real numbers. We use $\boldsymbol{a}, \ldots, \boldsymbol{z}$ with subscripts to denote (column) vectors and $A, \ldots, Z$ to denote matrices. For an $m \times n$ matrix A, the row vector A_i, for $1 \leq i \leq m$, denotes the i^{th} row. We define continuous systems using vector fields.

Definition 1 (Vector Field). *A vector field $\mathfrak{F}$ over a set $X \subseteq \mathbb{R}^n$ is a function $\mathfrak{F} : X \mapsto \mathbb{R}^n$ mapping each $\boldsymbol{x} \in X$ with a field direction $\mathfrak{F}(\boldsymbol{x})$.*

Vector fields commonly arise from the definition of time-invariant systems. A time-invariant system defined by the ODE $\frac{dx_1}{dt} = f_1(\boldsymbol{x}), \ldots, \frac{dx_n}{dt} = f_n(\boldsymbol{x})$ can be identified with the vector field $\mathfrak{F}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x}))$. Therefore, a continuous system $\mathcal{S} : \langle \mathfrak{F}, X \rangle$ is defined by a tuple consisting of the vector field $\mathfrak{F}$ and a domain (also referred to as a *mode invariant*) $X \subseteq \mathbb{R}^n$. We now define the time trajectories of a continuous system:

Definition 2 (Time Trajectories). *A time trajectory of a continuous system $\mathcal{S} : \langle \mathfrak{F}, X \rangle$ is a function $\tau : [0, T) \mapsto \mathbb{R}^n$ for some $T > 0$, such that: $\tau(t) \in X$, for all $t \in [0, T)$ and $\frac{d\tau}{dt} = \mathfrak{F}(\tau(t)),\ \forall\ t \in [0, T)$.*

Note 1. To facilitate presentation, we have (deliberately) restricted our attention to time-invariant and autonomous systems. The full generalization to time variant, non-autonomous systems will be presented in an extended version.

If the continuous system $\mathcal{S}$ is defined by a *Lipschitz continuous* vector field $\mathfrak{F}$, then for any $\boldsymbol{x}_0 \in X$, we can guarantee the existence of a unique time trajectory τ such that $\tau(0) = \boldsymbol{x}_0$ [25]. Henceforth, we will assume that the systems considered are defined by Lipschitz continuous vector fields. An affine system $\mathcal{S}$ is a continuous system whose dynamics are defined by an affine vector field $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} + \boldsymbol{b}$. If $f(\boldsymbol{x})$ is continuous and differentiable over $\boldsymbol{x}$ then we write $\partial_{\boldsymbol{x}} f$ to denote the vector of its partial derivatives w.r.t each $\boldsymbol{x}_i$. The *Lie derivative* of a function g with respect to a field $\mathfrak{F}$ is given by $\mathcal{L}_F(g) := \ (\partial_{\boldsymbol{x}} g) \ \cdot \ \mathfrak{F}(\boldsymbol{x})$, where '$\cdot$' computes the dot product of two vectors.

Positive Invariant Set: A set $M \subseteq X$ is an invariant set for the system $\mathcal{S}$ iff for any $\boldsymbol{x} \in M$, every time trajectory $\tau : [0, T) \mapsto X$ such that $\tau(0) = \boldsymbol{x}$ is entirely contained in M; that is, $(\forall\ t \in [0, T))\ \tau(t) \in M$.

Let M be a closed set defined by the assertion $\bigwedge_{j=1}^{m} g_j(\boldsymbol{x}) \leq 0$ for some finite m. For technical reasons, we assume that each $g_j(\boldsymbol{x})$ is continuous and differentiable, and M is a "practical set" satisfying the constraint qualification (Cf. Blanchini & Miani [6], page 104):

$$(\forall\ \boldsymbol{x} \in X),\ (\exists \boldsymbol{z})\ g_j(\boldsymbol{x}) + \partial_{\boldsymbol{x}} g_j \cdot\ \boldsymbol{z} < 0\,. \qquad (1)$$

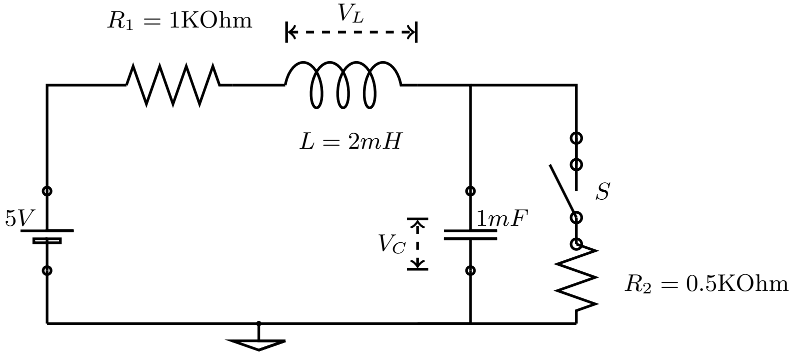


Fig. 1. Circuit diagram for an LCR circuit with a voltage-controlled switch S

Informally, the constraint qualifications ensure that $\bigwedge_j g_j(\boldsymbol{x}) < 0$ represents the (relative) interior of the set M and $\bigvee_j g_j(\boldsymbol{x}) = 0$ represents the boundary. It can be shown that all affine functions g_j and positive-semidefinite quadratic forms (defining n-dimensional ellipsoids) satisfy these conditions.

Theorem 1. *The set $M : \bigwedge_{j=1}^{m} g_j(\boldsymbol{x}) \leq 0$ is a positive invariant for the vector field $\mathfrak{F}$ if for each $j \in [1, m]$ the following assertion holds true: $\forall\ \boldsymbol{x} \in X : g_j(\boldsymbol{x}) = 0\ \wedge\ \bigwedge_{i \neq j} g_i(\boldsymbol{x}) \leq 0\ \Rightarrow\ \mathcal{L}_F(g_j) < 0$.*

The theorem states that under appropriate conditions, a closed set M is a positive invariant set if the vector field $\mathfrak{F}$ lies in the tangent cone at each point on the boundary of the set. It is a direct consequence of Nagumo's theorem, a more general result that holds for non-Lipschitz continuous dynamics and non-"practical" sets as well. The theorem above provides a basis for various techniques for generating invariants for continuous systems using quantifier elimination and constraint solving [32,36,19,29].

Hybrid Systems: Hybrid systems combine the continuous evolution of state with discrete, instantaneous jumps that can alter the state as well as the dynamics of a system [22].

Definition 3 (Hybrid System). *A hybrid system $\mathcal{H}$ is defined by a set of* discrete modes $\langle m_1, \ldots, m_k \rangle$, *wherein, each mode m_i is defined by a continuous system $\mathcal{S}_i : \langle \mathfrak{F}_i, X_i \rangle$. The system can change modes through a set of* discrete transitions $\tau_1, \ldots, \tau_m$. *Each transition is defined by a prior mode m_0, a post-mode m_1 and a transition relation $\rho[\boldsymbol{x}, \boldsymbol{x}'] \subseteq X_{m_0} \times X_{m_1}$, that relates the state $\boldsymbol{x} \in X_{m_0}$ before the transition to the state $\boldsymbol{x}' \in X_{m_1}$ obtained as a result of taking the transition. The initial conditions are given by the initial mode m_{init} with the initial state set $\Theta \subseteq X_{\mathsf{init}}$.*

A hybrid system is a *switched system* if each discrete transition of the system does not modify the continuous state variables. In other words, each discrete transition relation $\rho[\boldsymbol{x}, \boldsymbol{x}']$ can be written as $\rho : \gamma(\boldsymbol{x})\ \wedge\ \boldsymbol{x}' = \boldsymbol{x},$ for guard $\gamma(\boldsymbol{x})$.

Example 1 (Switched system). Figure 1 shows the circuit diagram for a voltage-controlled switch that closes whenever the voltage across the capacitor (V_C) exceeds $4V$, and opens whenever V_C goes below $1V$.

With the switch S open, the dynamics of the voltage across capacitor V_C and the voltage across the inductor V_L are given by $\frac{dV_C}{dt} = 5 - V_C - V_L, \quad \frac{dV_L}{dt} = -5 + V_C - V_L$. Likewise, with the switch S closed, the dynamics of the voltage across the inductor are given by $\frac{dV_C}{dt} = 5 - 3V_C - V_L, \quad \frac{dV_L}{dt} = -5 + 3V_C - V_L$. In each mode, we assume the mode invariant $(V_C, V_L) \in [-10, 10] \times [-10, 10]$.

3 Relational Abstractions

We define relational abstractions for continuous systems, and present proof rules for checking that a given relation is an abstraction of the time trajectories of a continuous system defined by ODEs.

Let $\mathcal{S} : \langle \mathfrak{F}, X \rangle$ be a continuous system defined by the vector field $\mathfrak{F}$, and domain (invariant) X. It is assumed that $\mathcal{S}$ arises from a mode of a larger hybrid system. Let $R(\boldsymbol{x}, \boldsymbol{y})$ be a relation over $X \times X$.

Definition 4 (Relational Abstraction Without Time). *The relation $R \subseteq \mathbb{R}^{2n}$ is a (timeless) relational abstraction of a continuous system $\mathcal{S}$ if for all time trajectories $\tau : [0, T) \mapsto X$ of the system $\mathcal{S}$, it is the case that $(\forall\ t \in [0, T))\ (\tau(0), \tau(t)) \in R$.*

Thus, for a time invariant system, a relational abstraction R captures all pairs of states $(\boldsymbol{x}, \boldsymbol{y})$ such that it is possible to reach $\boldsymbol{y}$ from $\boldsymbol{x}$ in a finite amount of time by evolving according to the dynamics of the system.

A relational abstraction $R \subseteq X \times X$ is said to be *complete* for a system $\mathcal{S}$ if whenever $R(\boldsymbol{x}, \boldsymbol{y})$ holds, there exists a time trajectory $\tau : [0, T) \mapsto X$ such that $\tau(0) = \boldsymbol{x}$ and $\tau(t) = \boldsymbol{y}$, for some time $0 \leq t < T$. Likewise, a relational abstraction R is linear if it can be expressed as an assertion in the theory of linear arithmetic over reals.

Note 2. A continuous system whose dynamics are defined by constants (such as a mode of a multirate hybrid automaton) has a complete, linear relational abstraction. For instance, the evolution of the ODE $\frac{dx}{dt} = 2, \ \frac{dy}{dt} = -3$ can be abstracted by the relation $R(x, y, x', y') :=\ x' - x \geq 0 \ \wedge\ \frac{1}{2}(x' - x) = \frac{-1}{3}(y' - y)$. In fact, we can show that hybrid systems with constant dynamics in each mode are bisimilar to a purely discrete transition system through relationalization. On the other hand, linear vector fields can fail to have complete abstractions.

We now define an "extended system" $\mathcal{S}'$ from a given system $\mathcal{S}$ such that invariants of $\mathcal{S}'$ will yield relational abstractions for $\mathcal{S}$.

Definition 5 (Extended System). *Let $\mathcal{S}$ be a continuous system over $\boldsymbol{x} \in \mathbb{R}^n$ defined by vector field $\mathfrak{F}$ and invariant region X. The extended system $\mathcal{S}'$ has state variables $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^{2n}$ with the dynamics.*

$$\frac{d\boldsymbol{y}}{dt} = \mathfrak{F}(\boldsymbol{y}), \ \frac{d\boldsymbol{x}}{dt} = \boldsymbol{0}\,, \tag{2}$$

invariant region given by $X \times X$ and with the initial conditions $\boldsymbol{x}(0) = \boldsymbol{y}(0) \in X$.

We now refine the notion of positive invariants from Section 2 to account for the presence of initial conditions in the system.

Definition 6 (Initialized Positive Invariant). *A set M is an initialized positive invariant for the system $\mathcal{S}$ with initial conditions $X_0 \subseteq M$ iff for all time trajectories $\tau : [0, T) \mapsto \mathbb{R}^n$ of $\mathcal{S}$ starting from $\tau(0) \in X_0$ we have $\tau(t) \in M$ for all $t \in [0, T)$.*

An initialized positive invariant is an over-approximation of all states reachable through a time trajectory starting from some pre-specified set of initial states. This is, in fact, the true analog of an invariant for a program.

Note that every positive invariant set M (following the definition in Section 2) that contains the initial set X_0 is an initialized positive invariant. On the other hand, an initialized positive invariant may not be a general positive invariant. This is because, it may be possible for trajectories that start from some state in the set $M - X_0$ to exit the invariant set M.

Lemma 1. *A relation R is a relational abstraction of $\mathcal{S}$ if and only if R is an* initialized *positive invariant for $\mathcal{S}'$.*

Proofs are provided in an extended version of the paper.

Therefore, if we can compute initialized positive invariants of the extended system $\mathcal{S}'$ with initial states $\boldsymbol{x}(0) = \boldsymbol{y}(0)$, we may use them to obtain relational abstractions. In this work, we use various techniques that can compute positive invariants M (using the definition in Section 2) of systems $\mathcal{S}$ that contain some initial set of states X_0.

Theorem 2. *Let M be a positive invariant of the extended system $\mathcal{S}'$ containing the initial states $X_0 = \{(\boldsymbol{x}, \boldsymbol{x}) \mid \boldsymbol{x} \in X\}$. Then M is a relational abstraction of the system $\mathcal{S}$.*

Proof. We note that a positive invariant M containing the initial states X_0 is also an initialized positive invariant. I.e, for any trajectory σ starting from X_0, we know that $\sigma(t) \in M$ since $\sigma(0) \in M$. The rest follows from Lemma 1.

The converse of the theorem above does not hold, in general. As discussed above, a positive invariant M containing the initial set of states X_0 is not necessarily an initialized positive invariant.

Note 3. The extended system can be expressed, equivalently, using the (time reversed) system instead of the system (2),

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{0},\ \frac{d\boldsymbol{x}}{dt} = -\mathfrak{F}(\boldsymbol{x}) \tag{3}$$

with the initial conditions $\boldsymbol{x}(0) = \boldsymbol{y}(0)$.

In other words, a relational abstraction $R(\boldsymbol{x}, \boldsymbol{y})$ is a positive invariant of one of two dynamical systems: System 2 where $\boldsymbol{x}$ is frozen in time and $\boldsymbol{y}$ evolves according to the vector field $\mathfrak{F}$, and System 3 where $\boldsymbol{y}$ is frozen in time and $\boldsymbol{x}$ evolves according to the time reversed field $-\mathfrak{F}$.

Proof Rule for Relational Abstractions: The proof rule for relational abstractions can be derived from the proof rule for invariant sets. Furthermore, techniques for synthesizing invariants can be directly used to synthesize relational abstractions. We now present a proof rule for checking if a relation R is a sound abstraction. We assume that the relation R is specified as an assertion of the form

$$R(\boldsymbol{x},\boldsymbol{y}):\ g_1(\boldsymbol{x},\boldsymbol{y}) \leq 0\ \wedge\ \ldots\ \wedge\ g_m(\boldsymbol{x},\boldsymbol{y}) \leq 0\,,$$

wherein $g_1,\ldots,g_m$ are continuous and differentiable functions over $\mathbb{R}^{2n}$. Furthermore, for technical reasons, we assume that the set $R \cap (X \times X)$ in $\mathbb{R}^{2n}$ defined by the relation R restricted to X is a *closed* set and g_j satisfy the constraint qualifications in (1).

Definition 7. *The following rules allow us to conclude that the relation R, as specified above, is a relational abstraction of a continuous system $\mathcal{S}$:*

Initialization: $\forall\ \boldsymbol{x}\ \in\ X,\ R(\boldsymbol{x},\boldsymbol{x})$*, and*
Flow Preservation: *We may use the rule for forward time:*

$$\forall j \in [1,m],\ \forall\ \boldsymbol{x},\boldsymbol{y}\ \in\ X,\ \bigwedge_{i \neq j} g_i(\boldsymbol{x},\boldsymbol{y}) \leq 0 \wedge g_j(\boldsymbol{x},\boldsymbol{y}) = 0\ \Rightarrow\ (\partial_{\boldsymbol{y}} g_j) \cdot \mathfrak{F}(\boldsymbol{y}) < 0\,,$$

or the rule for time reversed dynamics:

$$\forall j \in [1,m], \forall \boldsymbol{x},\boldsymbol{y} \in X, \bigwedge_{i \neq j} g_i(\boldsymbol{x},\boldsymbol{y}) \leq 0 \wedge g_j(\boldsymbol{x},\boldsymbol{y}) = 0 \Rightarrow (\partial_{\boldsymbol{x}} g_j) \cdot (-\mathfrak{F}(\boldsymbol{x})) < 0\,.$$

Example 2. We now consider relationalizations for the inductor-capacitor-resistor (LCR) circuit in Example 1. Consider the mode when the switch is open with dynamics given by $\frac{dV_C}{dt} = 5 - V_C - V_L,\quad \frac{dV_L}{dt} = -5 + V_C - V_L$.

We wish to show that the relation $R(V_{C0}, V_{L0}, V_C, V_L)$, represented by the assertion below, is a relational abstraction: $(V_{C0}, V_C, V_{L0}, V_L) \in [-10, 10]^4\ \wedge\ V_C + 5V_L \leq V_{C0} + 50\ \wedge\ 4V_L \leq V_{L0} + 30\ \wedge\ 2V_L - 3V_C \leq 2V_{L0} + 30$.

Let us consider the inequality $V_C + 5V_L - V_{C0} - 50 \leq 0$. For the initial condition, we set $V_C = V_{C0}$ and $V_L = V_{L0}$ and verify that $5V_{L0} \leq 50$ holds over the invariant region $(V_{C0}, V_{L0}) \in [-10, 10]^2$. Likewise, the Lie derivative of the left-hand side expression is given by $4V_C - 6V_L - 20$. We verify the following entailment using an SMT solver

$$R(V_{C0}, V_{L0}, V_C, V_L)\ \wedge\ V_C + 5V_L - V_{C0} - 50 = 0\ \models\ 4V_C - 6V_L - 20 < 0\,.$$

The remaining constraints are similarly verified.

Disjunctive Relational Abstraction: Often, the relational abstraction can be represented as the disjunction $R(\boldsymbol{x},\boldsymbol{y}):\ \bigvee_{j=1}^{m} R_j(\boldsymbol{x},\boldsymbol{y})$ of finitely many relations $R_1,\ldots,R_m$, such that (a) each relation R_j is represented by an assertion over $\boldsymbol{x},\boldsymbol{y}$ satisfying the flow preservation proof rule in Definition 7, and (b) the disjunctive relation $R(\boldsymbol{x},\boldsymbol{y})$ satisfies the initialization rule.

Example 3. Consider, once again, the LCR circuit in Example 1. The relation below is a disjunctive relational abstraction for the switch open mode:

$$|V_L| \leq \max\left(|V_{L0}|, |V_{C0} - 5|\right) \;\wedge\; |V_C - 5| \leq \max\left(|V_{L0}|, |V_{C0} - 5|\right) .$$

Verifying this fact can be performed by expanding the definitions of max and $|\cdot|$. The resulting assertion is cast in the disjunctive normal form, and the flow preservation proof rule in Definition 7 can be checked for each disjunct. The initialization rule can be checked for the whole disjunction.

3.1 Relational Abstractions of Hybrid Systems

A relational abstraction of a hybrid system is constructed by replacing each constituent continuous system by its relational abstraction and keeping the discrete transitions unchanged. Specifically, if $\mathcal{H}$ is a hybrid system (Definition 3) with k modes and n real-valued variables, then the relational abstraction of $\mathcal{H}$ is a state transition system over the state space $\{1, \ldots, k\} \times \mathbb{R}^n$ whose transition relation is the union of the discrete transitions of $\mathcal{H}$ and the relational abstractions of the k modes of $\mathcal{H}$.

Several classes of hybrid automata, such as timed automata and linear hybrid automata, have complete relational abstractions. Since the discrete transitions are not abstracted, we only need to ensure that the relational abstraction of the continuous dynamics are complete.

Timed Automata: The continuous dynamics of a timed automata with n clocks, $x_1, \ldots, x_n$, can be abstracted by the relation $\wedge_{i=2}^{n}(x_1 - x_{10} = x_i - x_{i0}) \wedge x_1 \geq x_{10}$. It is easy to check that this is a complete abstraction.

Multirate Automata: The continuous dynamics defined by ODEs $\frac{dx_1}{dt} = c_1, \ldots, \frac{dx_n}{dt} = c_n$, where $c_1, \ldots, c_n$ are nonzero constants, can be abstracted by the relation $\wedge_{i=2}^{n}(\frac{x_1 - x_{10}}{c_1} = \frac{x_i - x_{i0}}{c_i}) \wedge \frac{x_1 - x_{10}}{c_1} \geq 0$. Again, it is easy to check that this is a complete abstraction. Note that the result for timed automata is obtained as a special case where all c_i are 1.

Rectangular Automata: In rectangular automata, the dynamics in each mode are specified as $a_1 \leq \frac{dx_1}{dt} \leq b_1, \ldots, a_n \leq \frac{dx_n}{dt} \leq b_n$. Assuming $0 < a_i \leq b_i$ for all i, these dynamics can be abstracted by the relation

$$0 \leq \max\left(\frac{x_1 - x_{10}}{b_1}, \ldots, \frac{x_n - x_{n0}}{b_n}\right) \leq \min\left(\frac{x_1 - x_{10}}{a_1}, \ldots, \frac{x_n - x_{n0}}{a_n}\right)$$

Again, it is easy to check that this is a complete abstraction.

Linear Hybrid Automata: In linear hybrid automata, the dynamics in each mode are specified as a linear constraint $\phi(\dot{\boldsymbol{x}})$ over the dotted variables $\dot{\boldsymbol{x}}$. Without loss of generality, we can restrict ϕ to be of the form $A_1\dot{\boldsymbol{x}} \leq \boldsymbol{b}_1 \wedge A_2\dot{\boldsymbol{x}} \geq \boldsymbol{b}_2$, where A_1, A_2 are $n \times n$ rational matrices and $\boldsymbol{b}_1, \boldsymbol{b}_2$ are $n \times 1$ vectors consisting of *positive* rationals. These dynamics can be abstracted by the relation

$$0 \leq \max(A_1(\boldsymbol{x} - \boldsymbol{x}_0)./\boldsymbol{b}_1) \leq \min(A_2(\boldsymbol{x} - \boldsymbol{x}_0)./\boldsymbol{b}_2)$$

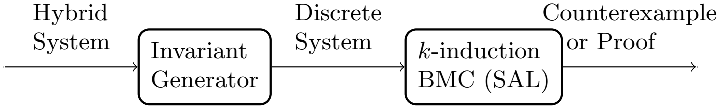


Fig. 2. Framework for implementing a safety verification engine using relationalization

where ./ is componentwise division. This is again a complete abstraction. Note that the result for rectangular automata is obtained as a special case where A_1, A_2 are identity matrices.

Linear Systems with Rational Eigenvalues: First consider the linear system $\frac{d\boldsymbol{x}}{dt} = D\boldsymbol{x}$, where D is a diagonal matrix with rational entries $\lambda_1, \ldots, \lambda_n$. For simplicity assume $\lambda_i \neq 0$ for all i. Since they are rational, the λ_i's can be written as integer multiples of some base rational λ, that is, $\lambda_1 = c_1\lambda, \ldots, \lambda_n = c_n\lambda$ for some rational λ and *integers* $c_1, \ldots, c_n$. A complete relational abstraction of the linear system is given by

$$(\exists r > 0) \bigwedge_{i=1}^{n} [x_i = r^{c_i} x_{i0}]$$

If A is not diagonal, but diagonalizable with all rational eigenvalues, then a relational abstraction can be obtained in the same way after doing a change of variables transformation. If A is nilpotent, then again the linear system can be shown to have a complete relational abstraction.

4 Implementation

Figure 2 shows the overall verification framework. It consists of two parts: (a) an invariant generator for generating the relational abstraction of the input hybrid system, and (b) a verifier for analyzing the relational abstraction using techniques such as k-induction and bounded model checking (BMC). Note that other verification techniques/tools are equally applicable here. Our framework abstracts each mode up front. It is possible, in practice, to implement the abstraction on-the-fly, whenever a previously unseen mode is entered.

We now discuss the implementation of relational abstraction, restricting our attention here to techniques that have been employed in our experiments. We primarily apply template-based methods for generating relational abstractions [9,19]. Template-based techniques formulate an unknown parameterized form for the required invariant and cast the problem of generating the invariant as an $\exists\forall$ formula. These $\exists\forall$ formulas can be solved directly using quantifier elimination techniques over the theory of reals [42,8], or they can be first converted into $\exists$ formulas through dualization. The $\exists$ formulas, which contain nonlinear constraints over the unknown parameters, can be solved using either fixed point iteration over cones [36], or using bit-vector solvers [19], or by simulating the system numerically [20]. In our experiments, we use a specialized quantifier elimination technique [38] and the tool TimePass, which implements a fixed point iteration with widening over polyhedral cones for affine ODEs [36]. We consider three types of abstractions *affine*, *eigen*, and *box*.

Affine abstractions: Affine abstractions employ the template: $\boldsymbol{a} \cdot \boldsymbol{x} + \boldsymbol{b} \cdot \boldsymbol{x}_0 \geq a_0$. In practice, the template: $\boldsymbol{a}(\boldsymbol{x} - \boldsymbol{x}_0) \geq a_0$ suffices after taking the initiation into account. Affine relational abstractions are computationally inexpensive to generate, but they are also of relatively poor quality.

