Proceedings of the 14th Conference on

Formal Methods in Computer-Aided Design (FMCAD 2014)

Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland, October 21 – 24, 2014

Edited by Koen Claessen and Viktor Kuncak

In cooperation with
ACM Special Interest Group on Programming Languages
ACM Special Interest Group on Software Engineering

SIGPLAN

Technical co-sponsorship of IEEE Council on
Electronic Design Automation

Proceedings of the 14th Conference on

Formal Methods in Computer-Aided Design

FMCAD 2014

October 21 – 24, 2014

Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland

Edited by Koen Claessen and Viktor Kuncak

ISBN: 978-0-9835678-4-4
Copyright owned jointly by the authors and FMCAD Inc.

Title page: Aerial view of Rolex Learning Center at EPFL
Photograph: Alain Herzog / EPFL
Photo provided with limited rights.

Preface

The International Conference on Formal Methods in Computer-Aided Design, FMCAD, is a series of conferences on the theory and application of formal methods to the computer-aided design and verification of hardware and systems. The fourteenth conference in the series, FMCAD 2014, takes place October 20-23, 2014 in Lausanne, Switzerland.

FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers the spectrum of formal aspects of computer-aided system design, including verification, specification, synthesis, and testing. This year, the conference once again received an in-cooperation status with ACM under the Special Interest Group on Programming Languages and the Special Interest Group on Software Engineering. It also received technical sponsorship from the IEEE Council on Electronic Design Automation. Two additional events are co-located with the conference this year: (1) MEMOCODE 2014, the 12th ACM-IEEE International Conference on Formal Methods and Models of Codesign and (2) DIFTS 2014, Workshop on Design and Implementation of Formal Tools and Systems.

FMCAD 2014 received 101 abstracts that materialized into 70 full submissions. Each full submission was reviewed by at least four, and on the average 4.1, program committee members. After a thorough peer-review process that often involved vigorous electronic discussions by program committee members, subreviewers, and additional external reviewers, 28 submissions were selected for presentation at the conference: 25 as regular papers and 3 as short papers. Accepted papers covered topics ranging from model checking, synthesis at various abstraction levels, solver and prover techniques (SAT and SMT, interpolation, theorem proving frameworks). The domains included hardware systems at various stages of the design, protocols, software, networks, interfaces between software and hardware, as well as biological systems.

In addition to reviewed submissions, the program includes two keynote presentations and four tutorials. Keynotes this year include Thomas A. Henzinger on *Computer-Aided Verification Technology for Biology* and Xavier Leroy on *Compiler verification for fun and profit*. The tutorials are hosted jointly by FMCAD and MEMOCODE. This year we have Armin Biere presenting *Challenges in Bit-Precise Reasoning*, Ziyad Hanna discussing *Challenging Problems in Industrial Formal Verification*, Morgan Deters, Andrew Reynolds and Timothy King, Clark Barrett, and Cesare Tinelli giving *A Tour of CVC4: How it works, and how to use it*, and Johannes Kinder on *Efficient symbolic execution for software testing*.

As in previous years, the 2014 FMCAD Proceedings are expected to be available through the ACM Digital Library, the IEEE Xplore Digital Library, and are also available as a free download from the FMCAD Website.

This year's edition includes again a Student Forum that provides a platform for students to present their research to the FMCAD community and obtain feedback. The forum includes short presentations, and a poster by a student author of each accepted submission. The student forum presentations were selected through a review process led by Ruzica Piskac.

We sincerely thank our industrial sponsors for their financial support of FMCAD 2014: ARM, Atrenta, Cadence Design Systems, Centaur, IBM, Intel, Jasper Design Automation, Mentor Graphics, Onespin, Oski Technology, Real Intent, and Synopsys. We thank FMCAD Inc. for continuous support of the conference series. We also thank EPFL's School for Computer and

Communication Sciences for substantially supporting this year's edition.