Eigen abstractions: For linear systems, such as $d\boldsymbol{x}/dt = A\boldsymbol{x}$, whenever A has real eigenvalues, useful relational abstractions can be generated using the eigenvectors of A^T corresponding to those real eigenvalues [39]. Here, A^T denotes the transpose of matrix A. Specifically, if $\boldsymbol{c}$ is such that $A^T\boldsymbol{c} = \lambda\boldsymbol{c}$, then by simple algebraic manipulation, we obtain $\frac{d}{dt}(c_1x_1 + \ldots + c_nx_n) = \lambda(c_1x_1 + \ldots + c_nx_n)$ where $\boldsymbol{c} := [c_1; \ldots; c_n]$ and $\boldsymbol{x} := [x_1; \ldots; x_n]$. Let p denote the linear expression $c_1x_1 + \ldots + c_nx_n$ and let p_0 denote the linear expression $c_1x_{10} + \ldots + c_nx_{n0}$. Here, x_{i0} denotes the old value of x_i. If $\lambda < 0$, then we know that the value of p approaches zero monotonically. Consequently, we get the relational abstraction $(p_0 < 0 \Rightarrow p_0 \leq p < 0) \ \wedge \ (p_0 > 0 \Rightarrow p_0 \geq p > 0)$. Similarly, we can write the relational invariants for the case when $\lambda > 0$ and $\lambda = 0$.

Box abstractions: Box relational abstractions are Boolean combinations of affine relational invariants of the form

$$max(a_1|x_1|, \ldots, a_n|x_n|) \leq max(a_1|x_{10}|, \ldots, a_n|x_{n0}|)$$

where a_i's are unknown nonnegative real numbers. Discovering appropriate values for a_is does not require expensive quantifier elimination. We can find box relational invariants in $O(n^3)$ time. Box invariants do not always exist: sufficient (and necessary) conditions for their existence are known [1]. Example 3 shows a box invariant for the switch open mode.

5 Experimental Evaluation

We evaluate our approach over the navigation benchmarks [14], to experimentally evaluate the usefulness of relational abstractions for verifying hybrid systems. The navigation benchmarks model a vehicle moving in a 2-dimensional rectangular space $[0, m-1] \times [0, n-1]$. This space is partitioned in $m \times n$ cells. Let x, y denote the position of the vehicle and v_x, v_y denote its velocity. Then the dynamics of the vehicle in any particular cell are given by the ODEs:

$$\begin{array}{ll} \frac{dx}{dt} = v_x & \frac{dv_x}{dt} = a_{11}(v_x - b) + a_{12}(v_y - c) \\ \frac{dy}{dt} = v_y & \frac{dv_y}{dt} = a_{21}(v_x - b) + a_{22}(v_y - c) \end{array}$$

where the matrix $A := [a_{11}, a_{12}; a_{21}, a_{22}]$ and the direction (b, c) are parameters that can potentially vary (for each of the cells)[2].

Every benchmark in the suite is specified by fixing the matrix A, the number of cells $m \times n$, the direction (b, c) in each cell, and initial intervals for each

[2] The matrix A is Hurwitz: the dynamics for (v_x, v_y) asymptotically converge to (b, c).

Table 1. Comparison of various abstractions over the NAV benchmarks. All experiments were performed on an Intel Xeon E5630 2.53GHz single-core processor (x86_64 arch) with 4GB RAM running Ubuntu Linux 2.6.32-26. Legend — **depth:** k-induction depth, **time:** time taken by verifier, **status: P**: Proved Property, **CE:** k-induction base case fails and counterexample is produced, **F**: inductive step fails, no proofs or counterexample. **Note:** Relational eigeninvariants are inapplicable for nav07, nav08 (indicated by -). k-induction timings reported as $t_1 + t_2$ indicate that an auxiliary lemma was used. t_1 is the time to prove the property, and t_2 to discharge the lemma.

Benchmark	Affine Invs			Affine+Eigen Invs			Affine+Eigen+Box Invs		
	depth	status	time(s)	depth	status	time(s)	depth	status	time(s)
nav01	4	F	0.63	4	F	0.88	4	F	1.91
nav01	5	P	0.75	5	P	0.91	5	P	1.36
nav02	4	F	0.64	4	F	0.87	4	F	1.8
nav02	5	P	0.68	5	P	1.04	5	P	3.33
nav03	4	F	0.60	4	F	0.91	4	F	1.72
nav03	5	P	0.67	5	P	1.05	5	P	2.7
nav04	3	CE	0.49	8	F	3.21	8	F	34.883
nav04				4	P	0.75+0.99	4	P	0.98+2.21
nav05	2	CE	0.47	8	F	3.85	8	F	37.31
nav05				8	P	2.15+2.50	8	P	5.38+11.05
nav06	4	CE	0.61	8	F	18.01	8	F	494.5
nav06*	4	CE	1.03	8	P	21.80+7.42	8	P	40.22+35.08
nav07	5	CE	0.66	-	-	-	5	F	69.9
nav07				-	-	-	6	P	6.25
nav08	4	CE	0.52	-	-	-	6	CE	0.95
nav09	4	CE	0.57	4	CE	1.45	4	CE	19.87
nav10	3	CE	0.44	3	CE	0.99	3	CE	0.95

of the four state variables x, y, v_x, v_y. Our experiments focus on proving the unreachability of a distinct cell marked B for each benchmark instance [14].

In our experiments, we verify the safety property for the navigation benchmarks using k-induction over the relational abstraction. We use the SAL infinite bounded model checker, with the k-induction flag turned on (`sal-inf-bmc -i`), which uses the SMT solver Yices in the back end. Table 1 reports the results. For each benchmark, we report the depth used for performing k-induction (under "depth"), the output of k-induction (under "status"), and the time it took (under "time"). There are three possible outputs: (a) the base case of k-induction fails and a counterexample is found (denoted by "CE"), (b) the base case is proved, but the induction step fails; i.e., no counterexample is found, but no proof is found either (denoted by "F"), (c) the base case and the induction step are successfully proved (denoted by "P"). Since we perform k-induction on an abstraction, the counterexamples may be spurious, but the proofs are not. As Table 1 indicates, relational abstractions are sufficient to establish safety of the benchmarks nav01–nav05, nav06*, and nav07. The system nav06* is the same as nav06 but with a slightly smaller set of initial states. However, the proof fails on nav06 and nav08–nav10. There are two reasons for failure: (a) poor quality of abstraction, which is reflected in entries "CE" in Table 1, and (b) inability to find suitable k-inductive lemmas. This happens in the case of nav06, where the proof fails without yielding a counterexample. As discussed in Section 4, we employed three kinds of relational abstractions for each mode: affine, eigen, and box. Table 1 also shows performance of each of these techniques.

Affine abstractions: In Table 1, Columns (2)–(4) report results on affine relational abstractions. We note that affine abstractions are sufficient to prove safety of benchmarks nav01–nav03, but they fail on all other benchmarks.

Eigen abstractions: The dynamics in each mode of the benchmarks nav01–nav06 and nav10 have negative real eigenvalues. In Table 1, Columns (5)–(7) present results using a relational abstraction obtained by combining affine and eigen abstractions. For nav04–nav06, the combination eliminates the spurious counterexamples. However, no such benefit is seen on nav08–nav10 benchmarks. The dynamics in benchmarks nav07–nav08 do not have any real eigenvalues.

Box abstractions: The dynamics of all modes of all benchmarks in Table 1 satisfy all the conditions for the existence of box invariants, which enables us to generate box relational invariants for each of them. Columns (8)–(10) report results using a relational abstraction obtained by combining affine, eigen, and box relational invariants. In the case of nav07, where there are no eigen invariants, addition of box invariants eliminated the counterexamples from the model and even enabled verification of safety using k-induction with depth 6. However, no such benefit is seen for benchmarks nav08–nav10. Also, when eigen invariants exist, then adding box invariants does not seem to improve the quality of abstraction. Note that the use of box invariants increases the time taken to perform k-induction: this is expected since box invariants have a complex Boolean structure, which increases the search space of the SMT solver.

Comparison with Other Tools: Comparing our timings with those reported in the literature for the very same benchmarks, especially previous work by one of the authors [35], we note that our techniques are at least an order of magnitude faster on the larger benchmarks (10s of seconds vs. $100s - 1000s$ of seconds using template-based flowpipes [35]). A detailed comparison will be made available in our extended version.

Disjunctive and conjunctive relational invariants: One plausible reason for the failure to prove nav08–nav10 benchmarks is that we do not consider invariants of richer Boolean structure, such as 2-disjunctive invariants of the form $p(\boldsymbol{x}_0) \geq 0 \Rightarrow p(\boldsymbol{x}, \boldsymbol{x}_0) \geq 0$. Even though eigen invariants have this form, there may be other invariants of this form that are not related to the eigenvectors of the A-matrix. We also do not consider conjunctive invariants of the form $p_1 \geq 0 \wedge p_2 \geq 0$. Note that $p_1 \geq 0$ and $p_2 \geq 0$ need not separately be inductive, but their conjunction could be inductive. For this reason, we often fail to find them by just considering templates for $p_1 \geq 0$ and $p_2 \geq 0$ separately.

Overcoming limitations of k-induction: Even if the relational abstractions are strong enough to rule out all unsafe behaviors, we may still fail to prove the system safe using k-induction. This will happen if the safety property is not k-inductive for any k. This is possibly the case for benchmarks nav04-nav06.

Table 2. Time (in milliseconds) to generate all affine equality, inequality and eigen relational invariants for all modes of all the benchmarks

Type of Relational Invariant	Time (no state invariant)	Time (with state invariant)
Affine Inequality	60ms	6740ms
Affine Equality + Eigen	70ms	340ms

However, we are able to successfully prove safety of nav04 and nav05 by using an auxiliary lemma. The auxiliary lemma was itself verified by k-induction again. For nav06, we are unable to find any suitable auxiliary lemma at this time.

Another plausible cause for the failure of k-induction is the introduction of spurious loops in the relational abstraction, where no such loops exist in the concrete system. Analysis of the counterexamples to the induction step in nav06 (generated by `sal-inf-bmc -i -ice`) strongly indicates this possibility.

One way to eliminate spurious loops in the abstract is based on assuming that the concrete system stays in a mode for some small, but fixed, amount of time. Under the assumption that the concrete system stayed in a mode for at least 0.1 second, we strengthened the affine invariants of nav06, allowing us to prove safety of nav06 (for a slightly smaller set of initial states than what is specified in the nav06 benchmarks). These results are reported in row nav06* in Table 1. We conjecture that this trick will eliminate all the spurious counterexamples in the other navigation benchmarks.

Quantifier elimination for generating relational invariants: The new redlog/qepcad combination [38] is quite effective in generating all the affine and eigen invariants used in our experiments. Table 2 provides the time taken by *all* runs of the quantifier elimination process to generate these invariants. We report times for two cases depending on whether we used a template of the form $\psi[\boldsymbol{x}_0] \Rightarrow R(\boldsymbol{x}_0, \boldsymbol{x})$, with a state invariant antecedent guarding the relation. The times are negligible since the benchmarks are 4-dimensional systems (they all involve only four real-valued variables) with relatively simple (linear) dynamics in each mode. As a final remark, note that quantifier elimination does not return specific values for the parameters, but constraints on the unknown parameters. We choose values by solving a satisfiability problem. In our examples, the constraints after elimination were simple enough to perform this step manually. The redlog files that were used to generate the relational invariants and SAL models of the relational abstraction are publicly available[3].

6 Conclusions

We have presented an approach for verifying hybrid systems based on relational abstractions. Relational abstractions can be constructed compositionally by abstracting each mode separately. Our initial results are quite encouraging. The technique successfully solves some of the standard benchmark examples an order of magnitude faster than symbolic model checkers. The abstractions can be

[3] `http://www.csl.sri.com/~tiwari/relational-abstraction/`

coarse, and k-induction itself can be challenging to apply on hybrid systems in practice. Our future work will focus on improving the speed and precision of relational abstraction generation to enable fast proofs for complex systems. We also wish to apply our techniques to nonlinear hybrid systems in order to derive linear arithmetic abstractions.

References

1. Abate, A., Tiwari, A., Sastry, S.: Box invariance in biologically-inspired dynamical systems. Automatica 45(7), 1601–1610 (2009)
2. Alur, R., Dang, T., Ivančić, F.: Counter-example guided predicate abstraction of hybrid systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
3. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Informatica 43, 451–476 (2007)
4. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: POPL, pp. 211–224. ACM, New York (2007)
5. Berz, M., Makino, K.: Performance of Taylor Model Methods for Validated Integration of ODEs. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 65–73. Springer, Heidelberg (2006)
6. Blanchini, F., Miani, S.: Set-Theoretic Methods in Control. Springer, Heidelberg (2008)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
8. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. Journal of Symbolic Computation 12(3), 299–328 (1991)
9. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
10. Colón, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
11. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Principles of Programming Languages, pp. 238–252 (1977)
12. Dang, T., Maler, O., Testylier, R.: Accurate hybridization of nonlinear systems. In: HSCC 2010, pp. 11–20. ACM, New York (2010)
13. Dang, T., Salinas, D.: Image Computation for Polynomial Dynamical Systems Using the Bernstein Expansion. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 219–232. Springer, Heidelberg (2009)
14. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
15. Frehse, G.: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. STTT 10(3) (June 2008)
16. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)

17. Guernic, C.L., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Analysis: Hybrid Systems 4(2), 250–262 (2010)
18. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI (2009)
19. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
20. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 262–276. Springer, Heidelberg (2009)
21. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. In: FMSD, vol. 11(2), pp. 157–185 (1997)
22. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE, Los Alamitos (1996)
23. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Transactions on Automatic Control 43, 540–554 (1998)
24. Kurzhanski, A.B., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 202–214. Springer, Heidelberg (2000)
25. Meiss, J.D.: Differential Dynamical Systems. SIAM publishers, Philadelphia (2007)
26. Mysore, V., Piazza, C., Mishra, B.: Algorithmic algebraic model checking II: Decidability of semi-algebraic model checking and its applications to systems biology. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 217–233. Springer, Heidelberg (2005)
27. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
28. Oishi, M., Mitchell, I., Bayen, A.M., Tomlin, C.J.: Invariance-preserving abstractions of hybrid systems: Application to user interface design. IEEE Trans. on Control Systems Technology 16(2) (March 2008)
29. Platzer, A., Clarke, E.: Computing differential invariants of hybrid systems as fixedpoints. Formal Methods in Systems Design 35(1), 98–120 (2009)
30. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE, Los Alamitos (2004)
31. Podelski, A., Wagner, S.: Model checking of hybrid systems: From reachability towards stability. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 507–521. Springer, Heidelberg (2006)
32. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
33. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 573–589. Springer, Heidelberg (2005)
34. Rushby, J., Lincoln, P., Owre, S., Shankar, N., Tiwari, A.: Symbolic analysis laboratory (SAL). Cf, http://www.csl.sri.com/projects/sal/
35. Sankaranarayanan, S., Dang, T., Ivančić, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008)
36. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Fixed point iteration for computing the time elapse operator. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 537–551. Springer, Heidelberg (2006)

37. 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)
38. Sturm, T., Tiwari, A.: Verification and synthesis using real quantifer elimination (2011) (submitted)
39. Tiwari, A.: Approximate reachability for linear systems. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 514–525. Springer, Heidelberg (2003)
40. Tiwari, A.: HybridSAL: A tool for abstracting HybridSAL specifications to SAL specifications (2007)
41. Tiwari, A.: Abstractions for hybrid systems. Formal Methods in Systems Design 32, 57–83 (2008)
42. Weispfenning, V. In: Applied Algebra and Error-Correcting Codes (AAECC)

Simplifying Loop Invariant Generation Using Splitter Predicates*

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken

Department of Computer Science
Stanford University
{sharmar,isil,tdillig,aiken}@cs.stanford.edu

Abstract. We present a novel static analysis technique that substantially improves the quality of invariants inferred by standard loop invariant generation techniques. Our technique decomposes *multi-phase* loops, which require disjunctive invariants, into a semantically equivalent sequence of single-phase loops, each of which requires simple, conjunctive invariants. We define *splitter predicates* which are used to identify phase transitions in loops, and we present an algorithm to find useful splitter predicates that enable the phase-reducing transformation. We show experimentally on a set of representative benchmarks from the literature and real code examples that our technique substantially increases the quality of invariants inferred by standard invariant generation techniques. Our technique is conceptually simple, easy to implement, and can be integrated into any automatic loop invariant generator.

Keywords: Static analysis, invariant generation, decomposition of multi-phase loops.

1 Introduction

A key problem in any automatic software verification system is the inference of loop invariants. A consistent theme in the literature is that most loops found in practice require only simple quantifier-free *conjunctive* invariants (i.e., invariants that are conjunctions of elementary facts) and that such invariants are relatively easy to infer using standard techniques such as [8,28,23].

However, some loops in real programs fundamentally require *disjunctive* invariants (i.e., an invariant with at least one disjunction). While relatively rare, obtaining accurate invariants for such loops is still important for successful verification, as imprecision in the analysis of even a small part of a program tends to spread, often reducing analysis precision for much of, or even the entire, program. As a result, a number of previous efforts have proposed techniques for inferring disjunctive invariants [19,5,20,18,1]. While there is considerable diversity in the approaches taken, the proposed disjunctive invariant generation techniques are considerably more involved than the more straightforward conjunctive case.

* This work was supported by NSF grants CNS-050955 and CCF-0702681.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 703–719, 2011.

To illustrate the problem of disjunctive invariant generation, consider Fig. 1(a), which is the motivating example of [19]. To prove the validity of the assertion, the following disjunctive invariant is required:

$$(x \leq 50 \wedge y = 50) \vee (50 \leq x \leq 100 \wedge y = x) \ . \tag{1}$$

Abstract interpretation-based techniques that generate only conjunctive invariants fail on this example. For instance, the widely-used abstract interpretation-based tool Interproc works over convex abstract domains [21] and computes invariants that are conjunctions of linear inequalities. For this loop, Interproc computes the post-condition $50y \geq 2599$ and cannot verify the assertion $y = 100$ tested in the last line of Fig. 1(a). Techniques such as [17] can infer disjunctive invariants, but for this example do so in a brute-force manner, performing 50 refinement iterations. Similarly, predicate abstraction techniques such as Slam [2] and Blast [3] generate a sequence of predicates of the form $x = 1, x = 2, \ldots$ during abstraction refinement and require 100 refinement iterations. Some much more elaborate techniques using interpolants [22] and probabilistic inference [19] can verify the correctness of this program without counting to the loop bound, but it is difficult to give an intuitive characterization of the class of loops for which these techniques can infer useful disjunctive invariants.

```
x=0;y=50;
while(x<100)
{
  x=x+1;
  if(x>50)
    y=y+1;
}
assert(y==100);
```

(a) Example from [19]

```
x=0;y=50;
while(x<=49)
{
  x=x+1;
}
while(x<100 && x>49)
{
  x=x+1;
  y=y+1;
}
assert(y==100);
```

(b) The example after splitting.

Fig. 1. Loop (a) requires a disjunctive invariant, but the equivalent program (b) requires only conjunctive invariants

While Fig. 1(a) is a synthetic example, it is representative of the loops found in practice that require disjunctive invariants. Specifically, this example has two important properties:

(a) The need for a disjunctive invariant arises from a particular conditional or conditionals in the loop body; in this case, the statement `if (x > 50)`. Not all conditionals imply that a disjunctive invariant is needed, but conditionals whose predicate is related to how many iterations the loop has executed, as in Fig. 1(a), usually do. For example, one of the more common patterns in practice is that the conditional causes the loop to do something different in its base (the first or first few iterations) and inductive cases (all subsequent iterations).

(b) The conditionals in question exhibit a fixed number of *phase transitions* during execution. A *phase* is a sequence of iterations in which the conditional, if it is evaluated, always evaluates to the same value, either *true* or *false*. A *phase transition* occurs when the conditional evaluates to b in one iteration, and the next time it is evaluated, it evaluates to $\neg b$. In Fig. 1, the conditional test `x > 50` has two phases and one phase transition: it is *false* for iterations 1-50, and *true* for iterations 51-100.

In principle, there are many loops requiring disjunctive invariants that do not satisfy conditions (a) and (b). However, it is our experience that the vast majority of loops arising in practice that require a disjunctive invariant do so because of conditionals with a fixed number of phases. For example, we have manually inspected the 95 loops found in OpenSSH, and found that exactly 9 of these loops require disjunctive invariants. Furthermore, of these 9 loops, all but one[1] satisfies conditions (a) and (b) above. Throughout this paper, we refer to loops satisfying conditions (a) and (b) as *multi-phase loops*.

The observation that multi-phase loops constitute a large majority of the loops that are not amenable to reasoning by standard invariant generation techniques motivates our approach: Rather than developing techniques to directly infer disjunctive invariants, we employ static analysis to identify phase transitions of multi-phase loops by computing *splitter predicates*. We then perform a program transformation that converts such multi-phase loops requiring disjunctive invariants to a semantically equivalent sequence of single-phase loops, each of which requires only conjunctive invariants. In general, if a loop has a conditional with k phases, it can be split into k successive loops without the conditional test.

As an example, consider again the loop from Fig. 1. Here, we can eliminate the phase transition by *splitting* the loop into two loops, one for each phase, as shown in Fig. 1(b). The resulting two loops have no conditionals and require only simple conjunctive invariants. Recall that the invariant for Fig. 1(a) is $(x \leq 50 \wedge y = 50) \vee (50 \leq x \leq 100 \wedge y = x)$. Here, the first disjunct, $x \leq 50 \wedge y = 50$, corresponds to the invariant of the first loop from Fig. 1(b), and the second disjunct, $50 \leq x \leq 100 \wedge y = x$, is the invariant of the second loop in Fig. 1(b). Furthermore, Interproc, which fails to verify the assertion for Fig. 1(a), easily discovers the loop invariants needed to prove the assertion for Fig. 1(b).

As this example illustrates, our approach effectively reduces the problem of inferring disjunctive invariants for a complex, multi-phase loop to the better understood problem of inferring conjunctive invariants for a sequence of single-phase loops. This strategy explicitly separates the task of identifying phase transitions from the inference of loop invariants, and allows standard invariant generation techniques to be successful for programs which previously might only be verified using much more sophisticated methods. Our technique is conceptually simple,

[1] This loop alternates its behavior from iteration to iteration.

easy to implement, and improves the quality of invariants discovered by a large class of invariant generation techniques.

1.1 An Overview of the Technique

Consider a loop $\texttt{while}(P)\{E[C]\}$ where E is an expression with one *hole* $[\cdot]$ for the predicate of an `if` statement, and C is the predicate plugged into the hole. For example, in Fig. 1(a), $P = \texttt{x} < \texttt{100}$, E `= x++; if(`$[\cdot]$`) y++;`, and $C = \texttt{x} > \texttt{50}$. We are interested in finding a predicate Q with two properties:

(a) Q should be a *splitter predicate*, which informally means that it can be used to divide the loop into two loops that execute one after the other.
(b) When Q (resp. $\neg Q$) is true on entry into the loop body, a particular conditional test C in the loop should be guaranteed to be *true* (resp. *false*).

If Q has both of these properties, which we formalize in Sect. 3, then the following semantic equivalence holds:

$$\texttt{while}(P)\{E[C]\} \equiv \texttt{while}(P \wedge \neg Q)\{E[\mathit{false}]\}; \texttt{while}(P \wedge Q)\{E[\mathit{true}]\} \ . \quad (2)$$

If we can find such a splitter predicate Q, then we can decompose the original loop into two loops, one in which the conditional's predicate is always *false* and the other in which it is always *true*. Constant folding then eliminates the conditionals, resulting in simpler loops.

Note that a predicate satisfying conditions (a) and (b) above identifies the loop iteration in which a phase transition occurs for C: When Q becomes *true*, the first of the split loops terminates, and this corresponds to the first iteration in which C evaluates to *true* in the original loop. Furthermore, observe that the the splitter predicate is, in general, different from the conditional test C. For example, $Q = \texttt{x} > \texttt{49}$ satisfies (2) for the program of Fig. 1(a).

It is straightforward to generalize (2) to transform loops with `if` conditions having any fixed number k of phase transitions into the composition of k loops. In this paper, we discuss only the case where the predicate of an `if` statement has at most one phase transition; besides being simpler to present, this case is also the only one we have thus far encountered in practice.

This paper makes the following contributions:

- We present a static analysis technique to decompose multi-phase loops requiring disjunctive invariants into a sequence of simpler single-phase loops, whose invariants can be inferred using standard techniques.
- We define phase splitter predicates, which are key to identifying phase transitions of multi-phase loops, and we present an algorithm for computing them.
- The proposed technique is simple to implement and relies only on SMT solvers already used in many verification systems.
- Our evaluation on a combination of representative examples from the literature and loops taken from real programs shows that our technique allows standard conjunctive invariant generation techniques to produce results comparable to some of the recently proposed advanced techniques for disjunctive invariant generation.

The rest of the paper is organized as follows: Section 2 presents a simple language for the formal development; Section 3 defines *splitter* and *phase splitter* predicates. Section 4 gives an algorithm for computing phase splitter predicates. Section 5 describes our prototype implementation, and Section 6 describes our experimental results. Section 7 surveys related work, and Section 8 concludes.

2 Language

Figure 2 gives the syntax of a simple imperative language we use for the formal development. We assume a family of integer-valued program variables x, y, z, $\dots$, a set of primitive relational operators (*RelOp*), and binary arithmetic operators (*BinOp*). We distinguish between a normal statement s and a *statement with one hole* h; the unique hole $[\cdot]$ in h indicates where a predicate can be inserted to complete the statement. If h is a statement with one hole and C is a predicate, then $h[C]$ is the statement (with no holes) formed by replacing the $[\cdot]$ in h by C. We omit the formal definition of the replacement operation, which is standard.

$$
\begin{array}{rcl}
s \in \mathit{Statement} & ::= & \texttt{skip} \mid x := e \mid \texttt{assert}(P) \\
& & \mid s; s \\
& & \mid \texttt{while}(P)\{s\} \\
& & \mid \texttt{if}(P)\ \{s\}\ \texttt{else}\ \{s\} \\
\\
P \in \mathit{Predicate} & ::= & \mathit{true} \mid \mathit{false} \mid e\ \mathit{RelOp}\ e \mid \neg P \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \\
\\
e \in \mathit{Iexpr} & ::= & \mathit{int} \mid x \mid x\ \mathit{BinOp}\ y \\
\\
h \in \mathit{StatementWithOneHole} & ::= & \mid h; s \mid s; h \\
& & \mid \texttt{if}([\cdot])\ \{s\}\ \texttt{else}\ \{s\} \\
& & \mid \texttt{if}(P)\ \{h\}\ \texttt{else}\ \{s\} \\
& & \mid \texttt{if}(P)\ \{s\}\ \texttt{else}\ \{h\}
\end{array}
$$

Fig. 2. The syntax of the language we use for the formal development

Figure 3 gives the small-step operational semantics for the language of Fig. 2. An environment E is a function from program variables to integers. Integer values are denoted by v. In each step we take a reducible expression, execute one step of computation, and possibly update E. Each transition maps an environment, statement pair $\langle E, s\rangle$ to a new pair $\langle E', s'\rangle$. The operational rules are standard; we note only that the rules for statement sequences always reduce the first statement $\langle E, s_1; s_2\rangle \to \langle E', s_1'; s_2\rangle$ until the first statement evaluates to `skip`, at which point the rule $\langle E, \texttt{skip}; s\rangle \to \langle E, s\rangle$ is applied to transfer control to the remainder of the statement sequence.

Definition 1 (Semantic Equivalence). *Two statements s_1 and s_2 are semantically equivalent, denoted $s_1 \equiv s_2$, if*

$$\forall E_1, E_2 \quad (\langle E_1, s_1\rangle \to^* \langle E_2, \texttt{skip}\rangle) \Leftrightarrow (\langle E_1, s_2\rangle \to^* \langle E_2, \texttt{skip}\rangle)$$

$$\frac{}{\langle E, \texttt{skip}; s\rangle \rightarrow \langle E, s\rangle}$$

$$\frac{}{\langle E, x := v\rangle \rightarrow \langle E[x \mapsto v], \texttt{skip}\rangle}$$

$$\frac{}{\langle E, x := y\rangle \rightarrow \langle E[x \mapsto E(y)], \texttt{skip}\rangle}$$

$$\frac{v = E(x)\ \mathit{BinOp}\ E(y)}{\langle E, z := x\ \mathit{BinOp}\ y\rangle \rightarrow \langle E[z \mapsto v], \texttt{skip}\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{assert}(P)\rangle \rightarrow \text{ABORT}}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{assert}(P)\rangle \rightarrow \langle E, \texttt{skip}\rangle}$$

$$\frac{\langle E, s_1\rangle \rightarrow \langle E_1, s_1'\rangle}{\langle E, s_1; s_2\rangle \rightarrow \langle E_1, s_1'; s_2\rangle}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{if}(P)\ \{s_t\}\ \texttt{else}\ \{s_f\}\rangle \rightarrow \langle E, s_t\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{if}(P)\ \{s_t\}\ \texttt{else}\ \{s_f\}\rangle \rightarrow \langle E, s_f\rangle}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{while}(P)\{s\}\rangle \rightarrow \langle E, s; \texttt{while}(P)\{s\}\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{while}(P)\{s\}\rangle \rightarrow \langle E, \texttt{skip}\rangle}$$

Fig. 3. The small-step operational semantics of the language from Fig. 2

3 Splitter Predicates

As discussed in Sect. 1, a key idea that allows our technique to identify phase transitions in loops is the concept of *splitter predicates*, which we define next.

Definition 2 (Splitter Predicate). A predicate Q is a *splitter predicate* for a loop $\texttt{while}(P)\{B\}$ if

$$\texttt{while}(P)\{B\} \equiv \texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\}$$

According to this definition, a predicate Q is a splitter predicate for a loop L if L can be decomposed as the sequence of two loops $L_1; L_2$ where $\neg Q$ always holds at the head of L_1 and Q always holds at the head of L_2. Thus, if Q is a splitter predicate, then Q's observed truth value changes at most once during the execution of L.

The following theorem describes how to verify whether a given predicate Q is a splitter predicate for a loop L by issuing a single query to a constraint solver:

Theorem 1. For a loop $L = \texttt{while}(P)\{B\}$, if Q satisfies the Hoare triple

$$\{P \wedge Q\}\ B\ \{Q \vee \neg P\}$$

then Q is a splitter predicate for L.

Proof. We claim the following:

$$\begin{array}{ll} \quad\texttt{while}(P)\{B\} & (a) \\ \equiv \texttt{while}(P)\{\texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\}\} & (b) \\ \equiv \texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\} & (c) \end{array}$$

For any predicate Q (whether Q is a splitter predicate or not), it is easily verified that loop (a) is equivalent to loop (b). Intuitively, loop (b) expresses that the truth-value of Q may in general change any number of times in the original loop. For the second step, if the outer loop executes its body either 0 or 1 times (i.e., Q's truth value changes at most once) then it is easy to check that (b) is equivalent to (c). Thus, it suffices to prove that the outer loop of (b) executes either 0 or 1 times if $\{P \wedge Q\}\, B\, \{Q \vee \neg P\}$. There are two cases:

- If $\neg P$ holds on entry to the outer loop, the outer loop executes its body 0 times.
- If P holds on entry to the outer loop, the outer loop's body is executed at least once. Clearly $\neg P \vee Q$ is a post-condition of the first inner loop. There are two cases on entry to the second inner loop:
    - If $\neg P$ holds, the second inner loop terminates without executing its body and then the outer loop terminates after one iteration.
    - Otherwise P holds, and therefore from the post-condition of the first inner loop we know Q also holds on entry to the second inner loop. Applying the assumption $\{P \wedge Q\}\, B\, \{Q \vee \neg P\}$, we conclude that Q is an invariant of the second inner loop whenever the second inner loop executes at least once. Since Q cannot become *false*, the second inner loop terminates only when $\neg P$ holds, and therefore the outer loop exits after completing one iteration.

Since the outer loop executes its body 0 or 1 times in all cases, (b) $\equiv$ (c), and therefore (a) $\equiv$ (c). Therefore, Q is a splitter predicate. □

Observe that not every splitter predicate is useful for the purpose of decomposing a loop into phases, because not every splitter identifies the phase transition associated with a conditional in the loop body. For example, in Fig. 1(a), $x > 60$ is a splitter predicate, as $x > 60$ is initially false, but stays true once it becomes true. On the other hand, $x > 60$ is not a useful splitter because it does not exactly split the loop into the two phases of the conditional test $x > 50$. We require a class of splitter predicates that satisfy a stronger condition:

Definition 3 (Phase Splitter Predicate). A splitter predicate Q that satisfies the additional requirement

$$\texttt{while}(P)\{E[C]\} \equiv \texttt{while}(P \wedge \neg Q)\{E[\mathit{false}]\}; \texttt{while}(P \wedge Q)\{E[\mathit{true}]\}$$

is called a *phase splitter predicate.*

According to this definition, a phase splitter predicate Q for a loop L decomposes the loop into two loops $L_1; L_2$, where both L_1 and L_2 have fewer branches in the loop body. Since a phase splitter predicate eliminates a conditional C in the original loop body, there is a relationship between the phase splitter for loop L and the conditional C. We now make this relationship precise.

Definition 4. Consider a loop $\texttt{while}(P)\{B[C]\}$. We define $\overline{B}$, the code that executes before the hole in B, by structural induction on B.

$$\begin{aligned}
\overline{\texttt{if}([\cdot])\,\{s\}\ \texttt{else}\ \{s\}} &= \texttt{skip}\\
\overline{\texttt{if}(P)\,\{h\}\ \texttt{else}\ \{s\}} &= \texttt{assert}(P);\overline{h}\\
\overline{\texttt{if}(P)\,\{s\}\ \texttt{else}\ \{h\}} &= \texttt{assert}(\neg P);\overline{h}\\
\overline{h;s} &= \overline{h}\\
\overline{s;h} &= s;\overline{h}
\end{aligned}$$

Note that if we allow holes inside `if` statements of nested loops, then the notion of code that executes before the hole is no longer straightforward. This is the primary reason for disallowing holes inside nested loops in the definition of *StatementWithOneHole* in Fig. 2.

Recall that our goal is to find a splitter predicate Q such that (i) if Q holds at the loop head, then the conditional C inside one `if` statement always evaluates to *true*, and (ii) if $\neg Q$ holds at the loop head, then the conditional C inside the same `if` statement always evaluates to *false*. The following lemma states the relationship between a predicate Q at the loop head and the conditional C in an `if` statement:

Lemma 1. Let Q be any predicate. Then,

$$\begin{aligned}
&\text{If } \{Q\}\,\overline{B}\,\{C\}, \text{ then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}\\
&\text{If } \{Q\}\,\overline{B}\,\{\neg C\}, \text{ then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{false}]\}
\end{aligned}$$

Proof. The proof of this lemma is given in the full version of the paper available at `http://www.stanford.edu/~isil/cav2011-full.pdf`. □

Just as Thm. 1 showed that we could use constraint solving techniques to determine whether Q is a splitter predicate, Lemma 1 shows that solving another constraint problem determines whether Q causes a conditional to have only one phase within the loop. Because we split the original loop into two loops, we must solve two constraint problems, one for each of the split loops, to ensure that the conditional in both loops can be eliminated. This leads us to the following theorem, which reduces the problem of checking phase splitter predicates to a constraint solving problem:

Theorem 2. Consider a loop $L = \texttt{while}(P)\{B[C]\}$ and let Q be a predicate such that

$$\{Q\} \quad \overline{B} \quad \{C\} \tag{3}$$

$$\{\neg Q\} \quad \overline{B} \quad \{\neg C\} \tag{4}$$

$$\{P \wedge Q\}\ B[C]\ \{Q \vee \neg P\} \tag{5}$$

Then Q is a phase splitter predicate.

Proof. By (5) and Thm. 1, Q is a splitter predicate for L. Hence $L \equiv L_1; L_2$ where $L_1 = \texttt{while}(P \wedge \neg Q)\{B[C]\}$ and $L_2 = \texttt{while}(P \wedge Q)\{B[C]\}$. By (4) and Lemma 1, $L_1 \equiv \texttt{while}(P \wedge \neg Q)\{B[\mathit{false}]\}$. By (3) and Lemma 1, $L_2 \equiv \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}$. Hence Q is a phase splitter predicate. □

4 Algorithm for Splitting

In Thm. 2 of the previous section, we showed how to check whether a predicate is a phase splitter, but we have not yet answered the question of how to find candidate splitter predicates. In this section, we discuss an algorithm for finding candidate phase splitter predicates and transforming a multi-phase loop into a sequence of simpler loops.

```
phase_split($L$)
 1:   foreach conditional test $C$ in $L = \texttt{while}(P)\{B[C]\}$ do
 2:     $Q = WP(\overline{B}, C)$
 3:     if $(\{\neg Q\}\ \overline{B}\ \{\neg C\}) \wedge (\{P \wedge Q\}\ B[C]\ \{Q \vee \neg P\})$ then
 4:       $L_1 = \texttt{while}(P \wedge \neg Q)\{B[\mathit{false}]\}$
 5:       $L_2 = \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}$
 6:       $B_1 = \texttt{phase\_split}(L_1)$
 7:       $B_2 = \texttt{phase\_split}(L_2)$
 8:       return $B_1; B_2$
 9:     endif
10:   done
11:   return $L$
```

Fig. 4. Phase splitting algorithm

Our algorithm considers loops in an inside-out fashion, starting with the innermost nested loops first. The pseudo-code for splitting a single loop is given in Fig. 4; in the figure, *WP* denotes a standard precondition computation. This precondition should be as weak as possible, and ideally it is the weakest precondition (hence *WP*) although in practice we must settle for a decidable approximation. Here, we repeatedly consider each `if` statement in the body of the outermost loop and attempt to find a splitter predicate for the `if`'s conditional test. Given the conditional C of some `if` statement, we first compute the precondition Q of C with respect to $\overline{B}$. Since Q is a precondition, it must satisfy condition (3) of Thm. 2. In our implementation we use a constraint solver to compute a precondition that is as weak as possible (see Sect. 5). We then explicitly check the other two conditions (4) and (5) of Thm. 2 to guarantee that we do not split the loop unless Q is a phase splitter predicate. If Q is indeed a predicate identifying phase transitions, then L is split into two loops L_1 and L_2 in lines 5 and 6 of Fig. 4. Since it may be possible to further decompose L_1 or L_2 into even simpler loops with fewer phases, we recursively invoke the `phase_split` algorithm, which transforms L_1 (resp. L_2) into a sequence of loops B_1 (resp. B_2). The final result of splitting L is then given by the sequence $B_1; B_2$.

Observe that the algorithm in Fig. 4 considers conditionals in the loop body in an arbitrary order. The reader might wonder whether the order in which conditionals are considered matters, as one order might yield a better decomposition than another. Fortunately, it turns out that the order in which we test the conditionals in the algorithm is irrelevant, because if C is a splitter predicate of the original loop, then it is guaranteed to remain a splitter predicate of the transformed loop. The following theorem makes this statement precise:

Theorem 3. If Q is a splitter predicate of $\texttt{while}(P)\{B\}$ satisfying the hypothesis of Thm. 1 and $P' \Rightarrow P$, then Q is a splitter predicate of $\texttt{while}(P')\{B\}$.

Proof. We show $\{P' \wedge Q\}\, B\, \{\neg P' \vee Q\}$. It then follows from Thm. 1 that Q is a splitter predicate for $\texttt{while}(P')\{B\}$. We reason as follows:

$$\begin{array}{l}(P' \Rightarrow P) \Rightarrow (Q \vee \neg P \Rightarrow Q \vee \neg P') \\ (P' \Rightarrow P) \Rightarrow (P' \wedge Q \Rightarrow P \wedge Q) \\ \ \{P \wedge Q\}\ B\ \{Q \vee \neg P\}\end{array}$$

Using Hoare's consequence rule we obtain $\{P' \wedge Q\}\, B\, \{Q \vee \neg P'\}$. Hence Q is a splitter predicate for $\texttt{while}(P')\{B\}$. □

Because the loop predicates in split loops are only stronger than the loop predicate of the original loop, Theorem 3 shows that if we have two splitters Q_1 and Q_2, then if we split on Q_1, Q_2 remains a splitter predicate for each of the new loops. Furthermore, it is easy to see from Thm. 2, that Q_2 still causes the same conditional(s) to be eliminated whether we split on Q_1 first or not. Thus, choosing one phase splitter predicate Q over another phase splitter predicate Q' cannot make further splitting by Q' illegal or vice versa.

As mentioned above, loops are split beginning with innermost loops and proceeding to outermost loops. The reason for selecting this order is that the weakest precondition of a code fragment containing a loop requires computing loop invariants. Thus, when splitting an outer loop, the weakest precondition computation may be required to compute loop invariants for any inner loops. By splitting the innermost loops first, the weakest precondition computation deals only with loops that have already been simplified as much as possible, making it easier to infer better invariants using standard techniques.

4.1 Revisiting the Running Example

We now illustrate the execution of our algorithm on the example from Fig. 1(a).

1. For this loop, we have:
$$\begin{array}{l}P = \texttt{x} \leq \texttt{99} \\ B = \texttt{x++;if([}\cdot\texttt{])y++} \\ C = \texttt{x} > \texttt{50} \\ \overline{B} = \texttt{x++}\end{array}$$
2. A candidate phase splitter predicate Q is computed as $Q = \mathit{WP}(\texttt{x} > \texttt{50}, \texttt{x++})$. Using a weakest precondition computation engine, we obtain $Q = x > 49$ as a candidate phase splitter predicate.
3. Now, we check whether the candidate predicate Q satisfies condition (4) of Thm. 2 by querying the validity of the formula $(\neg x > 49 \wedge x' = x + 1) \Rightarrow \neg x' > 50$, which is indeed valid.

4. Finally, we verify that candidate Q is a phase splitter predicate by checking the validity of the following constraint:

   $$(x > 49 \wedge x \leq 99 \wedge x' = x + 1 \wedge (x' > 50 \Rightarrow y' = y + 1)) \Rightarrow x' > 49 \vee \neg x' \leq 99$$

   This formula is valid, allowing us to perform the phase splitting transformation, which yields the decomposition shown in Fig. 1(b).

5 Implementation

We have implemented a prototype version of the algorithm described in this paper using the SAIL program analysis front-end [10] and the Mistral SMT solver [11,13] for a subset of the C programming language. The weakest precondition computation step in the algorithm is implemented by using the quantifier elimination capabilities of Mistral. More specifically, to compute the weakest precondition of C with respect to code fragment $\overline{B}$, we first convert $\overline{B}$ to single static assignment (SSA) form [9]. We then generate a constraint ϕ_s for any statement s in the following way: For each basic statement s (e.g., an assignment or assertion), we generate the corresponding atomic constraint in the constraint language, and a sequence $s_1; s_2$ is converted to the constraint $\phi_{s_1} \wedge \phi_{s_2}$ where ϕ_{s_1} and ϕ_{s_2} are the constraints derived from statements s_1 and s_2 respectively. For an `if` statement, `if(C) then s1 else s2`, we generate the constraint $(C \wedge \phi_{s_1}) \vee (\neg C \wedge \phi_{s_2})$.[2] To generate weakest preconditions for nested loops, we use a constraint obtained with the help of an invariant generation tool. Finally, we compute the weakest precondition of C with respect to $\overline{B}$ by computing the constraint $\phi_{\overline{B}}$ and then by existentially quantifying and eliminating all intermediate variables (i.e., variables with version number greater than one in SSA form).

6 Experiments

We evaluate our technique by comparing the quality of the loop invariants obtained from two publicly available invariant generation tools, Interproc [25] and InvGen [20], before and after decomposing multi-phase loops into a sequence of single-phase loops. Interproc is an abstract interpretation-based tool that implements the interval, octagon, and polyhedra abstract domains using the Apron [21] and Fixpoint[15] libraries. In contrast to Interproc, InvGen is a template-based invariant generator (see Sect. 7), which employs non-linear constraint solving to find valid instantiations for the unknown parameters of user-specified template invariants. Table 1 summarizes the results of our experiments on a set of challenging benchmarks, consisting of representative examples from the literature. All of our experimental benchmarks are available

[2] For soundness it is important that the negation here result in an overapproximation; for example, bracketing constraints [12] can be used.

Table 1. Comparison of the invariants generated by Interproc and InvGen on some benchmark programs before and after applying our technique

File	LOC	Interproc					InvGen				
		Before split		After split		Q	Before split		After split		Q
		time(s)	Proof?	time(s)	Proof?		time(s)	Proof?	time(s)	Proof?	
popl07	13	0.014	*N*	0.014	*Y*	+	0.425	*N*	0.215	*Y*	+
cav06	22	0.020	*N*	0.030	*Y*	+	0.318	*N*	0.28	*Y*	+
tacas08	30	0.018	*N*	0.021	*Y*	+	0.344	*Y*	0.298	*Y*	=
svd*	48	0.016	*Y*	0.014	*Y*	+	0.784	*Y*	0.794	*Y*	+
heapsort*	45	0.022	*Y*	0.036	*Y*	+	0.976	*Y*	1.55	*Y*	+
mergesort*	73	0.048	*N*	0.09	*N*	||	4.813	*Y*	12.138	*Y*	+
spam*	55	0.024	*Y*	0.029	*Y*	+	0.0521	*Y*	0.0397	*Y*	+
ex1	23	0.090	*N*	0.027	*Y*	+	416.985	*N*	0.621	*Y*	+
ex2	21	0.011	*N*	0.011	*Y*	+	123.945	*N*	0.553	*Y*	+
svd1	49	0.016	*N*	0.014	*Y*	+	0.456	*N*	0.784	*Y*	+
heapsort1	46	0.022	*N*	0.036	*Y*	+	2.291	*Y*	1.278	*Y*	+
mergesort1	74	0.048	*N*	0.090	*Y*	||	4.924	*Y*	11.431	*Y*	+
spam1	56	0.024	*N*	0.029	*Y*	+	0.46	*Y*	0.759	*Y*	+

from `http://www.stanford.edu/~isil/invariant-benchmarks.txt`. For generating invariants on each of these benchmarks, we used the polyhedra abstract domain of Interproc and the default templates provided by InvGen.

We now briefly describe the benchmark programs from Table 1. All of the benchmarks contain one or more assertions. The benchmark `popl07` is the program in Fig. 1 and the motivating example of [19]; `cav06` and `tacas08` are the motivating examples from [14] and [16] respectively. The next four benchmarks are programs from the test suite of InvGen. The program `spam` also occurs as `SpamAssassin-loop` in [24]. The next benchmark, `ex1`, is an interesting variation of `cav06`, and `ex2` is an example illustrating that splitting can be carried out in any order to obtain equivalent results in the presence of multiple splitter predicates. The programs `svd1`, `heapsort1`, and `spam1` have code similar to `svd`, `heapsort`, and `spam` but require stronger assertions to be proved; similarly, `mergesort1` differs only in having weaker assertions than `mergesort`.

The entries in the table marked *Y* indicate that a given tool was able to prove the assertions correct for the benchmark and *N* indicates that the tool could not prove at least one assertion. The column labeled "Before Split" shows whether a given tool was able to prove the assertions without using our technique, and the column labeled "After Split" describes whether the same tool could prove the same assertions on the loops decomposed by our technique. The column labeled "Q" compares the quality of the invariants obtained before and after using our technique. An entry labeled + in this column means that the tool generates better, i.e., logically stronger, invariants on the loops transformed by our technique, a || indicates that the invariants are incomparable (there are several invariants and some are stronger and some are weaker), and an = indicates that the inferred invariants were logically equivalent. Benchmarks marked with * indicate that the original benchmark code used a feature which is not part of the input language of Interproc; we manually modified these benchmarks before using them as input to Interproc. The time taken for computing phase splitter predicates was negligible; our algorithm took no longer than 90 milliseconds on any benchmark from Table 1.

The results shown in Table 1 demonstrate that our technique substantially improves the quality of the invariants generated by both Interproc and InvGen and allows them to verify assertions they could not verify previously. Consider only the first 9 programs (those above the double line); we discuss the 4 variations separately below. The invariants discovered by Interproc improve (i.e., are logically strengthened) in 8 out the 9 benchmarks. On `mergesort` the invariants discovered by Interproc are incomparable before and after splitting due to the non-monotonicity of the widening operator [7,8]. On the same set of 9 benchmarks, InvGen discovers logically stronger invariants on 8 benchmarks after splitting, and obtains a logically equivalent invariant on one benchmark.

Table 1 shows not only that there is an improvement in the quality of invariants after splitting, but also that Interproc and InvGen can prove many assertions after splitting that they could not previously verify. More specifically, in 6 of the first 9 benchmarks, Interproc fails to prove at least one assertion in the original program, but can verify all assertions in these programs after splitting. Similarly, InvGen cannot verify 4 of the 9 original benchmarks, but it can prove all the assertions in these programs after our transformation. Recall that five of the benchmarks (`tacas08` through `spam`) are included in InvGen's test suite; thus, it is not surprising that InvGen can verify the assertions in programs on which it was developed. We note that InvGen was unable to verify the four programs that were not taken from its test suite, but was able to verify all four of them after performing our transformation.

Observe that there are some benchmarks in Table 1 where the tools are able to prove the assertions both before and after splitting, but yield strictly stronger invariants after splitting. To demonstrate that this extra precision is useful, we created variants `svd1`, `heapsort1`, and `spam1` of `svd`, `heapsort`, and `spam` with stronger assertions (shown below the double line). Notice that Interproc can verify these three benchmarks after splitting, but is unable to do so before.

Also observe that for `mergesort1` Interproc is unable to prove the assertions both before and after splitting, despite yielding new facts after splitting. We demonstrate that these new invariants are again useful by designing `mergesort1` with weaker assertions. As shown below the double line in Table 1, Interproc can take advantage of these new facts obtained through splitting.

7 Related Work

Techniques for Multi-Phase Loops. There is an existing body of work whose goal is to improve the quality of invariants for multi-phase loops [27,15,14,1,18]. For example, Mauborgne and Rival address this problem in [27]. In contrast to our technique, their method is not fully automatic; it critically depends on user input for partitioning directives. The techniques described in [15,14] also attempt to improve the quality of invariants for multi-phase loops by guiding a static analysis to compute a fix-point on one phase of the loop before considering the next phase. Since our algorithm for identifying phases is independent of the particular abstract domain used for inferring invariants, our approach can recover precision irrespective of the abstract domain used for invariant generation.

Two recent works [18,1] take a similar approach to the one we present: splitting a loop to produce multiple loops with simpler invariants. These approaches first enumerate all the paths through a loop body and then they either search for all the possible sequences in which these paths can execute [18] or they eliminate the infeasible path sequences [1]. Since the number of paths is, in general, best case exponential in the number of conditionals in the loop body, both techniques have to rely on heuristics to keep the number of paths under consideration tractable. In contrast to both [18] and [1], our approach is less eager: we delay the worst-case exponential search to an SMT solver; if the solver can prove the properties of splitter predicates without reasoning about all the paths, we take advantage of that fact. Our approach is also much simpler and easier to implement, does not use heuristics, and yields some new insight into the nature of the problem (e.g., the independence of splitter predicates).

Direct Techniques for Inferring Disjunctive Invariants. Many different approaches have been developed for directly inferring disjunctive invariants, and some of these techniques are capable of discovering precise invariants for some of our benchmarks without decomposing the loop into phases. These approaches include (i) template-based techniques, such as [6,20,4], (ii) techniques based on predicate abstraction such as [2,3,22,16,5], and (iii) techniques based on probabilistic inference [19]. While some of these approaches are, in principle, capable of discovering precise invariants in loops exhibiting multiple phases, they are significantly more complicated, less efficient, and less widely-used than standard abstract interpretation-based techniques for generating conjunctive numeric invariants such as [8,28,26]. We discuss each of these three classes in more detail below.