We owe a large debt of gratitude to this year's organizing committee, which, in addition to PC chairs and local chairs include (alphabetically): Jason Baumgartner (Sponsorship and Steering committee contact), Shilpi Goel (Webmaster), Warren A. Hunt Jr. (FMCAD Inc.), Barbara Jobstmann (Publication and Registration Chair), Ruzica Piskac (Student Forum Chair), Mitra Purandare (Publicity Chair), Vigyan Singhal (Sponsorship Co-Chair). Viktor Kuncak would also like to thank Yvette Gallay from EPFL for her immense local organization effort, without which it would not have been possible to have the conference at EPFL. We thank the best paper award committee chaired by Alan J. Hu and consisting additionally of Bruno Dutertre, Cindy Eisner, and Aarti Gupta. We thank all members of the FMCAD Steering Committee: Jason Baumgartner, Armin Biere, Alan J. Hu, and Warren A. Hunt, for their kind advice during the conference preparation process. Big thanks to all members of the Program Committee and all reviewers, who did a stellar job not only of selecting this year's exciting program, but also of providing feedback to the authors to help them improve their papers for publication. The conference would not be possible without all the authors that submitted high-quality papers. We thank especially keynote and tutorial speakers that accepted to present their exciting research. Last, but not the least, we thank all attendees, whose presence justifies the effort of organizing an exciting physical meeting on the EPFL campus.

FMCAD 2014 Program Co-Chairs
Koen Claessen and Viktor Kuncak
14 September 2014

Organization Committee

Program Co-Chairs

Koen Claessen	Chalmers University of Technology
Viktor Kuncak	EPFL

Local Arrangement Chair

Viktor Kuncak	EPFL

Publication and Registration Chair

Barbara Jobstmann	EPFL and CNRS/Verimag

Publicity Chair

Mitra Purandare	IBM Research Lab, Zurich

Student Forum Chair

Ruzica Piskac	Yale University

Webmaster

Shilpi Goel	University of Texas at Austin

Steering Committee

Jason Baumgartner	IBM, USA
Armin Biere	Johannes Kepler University in Linz, Austria
Alan J. Hu	University of British Columbia, Canada
Warren A. Hunt, Jr.	University of Texas at Austin, USA

Program Committee

Jason Baumgartner	IBM
Dirk Beyer	University of Passau
Armin Biere	Johannes Kepler University
Per Bjesse	Synopsys
Nikolaj Bjorner	Microsoft Research
Roberto Bruttomesso	Atrenta
Gianpiero Cabodi	Politecnico di Torino
Hana Chockler	King's College
Alessandro Cimatti	FBK-irst
Koen Claessen	Chalmers University of Technology
Bruno Dutertre	SRI international
Ziyad Hanna	Cadence Design Systems
Keijo Heljanko	Aalto University
Alan J. Hu	University of British Columbia
Warren A. Hunt	University of Texas
Susmit Jha	Strategic CAD Lab, Intel
Daniel Kroening	University of Oxford
Viktor Kuncak	EPFL
Panagiotis Manolios	Northeastern University
Ken McMillan	Microsoft Research
Katell Morin-Allory	TIMA Laboratory, Grenoble
Lee Pike	Galois, Inc.
Ruzica Piskac	Yale University
Mitra Purandare	IBM Research Zurich
Sandip Ray	Intel Corporation
Andrey Rybalchenko	Microsoft Research Cambridge
Philipp Rümmer	Uppsala University
Julien Schmaltz	Open University of the Netherlands
Natasha Sharygina	Universita' della Svizzera Italiana
Anna Slobodova	Centaur Technology
Daryl Stewart	ARM
Niklas Sörensson	Mentor Graphics Corporation
Cesare Tinelli	The University of Iowa
Martin Vechev	ETH Zurich
Thomas Wahl	Northeastern University
Georg Weissenbacher	Vienna University of Technology

Additional Reviewers

Abd Elkader, Karam
Albarghouthi, Aws
Alberti, Francesco
Aleksandrowicz, Gadi
Alglave, Jade
Alt, Leonardo
Andrikos, Nikolaos
Appold, Christian
Atig, Mohamed Faouzi

Backeman, Peter
Backes, John
Barner, Sharon
Bayless, Sam
Bhunia, Swarup
Boden, Eric
Borrione, Dominique
Brockschmidt, Marc
Bustan, Doron

Chau, Cuong
Cotton, Scott

Dagit, Jason
Darulova, Eva
David, Cristina
Deters, Morgan
Dimitrova, Rayna

Ebergen, Jo
Eisner, Cindy

Fedyukovich, Grigory
Finkbeiner, Bernd
Franzén, Anders
Frehse, Goran

Gacek, Andrew
Garg, Deepak
Goel, Shilpi
Goldberg, Eugene
Gopalakrishnan, Ganesh
Griggio, Alberto