Template-Based Techniques. Given an input template (i.e., parametrized form of invariant) provided by the user, template-based techniques find values for the parameters such that these instantiated templates correspond to inductive invariants [6,20,4]. While these techniques can, in principle, find precise invariants for multi-phase loops if the user provides appropriate disjunctive templates, they suffer from two drawbacks: First, they are not fully automatic since they require the user to specify the shape of the desired invariant. Second, since many template-based techniques require solving non-linear constraints, their applicability is limited by the lack of efficient algorithms for solving such constraints.

Predicate Abstraction Techniques. Techniques based on the basic form of counterexample guided abstraction refinement such as [2,3], are, in principle, capable of inferring disjunctive invariants. However, these techniques often diverge or take a very large number of refinement steps. More sophisticated invariant generation techniques based on predicate abstraction are considered in [22,16,5]. The basic idea underlying [22] is to use interpolants to generate counterexamples. To guarantee convergence, this technique restricts the language of the interpolants to some finite language L, and can therefore only find invariants in this restricted language. Hence, a poor choice of language degrades its performance. The technique described in [16] uses counterexample guided abstraction refinement to tune widening strategies in an abstract interpretation framework. While this technique can sometimes be helpful for generating more precise invariants for multi-phase loops, it is difficult to characterize the class of loops for which this technique will generate useful invariants. The technique presented in [5] combines counterexample-guided abstraction refinement with template-based invariant generation techniques. More specifically, the counterexamples produced in this technique are not finite program paths, but full-fledged programs called *path programs*. The algorithm in [5] then employs template-based techniques to infer invariants of the path program, which are used to refine the analysis. While this technique is capable of finding disjunctive invariants, it is not directly helpful for multi-phase loops.

Probabilistic Techniques. Gulwani and Jojic formulate the problem of invariant generation as probabilistic inference, and use machine learning techniques to infer invariants [19]. Their technique is capable of inferring the disjunctive invariant from Fig. 1. However, this technique is not guaranteed to converge, and it is difficult to characterize the class of loops for which it succeeds. Furthermore, this approach is significantly more involved than our algorithm for splitting loops with multiple phases.

8 Conclusion

We have proposed a static analysis technique to identify phase transitions in loops and decompose multi-phase loops into a sequence of simpler loops with fewer phases. We have demonstrated that standard invariant generation tools benefit substantially from the technique proposed in this paper, raising their level of precision to that of recently proposed methods for disjunctive invariant generation. Our technique is conceptually simple, easy to implement, and can be integrated into any invariant generation technique.

Acknowledgments

We thank the anonymous reviewers for their comments and for bringing related work to our attention. We would like to thank Denis Gopan, Francesco Logozzo, and Sumit Gulwani for their helpful pointers to benchmarks and invariant generation tools. We also thank Bertrand Jeannet for his guidance with the usage of Interproc and Ashutosh Gupta for his help with InvGen. Finally we would like to thank Yannick Moy for pointing out an error in the definition of $\overline{B}$ in an earlier version of this paper.

References

1. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT, pp. 49–58 (2009)
2. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT 9(5-6), 505–525 (2007)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
5. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309 (2007)
6. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
7. 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)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
10. Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language with a Two-Level Representation. Stanford University Technical Report (2009)
11. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 233–247. Springer, Heidelberg (2009)
12. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. Weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
13. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 236–252. Springer, Heidelberg (2010)
14. Gopan, D., Reps, T.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
15. Gopan, D., Reps, T.: Guided static analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007)
16. 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)
17. 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)
18. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)
19. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL, pp. 277–289 (2007)

20. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
21. 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)
22. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
23. Karr, M.: Affine relationships among variables of a program. Acta Inf. 6, 133–151 (1976)
24. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE, pp. 389–392 (2007)
25. Lalire, G., Argoud, M., Jeannet, B.: The Interproc Analyzer, `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
26. Laviron, V., Logozzo, F.: SubPolyhedra: A (More) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
27. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
28. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)

Monitorability of Stochastic Dynamical Systems

A. Prasad Sistla, Miloš Žefran, and Yao Feng

University of Illinois at Chicago
{sistla,mzefran,yfeng9}@uic.edu

Abstract. Monitoring is an important run time correctness checking mechanism. This paper introduces the notions of *monitorability* and *strong monitorability* for partially observable stochastic systems, and gives necessary and sufficient conditions characterizing them. It also presents important decidability and complexity results for checking these properties for finite state systems. Furthermore, it presents general monitoring techniques for the case when systems are modeled as quantized probabilistic hybrid automata, and the properties are specified as safety or liveness automata. Experimental results showing the effectiveness of the methods are given.

1 Introduction

The growing complexity of modern engineered systems and their increased reliance on computation calls for novel approaches to guaranteeing their correct functioning. This is especially important for safety critical systems such as medical devices and transportation systems where a failure can have catastrophic consequences.

One way to ensure correctness of a complex system is to thoroughly test and/or verify it. While testing can increase confidence in a component, it can not guarantee correctness. Verification, on the other hand, can guarantee correctness, but it is simply not feasible, for example, for a car with advanced engine controls and numerous networked microprocessors. In other cases, the component might have been verified for correctness on a model which was not accurate. And more importantly, even if through verification a component is found to be defective, we may still want to use it if the incorrect behavior only occurs rarely.

An alternative to testing and verification is to monitor the behavior of the component at run time. The monitor observes the inputs and outputs of the component and checks whether the behavior of the system is consistent with the expected behavior. Monitors can be especially useful if a fail-safe shutdown procedures can be developed, which is true for a broad class of systems. The fundamental advantage of monitors is that they are in principle easy to implement, and they are independent of the design procedures used to develop a component. While wrong assumptions might lead to a faulty design, the monitor is independent of design decisions and can therefore easily detect that the component is failing to perform its function.

In control systems literature, it is commonly assumed that system behavior is stochastic and the state of the system is not directly observable. Furthermore, in digital control systems the state is typically quantized. We thus consider Hidden

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 720–736, 2011.

Markov Chains (HMC) to model such discrete state systems. In our earlier work [10,24], we addressed the problem of monitoring a system, modeled as a HMC H, when the correctness specification is given by a deterministic Streett automaton $\mathcal{A}$ on infinite strings. In these works, we considered *external* monitoring where the automaton $\mathcal{A}$ is defined on the outputs generated by the system. There we defined two measures, called *Acceptance Accuracy* (AA) and *Rejection Accuracy* (RA) that capture the effectiveness of the monitor. Monitoring algorithms for achieving arbitrary high values of accuracies were presented when H and $\mathcal{A}$ are finite state systems. The values $(1 - AA)$ and $(1 - RA)$ are measures of false alarms and missed alarms, respectively, and should be kept low.

In this paper, we consider *internal* monitoring where the automaton $\mathcal{A}$ is specified on the states of the system, not on its outputs. Further, we allow both the system H and the automaton $\mathcal{A}$ to have infinite number of states. Since states are not directly observable, internal monitoring is significantly harder than external monitoring. In this setting, we define two notions of monitorability. We say that a system H is *strongly monitorable* with respect to an automaton $\mathcal{A}$, if there is a monitor such that both of its accuracies are 1. We give a necessary and sufficient condition for strong monitorability. We show that, for finite state systems, the problem of deciding whether a system H is strongly monitorable with respect to an automaton $\mathcal{A}$ is PSPACE-complete.

We also define a more realistic notion called *monitorability.* A system H is said to be *monitorable* with respect to an automaton $\mathcal{A}$ if accuracies arbitrarily close to 1 can be achieved, i.e., for every $x \in [0, 1)$, there is a monitor such that both of its accuracies are greater than or equal to x. We present a fundamental result that exactly characterizes monitorability. It states that a system H is monitorable with respect to $\mathcal{A}$ exactly when the probability measure of the set of so-called 0/1-limit sequences is 1. We show that even for finite state systems, determining monitorability is undecidable; more specifically, it is shown to be r.e.-complete. However, we identify some sufficient conditions for monitorability.

Finally, we consider systems specified as quantized probabilistic hybrid automata [11]. We assume that the correctness automaton, called property automaton, is specified by a non-probabilistic quantized hybrid automaton. We present monitoring algorithms for these cases that employ particle filters for state estimation. When the property automaton $\mathcal{A}$ specifies a safety property on the discrete states of the system, then one can achieve arbitrary high levels of accuracies by appropriately specifying threshold probability parameter. When $\mathcal{A}$ also includes a liveness property, one can achieve high levels of accuracies by approximating it using safety automata with timeouts. High accuracies can be achieved by adjusting the time-out parameters. We implemented these techniques in Matlab and evaluated them for an example of a train equipped with electronically controlled pneumatic (ECP) brakes. Experimental results showing the effectiveness of the monitoring algorithms are presented.

In summary the main contributions of the paper are as follows: (1) a Hidden Markov model for modeling infinite state systems and accuracy measures when the property automaton is specified on system states; (2) definitions of monitorability and strong monitorability, and exact characterizations of systems that satisfy these properties; PSPACE-completeness result for checking strong monitorability and undecidability for checking monitorability for finite state systems; identification of sufficient conditions, that hold in practice, for monitorability; (3) monitoring algorithms when the system and property automata are specified by probabilistic hybrid automata and hybrid automata, respectively; and (4) experimental results showing the effectiveness of our approach. Due to the space limitations, most proofs have been omitted in the current version; for complete details we refer the reader to [26].

2 Related Work

For external monitoring, it is easy to see that every safety property is strongly monitorable and [10] shows that, for finite state systems, all properties including liveness properties are monitorable. There we employ dynamically increasing timeouts to achieve monitorability of liveness properties. The current paper employs completely different techniques and presents some fundamentally new results on monitorability for internal monitoring.

Several authors consider monitoring temporal properties of deterministic systems capable of measuring system state [30, 20, 8, 3, 6]. Some of them define monitorability, but it is only with respect to a property. In contrast, we consider partially observable stochastic systems (i.e., HMCs). In this case, the monitoring problem is significantly more difficult and both the system and the property need to be considered when defining monitorability.

A problem that has been extensively studied is monitoring and diagnosis of hybrid automata [14,29,4,17,2], where the aim is to detect when the automaton enters a fail state so that the system can appropriately react. In most cases, these works employ techniques that depend on the specific possible modes of failure. None of the above works addresses the general problem of monitoring system behaviors against specifications given in an expressive formal system such as the hybrid automata. Furthermore, they do not address the problem of monitoring liveness properties.

Control synthesis for stochastic discrete-event systems has been studied in [15, 18]. In contrast to our work, the authors only consider finite-state systems with directly observable state. Similarly, the literature on diagnosability of partially-observable discrete-event systems (e.g. [31]) only considers deterministic finite-state systems.

A method for monitoring and checking quantitative and probabilistic properties of real-time systems has been given in [23], [21] considers monitoring interfaces for faults using game-theoretical framework, and *conservative* run time monitors were proposed in [16,25]; none of these works is intended for monitoring of hybrid systems.

3 Definitions and Notation

Sequences. Let S be a set. Let $\sigma = s_0, s_1, \ldots$ be a possibly infinite sequence over S. For any $i \geq 0$, $\sigma[0, i]$ denotes the prefix of σ up to s_i. If α_1 is a finite sequence and α_2 is either a finite or an ω-sequence then $\alpha_1\alpha_2$ denotes the concatenation of the two sequences in that order. We let S^*, S^ω denote the set of finite sequences and the set of infinite sequences over S. If $C \subseteq S^\omega$ and $\alpha \in S^*$ then αC denotes the set $\{\alpha\beta \ : \beta \in C\}$.

Safety Properties. For any $\sigma \in S^\omega$, let prefixes(σ) denote the set of prefixes of σ and for any $C \subseteq S^\omega$, let prefixes$(C) = \cup_{\sigma \in C}(\text{prefixes}(\sigma))$. We say that $C \subseteq S^\omega$ is a *safety* property if the following condition holds: for any $\sigma \in S^\omega$, if prefixes$(\sigma) \subseteq$ prefixes(C) then $\sigma \in C$. For any $C \subseteq S^\omega$, let $closure(C)$ be the smallest safety property such that $C \subseteq closure(C)$.

Automata. We consider deterministic Streett automata to specify properties over infinite sequences. Each such automaton $\mathcal{A}$ has an input alphabet Σ and defines a language $L(\mathcal{A}) \subseteq \Sigma^\omega$. These automata can have countable number of states. Throughout sections 3 and 4, an automaton refers to a Streett automaton. We also consider Büchi automata, a subclass of Streett automata, and a subclass of Büchi automata called *safety automata* whose language is a safety property.

Markov Chains. We assume that the reader is familiar with basic probability theory, random variables and Markov chains. We consider stochastic systems given as Markov Chains [19] and monitor their computations for satisfaction of a given property specified by an automaton or a temporal formula. A Markov chain $G = (S, R, \phi)$ is a triple satisfying the following: S is a set of countable states; $R \subseteq S \times S$ is a total binary relation (i.e., for every $s \in S$, there exists some $t \in S$ such that $(s,t) \in R$); and $\phi : R \rightarrow (0,1]$ is a probability function such that for each $s \in S$, $\sum_{(s,t)\in R} \phi((s,t)) = 1$. Note that, for every $(s,t) \in R$, $\phi((s,t))$ is non-zero. Intuitively, if at any time the system is in a state $s \in S$, then in one step, it goes to some state t such that $(s,t) \in R$ with probability $\phi((s,t))$. A finite path p of G is a sequence $s_0, s_1, \ldots, s_n$ of states such that $(s_i, s_{i+1}) \in R$ for $0 \leq i < n$. For any such p, if $n > 0$, then let $\phi(p) = \prod_{0 \leq i < n} \phi((s_i, s_{i+1}))$; if $n = 0$ then let $\phi(p) = 1$. An infinite path of G is an infinite sequence of states $s_0, s_1, ...$ such that $\forall i \geq 0$, $(s_i, s_{i+1}) \in R$. We let $Paths(G)$ and $Paths(G, s)$ for any $s \in S$, respectively, denote the set of all infinite paths in G and the set of all infinite paths in G starting from s.

For any Markov chain G, as given above, we define a class $\mathcal{E}_G$ of measurable sets of infinite sequences over S. $\mathcal{E}_G$ is the σ-algebra [19] generated by sets of sequences of the form pS^ω where $p \in S^*$. Now, for any system state $r \in S$, we define a probability function $\mathcal{F}_{G,r}$ defined on $\mathcal{E}_G$ as follows. Intuitively, for any $C \in \mathcal{E}_G$, $\mathcal{F}_{G,r}(C)$ denotes the probability that a sequence of states generated from the system state r, is in C. $\mathcal{F}_{G,r}$ is the unique probability measure satisfying all the probability axioms [19], such that for every $p \in S^*$ and $C = pS^\omega$, if p is the empty sequence then $\mathcal{F}_{G,r}(C) = 1$, if p is a finite path starting from state r then $\mathcal{F}_{G,r}(C) = \phi(p)$, otherwise $\mathcal{F}_{G,r}(C) = 0$.

Although, for convenience, we have considered all sequences in S^ω in defining $\mathcal{E}_G$, sequences that are not paths in G do not contribute to the probability of any $C \in \mathcal{E}_G$, as shown below. Since S is a countable set, it is not difficult to see that $Paths(G)$, $Paths(G, r) \in \mathcal{E}_G$. Further more, for any $C \in \mathcal{E}_G$, it can be shown that $\mathcal{F}_{G,r}(C) = \mathcal{F}_{G,r}(C \cap Paths(G)) = \mathcal{F}_{G,r}(C \cap Paths(G, r))$.

For any $D \in \mathcal{E}_G$, we let $\mathcal{F}_{G,r|D}$ denote the conditional probability function given D; formally, for any $C, D \in \mathcal{E}_G$, $\mathcal{F}_{G,r|D}(C) = \frac{\mathcal{F}_{G,r}(C\cap D)}{\mathcal{F}_{G,r}(D)}$. For any $\alpha \in S^*$ and $C = \alpha S^\omega$, we let $\mathcal{F}_{G,r}(\alpha)$ denote the probability $\mathcal{F}_{G,r}(C)$ and $\mathcal{F}_{G,r|\alpha}$ denote the conditional probability function $\mathcal{F}_{G,r|C}$. For a set $C \subseteq S^*$, we let $\mathcal{F}_{G,r}(C)$ denote $\mathcal{F}_{G,r}(CS^\omega)$.

We will use automata to specify properties over sequences of states of a Markov chain G. The input symbols to the automata are states of G, i.e., members of S. It has been shown that, for any automaton $\mathcal{A}$, $L(\mathcal{A})$ is measurable [28]. We will be interested in monitoring sequences of states of a system modeled by G, i.e., computations generated by G, to ensure that it satisfies the property given by an automaton $\mathcal{A}$. However, the monitor can not observe the actual states of the system.

Hidden Markov Chains. A Hidden Markov Chain (HMC) [5] $H = (G, O, r_0)$ is a triple where $G = (S, R, \phi)$ is a Markov chain, $O : S \rightarrow \Sigma$ is the output function and $r_0 \in S$ is the initial state. Intuitively, for any $s \in S$, $O(s)$ is the output generated in state s and this output is generated when ever a transition entering state s is taken. The generated symbols become inputs to the monitor. H is called Hidden Markov chain because one only observes the outputs generated in each state but not the actual state[1]. We extend the output function O to paths of G as follows. For any finite or infinite path $p = s_0, s_1, \ldots, s_i, \ldots$ in G, $O(p) = O(s_0), O(s_1), \ldots, O(s_i), \ldots$. For any finite or infinite sequence α in $\Sigma^* \cup \Sigma^\omega$, we let $O^{-1}(\alpha)$ denote the set of $p \in S^* \cup S^\omega$ such that $O(p) = \alpha$. For any $C' \subseteq \Sigma^* \cup \Sigma^\omega$, we let $O^{-1}(C') = \cup_{\alpha\in C'}(O^{-1}(\alpha))$.

For any HMC H as given above, we define a class $\mathcal{E}_H$ of sets of infinite sequences over Σ and for any $r \in S$, we define a probability measure $\mathcal{F}_{H,r}$ on $\mathcal{E}_H$ as follows. $\mathcal{E}_H$ is the σ-algebra generated by the sets $\alpha\Sigma^\omega$ for $\alpha \in \Sigma^*$. For any system state $r \in S$ and $C' \in \mathcal{E}_H$, $\mathcal{F}_{H,r}(C') = \mathcal{F}_{G,r}(O^{-1}(C'))$. Intuitively, $\mathcal{F}_{H,r}(C')$ denotes the probability that an output sequence generated from the system state r, is in C'.

Quantized Probabilistic Hybrid Automata. Quantized probabilistic hybrid automata (QPHA) are probabilistic hybrid automata [11] whose continuous variables are quantized. Their semantics is given by a HMC, but they provide a convenient formalism for specifying systems. A quantized probabilistic hybrid automaton $\mathcal{A}$ is a tuple $(Q, V, \Delta t, \mathcal{E}, \mathcal{T}, c_0)$ where Q is a finite set of *discrete* states (modes); $V = \{x_q\}_{q\in Q} \cup \{y_q\}_{q\in Q} \cup \{n_q\}_{q\in Q}$ is the finite set of real-valued *continuous*, *output* and *noise* variables, respectively, that will be assumed to be quantized ;Δt is the sampling time; $\mathcal{E}$ is a function that with each $q \in Q$ associates a set $\mathcal{E}(q)$ of difference equations describing the evolution of the continuous state and the output at time $t + \Delta t$ as a function of the state at t and the noise variables; $\mathcal{T}$ is a function that assigns to each $q \in Q$ a set of *transition*

[1] In the traditional definition of HMCs considered in literature, the output of a state can be any symbol in Σ generated with a probability distribution that is specific to the state; since Σ is a countable set, it is not difficult to see that by duplicating each state as many times as there are output symbols, such a HMC can be converted into an equivalent HMC consistent with our model.

triplets (p_{qi}, ϕ_i, ψ_i), where the *guard* ϕ_i is a predicate over the set of continuous and discrete variables, p_{qi} is a probability distribution over *transition target discrete states*, and the *reset relation* ψ_i is a set of assignments that update or reset some of the continuous variables; and c_0 denotes the initial discrete and continuous states of the automaton. If no noise variables are present and for each all transition triplets the transition target state set is a singleton, a QPHA becomes a quantized hybrid automaton (QHA) [1]. A property can be specified if an appropriate acceptance condition is defined for a QHA. In fact, in this case the QHA is equivalent to a Streett automaton.

Within each mode q, the evolution of the QPHA is given by the difference equations. Since the continuous, noise and output variables are assumed to be quantized, these transitions can be interpreted as an HMC. When a guard ϕ_i becomes satisfied, a transition takes place from q to some target mode q' according to the probability distribution p_{qi}. The overall evolution of the QPHA can be thus interpreted as the evolution of an appropriate HMC. See [11,12] for details. We will use QPHA to model systems. A property can be modeled using QHA by defining an appropriate acceptance condition.

Monitors. A monitor $M : \Sigma^* \rightarrow \{0, 1\}$ is a function with the property that, for any $\alpha \in \Sigma^*$, if $M(\alpha) = 0$ then $M(\alpha\beta) = 0$ for every $\beta \in \Sigma^*$. For an $\alpha \in \Sigma^*$, we say that M rejects α, if $M(\alpha) = 0$, otherwise we say M accepts α. Thus if M rejects α then it rejects all its extensions. For an infinite sequence $\sigma \in \Sigma^\omega$, we say that M rejects σ iff there exists a prefix α of σ that is rejected by M; we say M accepts σ if it does not reject it. Let $L(M)$ denote the set of infinite sequences accepted by M. It is not difficult to see that $L(M)$ is a safety property and $O^{-1}(L(M))$ is measurable (it is in $\mathcal{E}_G$).

Note that for a monitor to be implementable, M has to be a computable function as observed in [30] for deterministic systems. We will not further consider this issue in this paper.

Accuracy Measures. Let $\mathcal{A}$ be an automaton on states of H. The *acceptance accuracy* of M for $\mathcal{A}$ with respect to the HMC H, denoted by $AA(M, H, \mathcal{A})$, is the probability $\mathcal{F}_{G,r_0|L(\mathcal{A})}(O^{-1}(L(M)))$ where r_0 is the initial state of H. Intuitively, it is the conditional probability that a sequence generated by the system is accepted by M, given that it is in $L(\mathcal{A})$. We define the *rejection accuracy* of M for $\mathcal{A}$ with respect to H, denoted by $RA(M, H, \mathcal{A})$, to be the probability that a sequence generated by the system is rejected by M, given that it is not in $L(\mathcal{A})$; formally, it is the probability $\mathcal{F}_{G,r_0|C}(D)$, where C, D are the complements of $L(\mathcal{A})$ and $O^{-1}(L(M))$ respectively.

Monitorability. We say that a system H is *strongly monitorable* with respect to an automaton $\mathcal{A}$ if there exists a monitor M such that $AA(M, H, \mathcal{A}) = RA(M, H, \mathcal{A}) = 1$, i.e., both of its accuracies are 1. We say that a system H is *monitorable* with respect to an automaton $\mathcal{A}$ if for every $x \in [0, 1)$ there exists a monitor M such that $AA(M, H, \mathcal{A}) \geq x$ and $RA(M, H, \mathcal{A}) \geq x$. Strong monitorability is a property that is difficult to satisfy. In the next section, we give necessary and sufficient conditions for these properties to be satisfied.

It is worth noting that monitorability, while related to the classical notion of observability, is fundamentally different from it. It is not difficult to construct hybrid systems that are not observable or even discrete-state observable but are monitorable.

4 Conditions for Monitorability

In this section we give necessary and sufficient conditions for strong monitorability and monitorability. Let $H = (G, O, r_0)$ be a HMC where $G = (S, R, \phi)$ is the associated Markov chain. Let $\mathcal{A}$ be an automaton with input alphabet S. $H, G, \mathcal{A}$ are fixed throughout this section unless otherwise stated.

4.1 Strong Monitorability

We define a set of infinite paths $\mathit{OverlapSeq}(H, \mathcal{A})$ that intuitively captures non-trivial overlap, based on the generated outputs, between sets of infinite paths of G that are accepted and those that are rejected by $\mathcal{A}$. We say that a finite path p in G is *good* if it starts from r_0 and the set C of infinite paths, accepted by $\mathcal{A}$, having p as a prefix, has non-zero measure, i.e., $\mathcal{F}_{G,r_0}(C) > 0$ where $C = (pS^\omega \cap \mathit{Paths}(G, r_0) \cap L(\mathcal{A}))$. Let $\mathit{GoodPaths}(H, \mathcal{A})$ be the set of infinite paths in G having only good prefixes. Now we define $\mathit{OverlapSeq}(H, \mathcal{A}) = (\mathit{Paths}(G, r_0) - L(\mathcal{A})) \cap O^{-1}(O(\mathit{GoodPaths}(H, \mathcal{A})))$. Intuitively, $\mathit{OverlapSeq}(H, \mathcal{A})$ is the set of $p \in \mathit{Paths}(G, r_0)$ such that p is rejected by $\mathcal{A}$ and each of its prefix generates the same output sequence as some good path in G, i.e., it can not be distinguished from a good path based on the outputs.

The following theorem gives a necessary and sufficient condition for strong monitorability.

Theorem 1. *Let $H = (G, O, r_0)$ be a hidden Markov chain where $G = (S, R, \phi)$ is the associated Markov chain. Let $\mathcal{A}$ be an automaton with input alphabet S. H is strongly monitorable with respect to $\mathcal{A}$ iff $\mathcal{F}_{G,r_0}(\mathit{OverlapSeq}(H, \mathcal{A})) = 0$.*

The lower bound in next theorem is proved by reduction from the non-universality problem for non-deterministic safety automata:

Theorem 2. *Given a finite HMC H and a finite state automaton $\mathcal{A}$, the problem of determining if H is strongly monitorable with respect to $\mathcal{A}$ is PSPACE-complete.*

If H is strongly monitorable with respect to $\mathcal{A}$, using the techniques employed in the proof of the above theorem, we can construct a monitor M' both of whose accuracies equal 1. M' simply constructs a deterministic automaton $\mathcal{C}$ and runs it on the output generated by H. It rejects iff $\mathcal{C}$ rejects. M' does not estimate any probabilities.

4.2 Monitorability

Consider any $\alpha \in \Sigma^*$. According to our notation $\mathcal{F}_{H,r}(\alpha)$ is the probability that an output sequence of length n, generated by H from state r, is α. Let $\alpha \in \Sigma^*$ be such that $\mathcal{F}_{H,r}(\alpha) > 0$. Now, we define a probability measure $\mathit{AccProb}(\alpha)$ which is the conditional probability that an execution of the system H that initially generated the output sequence α is accepted by $\mathcal{A}$. Formally, $AccProb(\alpha) = \mathcal{F}_{H,r_0|C}(L(\mathcal{A}))$ where $C = O^{-1}(\alpha)S^{\omega}$. Let $RejProb(\alpha) = 1 - AccProb(\alpha)$. Observe that $RejProb(\alpha)$ is the conditional probability that an execution of the system that initially generated the output sequence α is rejected by $\mathcal{A}$.

Recall that for any $\beta \in \Sigma^{\omega}$ and integer $i \geq 0$, $\beta[0,i]$ denotes the prefix of β of length $i+1$. Now, let $OneSeq(H,\mathcal{A})$ be the set of all $\beta \in \Sigma^{\omega}$ such that $\lim_{i\to\infty} AccProb(\beta[0,i])$ exists and its value is 1. Similarly, let $ZeroSeq(H,\mathcal{A})$ be the set of all $\beta \in \Sigma^{\omega}$ such that the above limit exists and is equal to 0. Let $ZeroOneSeq(H,\mathcal{A}) = OneSeq(H,\mathcal{A}) \cup ZeroSeq(H,\mathcal{A})$. The following lemma states that the sets $OneSeq(H,\mathcal{A})$ and $ZeroSeq(H,\mathcal{A})$ are measurable. It also states that the measure of those executions of H that generate output sequences in $OneSeq(H,\mathcal{A})$ (resp., sequences in $ZeroSeq(H,\mathcal{A})$) and that are rejected by $\mathcal{A}$ (respectively, accepted by $\mathcal{A}$), is zero.

Lemma 1. *The sets $OneSeq(H,\mathcal{A})$ and $ZeroSeq(H,\mathcal{A})$ are measurable (both are members of $\mathcal{E}_H$). Furthermore,*

$$\mathcal{F}_{G,r_0}(O^{-1}(OneSeq(H,\mathcal{A})) - L(\mathcal{A})) = 0 \quad \textit{and}$$
$$\mathcal{F}_{G,r_0}(O^{-1}(ZeroSeq(H,\mathcal{A})) \cap L(\mathcal{A})) = 0.$$

The following theorem, a central result of the paper, gives a necessary and sufficient condition for the monitorability of H with respect to $\mathcal{A}$. But in addition to providing a characterization of the monitorability, the theorem also provides a method for constructing monitors as explained in Section 4.3.

Theorem 3. *For any HMC H and deterministic Streett automaton $\mathcal{A}$, H is monitorable with respect to $\mathcal{A}$ iff $\mathcal{F}_{H,r_0}(ZeroOneSeq(H,\mathcal{A})) = 1$.*

Proof. Let $H = (G, O, r_0)$ be a HMC where $G = (S, R, \phi)$ is a Markov chain, and $\mathcal{A}$ be a deterministic Streett automaton with input alphabet S. Assume that H is monitorable with respect to $\mathcal{A}$.

Suppose that $\mathcal{F}_{H,r_0}(ZeroOneSeq(H,\mathcal{A})) < 1$. Let $F = \Sigma^{\omega} - ZeroOneSeq(H,\mathcal{A})$. Clearly $\mathcal{F}_{H,r_0}(F) > 0$. Consider any $\beta \in F$. It should be easy to see that for some $m > 0$, the following property holds (otherwise β is in $(ZeroOneSeq(H,\mathcal{A}))$):

Property 1. $AccProb(\beta[0,i]) < (1 - \frac{1}{2^m})$ for infinitely many values of i and $RejProb(\beta[0,j]) < (1 - \frac{1}{2^m})$ for infinitely many values of j.

For each $m > 0$, define F_m to be the set of sequences $\beta \in F$ such that m is the smallest integer that satisfies Property 1. It is easy to see that the set $\{F_m : m > 0\}$ is a partition of F. It should also be easy to see that $F_m \in \mathcal{E}_H$, i.e., is measurable, for each $m > 0$. Since $\mathcal{F}_{H,r_0}(F) > 0$, it follows that for some $m > 0$, $\mathcal{F}_{H,r_0}(F_m) > 0$. Fix such an m and let $x = \mathcal{F}_{H,r_0}(F_m)$. From Property 1, it can be shown that for any $C \in \mathcal{E}_H$, such that $C \subseteq F_m$ and $y = \mathcal{F}_{H,r_0}(C) > 0$, the following property holds:

Property 2. $\mathcal{F}_{H,r_0}(O^{-1}(C) \cap L(\mathcal{A})) \geq \frac{y}{2^m}$ and $\mathcal{F}_{H,r_0}(O^{-1}(C) - L(\mathcal{A})) \geq \frac{y}{2^m}$.

Now, consider any monitor M. Recall that $L(M)$ is the set of infinite sequences over Σ that are accepted by M. Since $L(M)$ is a safety property, it is easily seen that $L(M) \in \mathcal{E}_H$, i.e., it is measurable. Now, since $x = \mathcal{F}_{H,r_0}(F_m)$ and $F_m = (F_m \cap L(M)) \cup (F_m - L(M))$, it is the case that either $\mathcal{F}_{H,r_0}(F_m \cap L(M)) \geq \frac{x}{2}$ or $\mathcal{F}_{H,r_0}(F_m - L(M)) \geq \frac{x}{2}$. In the former case, by taking $C = F_m \cap L(M)$ and using Property 2, we see that the measure of bad executions of the system (i.e., those in $S^\omega - L(\mathcal{A})$) that are accepted by the monitor is $\geq \frac{x}{2^{m+1}}$ and in the latter case, by taking $C = (F_m - L(M))$, the measure of good executions of the system that are rejected is $\geq \frac{x}{2^{m+1}}$. Now, let $z = \max\{\mathcal{F}_{H,r_0}(Paths(G, r_0) \cap L(\mathcal{A})), \mathcal{F}_{H,r_0}(Paths(G, r_0) - L(\mathcal{A}))\}$. From the above arguments, we see that for every monitor M, either $RA(M, H, \mathcal{A}) \leq 1 - \frac{x}{z \cdot 2^{m+1}}$ or $AA(M, H, \mathcal{A}) \leq 1 - \frac{x}{z \cdot 2^{m+1}}$. This contradicts our assumption that H is monitorable with respect to $\mathcal{A}$.

Now, assume that $\mathcal{F}_{H,r_0}(ZeroOneSeq(H, \mathcal{A})) = 1$. Let $z \in (0, 1)$. Let $M_z : \Sigma^* \to \{0, 1\}$ be a function such that for any $\alpha \in \Sigma^*$, $M_z(\alpha) = 0$ iff there exists a prefix α' of α such that $RejProb(\alpha') \geq z$. Clearly M_z is a monitor. When extended to infinite sequences $M_z(\beta) = 0$ for every $\beta \in ZeroSeq(H, \mathcal{A})$, i.e., it rejects all of them. The second part of lemma 3 implies that $\mathcal{F}_{G,r_0}(O^{-1}(ZeroSeq(H, \mathcal{A})) \cap L(\mathcal{A})) = 0$, and it can further be deduced that $\mathcal{F}_{G,r_0}((S^\omega - O^{-1}(ZeroSeq(H, \mathcal{A})) - L(\mathcal{A})) = 0$. From these observations, it follows that $RA(M_z, H, \mathcal{A}) = 1$. It should be easy to see that the measure of good executions of H that are rejected by M_z is $\leq \min\{y,\ 1 - z\}$ where $y = \mathcal{F}_{G,r_0}(Paths(G, r_0) \cap L(\mathcal{A}))$. Therefore, $AA(M_z, H, \mathcal{A}) \geq 1 - \frac{\min\{y, 1-z\}}{y}$. Now, for any given $x \in (0, 1)$, we can chose a value of z such that $AA(M_z, H, \mathcal{A}) \geq x$ and $RA(M_z, H, \mathcal{A}) = 1$. This implies that H is monitorable with respect to $\mathcal{A}$. □

The following result can be obtained by reducing the non-universality problem for Probabilistic Finite State Automata.

Theorem 4. *The problem of deciding if a finite state HMC is monitorable with respect to a finite state automaton is r.e.-complete and hence is undecidable, where r.e. is the set of recursively enumerable sets.*

Remark 1. Theorem 3 generalizes to the case when we replace $\mathcal{A}$ by an arbitrary measurable set $C \in \mathcal{E}_H$ by defining $AA(M, H, C)$ to be simply $\mathcal{F}_{G,r_0|C}(O^{-1}(L(M)))$.

4.3 Monitoring Algorithms

Although, the problem of determining if a HMC is monitorable with respect to an automaton $\mathcal{A}$ for finite state systems, is undecidable, we can give sufficient conditions that ensure monitorability.

Intuitively, $H = (G, O, r_0)$ is going to be monitorable with respect to an automaton $\mathcal{A}$ if the statistics (i.e., probability distributions) of outputs generated in paths (in $Paths(G, r_0)$) that are accepted by $\mathcal{A}$ is different from those generated in paths that are rejected by $\mathcal{A}$. Many times, this property may be known. For example, consider a system that can fail, i.e., can get into any of a set of failure states and once it gets into these states, it remains in these states. Further more, assume that the probability distributions of outputs generated in failure states is different from that in non-failure states. Such systems are monitorable with respect to properties that hold only on computations without failure states.

Consider the HMC given in Figure 1 with initial state s. Its output symbols are a, b. Let $\mathcal{A}$ be the automaton that accepts all paths in which state v appears infinitely often. This HMC is monitorable with respect to $\mathcal{A}$, but is not strongly monitorable with respect to $\mathcal{A}$. The probabilities of generation of a, b are different in the two strongly connected components.

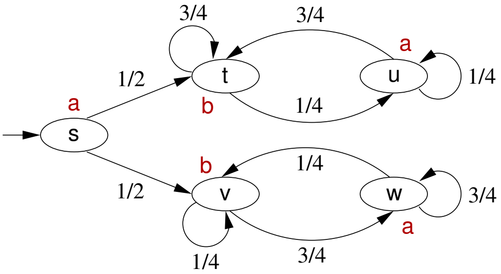


Fig. 1. A HMC that is monitorable

Assume that H is monitorable with respect to $\mathcal{A}$. Now, we address the problem of constructing accurate monitors for it. The second part of the proof of theorem 4 gives an approach. Here we choose some probability threshold value z. After each output symbol generated by H, if α is the output sequence generated thus far, we compute $RejProb(\alpha)$ and reject (i.e., raise an alarm) if $RejProb(\alpha) \geq z$. Theorem 3 implies that as the threshold probability value z approaches 1[2], we get a sequence of monitors that have accuracies approaching 1. This method requires computation of $RejProb(\alpha)$. Since $\mathcal{A}$ is deterministic, if H and $\mathcal{A}$ are finite state systems then we can construct their product Markov chain and using standard techniques [19] we can compute $RejProb(\alpha)$. For infinite state systems the above approach does not work and it may not be efficient even for finite state systems.

Monitoring Safety Properties. Assume that the system is modeled using a probabilistic hybrid automaton and a property is specified by a safety automaton $\mathcal{A}$. We construct the product of the hybrid automaton model of the system and the automaton $\mathcal{A}$. As the system runs, using an appropriate estimator that uses the product automaton, after each output generated by the actual system, we estimate the probability that the system execution is bad. This is estimated to be the probability that the component denoting the state of $\mathcal{A}$ is an error state in the product automaton, given that it has generated the observed output sequence. If this estimated value is $\geq z$ then we reject. When the system is monitorable with respect to $\mathcal{A}$, we can achieve high accuracies by increasing z.

[2] Making $z = 1$ may result in rejection accuracy being 0.

Monitoring Liveness Properties. Monitoring of properties specified by liveness automata can be achieved using the methods given in [25, 16]. Let $\mathcal{A}$ be a Büchi automaton[3]. We convert $\mathcal{A}$ into a safety automaton $\mathcal{A}'$ by using timeouts. Let T' be positive time out value. $\mathcal{A}$ is modified so that if an accepting state is not reached within T' units from the start or from the last time an accepting state is reached, then the automaton goes to the error state. It is fairly easy to show that any input sequence that is rejected by $\mathcal{A}$ is also rejected by $\mathcal{A}'$; however $\mathcal{A}'$ rejects more input sequences. Thus, $\mathcal{A}'$ is an approximation of $\mathcal{A}$. Note that we get better approximations by choosing larger values of T'. The above construction can be incorporated by including a *counter* variable in the HA model. The details are straightforward. This approach will be used in the experimental section.

5 Example

Consider the operation of a train with electronically-controlled pneumatic (ECP) brakes [9]. In this case, a braking signal is sent to each of the N cars of the train that subsequently engage their own brakes. We consider the case when the braking systems of individual cars can fail. If this happens to more than a given number of cars ($2N/3$ in our example) the train might not be able to stop and it should start an emergency stopping procedure (for example, engaging the brakes using the traditional pneumatic system). We would therefore like to develop a monitor that can correctly trigger the stopping mechanism in the event when several of the cars have faulty brakes, allowing the train operators to take the advantage of the superior braking performance of the ECP while not sacrificing the safety of the train.

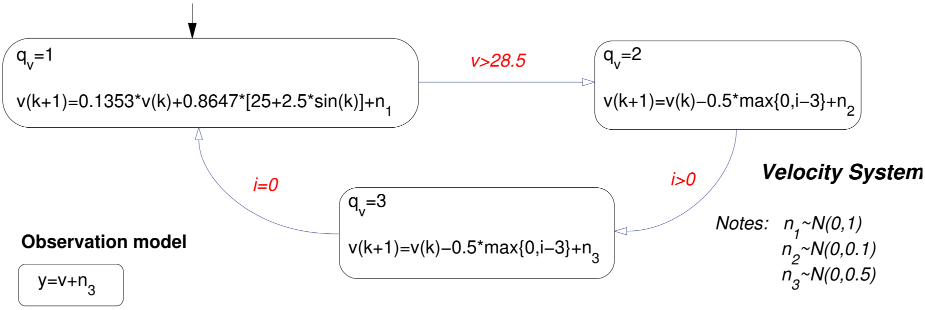


Fig. 2. Velocity subsystem for the train with ECP brakes

Figure 2 describes how the train velocity v evolves. The train starts in the discrete state $q_v = 1$ and remains in that state until the velocity exceeds a threshold $V_U = 28.5$, when it switches to the discrete state $q_v = 2$. The train remains in the state $q_v = 2$ until one of the brakes engages and it switches to state $q_v = 3$. The velocity in states $q_v = 2$ and $q_v = 3$ depends on the number of brakes

[3] The construction can be easily extended to an arbitrary Streett automaton.

that have been engaged through the braking force term $-0.5\max\{0, i - \lfloor N/3 \rfloor\}$, where i is the number of brakes that have been engaged and N is the number of cars ($N = 10$ for the simulations). Note that in order for the brakes to slow down the train at least $\lfloor N/3 \rfloor$ brakes need to be engaged. When all the brakes disengage, the velocity system switches back to the state $q_v = 1$. When in the state $q_v = 1$, the train accelerates to a constant velocity $V_C = 25$ and oscillates around it with the amplitude 2.5. The measured variable y is assumed to be v, however the measurements are corrupted by a measurement noise n_3. It is worth noting that the dynamics of the system (and thus statistical properties of output sequences) are different for $q_v = 1$ and $q_v = 3$. The system thus satisfies the sufficient conditions for monitorability given in Section 4.3.

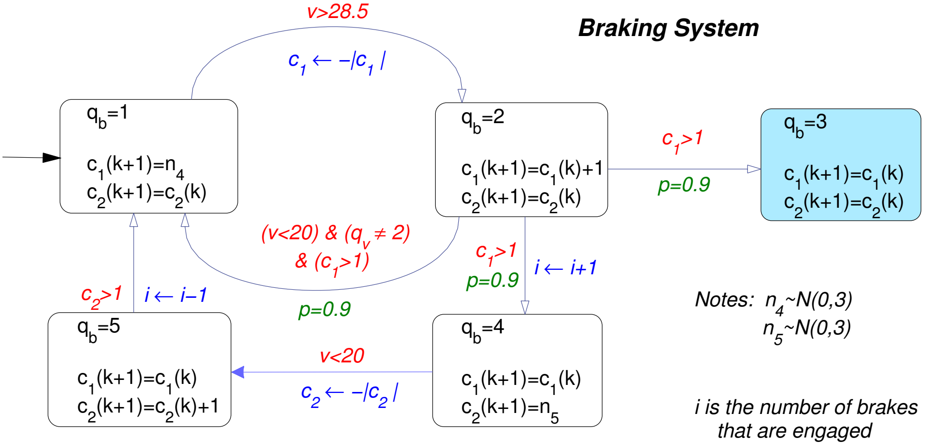


Fig. 3. ECP braking subsystem of each car of the train

Figure 3 describes the operation of the braking system of each of the cars. The braking system starts in the discrete state $q_b = 1$ and remains in that state until the velocity exceeds a threshold $V_U = 28.5$, when it switches to the discrete state $q_b = 2$. The braking system remains in the state $q_b = 2$ until the timer c_1 reaches $L_1 = 1$ (modeling delays in actuation and computational delays). Note that the initial value of the timer c_1 in the state $q_b = 2$ is not deterministic, so the duration of time the system remains in $q_b = 2$ is a random variable. After the timer reaches L_1, the braking system can fail with a probability $p = 0.1$ and permanently switch to $q_b = 3$. With the probability $p = 0.9$ it either returns to state $q_b = 1$ if the velocity already fell below the threshold $V_L = 20$, or switches to $q_b = 4$ and engages in braking sequence otherwise. When the brake engages the variable i is increased by 1, thereby affecting the velocity of the train as described above. When the velocity falls below $V_L = 20$, the brake disengages after a random amount of time (modeled by the timer c_2 in the state $q_b = 5$), when it switches to the state $q_b = 1$.

Since a braking system is defined for each car, the overall model of the system is roughly a product of N copies of the braking system with the velocity system. For $N = 10$, the number of discrete states of the resulting automaton approaches

30 million. Observe that if the system above is allowed to run forever then all the breaks will eventually fail with probability one. To prevent this, we assume that the brakes can only fail in the first τ units of time. To capture this, we add an additional counter in the breaking subsystem that allows the transition from $q_b = 2$ to the $q_b = 3$ only if this counter is less than τ; this is not shown in the figure. For the simulations, $\tau = 500$.

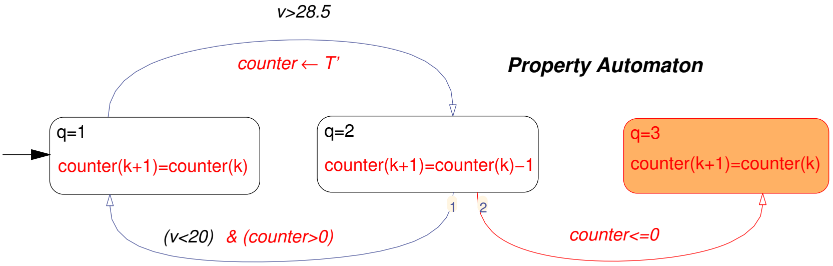


Fig. 4. Property automaton for the train with ECP brakes

The desired behavior of the train is given by the following specification: every time the train velocity increases beyond V_U, the train should brake so that the velocity decreases below V_L. This is given by a liveness automaton that has two states $q = 1$ and $q = 2$, and whose initial state as well as the single acceptance state is $q = 1$. This automaton $\mathcal{P}$ is converted to a safety automaton $\mathcal{P}'$ using a static time out T' according to the approach given in Section 4.3. Figure 4 shows the modified automaton $\mathcal{P}'$, where all the modifications of the original liveness automaton are shown in red. Note that $\mathcal{P}'$ has an additional state $q = 3$ which is the error/bad state.

State estimation. Let $\mathcal{S}$ be the system automaton and $\mathcal{P}'$ the modified property automaton given in Figure 4. Note that $\mathcal{P}'$ has a single error/bad state $q = 3$. We construct the product of $\mathcal{S}$ and $\mathcal{P}'$ to obtain the product automaton $\mathcal{S} \times \mathcal{P}'$. Using this product automaton and using the outputs generated by the actual system, our Monitor M estimates the probability that the state component of the property automaton $\mathcal{P}'$ is the bad state. Thus, it becomes necessary to estimate the probability that the property automaton $\mathcal{P}'$ enters the bad state. This can be achieved by propagating the belief (probability distribution over the states of the product automaton) from the current state to the next state, given the new observation [22]. Particle filters were developed as a computationally efficient approximation of the belief propagation [7, 13, 27]. They have been successfully applied in the hybrid system community for state estimation [17, 4, 14, 29]. These methods become impractical for realistic systems with high number of states and several improvements have been suggested in recent years. It is also worth noting that for particle filters, both estimation accuracy and time complexity increase with the number of particles. The exact relationships depend on the structure of the system and transition probabilities and are difficult to characterize. All these issues are beyond the scope of the present paper.

Experimental results. As described in Section 4, the monitor M computes the probability that the property automaton $\mathcal{P}'$ is in a bad state and raises an alarm when this probability surpasses a given threshold P. In order to evaluate the performance of the monitor numerically, the system was run 1000 times. Particle filter was used to estimate the probability of each state of the product automaton $\mathcal{S} \times \mathcal{P}'$. The number of particles for the particle filter was $\eta = 200$. This number was chosen so that particles were not depleted during the transients corresponding to discrete state transitions. The simulation was terminated when either an alarm was raised, or the discrete time (number of steps the system has taken) reached $T_d = 700$. As explained above, the brakes can only fail during the first $\tau = 500$ units of time. For each run, the number of brakes that failed (the braking system was in $q_b = 3$) was recorded, as well as the state of the monitor. The acceptance and rejection accuracies, respectively, denoted by $AA(M, \mathcal{S}, \mathcal{P}')$ and $RA(M, \mathcal{S}, \mathcal{P}')$ were computed according to:

$$AA(M, \mathcal{S}, \mathcal{P}') = \frac{g_a}{g_a + g_r} \qquad RA(M, \mathcal{S}, \mathcal{P}') = \frac{b_r}{b_a + b_r},$$

where g_a (resp., g_r) is the number of good runs that were accepted (resp., rejected), and b_r (resp., b_a) is the number of bad runs that were rejected (resp., accepted). Note that g_r corresponds to the number of false alarms, and b_a to the number of missed alarms; accuracies approach 1 as these numbers approach 0. A run was considered good if the state of the property automaton at $T_d = 700$ was not equal to 3, and bad otherwise.

Table 1. Results of experiments for different values of z and T'

(a) Monitor outcomes for 1000 runs

z	$T' = 20$					
	g_a	g_r	b_a	b_r	AA	RA
0.050	254	262	0	484	0.492	1.000
0.100	256	222	0	522	0.536	1.000
0.150	256	200	0	544	0.561	1.000
0.300	258	155	0	587	0.625	1.000
0.500	259	117	0	624	0.689	1.000
0.750	259	80	0	661	0.764	1.000
0.875	259	56	0	685	0.822	1.000
0.950	259	43	1	697	0.858	0.999

(b) Monitor accuracies

z	$T' = 20$		$T' = 40$		$T' = 60$	
	AA	RA	AA	RA	AA	RA
0.050	0.49	1.00	0.98	1.00	1.00	1.00
0.100	0.54	1.00	0.99	1.00	1.00	1.00
0.150	0.56	1.00	0.99	1.00	1.00	1.00
0.300	0.62	1.00	1.00	1.00	1.00	1.00
0.500	0.69	1.00	1.00	1.00	1.00	1.00
0.750	0.76	1.00	1.00	1.00	1.00	1.00
0.875	0.82	1.00	1.00	1.00	1.00	1.00
0.950	0.86	1.00	1.00	1.00	1.00	1.00

An example of the monitor performance for $T' = 20$ and different values of the threshold probability z is shown in Table 1a. As expected, if z increases the acceptance accuracy AA increases as it becomes more difficult to reject a run. However, even for $z = 0.95$ the acceptance accuracy does not reach 1 because the small value of T' makes it easy for a run to be rejected. Also, as z increases, the rejection accuracy decreases since it becomes more difficult for the particle filter to estimate that the property automaton entered the fail state with such a high probability.

Table 1b shows accuracy measures of the monitor for different values of the probability threshold z and time out value T'. It can be seen that as z increases the acceptance accuracy increases. Similarly, as T' increases the acceptance accuracy increases. The reason for this is that as z increases, the estimate of the probability that the state of the property automaton $\mathcal{P}'$ is the bad state must be higher before the monitor raises an alarm. Clearly for $z_1 < z_2$, if the monitor with the threshold z_2 would raise an alarm so would the monitor with the threshold z_1, while the reverse is not true. So with lower z, the probability that a false alarm is declared is higher. The same reasoning explains the trends as T' increases. As discussed above, the rejection accuracy may decrease as z increases; similarly, larger T' could lead to more missed alarms. However, these trends can not really be observed in our example due to excellent performance of the particle filter.

6 Conclusion

In this paper we formulated the problem of monitoring both safety and liveness properties for partially observable stochastic systems. Two different notions of monitorability are defined and necessary and sufficient conditions are given for systems to satisfy these properties. Complexity and decidability results for checking these two notions of monitorability for finite state systems are presented. We also presented a general approach for monitoring when the system is specified as a (quantized) probabilistic hybrid automaton and the property is specified as a safety or liveness hybrid automaton. The monitors have been implemented for an example of a train equipped with electronically controlled pneumatic brake. Particle filters were used as state estimators. Experimental results showing the effectiveness of our approach are presented.

References

1. Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
2. Balluchi, A., Benvenuti, L., Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L.: Design of observers for hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 76–80. Springer, Heidelberg (2002)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (2011)
4. Blom, H., Bloem, E.: Particle filtering for stochastic hybrid systems. In: 43rd IEEE Conference on Decision and Control, CDC 2004, vol. 3 (2004)
5. Cappe, O., Moulines, E., Riden, T.: Inferencing in Hidden Markov Models. Springer, Heidelberg (2005)
6. D'Amorim, M., Roşu, G.: Efficient monitoring of ?-languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)
7. Doucet, A., de Freitas, N., Murphy, K., Russell, S.: Rao-Blackwellised particle filtering for dynamic Bayesian networks. In: Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence, pp. 176–183 (2000)

8. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009)
9. Federal Railroad Administration. ECP brake system for freight service (2006), `http://www.fra.dot.gov/downloads/safety/ecp_report_20060811.pdf`
10. Gondi, K., Patel, Y., Sistla, A.P.: Monitoring the full range of ω-regular properties of stochastic systems. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 105–119. Springer, Heidelberg (2009)
11. Hofbaur, M.W., Williams, B.C.: Mode estimation of probabilistic hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 81–91. Springer, Heidelberg (2002)
12. Hofbaur, M.W.: *Hybrid Estimation of Complex Systems*. Lecture Notes in Control and Information Sciences, vol. 319. Springer, Heidelberg (2005)
13. Isard, M., Blake, A.: Condensation–conditional density propagation for visual tracking. International Journal of Computer Vision 29(1), 5–28 (1998)
14. Koutsoukos, X.D., Kurien, J., Zhao, F.: Estimation of distributed hybrid systems using particle filtering methods. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 298–313. Springer, Heidelberg (2003)
15. Kumar, R., Garg, V.: Control of stochastic discrete event systems modeled by probabilistic languages. IEEE Transactions on Automatic Control 46(4), 593–606 (2001)
16. Margaria, T., Sistla, A.P., Steffen, B., Zuck, L.D.: Taming interface specifications. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 548–561. Springer, Heidelberg (2005)
17. McIlraith, S.A., Biswas, G., Clancy, D., Gupta, V.: Hybrid systems diagnosis. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 282–295. Springer, Heidelberg (2000)
18. Pantelic, V., Postma, S., Lawford, M.: Probabilistic Supervisory Control of Probabilistic Discrete Event Systems. IEEE Transactions on Automatic Control 54(8), 2013–2018 (2009)
19. Papoulis, A., Pillai, S.U.: Probability, Random Variables and Stochastic Processes. McGrawHill, NewYork (2002)
20. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
21. Pnueli, A., Zaks, A., Zuck, L.D.: Monitoring interfaces for faults. In: Proceedings of the 5^{th} Workshop on Runtime Verification, RV 2005 (2005); To appear in a special issue of ENTCS
22. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2002)
23. Sammapun, U., Lee, I., Sokolsky, O.: Rt-mac:runtime monitoring and checking of quantitative and probabilistic properties. In: Proc. of 11th IEEE International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA 2005), pp. 147–153 (2005)
24. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008)
25. Sistla, A.P., Zhou, M., Zuck, L.D.: Monitoring off-the-shelf components. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 222–236. Springer, Heidelberg (2005)

26. Sistla, A.P., Žefran, M., Feng, Y.: Monitorability of stochastic dynamical systems. Technical Report CVRL-2011-01, Computer Vision and Robotics Laboratory, University of Illinois at Chicago, Chicago, IL (2011), http://www.cvrl.cs.uic.edu/~milos/publications/papers/cav2011_long.pdf
27. Thrun, S., Fox, D., Burgard, W., Dellaert, F.: Robust Monte Carlo localization for mobile robots. Artificial Intelligence 128(1-2), 99–141 (2001)
28. Vardi, M.: Automatic verification of probabilistic concurrent systems. In: 26th Annual Symposium on Foundations of Computer Science, pp. 327–338. IEEE Computer Society Press, Los Alamitos (1985)
29. Verma, V., Gordon, G., Simmons, R., Thrun, S.: Real-time fault diagnosis. IEEE Robotics & Automation Magazine 11(2), 56–66 (2004)
30. Viswanathan, M., Kim, M.: Foundations for the run-time monitoring of reactive systems - *fundamentals of the maC language*. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 543–556. Springer, Heidelberg (2005)
31. Yoo, T., Lafortune, S.: Polynomial-time verification of diagnosability of partially observed discrete-event systems. IEEE Transactions on Automatic Control 47(9), 1491–1495 (2002)

Equality-Based Translation Validator for LLVM

Michael Stepp, Ross Tate, and Sorin Lerner

University of California, San Diego
{mstepp,rtate,lerner}@cs.ucsd.edu

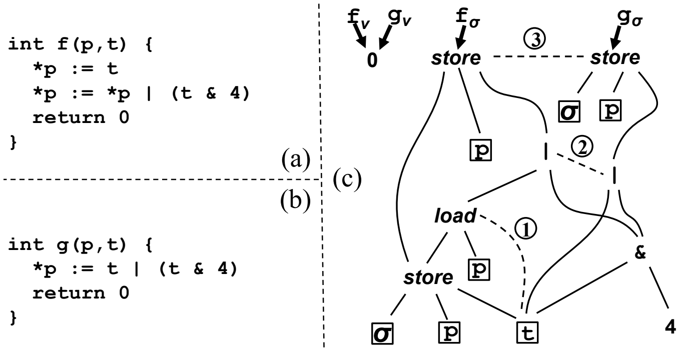
Abstract. We updated our Peggy tool, previously presented in [6], to perform translation validation for the LLVM compiler using a technique called Equality Saturation. We present the tool, and illustrate its effectiveness at doing translation validation on SPEC 2006 benchmarks.

1 Introduction

Compiler optimizations have long played a crucial role in the software ecosystem, allowing programmers to express their programs at higher levels of abstraction without paying the performance cost. At the same time, however, programmers also expect compiler optimizations to be correct, in that they preserve the behavior of the programs they transform. Unfortunately, this seemingly simple requirement is hard to ensure in practice. One approach to improving the reliability of compiler optimizations is a technique called *Translation Validation* [4]. After each run of the optimizer, a separate tool called a translation validator tries to show that the optimized program is equivalent to the corresponding original program. Therefore, a translation validator is just a tool that tries to show two programs equivalent. In our own previous work [6], we developed a technique for reasoning about program equivalence called *Equality Saturation*, and a tool called Peggy that implements this technique. Although our paper focused mostly on performing compiler optimizations using Peggy, we also showed how Equality Saturation can be used to perform translation validation, and that Peggy is an effective translation validator for Soot, a Java bytecode-to-bytecode optimizer.

Inspired by the recent results of Tristan, Govereau and Morrisett [7] on translation validation for LLVM [1], we have updated our Peggy tool so that it can be used to perform translation validation for LLVM, a more aggressive and more widely used compiler than Soot. This updated version of Peggy reuses our previously developed equality saturation engine described in [6]. However, we had to develop several new, LLVM-specific components for Peggy: an LLVM front-end for the intermediate representation used by our equality-saturation engine; new axioms for our engine to reason about LLVM loads, stores and calls; and an updated constant folder that takes into account LLVM operators. Finally, we present new experimental results showing the effectiveness of Peggy at doing translation validation for LLVM on SPEC 2006 C benchmarks.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 737–742, 2011.



(a)

```
int f(p,t) {
  *p := t
  *p := *p | (t & 4)
  return 0
}
```

(b)

```
int g(p,t) {
  *p := t | (t & 4)
  return 0
}
```

Fig. 1. (a) Original code (b) Optimized code (c) Combined E-PEG

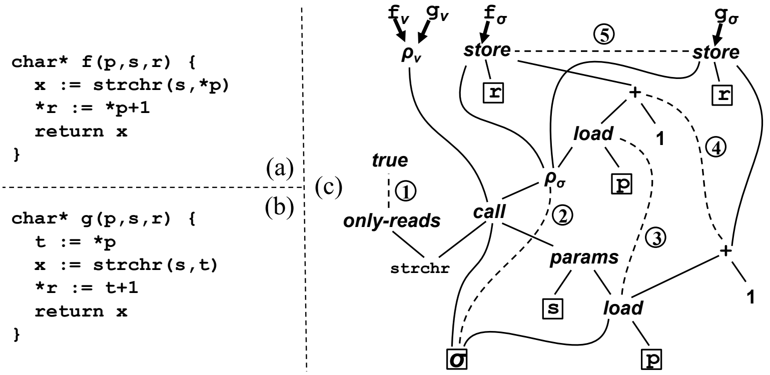
2 Overview

We first present several examples demonstrating how Peggy performs translation validation. These examples are distilled versions of real examples that we found while doing translation validation for LLVM on SPEC 2006.

Example 1. Consider the original code in Figure 1(a) and the optimized code in Figure 1(b). There are two optimizations that LLVM applied here. First, LLVM performed copy propagation through the location `*p`, thus replacing `*p` with `t`. Second, LLVM removed the now-useless store `*p := t`.

Our approach to translation validation uses a representation that we developed and presented previously called Program Expression Graphs [6] (PEGs). A PEG is a pure functional representation of the program which has nice properties for reasoning about equality. Figure 1(c) shows the PEGs for `f` and `g`. The labels $\texttt{f}_v$ and $\texttt{f}_\sigma$ point to the value and heap returned by `f` respectively, and likewise for `g` – for now, let us ignore the dashed lines. In general, a PEG is a (possibly cyclic) expression graph. The children of a node are shown graphically below the node. Constants such as 4 and 0 have no children, and nodes with a square around them represent parameters to the function and also have no children. PEGs encode the heap in a functional way using the standard operators *load* and *store*. In particular, given a heap σ and a pointer p, $\mathit{load}(\sigma, p)$ returns the value at address p; given a heap σ, a pointer p, and a value v, $\mathit{store}(\sigma, p, v)$ returns a new heap identical to σ, except that the value at address p is v. The PEG for `f` takes a heap σ as an input parameter (in addition to `p` and `t`), and produces a new heap, labeled $\texttt{f}_\sigma$, and a return value, labeled $\texttt{f}_v$.

Our approach to translation validation builds the PEGs for both the original and the optimized programs in the same PEG space, meaning that nodes are reused when possible. In particular note how `t & 4` is shared. Once this combined PEG has been constructed, we apply equality saturation, a process by which equality axioms are repeatedly applied to infer equalities between PEG nodes. If



```
char* f(p,s,r) {
  x := strchr(s,*p)
  *r := *p+1
  return x
}
```

```
char* g(p,s,r) {
  t := *p
  x := strchr(s,t)
  *r := t+1
  return x
}
```

Fig. 2. (a) Original code (b) Optimized code (c) Combined E-PEG

through this process Peggy infers that node $\texttt{f}_\sigma$ is equal to node $\texttt{g}_\sigma$ and that node $\texttt{f}_v$ is equal to node $\texttt{g}_v$, then Peggy has shown that the original and optimized functions are equivalent. In the diagrams we use dashed lines to represent PEG node equality (in the implementation, we store equivalence classes of nodes using Tarjan's union-find data structure). Note that E-PEGs are similar to E-graphs from SMT solvers like Simplify [3] and Z3 [2], but specialized for representing programs and with algorithms specialized to handle cyclic expressions which arise much more frequently in E-PEGs than in typical SMT problems.

Peggy proves the equivalence of f and g in the following three steps:

Peggy adds equality① using axiom: $load(store(\sigma, p, v), p) = v$
Peggy adds equality② by congruence closure: $a = b \Rightarrow f(a) = f(b)$
Peggy adds equality③ by axiom: $store(store(\sigma, p, v_1), p, v_2) = store(\sigma, p, v_2)$

By equality③, Peggy has shown that f and g return the same heap, and are therefore equivalent since they are already known to return the same value 0.

Example 2. As a second example, consider the original function f in Figure 2(a) and the optimized version g in Figure 2(b). p is a pointer to an int, s is a pointer to a char, and r is a pointer to an int. The function strchr is part of the standard C library, and works as follows: given a string s (i.e., a pointer to a char), and an integer c representing a character[1], strchr(s,c) returns a pointer to the first occurrence of the character c in the string, or null otherwise. The optimization is correct because LLVM knows that strchr does not modify the heap, and the second load *p is redundant.

The combined PEGs are shown in Figure 2(c). The call to strchr is represented using a *call* node, which has three children: the name of the function, the incoming heap, and the parameters (which are passed as a tuple created by the *params* node). A *call* node returns a pair consisting of the return value and the resulting heap. We use projection operators ρ_v and ρ_σ to extract the return value and the heap from the pair returned by a *call* node.

[1] It may seem odd that c is not declared a char, but this is indeed the interface.

```
int f(x,y,z) {
   for (t:=0; t<z; t:=x*y+t) {}
   return t
}
```

(a)

(b)

```
int g(x,y,z) {
   xy := x*y
   for (t:=0; t<z; t:=xy+t) {}
   return t
}
```

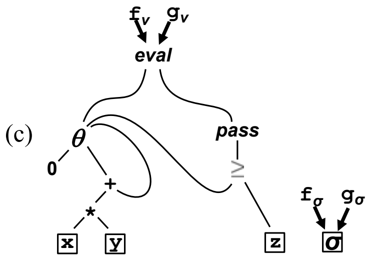


Fig. 3. (a) Original code (b) Optimized code (c) Combined E-PEG

To give Peggy the knowledge that standard library functions like `strchr` do not modify the heap, we have annotated such standard library functions with an *only-reads* annotation. When Peggy sees a call to a function foo annotated with *only-reads*, it adds the equality $\textit{only-reads}(foo) = \textit{true}$ in the PEG. Equality① in Figure 2(c) is added in this way.

Peggy adds equality② using: $\textit{only-reads}(n) = \textit{true} \Rightarrow \rho_\sigma(\textit{call}(n, \sigma, p)) = \sigma$. This axiom encodes the fact that a read-only function call does not modify the heap. Equalities③,④, and⑤ are added by congruence closure.

In these 5 steps, Peggy has identified that the heaps $\mathtt{f}_\sigma$ and $\mathtt{g}_\sigma$ are equal, and since the returned values $\mathtt{f}_v$ and $\mathtt{g}_v$ are trivially equal, Peggy has shown that the original and optimized functions are equivalent.

Example 3. As a third example, consider the original code in Figure 3(a) and the optimized code in Figure 3(b). LLVM has pulled the loop-invariant code `x*y` outside of the loop. The combined PEG for the original function `f` and optimized function `g` is shown in Figure 3. As it turns out, `f` and `g` will produce the exact same PEG, so let us focus on understanding the PEG itself. The θ node represents the *sequence* of values that `t` takes throughout the loop. The left child of the θ node is the first element of the sequence (0 in this case) and the right child provides any element in the sequence after the first one in terms of the previous element. The *eval*/*pass* pair is used to extract the value of `t` after the loop. The $\geq$ node is a lifting of $\geq$ to sequences, and so it represents the sequence of values that `t` $\geq$ `z` takes throughout the loop. *pass* takes a sequence of booleans and returns the index of the first boolean in the sequence that is true. Therefore *pass* in this case returns the index of the last iteration of the loop. *eval* takes a sequence of values and an integer index, and returns the element of the sequence at that index. Therefore, *eval* returns the value of `t` after the last iteration of the loop (a denotational semantics of PEGs can be found in [6]).

As Figure 3 shows, the PEG for the optimized function `g` is the same as the original function `f`. Peggy has validated this example just by converting to PEGs, without even running equality saturation. One of the key advantages of PEGs is that they are agnostic to code-placement details, and so Peggy can validate code placement optimizations such as loop-invariant code motion, lazy code motion, and scheduling by just converting to PEGs and checking for syntactic equality.

3 Implementation

Axioms. Peggy uses a variety of axioms to infer equality information. Some of these axioms were previously developed and state properties of built-in PEG operators like θ, *eval*, *pass*, and ϕ (which are used for conditionals). We also implemented LLVM specific axioms to reason about *load* and *store*, some of which we have already seen. An additonal such axiom is important for moving unaliased loads/stores across each other: $p \neq q \Rightarrow load(store(\sigma, q, v), p) = load(\sigma, p)$.

Alias Analysis. The axiom above can only fire if $p \neq q$, requiring alias information. Our first attempt was to encode an alias analysis using axioms, and run it using the saturation engine. However, this added a significant run-time overhead, and so we instead took the approach from [7], which is to pre-compute alias information; we then used this information when applying axioms.

Generating Proofs. As described in our follow-up work [5], after Peggy validates a transformation it can use the resulting E-PEG to generate a proof of equivalence of the two programs. This proof has already helped us determine how often axioms are useful. In the future, we could also use this proof to improve the run-time of our validator: after a function f has been validated, we could record which axioms were useful for f, and enable only those axioms for subsequent validations of f (reverting back to all axioms if the validation fails).

4 Results

We used Peggy to perform translation validation for LLVM 2.8 on SPEC 2006 C benchmarks. We enabled the following optimizations: dead code elimination, global value numbering, partial redundancy elimination, sparse conditional constant propagation, loop-invariant code motion, loop deletion, loop unswitching, dead store elimination, constant propagation, and basic block placement.

Figure 4 shows the results: "#func" and "#instr" are the number of functions and instructions; "%success" is the percentage of functions whose compilation Peggy validated ("All" considers all functions; "OC", which stands for "Only Changed", ignores functions for which LLVM's output is identical to the input); "To PEG" is the average time per function to convert from CFG to PEG; "Avg Engine Time" is the average time per function to run the equality saturation engine ("Success" is over successful runs, and "Failure" over failed runs).

Overall our results are comparable to [7]. However, because of implementation differences (including the set of axioms), an in-depth and meaningful experimental comparison is difficult. Nonetheless, conceptually the main difference is that [7] uses axioms for destructive rewrites, whereas we use axioms to add equality information to the E-PEG, thus expressing multiple equivalent programs at once. Our approach has several benefits over [7]: (1) we simultaneously explore an exponential number of paths through the space of equivalent programs, whereas [7] explores a single linear path – hence we explore more of the search space; (2) we need not worry about axiom ordering, whereas [7] must

Benchmark	#func	#instr	%success		To	Avg Engine Time	
			All	OC	PEG	Success	Failure
400.perlbench	1,864	269,631	79.0%	73.3%	0.531s	1.028s	11s
401.bzip2	100	16,312	82.0%	76.9%	0.253s	0.733s	19s
403.gcc	5,577	828,962	80.8%	74.9%	0.558s	0.700s	19s
429.mcf	24	2,541	87.5%	87.0%	0.216s	0.500s	19s
433.milc	235	21,764	80.4%	75.0%	0.246s	0.188s	9s
456.hmmer	538	57,102	86.4%	84.6%	0.285s	0.900s	11s
458.sjeng	144	23,807	77.1%	72.5%	1.099s	0.253s	7s
462.libquantum	115	5,864	73.9%	64.3%	0.123s	0.167s	8s
464.h264ref	590	131,627	74.2%	70.5%	0.587s	0.846s	12s
470.lbm	19	3,616	78.9%	76.5%	0.335s	0.154s	3s
482.sphinx3	369	28,164	88.1%	86.0%	0.208s	0.480s	12s

Fig. 4. Results of running Peggy's translation validator on SPEC 2006 benchmarks

pick a good ordering of rewrites for LLVM – hence it is easier to adapt our approach to new compilers, and a user can easily add/remove axioms (without worrying about ordering) to balance precision/speed or to specialize for a given code base; (3) our approach effectively reasons about loop-induction variables, which is more difficult using the techniques in [7]. However, the approach in [7] is faster, because it explores a single linear path through the space of programs.

Failures were caused by: (1) incomplete axioms for linear arithmetic; (2) insufficient alias information; (3) LLVM's use of pre-computed interprocedural information, even in intraprocedural optimizations. These limitations point to several directions for future work, including incorporating SMT solvers and better alias analyses, and investigating interprocedural translation validation.

References

1. The LLVM compiler infrastructure, `http://llvm.org`
2. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
3. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
4. Pnueli, A., Siegel, M.D., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, p. 151. Springer, Heidelberg (1998)
5. Tate, R., Stepp, M., Lerner, S.: Generating compiler optimizations from proofs. In: POPL (2010)
6. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL (2009)
7. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011)

Model Checking Recursive Programs with Numeric Data Types

Matthew Hague and Anthony Widjaja Lin

Oxford University Computing Laboratory

Abstract. Pushdown systems (PDS) naturally model sequential recursive programs. Numeric data types also often arise in real-world programs. We study the extension of PDS with unbounded counters, which naturally model numeric data types. Although this extension is Turing-powerful, reachability is known to be decidable when the number of reversals between incrementing and decrementing modes is bounded. In this paper, we (1) pinpoint the decidability/complexity of reachability and linear/branching time model checking over PDS with reversal-bounded counters (PCo), and (2) experimentally demonstrate the effectiveness of our approach in analysing software. We show reachability over PCo is NP-complete, while LTL is coNEXP-complete (coNP-complete for fixed formulas). In contrast, we prove that EF-logic over PCo is undecidable. Our NP upper bounds are by a direct poly-time reduction to satisfaction over existential Presburger formulas, allowing us to tap into highly optimized solvers like Z3. Although reversal-bounded analysis is incomplete for PDS with unbounded counters in general, our experiments suggest that some intricate bugs (e.g. from Linux device drivers) can be discovered with a small number of reversals. We also pinpoint the decidability/complexity of various extensions of PCo, e.g., with discrete clocks.

1 Introduction

Pushdown systems (PDS) are natural abstractions of sequential programs with unbounded recursions whose verification problems have been extensively studied (cf. [4,12,35]). In addition to recursions, numerical data types commonly arise in real-world programs. A standard approach to these potentially infinite domains is to map them into abstract domains that are more amenable to analysis (e.g. finite ones like $\{\mathrm{Pos}, \mathrm{Neg}, \mathrm{Zero}\}$, or infinite ones expressed by intervals, difference bound matrices, polyhedra, etc.). For other types of program analysis, it is also common to simply ignore numerical data types. For a comprehensive treatment of these techniques, and others, the reader is referred to the survey [11].

In this paper, we study a different approach. Motivated by the success of pushdown systems (or similar models like boolean programs) in software model checking (cf. [2,3,29]), we aim to investigate extensions of pushdown systems with numerical data types and preserve nice properties, such as decidability and good complexity. A clean and simple approach is to enrich pushdown systems with unbounded counters, which can be incremented/decremented and tested for zero. Unfortunately, this model is Turing-powerful, even without the stack.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 743–759, 2011.

One way to retain decidability of reachability is to impose an upper bound r on the number of reversals between incrementing and decrementing modes for each counter (cf. [9,19]). In fact, decidability holds even if discrete clocks are added to the model [9]. On the other hand, the complexity of reachability over these models is still open; a simple analysis of [9,19] yields at least double exponential-time complexity for their algorithms. Recently, the authors of [13] observed that replacing the use of Parikh's Theorem [24] in [19] by a recently improved version of [34] immediately yields an NP procedure for this reachability problem (without clocks) for *fixed* numbers of reversals and counters, yielding only an NEXP procedure in general. Furthermore, the decidability of linear/branching time model checking over these models is also unknown.

There are at least two potential applications of reversal-bounded model checking (cf. [9,20]). First, it could be used as a sound but *incomplete* verification technique for the case of unbounded reversals. Despite this incompleteness, results in bounded model checking suggest that "shallow" bugs are common in practice (cf. [11]). Clearly, reversal-bounded model checking is an infinite-state generalization of bounded model checking over counter systems. A second application is the use of reversal-bounded counters for tracking the number of times certain actions have been executed to reach the current configuration. For example, we can check the existence of a computation path in a recursive program where the number of invocations for the functions f_1, f_2, f_3, and f_4 are the same. Similar counting properties (and their model checking problems) have been studied in many other contexts (cf. [22] and references therein).

Contributions. We begin by studying pushdown systems enriched with reversal-bounded counters, which can be compared against and incremented by constants given in binary, but without clocks (PCo). This model is more general than the model studied in [13,19], which allows only counters to be compared against 0 and incremented by $\{-1, 0, 1\}$, though at the cost of an exponential blow-up (cf. [20]) our model can be translated into their model. Our main contributions are (1) to pinpoint the decidability/complexity of reachability and linear/branching time model checking over PCo, and (2) experimentally demonstrate the effectiveness of our approach in the analysis of software.

We show that reachability over PCo is NP-complete, while LTL model checking is coNEXP-complete (coNP-complete for fixed formulas). In contrast, we prove that model checking EF-logic over PCo is undecidable. All of our lower bounds hold already for PDS with one 1-reversal counter wherein numeric constants, which can be either be compared against or used to increment/decrement counters, are restricted to 0 or 1. Our NP upper bounds are established by a direct poly-time reduction to satisfaction over existential Presburger formulas, allowing us to tap into highly optimized solvers like Z3 [10]. This reduction also permits additional constraints on the number of actions executed and the values of the counters at the end of the run (also specified in existential Presburger arithmetic) without further computation overhead in the reduction. We have implemented our algorithm and use it to analyse several examples, including some derived from memory management issues in Linux device drivers. Although reversal-bounded analysis is only complete up to the bound on the number of reversals, our experiments suggest that many subtle bugs manifest themselves even within a small number of reversals, which our tool can detect reasonably fast. Without increasing complexity, our algorithm can also check whether a given PDS $\mathcal{P}$ with unbounded counters is r-reversal bounded, for a given input $r \in \mathbb{N}$; note that this is not the same as deciding whether $\mathcal{P}$ is reversal-bounded, which is undecidable [14]. In the case when $\mathcal{P}$ is r-reversal bounded, our technique gives a complete coverage of the infinite state space, which suggests the usefulness of our technique in proving correctness as well as finding bugs.

We then study the extension of PCo with discrete clocks (PCC). We show that LTL model checking over PCC is still coNEXP-complete, but hardness holds even for a fixed formula. Similarly, we show that reachability over PCC is NEXP-complete. We also show that, without reversal-bounded counters, model checking EF-logic over PCC is EXPSPACE-complete even for a fixed formula.

Related work. The complexities of most standard model checking problems over pushdown systems are well-understood. In relation to our results, we mention that LTL model checking over PDS is EXP-complete and is P-complete for fixed formulas [4] (the latter is also the complexity for reachability), whereas model checking EF-logic is PSPACE-complete [4,35]. Therefore, adding reversal-bounded counters yields computationally harder problems in both cases.

Over reversal-bounded counter systems (without stack), reachability is NP-complete but becomes NEXP-complete when the number of reversals is given in binary [18]. On the other hand, when the numbers of reversals and counters are fixed, the problem is solvable in P [16]. The techniques developed by [16,18], which reason about the maximal counter values, are very different to our techniques, which exploit the connection to Parikh images of pushdown automata (first explicated in Ibarra's original paper [19] though not in a way that gives optimal complexity or practical algorithm). For LTL model checking, the problem is solvable in EXP even in the presence of discrete clocks [31], whereas EF-logic model checking is still decidable but becomes undecidable for CTL [31].

For discrete-timed systems, reachability is known to be PSPACE-complete [1], where hardness holds already for three clocks [8]. Using region graph constructions [1], LTL model checking and EF-logic can also be easily shown to be PSPACE-complete. Note that we do not consider timed logics (cf. [5]). The complexities of pushdown systems with clocks have also been studied, e.g., [7].

Organization. §2 contains preliminaries. In §3, we define the basic model PCC that we study. §4 and §5, contain upper and lower bounds for model checking PCo. In §6, we extend our results to PCC. Experimental results are given in §7. Other extensions and future work are given in §8. Due to the space limit, some proofs are in the full version from the project page `http://www.cs.ox.ac.uk/recount`.

2 Preliminaries

Transition systems. An *action alphabet* ACT is a finite nonempty set of *actions.* A *transition system* over ACT is a tuple $\mathfrak{S} = \langle S, \{\rightarrow_a\}_{a \in \mathsf{ACT}}\rangle$, where S is a set of *configurations* and each $\rightarrow_a \subseteq S \times S$ is an a-labeled *transition relation* containing the set of all a-labeled *transitions.* We use $\rightarrow$ to denote the union of all $\rightarrow_a$. A *(computation) path* in $\mathfrak{S}$ is a finite or infinite sequence $\pi = \alpha_0 \rightarrow_{a_1} \alpha_1 \rightarrow_{a_2} \ldots$ such that $\alpha_i \in S$, $\alpha_i \rightarrow_{a_{i+1}} \alpha_{i+1}$ and $a_i \in \mathsf{ACT}$ for each i. If π is finite, let $\mathbf{last}(\pi)$ denote the last configuration in π. In this case, $a_1 a_2 \ldots$ is said to be a *(finite) trace in $\mathfrak{S}$ from α_0 to **last**(π)*. **Automata** We assume familiarity with automata theory. In particular, nondeterministic Büchi automata (NBWA), cf. [32], and pushdown automata (PDA), cf. [27]. For an NBWA $\mathcal{A}$, we denote by $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$. Similarly, given a PDA we also write $\mathcal{L}(\mathcal{P})$ for the language $\mathcal{P}$ recognizes.

Parikh images. Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$ and a word $w \in \Sigma^*$, we write $\mathbb{P}(w)$ to denote a tuple with $|\Sigma|$ entries where the ith entry counts the number of occurrences of a_i in w. Given a language $\mathcal{L} \subseteq \Sigma^*$, we write $\mathbb{P}(\mathcal{L})$ to denote the set $\{\mathbb{P}(w) : w \in \mathcal{L}\}$. We say that $\mathbb{P}(\mathcal{L})$ is the *Parikh image* of $\mathcal{L}$.

Logic The syntax of LTL (cf. [17,31,32]) over ACT is given by: $\varphi, \varphi' := a$ $(a \in \mathsf{ACT})$ $\mid \neg\varphi$ $\mid \varphi \vee \varphi'$ $\mid \varphi \wedge \varphi'$ $\mid \mathbf{X}\varphi$ $\mid \varphi \mathbf{U} \varphi'$. Given an ω-word $w \in \mathsf{ACT}^\omega$ and an LTL formula φ over ACT, we define the satisfaction relation $w \models \varphi$ in the standard way. Write $[\![\varphi]\!]$ for all $w \in \mathsf{ACT}^\omega$ such that $w \models \varphi$. EF-logic (over ACT) is a fragment of CTL (cf. [17,31]) with the syntax $\varphi, \psi := \top \mid \neg\varphi \mid \varphi \vee \psi \mid \langle a \rangle \varphi$ $(a \in \mathsf{ACT})$ $\mid \mathsf{EF}\varphi$. Given an EF formula φ, a transition system $\mathfrak{S} = \langle S, \{\rightarrow_a\}_{a \in \mathsf{ACT}}\rangle$ and $s \in S$, we may define $\mathfrak{S}, s \models \varphi$ in the standard way.

Presburger formulas. Presburger formulas are first-order formulas over natural numbers with addition. Here, we use extended existential Presburger formulas $\exists x_1, \ldots, x_k.\varphi$ where φ is a boolean combination of expressions $\sum_{i=1}^{k} a_i x_i \sim b$ for constants $a_1, \ldots, a_k, b \in \mathbb{Z}$ and $\sim \in \{\leq, \geq, <, >, =\}$ with constants represented in binary. It is known that satisfiability of existential Presburger formulas is NP-complete even with these extensions (cf. [26]).

3 Pushdown Systems with Counters and Clocks

The model. An *atomic clock constraint* on clock variables $Y = \{y_1, \ldots, y_m\}$ is simply an expression of the form $y_i \sim c$ or $y_i - y_j \sim c$, where $\sim \in \{<, >, =\}$, $1 \leq i, j \leq m$ and $c \in \mathbb{Z}$. An *atomic counter constraint* on counter variables $X = \{x_1, \ldots, x_n\}$ is simply an expression of the form $x_i \sim c$, where $c \in \mathbb{Z}$. A *clock-counter (CC) constraint* θ on (X, Y) is simply a boolean combination of atomic counter constraints on X and atomic clock constraints on Y. Given a valuation $\nu : X \cup Y \rightarrow \mathbb{N}$ to the counter/clock variables, we can determine whether $\theta[\nu]$ is true or false by replacing a variable z by $\nu(z)$ and evaluting the resulting arithmetic expressions in the obvious way. Let $Const_{X,Y}$ denote the set of all CC constraints on (X, Y). Intuitively, these formulas will act as "enabling conditions" (or "guards") to determine whether certain transitions can be fired.

A *pushdown system with* n *counters and* m *discrete clocks* is a tuple $\mathcal{P} = (Q, \mathsf{ACT}, \Gamma, \delta, X, Y)$ where (1) Q is a finite set of states, (2) ACT is a set of action labels, (3) Γ is a stack alphabet, (4) $X = \{x_1, \ldots, x_n\}$ is a set of counter variables, (5) $Y = \{y_1, \ldots, y_m\}$ is a set of clock variables, and (6) δ is a transition relation of $\mathcal{P}$, which is defined to be a finite subset of $(Q \times \Gamma^* \times Const_{X,Y}) \times \mathsf{ACT} \times (Q \times \Gamma^* \times 2^Y \times \mathbb{Z}^n)$. A *configuration* of $\mathcal{P}$ is a tuple $(q, u, \mathbf{u}, \mathbf{v}) \in Q \times \Gamma^* \times \mathbb{N}^n \times \mathbb{N}^m$. Given $a \in \mathsf{ACT}$ and two configurations $\alpha_1 = (q, u, \mathbf{u}, \mathbf{v})$ and $\alpha_2 = (q', u', \mathbf{u}', \mathbf{v}')$ of $\mathcal{P}$, $\alpha_1 \rightarrow_{a,\mathcal{P}} \alpha_2$ iff there exists $\langle (q, w, \theta), a, (q', w', Y', \mathbf{w}) \rangle \in \delta$ such that

- θ holds under the valuation $(\mathbf{u}, \mathbf{v})$,
- for some $v \in \Gamma^*$, $u = wv$ and $u' = w'v$,
- if $Y' = \emptyset$, then each clock progresses by one time unit, i.e., $\mathbf{v}' = \mathbf{v} + \mathbf{1}$; if $Y' \neq \emptyset$, then the clocks in Y' are reset, while other clocks do not change, i.e., for each $y_i \in Y$, set $v'_i := 0$ and, for each $y_i \notin Y$, set $v'_i := v_i$.
- $\mathbf{u}' := \mathbf{u} + \mathbf{w}$ (note, all elements of $\mathbf{u}'$ must be non-negative).

The transition system generated by $\mathcal{P}$ is $\mathfrak{S}_{\mathcal{P}} = \langle S, \{\rightarrow_a\}_{a \in \mathsf{ACT}} \rangle$, where S denotes the set of configurations of $\mathcal{P}$ and $\rightarrow_a$ is defined to be $\rightarrow_{a,\mathcal{P}}$.

Notice that in this model, which is similar to the model given in [9], clocks are updated via transitions. This is slightly different from the usual definition of discrete timed systems (cf. [1]) in which (1) clocks may progress within any particular state of the system without taking any transitions as long as some *invariants* are satisfied, and (2) transitions are *instantaneous*. However, by introducing self-looping transitions with no reset and adding an extra "dummy" clock which always resets for old transitions, we can easily construct a weakly bisimilar system in our model. See [9] for more details.

Let us now define the r-reversal-bounded variant of this model for each $r \in \mathbb{N}$. Syntactically, a *pushdown system with* n r*-reversal bounded counters and* m *discrete clocks* is simply a pair $(r, \mathcal{P})$ of number r and a pushdown system $\mathcal{P}$ with n counters and m clocks. Together with a given initial configuration α of $\mathcal{P}$, the system $(r, \mathcal{P})$ generates a transition system $\mathfrak{S}^{\alpha}_{(r,\mathcal{P})} = \langle S, \{\rightarrow_a\}_{a \in \mathsf{ACT}} \rangle$ defined as follows. Let $\mathfrak{S}_{\mathcal{P}} = \langle S', \{\rightarrow'_a\}_{a \in \mathsf{ACT}} \rangle$ be the transition system generated by $\mathcal{P}$. A path π in $\mathfrak{S}_{\mathcal{P}}$ from α is said to be r*-reversal-bounded* if each counter of $\mathcal{P}$ changes from a non-incrementing mode to non-decrementing mode (or vice versa) at most r times in the path π. For example, if the values of a counter x in a path π from α are $1, 1, 1, 2, 3, 4, 4, \overline{4, 3}, 2, \overline{2, 3}$, then the number of reversals of x is 2 (reversals occur in between the overlined positions). This sequence has three *phases* (i.e. subpaths interleaved by consecutive reversals or end points): non-decrementing, non-incrementing, and finally non-decrementing. This intuition suffices for understanding the main ideas in this paper, though we provide the detailed definitions in the full version. The set S is then defined to be the set of all finite r-reversal-bounded paths from α. Given two such paths π and π' such that $\pi' = \pi, \alpha'$, we define $\pi \rightarrow_a \pi'$ iff $\mathbf{last}(\pi) \rightarrow'_a \alpha'$. Notice that $\mathfrak{S}^{\alpha}_{(r,\mathcal{P})}$ is a tree.

We write (r, n, m)-PCC to denote the set of all pushdown systems with n r-reversal-bounded counters and m discrete clocks. We write PCC to denote the union of all (r, n, m)-PCC for all $r, n, m \in \mathbb{N}$. Similarly, we use (r, n)-PCo to denote $(r, n, 0)$-PCC and (r, m)-PCl to denote $(r, 0, m)$-PCC. We use PCo and PCl as well in the same way.

Unless stated otherwise, we make the following conventions: (1) the number r of reversals is given in unary, and (2) numeric constants in CC constraints and counter increments are given in binary. In the sequel, we will deal with (control-state) reachability, model checking LTL and EF over PCC (and their variants) defined as follows:

- *Reachability*: given a PCC $\mathcal{P}$ and two configurations α, α' of $\mathcal{P}$ (in binary representation), decide whether α' is reachable in $\mathfrak{S}_{\mathcal{P}}^{\alpha}$.
- *Control-state reachability*: given a PCC $\mathcal{P}$ and two states q, q' of $\mathcal{P}$, decide whether there exist stack contents $u, u' \in \Gamma$ and counter values $\mathbf{c}, \mathbf{c}' \in \mathbb{N}^k$ such that $(q', u', \mathbf{c}')$ is reachable in $\mathfrak{S}_{\mathcal{P}}^{(q,u,\mathbf{c})}$.
- *LTL model checking*: given a PCC $\mathcal{P}$, a configuration α of $\mathcal{P}$ (in binary), and an LTL formula φ, decide whether $\mathfrak{S}_{\mathcal{P}}^{\alpha}, \alpha \models \varphi$.
- *EF model checking*: given a PCC $\mathcal{P}$, a configuration α of $\mathcal{P}$ (in binary), and an EF formula φ, decide whether $\mathfrak{S}_{\mathcal{P}}^{\alpha}, \alpha \models \varphi$.

4 Upper Bounds for Model Checking PCo

In this section, we show that reachability over PCo is in NP by providing a direct poly-time reduction to satisfactions of existential Presburger formulas. As applications of our technique, we will provide: (1) an NP upper bound for control-state reachability with additional constraints on counter values at the beginning/end of the run and how many times actions are executed at the end of the run, and (2) a coNEXP upper bound for LTL model checking over PCo (coNP for fixed formulas). Lower bounds are proved in the next section.

An NP procedure for reachability over PCo. We prove the following.

Theorem 1. *Reachability over PCo is NP-complete. In fact, it is poly-time reducible to checking satisfactions of existential Presburger formulas.*

Given an (r, k)-PCo $\mathcal{P} = (Q, \mathsf{ACT}, \Gamma, \delta, X)$, and two configurations $\alpha = (q, u, \mathbf{c})$ and $\alpha' = (q', u', \mathbf{c}')$ of $\mathcal{P}$, our algorithm decides if α' is reachable from α in $\mathfrak{S}_{\mathcal{P}}$. Let $\mathbf{c} = (c_1, \ldots, c_k)$ and $\mathbf{c}' = (c_1', \ldots, c_k')$. We assume that $u = u' = \perp$ since, by hardwiring u and u' into the finite control of $\mathcal{P}$ the standard way, the PCo $\mathcal{P}$ may initialize the stack content to u and make sure the final stack content is precisely u'. In addition, let $d_1 < \ldots < d_m$ denote all the numeric constants appearing in an atomic counter constraint as a part of CC constraints in $\mathcal{P}$. Without loss of generality, we assume that $d_1 = 0$ for notational convenience. Let $\mathtt{REG} = \{\varphi_1, \ldots, \varphi_m, \psi_1, \ldots, \psi_m\}$ be a set of formulas defined as follows. Note that these formulas partition $\mathbb{N}$ into $2m$ pairwise disjoint regions.

$$\varphi_i(x) \equiv x = d_i, \quad \psi_i(x) \equiv d_i < x < d_{i+1} \ (1 \leq i < m), \quad \psi_m(x) \equiv d_m < x \ .$$

A vector $\mathbf{v}$ in $\mathbf{Modes} = \mathtt{REG}^k \times [0, r]^k \times \{\uparrow, \downarrow\}^k$ is said to be a *mode vector*. Given a path $\pi = \alpha_0 \alpha_1 \ldots \alpha_h$ from α to α', we may associate a mode vector $\mathbf{v}_j$

to each configuration α_i in π that records for each counter: which region its value is in, how many reversals its used, and whether its phase is non-decrementing ($\uparrow$) or non-incrementing ($\downarrow$). Consider the sequence $\sigma = \{\mathbf{v}_j\}_{j=0}^{h}$ of mode vectors. A crucial observation is that each mode vector $\mathbf{v} \in \mathbf{Modes}$ in this sequence occurs in a contiguous block, i.e., if $0 \leq j \leq j' \leq h$ are such that $\mathbf{v}_j$ (resp. $\mathbf{v}_{j'}$) is the first (resp. last) time $\mathbf{v}$ appearing in σ, then $\mathbf{v}_l = \mathbf{v}$ for all $l \in [j, j']$. Intuitively, once a change occurs in σ, we cannot revert to the previous vector. This is because any such change will incur an extra reversal for at least one counter. There are at most $N_{\max} := |\mathtt{REG}| \times (r+1) \times k = 2mk(r+1)$ distinct mode vectors in σ.

In outline, to avoid an exponential blow-up in our reduction, we will first construct a very rough "upperapproximation" of the PCo $\mathcal{P}$ as a PDA $\mathcal{P}'$. Intuitively, $\mathcal{P}'$ will simulate $\mathcal{P}$, while guessing and remembering *only* how many mode vector changes have occurred, but disregarding the counter information. In this case, there are runs in $\mathcal{P}'$ that are not valid in $\mathcal{P}$. Each time $\mathcal{P}'$ fires a transition t (derived from a transition t' of $\mathcal{P}$ by disregarding counters), it will also output information about the counter tests and in(de)crements associated with t', and how many changes in the mode counter have occurred thus far (recorded in the states of $\mathcal{P}'$). Since $\mathcal{P}'$ is of polynomial size, we apply on $\mathcal{P}'$ the linear-time algorithm of Verma *et al.* [34] to compute the Parikh images of CFGs (equivalently, PDA). Building on the output formula, we use further existential quantifications to guess the evolution of the mode vectors, on which we impose further constraints to eliminate invalid runs. We give the details below.

Building the PDA $\mathcal{P}$. Define $\mathcal{P}' = (Q', \mathsf{ACT}', \Gamma, \delta', (q, 0), F')$ allowing transitions to execute a (finite) *sequence* of actions, instead of just one. These are for convenience and can be encoded in the states of $\mathcal{P}'$. Let $Q' = Q \times [0, N_{\max} - 1]$ and define ACT' implicitly from δ'. In fact, ACT' is a (finite) subset of $\{(\mathtt{ctr}_i, u, j, l) : i \in [1, k], u \in \mathbb{Z}, j \in [0, N_{\max} - 1], l \in \{0, 1\}\} \cup (Const_{X,\emptyset} \times [0, N_{\max} - 1])$. Here, $l \in \{0, 1\}$ signifies whether this action changes the mode vector. We define δ' by initially setting $\delta' = \emptyset$ and adding rules as follows. If $\langle (q, w, \theta), a, (q', w', \mathbf{u}) \rangle \in \delta$ where $\mathbf{u} = (u_1, \ldots, u_k)$ then for each $i \in [0, N_{\max} - 1]$ we add

$$\langle ((q, i), w), (\theta, i)(\mathtt{ctr}_1, u_1, i, 0)(\mathtt{ctr}_2, u_2, i, 0) \ldots (\mathtt{ctr}_k, u_k, i, 0), ((q, i), w') \rangle$$

to δ'. If $i \in [0, N_{\max} - 1)$, we also add the following rule:

$$\langle ((q, i), w), (\theta, i)(\mathtt{ctr}_1, u_1, i, 1)(\mathtt{ctr}_2, u_2, i, 1) \ldots (\mathtt{ctr}_k, u_k, i, 1), ((q, i+1), w') \rangle$$

In this way, $\mathcal{P}'$ makes "visible" the counter tests and the counter updates performed. Finally, the set F' of final states are defined to be $\{q'\} \times [0, N_{\max} - 1]$ and the initial control state is $(q, 0)$.

Constructing the formula. Fix an ordering on ACT', say $\alpha_1 < \ldots < \alpha_l$. For convenience, by f we denote a function mapping α_i to i for each $i \in [1, l]$. We apply the linear-time algorithm of [34] on $\mathcal{P}'$ above to obtain $\chi(\mathbf{z})$, where $\mathbf{z} = (z_1, \ldots, z_l)$, such that for each $\mathbf{n} \in \mathbb{N}^l$ we have $\mathbf{n} \in \mathbb{P}(\mathcal{L}(\mathcal{P}'))$ iff $\chi(\mathbf{n})$ holds. We impose constraint χ to eliminate vectors that do not correspond to traces of $\mathcal{P}'$. Currently, $\mathcal{P}'$ knows only the maximum number of allowed changes in mode vectors, but is not "aware" of other information about the counters. Therefore, the formula that we construct should assert the existence of a valid sequence of mode vectors that respect the counter tests and updates that $\mathcal{P}'$ outputs. We construct `HasRun` of the form

$$\exists \mathbf{z} \exists \mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1} \left(\begin{array}{c} \texttt{Init}(\mathbf{m}_0) \wedge \texttt{GoodSeq}(\mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1}) \wedge \chi(\mathbf{z}) \wedge \\ \texttt{Respect}(\mathbf{z}, \mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1}) \wedge \texttt{EndVal}(\mathbf{z}) \end{array} \right)$$

where $\mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1}$ are variables for representing a valid sequence of mode vectors occurring in the run of $\mathcal{P}$.

Let us now elaborate `HasRun`. First, since a mode vector is a member of $\mathbf{Modes} = \texttt{REG}^k \times [0, r]^k \times \{\downarrow, \uparrow\}^k$, we set $\mathbf{m}_i$ to be a tuple of variables $\{reg_j^i, rev_j^i, arr_j^i : j \in [1, k]\}$, where:

- reg_j^i is a variable that will range over $[1, 2m]$ denoting which region the jth counter is in (a number of the form $2i + 1$ refers to φ_i, while $2i$ refers to ψ_i).
- rev_j^i is a variable that will range over $[0, r]$ denoting the number of reversals that have been used thus far by the jth counter.
- arr_j^i is a variable that will range over $\{0, 1\}$ denoting the current arrow direction, e.g., $0/1$ for $\uparrow/\downarrow$ (non-decrementing/non-incrementing mode).

Using $\texttt{Init}(\mathbf{m}_0)$, we assert that the initial mode vector needs to respect the given initial configuration $\alpha = (q, u, \mathbf{c})$, where $\mathbf{c} = (c_1, \ldots, c_k)$. More precisely,

$$\texttt{Init}(\mathbf{m}_0) \equiv \bigwedge_{j=1}^{k} \left(rev_j^0 = 0 \wedge \bigwedge_{i=1}^{m} \left(\begin{array}{c} reg_j^0 = 2i - 1 \leftrightarrow \varphi_i(c_j) \bigwedge \\ reg_j^0 = 2i \leftrightarrow \psi_i(c_j) \end{array} \right) \right).$$

In fact, since $\mathbf{c}$ is given, we could replace some of these variables by constants. However, being able to define this in the formula allows us to prove something more general, as we will see later.

Recall that the target configuration is $\alpha' = (q', u', \mathbf{c}')$, where $\mathbf{c}' = (c_1', \ldots, c_k')$. We assert that the end counter values match $\mathbf{c}'$. This definition is given as follows. Note, multiplications by constants are allowed within Presburger arithmetic.

$$\texttt{EndVal}(\mathbf{z}) \equiv \bigwedge_{j=1}^{k} \left(\sum_{i=0}^{r} \sum_{(\texttt{ctr}_j, d, i, l) \in \mathsf{ACT}'} d \times z_{f(\texttt{ctr}_j, d, i, l)} \right) = c_j'.$$

We define $\texttt{GoodSeq}(\mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1})$ to express that $\mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1}$ is a valid sequence of mode counters. The formula is a conjunction of smaller formulas defined below. One conjunct says that each rev_j^i must be a number in $[0, r]$. Likewise, we add conjuncts expressing that each reg_j^i (resp. arr_j^i) ranges over $[1, 2m]$ (resp. $\{0, 1\}$). We also need to state that changes in the direction arrows incur an extra reversal (otherwise, no reversal is incurred):

$$\bigwedge_{j=1}^{k} \bigwedge_{i=0}^{N_{\max}-1} 0 \leq rev_j^i \leq r. \wedge \bigwedge_{j=1}^{k} \bigwedge_{i=0}^{N_{\max}-2} \left(\begin{array}{c} arr_j^i \neq arr_j^{i+1} \rightarrow rev_j^{i+1} = rev_j^i + 1 \bigwedge \\ arr_j^i = arr_j^{i+1} \rightarrow rev_j^{i+1} = rev_j^i \end{array} \right).$$

Finally, the sequence $\{reg_j^i\}_{i=0}^{N_{\max}-1}$ must obey the changes in $\{arr_j^i\}_{i=0}^{N_{\max}-1}$:

$$\bigwedge_{j=1}^{k} \bigwedge_{i=0}^{N_{\max}-2} \left(reg_j^i < reg_j^{i+1} \rightarrow arr_j^{i+1} = 0 \bigwedge reg_j^i > reg_j^{i+1} \rightarrow arr_j^{i+1} = 1 \right).$$

I.e., since regions denote increasing intervals, going to higher (resp. lower) regions mean the counter mode must be non-decrementing (resp. non-incrementing).

Lastly, we give $\texttt{Respect}(\mathbf{z}, \mathbf{m}_0, \ldots, \mathbf{m}_{N_{\max}-1})$. Again, this is a conjunction. First, when the jth counter is non-incrementing (resp. non-decrementing), we allow only non-negative (resp. non-positive) counter increments:

$$\bigwedge_{j=1}^{k} \bigwedge_{i=1}^{N_{\max}-1} \begin{pmatrix} arr_j^i = 0 \rightarrow \bigwedge_{(\texttt{ctr}_k, d, i, l) \in \mathsf{ACT}', d<0} z_{f(\texttt{ctr}_k, d, i, l)} = 0 \bigwedge \\ arr_j^i = 1 \rightarrow \bigwedge_{(\texttt{ctr}_k, d, i) \in \mathsf{ACT}', d>0} z_{f(\texttt{ctr}_k, d, i, l)} = 0 \end{pmatrix}.$$

Secondly, the value of the jth counter at the beginning and end of each mode must respect the guessed mode vector. Let us first introduce some notations. Observe that the value of the jth counter at the *end* of ith mode vector can be expressed by $c_j + \left(\sum_{i'=0}^{i-1} \sum_{(\texttt{ctr}_j, d, i', l) \in \mathsf{ACT}'} d \times z_{f(\texttt{ctr}_j, d, i', l)} \right) + \sum_{(\texttt{ctr}_j, d, i, 0)} d \times z_{f(\texttt{ctr}_j, d, i, 0)}$. Let us denote this term by $EndCounter_j^i$. Similarly, the value at the *beginning* of the ith mode is $c_j + \sum_{i'=0}^{i-1} \sum_{(\texttt{ctr}_j, d, i', l) \in \mathsf{ACT}'} d \times z_{f(\texttt{ctr}_j, d, i', l)}$. We denote this term by $StartCounter_j^i$. Hence, this second conjunct is

$$\bigwedge_{j=1}^{k} \bigwedge_{i=0}^{N_{\max}-1} \bigwedge_{l=1}^{m} \begin{pmatrix} reg_j^i = 2l - 1 \rightarrow (\varphi_l(EndCounter_j^i) \wedge \varphi_l(StartCounter_j^i)) \bigwedge \\ reg_j^i = 2l \rightarrow (\psi_l(EndCounter_j^i) \wedge \psi_l(StartCounter_j^i)) \end{pmatrix}.$$

Finally, we need to express that no invalid counter tests are executed in a given mode. To test whether a CC constraint θ is satisfied by the values $\mathbf{b} = (b_1, \ldots, b_k)$ of the counters, it is necessary and sufficient to test whether θ is satisfied by *some* vector $\mathbf{b}' = (b_1', \ldots, b_k')$, where each b_i lies in the same region in REG as b_i'. Therefore, the desired property can be expressed as:

$$\bigwedge_{i=0}^{N_{\max}-1} \bigwedge_{(\theta, i) \in \mathsf{ACT}'} z_{f(\theta, i)} > 0 \rightarrow \theta(StartCounter_1^i, \ldots, StartCounter_k^i).$$

Of course, we might want to make the formula smaller by associating new variables for all terms $StartCounter_j^i$ and $EndCounter_j^i$. Since the translation from PDA to CFGs produces an output of cubic size, it is easy to check that the size of $\texttt{HasRun}$ is cubic in $\|\mathcal{P}\|$, r, and k.

Applications. We start with a straightforward application of the above proof: Control-state reachability over PCo with additional existential Presburger constraints on counter values at the beginning/end of the run and on how many times actions are executed at the end of the run can be checked in NP. In fact, it is poly-time reducible to checking satisfactions over existential Presburger formulas. To see this, observe that the counter values in α and α', which were treated as constants in HasRun, could be treated as *variables*. Hence, we can add the additional constraint within the inner bracket of HasRun as a conjunct, and quantify the new variables for counter values. Secondly, we have the following:

Theorem 2. *LTL model checking over PCo is* coNEXP*-complete. For fixed formulas, it is* coNP*-complete.*

For this, we begin with the standard Vardi-Wolper automata-theoretic approach [32] reducing the *complement* of this problem to *recurrent reachability* over PCo: given a PCo $\mathcal{P} = (Q, \mathsf{ACT}, \Gamma, \delta, X)$ and a subset $F \subseteq Q$, decide if there is a run of $\mathcal{P}$ visiting configurations of the form $(q, u, \mathbf{c})$ where $q \in F$ infinitely often. This reduction obtains an NBWA of exponential size from the negation of the given LTL formula, and builds the product of this NBWA and $\mathcal{P}$. Thus, the reduction is exponential in the LTL formula but polynomial in the PCo. Therefore, it suffices to show that recurrent reachability is in NP. Observe that all infinite runs π of $\mathcal{P}$ stabilise in some mode, expressed by a mode vector $\mathbf{m}$. We split π into a subpath from the initial configuration α to a configuration $\alpha' = (q', u', \mathbf{c}')$ in π in mode $\mathbf{m}$, and the rest of the path from α'. In fact, if $\mathbf{c}' = (c'_1, \ldots, c'_k)$, then α' could be chosen such that if the value of the jth counter after α' in π has a maximum c (e.g. when jth counter value stabilises in a region $< 2m$, which is bounded), then $c'_j = c$. By allowing only rules which do not decrement counters whose values do not stabilise, we may treat the path from α' as a path of a PDS with no counters. Hence, we use a well-known lemma of PDS (w.l.o.g. assume that transitions of PDS/PCo only check the top stack character):

Lemma 3 ([4]). *Given a PDS $\mathcal{P} = (Q, \mathsf{ACT}, \Gamma, \delta)$, an initial configuration α, and a set $F \subseteq Q$, then there exists an infinite computation path of $\mathcal{P}$ from α visiting F infinitely often iff there exist configurations $(p, \alpha) \in Q \times \Gamma$, $(g, u), (p, \alpha v)$ such that $g \in F$ and (i) α can reach $(p, \alpha w)$ for some $w \in \Gamma^*$, and (ii) (p, α) can reach (g, u), from which $(p, \gamma v)$ is reachable.*

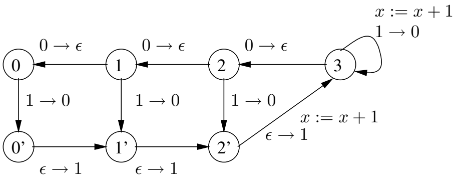
We construct a PCo $\mathcal{P}'$ that simulates $\mathcal{P}$ (for the initial subpath of the recurrent reachability witness). Eventually, it decides to stop simulating $\mathcal{P}$ at some configuration $(p, \alpha w, \mathbf{v})$. The state and the top-stack character are then made visible by $\mathcal{P}'$ with an action $\overline{(p, \alpha)}$. The PCo $\mathcal{P}'$ then empties the stack and continues the simulation of $\mathcal{P}$ from (p, α) except that (1) only rules which increment counters by non-negative values are allowed, but instead of actually modifying the counter values, actions of the form $+\mathtt{ctr}_j$ will be executed if the rule increments the jth counter by a positive value (2) counter tests θ are no longer performed, but instead we execute actions that signify θ. When a configuration of the form $(g, u, \mathbf{v})$ is reached with $g \in F$, it will make it visible by performing an action of the form $g!$. At any given time after this, it may output a character $\overline{(p', \alpha')}$, where p' is the current state and α' is the current top stack character, and go into the state Finish. There exists a computation path of $\mathcal{P}$ from α that visits F infinitely often iff there exists a path from α to the Finish in $\mathcal{P}'$ such that: (a) the number executions of some $\overline{(p, \alpha)}$ is 2, (b) some $g!$ action has been executed, (c) if the region of the end value of the jth counter (corresponding to the jth counter value of the initial witnessing subpath of $\mathcal{P}$) is bounded (i.e.

$< 2m$), then no action $+\mathtt{ctr}_j$ must have been executed, and (d) no counter test actions violating the regions of the end counter values have been executed. Using the techniques from Theorem 1, we may express these constraints as a poly-size existential Presburger formula. In conclusion, we have reduced recurrent reachability over PCo to control-state reachability over PCo with constraints, which we already saw to be in NP.

5 Lower Bounds for Model Checking PCo

In this section, we prove coNEXP and coNP lower bounds for model checking LTL and reachability over PCo. We will also show that model checking EF-logic is undecidable even for a fixed formula. All our lower bounds hold for $(1, 1)$-PCo where counters can only be compared against 0 and incremented by $\{-1, 0, 1\}$.

Lower bound for LTL and reachability. We start with coNP-hardness for a fixed LTL formula over $(1, 1)$-PCo. We show that the complement of this model checking problem is NP-hard. It will be clear later that the same proof can be used to show that reachability is NP-hard over $(1, 1)$-PCo. The idea is to reduce from the NP-complete `0-1 Knapsack` problem ([23]): given $a_1, \ldots, a_k, b \in \mathbb{N}$ in binary, decide whether $\sum_{i=1}^{k} a_i x_i = b$ for some $x_1, \ldots, x_k \in \{0, 1\}$. The reduction constructs a $(1, 1)$-PCo $\mathcal{P}$ which initializes the counter to 0 and repeats for each $i \in [1, k]$: guess x_i, add $a_i x_i$ to the counter. The PCo then checks whether the counter is b by substracting b and checking whether the result is 0. If the test is positive, execute a special action *success* and then loop silently. The LTL formula is $\mathsf{G}\neg success$. Note, this PCo makes one reversal. The problem with this reduction, however, is that the numbers $a_1, \ldots, a_k, b$ are given in binary, whereas each transition in $\mathcal{P}$ can add -1 or 1. Hence, we cannot naively hardwire the "intermediate" values of $a_1, \ldots, a_k, b$ during the computation in the finite control. Instead, we need to use the stack. We illustrate this technique by the example in the following figure on the right. This is a PCo with one counter x that increases x by the number represented by the topmost four bits on the stack (edge label $u \to v$ defines the stack operation). For example, if the PCo starts with the configuration $(3, 1101, 0)$, then it will end at the configuration $(0, 0, 13)$. Using this technique, we will only need at most $\sum_{i=1}^{k} \log(a_i) + \log(b)$ extra states and therefore avoiding an exponential blow-up.



For the coNEXP lower bound for non-fixed formulas, we reduce succinct `0-1 Knapsack`. We define `Succinct 0-1 Knapsack` as the `0-1 Knapsack` problem where the input is given as a boolean formula θ with variables $x_1, \ldots, x_{k+m}$ where $k, m \in \mathbb{Z}_{>0}$ are given in unary. Here θ represent the numbers $a_1, \ldots, a_{2^k-1}, b \in \mathbb{N}$ each with precisely 2^m bits (leading 0s permitted) as follows:

- The ith bit of $\mathbf{bin}(b)$ is defined to be $x \in \{0, 1\}$ iff the formula θ evaluates to x when $x_1, \ldots, x_{k+m}$ are evaluated to $0^k\mathbf{bin}^m(i)$.

– The ith bit of $\mathbf{bin}(a_j)$ is defined to be $x \in \{0,1\}$ iff the formula θ evaluates to x when the inputs to $x_1, \ldots, x_{k+m}$ are $\mathbf{bin}^k(j)\mathbf{bin}^m(i)$.

The problem is to check whether $\sum_{i=1}^{2^k-1} a_i z_i = b$ for some $z_1, \ldots, z_{2^k-1} \in \{0,1\}$. The problem of `Succinct 0-1 Knapsack` can be shown to be NEXP-complete in the same way that the problem `Succinct Knapsack`, where natural numbers can be assigned to z_i's (instead of only $\{0,1\}$), is shown in [33] to be NEXP-complete.

We now show a NEXP lower bound for the complement of LTL model checking over $(1,1)$-PCo by reducing from `Succinct 0-1 Knapsack`. The idea of the reduction is the same as for the case of fixed formulas, but we will have to use the LTL formula to count up to doubly exponential values.

The stack alphabet includes $\#, c_1^0, c_1^1, \ldots, c_m^0, c_m^1$. If the ith bit of a_j is 1 and z_j has been guessed to be 1, then we need to add 2^i to the counter. E.g., suppose we're on the $11\ldots1$th bit of the a_0. Using the techniques below, we calculate the value of this bit. If it is 1, we increment $2^{(2^m-1)}$ times. To do this, we push the following sequence on the stack, again using techniques described below.

$$0c_m^0 \ldots c_1^0 \# 0 c_m^1 c_{m-1}^0 \ldots c_1^0 \# \ldots \;\ldots\; \ldots \# 0 c_m^1 \ldots c_1^1 \# \qquad (*)$$

That is, a bit-string with 2^m many 0s, annotated with their bit positions (least significant bit on the top/left of the stack). From this stack configuration, the system counts up to $1c_m^0 \ldots c_1^0 \# 1 c_m^1 c_{m-1}^0 \ldots c_1^0 \# \ldots \;\ldots\; \ldots \# 1 c_m^1 \ldots c_1^1 \#$, i.e., a bit-string with 2^m many 1s, taking $2^{(2^m-1)}$ steps. We increment the counter $+1$ at each step.

To complete the proof, we need to know how to: (1) enumerate all assignments to $x_1, \ldots, x_{k+m}$ and evaluate θ, (2) initialise the large counter described above, and (3) increment the large counter from $00\ldots0$ to $11\ldots1$.
Problem 1: We store the current assignment on the bottom of the stack, using $x_1^0, x_1^1, \ldots, x_{k+m}^0, x_{k+m}^1$. Initially, we push $x_1^0, \ldots, x_{k+m}^0$, making the pushed characters visible using suitable action symbols. We then guess whether θ evaluates to 1. An LTL formula encoding θ asserts this guess is correct. To move to the lexicographically next assignment, we erase the stack, making the characters visible, and then guess the next assignment, using the LTL to check it.
Problem 2: Guess and push $0c_1^{y_1} \ldots c_m^{y_m}\#$, using the formula to check this matches the $x_{k+1}^{y_1}, \ldots, x_{k+m}^{y_m}$ on the stack. We then push $0c_1^{Y_1} \ldots c_m^{Y_m}\#$ (with the values of $Y_1, \ldots, Y_m \in \{0,1\}$ guessed) to the stack. This is done arbitrarily many times and the PCo may stop at some point. To make sure we obtain (*) on the top of the stack, we assert the successor property using the LTL formula.
Problem 3: This uses the idea for the fixed formula case: pop from the stack until the first 0, then push a correct number of 1s annotated with the bit positions. For this, an LTL formula asserts the successor property, cf. [28].

Undecidability results for EF-logic. We now turn our attention to model checking EF-logic over PCo. It turns out that the problem is already undecidable for a fixed formula with two operators. We reduce from the emptiness of linear bounded Turing machines (LTM), which is undecidable (cf. [27]). Given an LTM $\mathcal{M}$ that accept/rejects an input w using at most $c|w|$ space, we compute a

$(1,1)$-PCo $\mathcal{P}$, an EF formula φ, and a start state q_0 of $\mathcal{P}$ such that $\mathfrak{S}_{\mathcal{P}}, (q_0, \epsilon) \models \varphi$ iff $\mathcal{M}$ is nonempty. We make $\mathcal{P}$ guess a word w and an accepting computation path of $\mathcal{M}$, which is stored in the stack. The length $c|w|$ is stored in the counter. Once the path has been guessed, it suffices to show that: (P1) the length of *each* guessed configuration is $c|w|$, and (P2) *each* non-initial configuration is a successor of its previous configuration. The guessing and checking stages incur one alternation for the EF formula. Since checking P2 requires us to check at most four consecutive tape cells of $\mathcal{M}$ (once a cell is chosen), we can remember this location by decrementing the counter, moving to the previous configuration of $\mathcal{M}$ that is stored in the stack, and then further decrementing the counter making sure that the end value is 0. See the full version for the proof.

Theorem 4. *Model checking EF-logic over* $(1,1)$*-PCo is undecidable even for a fixed formula with two* EF *operators.*

6 Adding clocks

To extend our results to the case of PCC, we use the region construction of Alur and Dill [1] to reduce a PCC to a PCo of exponential size (in exponential time) such that every run of the PCC can be projected, state-for-state, onto a run of the PCo. From S4, we obtain a coNEXP (resp. NEXP) upper bound for LTL model checking (resp. reachability) over PCC. We next provide a lower bound.

Theorem 5. *Reachability is* NEXP*-hard for PDS with discrete clocks and one 1-reversal-bounded counter. When clock constraint constants are given in binary, only three clocks and one single-reversal counter are needed. Similarly, LTL model-checking is* coNEXP*-hard, even for a fixed formula.*

We first consider unary constants, and a non-fixed number of clocks. We adapt the previous reduction from `Succinct 0-1 Knapsack`, except the formula can no longer be used to evaluate the boolean formulas. Instead, we encode bits with two clocks, which are equal iff the bit is 1. We evaluate boolean formulas using the transition guards. To test all assignments to $x_1, \ldots, x_{k+m}$ we store the valuation in the counters (not the stack). This is straightforward. Then, to build the large counter on the stack, we use another set of clocks to store the bit position values of the last two blocks pushed on to the stack. These clocks can be used to ensure the successor relation between the two values. For binary clock constraints, we use Courcoubetis and Yannakakis [8] to reduce to three clocks.

Recall that model checking EF-logic over PCo is undecidable. It turns out that this problem is decidable over PCl. See the full version for a proof.

Theorem 6. *Model checking EF over PCl is* EXPSPACE*-complete. The lower bound holds for fixed formulas with two clocks with binary constraint constants, or with a non-fixed number of clocks when the constraints are in unary.*

7 Experimental Results

We provide a prototypical C++ implementation of an optimised version of the reduction in Section 4. In particular, from the Verma *et al.*, we can derive, at no cost, the number of times each rule of the PCo is fired. From this, we infer the number of action symbols output, and hence, do not need additional variables and transitions for these. In addition, the per mode information per counter uses a single variable. Finally, we look for pairs of pushdown rules $\langle (q_1, a_1, \top), \epsilon, (q_2, a_2, \emptyset, \emptyset^n) \rangle$ and $\langle (q_2, a_2, \top), \epsilon, (q_3, a_3, \emptyset, \emptyset^n) \rangle$ such that a_1 and a_2 are single characters and the pair q_2, a_2 does not appear together in any other rule. Furthermore, a_2 does not appear in a rule $\langle (q, w, \theta), a, (q', w' a_2 w'', Y, \mathbf{w}) \rangle$ where $|w'| > 0$. These rules can be replaced by $\langle (q_1, a_1, \top), \epsilon, (q_3, a_3, \emptyset, \emptyset^n) \rangle$. Then we remove unreachable productions from the generated CFGs. We used the 64-bit Linux binary of Z3 2.16 [10] on the Presburger formulas on a quad-core, 2.4Gz Intel® Xeon® machine with 12GB of RAM, running Fedora™12. We report the time from the Linux command `time` for the translation and the run of Z3.

Double free in dm-target.c. In version 2.5.71 of the Linux kernel, a double free was introduced to `drivers/md/dm-target.c` [30]. This was introduced to fix a perceived memory-leak. When registering a new target, memory was allocated and a check made to see if the target was already known. If so, it was freed and an "exists" flag set. Otherwise the target was added to the target list. Before returning, the exists flag was checked and the object was freed (again) if it was set. We created, by hand, a model of this file using a counter to store the length of the target list. The complete control flow of the file was maintained and data only tracked when relevant to the memory management. The model outputs special symbols to mark when memory is allocated or freed. We then look for a run where, either an item was removed from the empty list, the number of free calls was greater than the number of allocations, or, the code exited normally, but more memory was allocated than freed. We were able to verify that the code contained a memory error in version 2.5.71 and that the memory management was correct in earlier versions (for a bounded number of reversals) provided as many targets were registered as unregistered. Note, the counter is required to track the size of the list which ensures that the number of allocations matches the number of frees. The size of the `dm-target.c` is approximately 175 lines without comments. Detecting the bug took 2s, and proving correctness took 1.7s, 2.9s, 16s, 24s and 77s for 1, 2, 3, 4 and 5 reversals respectively.

Memory leak in aer_inject.c. In version 2.6.32 of the Linux kernel, the file `drivers/pci/pcie/aer/aer_inject.c` contains a memory leak that was patched in the next version [25]: two lists of allocated objects are maintained, but, when exiting, the code empties the items from the first list and frees them, then, empties and frees the first list again, instead of the second. We created a model of this driver with two counters to track the size of the lists and searched for memory errors as in the previous example. Only one reversal was required to detect the memory leak. We showed that the patch corrects the problem (up to one reversal). Note, without counters, it would always be possible for the number of allocations to differ from the number of frees. The file `aer_inject.c` is approximately 470 lines without comments. Detecting the bug required 220s and proving correctness for a single reversal took 508s.

Buffer overflow. Jhala and McMillan have a buffer overflow example which their technique, SatAbs and Magic, failed to verify [21]. There are three buffers, x, y and z of sizes $100, 100$ and 200 and two counters i and j. First, i is used to copy up to 100 positions of x into z. Then, counters i and j are used to copy up to 100 positions of y into the remainder of z. There is overflow if i, which indexes z, finishes greater than 199. This simply encodes into our model[1] and could be verified correct (since the example is trivially reversal bounded) in 1.6s.

David Gries's coffee can problem [15]. We have an arbitrary number of black and white coffee beans in a can. We pick two beans at random. If they are the same colour, they are discarded and an extra black bean is put in the tin. If they differ, the white is kept but the black discarded. The last bean in the can is black iff the number of white beans is odd. This problem can be modelled without abstraction by using the stack to count the number of black beans, and a single-reversal counter to track the number of white beans. We verified in 1.3s that the last bean cannot be white if the number of white beans is odd, and in 1.1s that the last bean may be white if the number of white beans is even.

8 Extensions and Future Work

We prove, in the full version of this paper, that (i) reachability and LTL for a prefix-recognisable version of PCo is NEXP-complete and coNEXP-complete respectively, even with one 1-reversal-bounded counter and a fixed formula; (ii) LTL model checking for a fixed formula is coNEXP when the number of reversals is given in binary, whereas reachability is in NEXP (a matching lower bound is in [18], even without the stack); and (iii) the reachability problem for second-order pushdown systems (cf. [17]) with reversal bounded counters is undecidable.

For future work we may investigate counter-example guided abstraction refinement. We would need counter-examples from the models of the existential Presburger formula, and refinement techniques to add counters and reversals as well as predicates. Furthermore, as we allow user defined numerical constraints on reachability, we may also restrict LTL model checking to runs satisfying additional numerical fairness constraints.

Acknowledgments. We thank the authors of [7] and [33] for their correspondence, V. D'Silva for recommending [21], J. Leroux and P. Ruemmer for suggesting Z3, A. Sangnier for [13], and anonymous referees for their comments. We thank EPSRC (EP/F036361 and EP/H026878/1) for their support.

[1] Comparison with constants is not implemented. We adjusted the formula by hand.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
3. Ball, T., Rajamani, S.K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: SPIN 2000, pp. 113–130 (2000)
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. Bouyer, P.: Model-checking timed temporal logics. Electr. Notes Theor. Comput. Sci. 231, 323–341 (2009)
6. Cachat, T.: Uniform Solution of Parity Games on Prefix-Recognizable Graphs. Electr. Notes Theor. Comput. Sci. 68(6) (2002)
7. Chadha, R., Legay, A., Prabhakar, P., Viswanathan, M.: Complexity Bounds for the Verification of Real-Time Software. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 95–111. Springer, Heidelberg (2010)
8. Courcoubetis, C., Yannakakis, M.: Minimum and Maximum Delay Problems in Real-Time Systems. FMSD 1(4), 385–415 (1992)
9. Dang, Z.: Binary reachability analysis of pushdown timed automata with dense clocks. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 506. Springer, Heidelberg (2001)
10. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
12. Esparza, J., Kucera, A., Schwoon Model, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. 186(2), 355–376 (2003)
13. Filiot, E., Raskin, J.-F., Reynier, P.-A., Servais, F., Talbot, J.-M.: Properties of visibly pushdown transducers. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 355–367. Springer, Heidelberg (2010)
14. Finkel, A., Sangnier, A.: Reversal-bounded counter machines revisited. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 323–334. Springer, Heidelberg (2008)
15. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
16. Gurari, E.M., Ibarra, O.H.: The Complexity of Decision Problems for Finite-Turn Multicounter Machines. J. Comput. Syst. Sci. 22(2), 220–229 (1981)
17. Hague, M.: Saturation Methods for Global Model-Checking Pushdown Systems. PhD thesis, Oxford University Computing Laboratory (2009)
18. Howell, R., Rosier, L.: An Analysis of the Nonemptiness Problem for Classes of Reversal-Bounded Multicounter Machines. J. Comput. Syst. Sci. 34(1), 55–74
19. Ibarra, O.H.: Reversal-Bounded Multicounter Machines and Their Decision Problems. J. ACM 25(1), 116–133 (1978)
20. Ibarra, O., Su, J., Dang, Z., Bultan, T., Kemmerer, R.: Counter machines and verification problems. Theor. Comput. Sci. 289, 165–189 (2002)

21. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
22. Laroussinie, F., Meyer, A., Petonnet, E.: Counting CTL. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 206–220. Springer, Heidelberg (2010)
23. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Mineola (1998)
24. Parikh, R.: On context-free languages. J. ACM 13(4), 570–581 (1966)
25. Patterson, A.: PCI: fix memory leak in aer_inject (2003), `https://patchwork.kernel.org/patch/53058/`
26. Scarpellini, B.: Complexity of Subcases of Presburger Arithmetic. Trans. of AMS 284(1), 203–218 (1984)
27. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing Co. (1997)
28. Sistla, A.P., Clarke, E.M.: The Complexity of Propositional Linear Temporal Logics. J. ACM 32(3), 733–749 (1985)
29. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java Bytecode Checker Based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
30. Thornbur, J.: dm: Fix memory leak in dm_register_target() (2003), `http://lkml.org/lkml/2003/6/9/70`
31. To, A.W.: Model Checking Infinite-State Systems: Generic and Specific Approaches. PhD thesis, School of Informatics, University of Edinburgh (2010)
32. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344 (1986)
33. Veith, H.: Languages Represented by Boolean Formulas. Inf. Process. Lett. 63(5), 251–256 (1997)
34. Verma, K.N., Seidl, H., Schwentick, T.: On the Complexity of Equational Horn Clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)
35. Walukiewicz, I.: Model Checking CTL Properties of Pushdown Systems. In: FSTTCS 2000, pp. 127–138 (2000)

Author Index