Hadarean, Liana
Heizmann, Matthias
Hendrix, Joe
Heule, Marijn
Hicks, Matthew
Hjort, Håkan
Hyvärinen, Antti

Iabrudi, Andréa
Ivrii, Alexander

Jagadeesan, Radha
Jain, Himanshu
Jain, Mitesh
Jha, Sumit Kumar
Jobstmann, Barbara
Joosten, Sebastiaan

Kahsai, Temesghen
Kini, Keshav
Kiniry, Joseph
Kneuss, Etienne
Korthikanti, Vijay
Kosikinen, Eric
Kravets, Victor
Kretinsky, Jan

Lampka, Kai
Leslie-Hurd, Joe
Lewis, Matt
Li, Juncao
Liu, Lingyi
Liu, Peizun
Loiacono, Carmelo
Lopes, Nuno P.
Lundgren, Lars
Löwe, Stefan

Majumdar, Rupak
Margalit, Oded
Martins, Ruben
Mattarei, Cristian
Micheli, Andrea
Mitchell, Ian
Mover, Sergio
Mukherjee, Rajdeep

Nikolic, Mladen
Niksic, Filip
Nocco, Sergio
Nori, Aditya

Oliveras, Albert
Orni, Avigail

Palena, Marco
Papavasileiou, Vasilis
Parlato, Gennaro
Pasini, Paolo
Peter, Hans-Jörg
Pidan, Dmitry
Pinto, Alessandro
Poetzl, Daniel
Preiner, Mathias

Quer, Stefano

Rabe, Markus N.
Rager, David L.
Ramachandran, Jaideep
Roveri, Marco
Rumley, Sebastien

Saarikivi, Olli
Saha, Indranil
Schrammel, Peter
Schäf, Martin
Seidl, Martina
Selfridge, Ben
Siirtola, Antti Tapani
Singh, Satnam
Stergiou, Christos
Stigge, Martin
Sousa, Marcelo

Tabaei Befrouei, Mitra
Talupur, Murali
Tentrup, Leander
Thiemann, René

van Gastel, Bernard
Vendraminetto, Danilo
Verbeek, Freek
Viaud, Emmanuel
Vizel, Yakir

Wachter, Björn
Wendler, Philipp
Wetzler, Nathan
Wolfovitz, Guy

Xu, Jiazhao

Yorav, Karen
Yu, Andy

Table of Contents

Tutorials

Challenging Problems in Industrial Formal Verification – Ziyad Hanna 1

Challenges in Bit-Precise Reasoning – Armin Biere 3

Efficient symbolic execution for software testing – Johannes Kinder 5

A Tour of CVC4: How it works, and how to use it – Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett and Cesare Tinelli 7

Invited Talks and Student Forum

Compiler verification for fun and profit – Xavier Leroy 9

Computer-Aided Verification Technology for Biology – Thomas A. Henzinger 11

Student Forum – Ruzica Piskac 13

Contributed Articles

Response property checking via distributed state space exploration – Brad Bingham and Mark Greenstreet 15

Towards Pareto-Optimal Parameter Synthesis for Monotonic Cost Functions – Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, Marco Gario and Alberto Griggio 23

SAT-Based Methods for Circuit Synthesis – Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Koenighofer and Florian Lonsing 31

Synthesis of Synchronization using Uninterpreted Functions – Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Außerlechner and Raphael Spörk 35

Interpolation with Guided Refinement: revisiting incrementality in SAT-based Unbounded Model Checking – Gianpiero Cabodi, Marco Palena and Paolo Pasini 43

Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots – Sagar Chaki, Arie Gurfinkel and Nishant Sinha 51

Under-approximate Flowpipes for Non-linear Continuous Systems – Xin Chen, Sriram Sankaranarayanan and Erika Abraham 59

Disproving termination with overapproximation – Byron Cook, Carsten Fuhs, Kaustubh Nimkar and Peter O'Hearn 67

Faster Temporal Reasoning for Infinite-State Programs – Byron Cook, Heidy Khlaaf and Nir Piterman 75

Template-based Circuit Understanding – Adria Gascon, Ashish Tiwari, Bruno Dutertre, Pramod Subramanyan, Sharad Malik and Dejan Jovanovic 83

Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls – Shilpi Goel, Warren Hunt, Matt Kaufmann and Soumava Ghosh 91

DRUPing for Interpolants – Arie Gurfinkel and Yakir Vizel **99**

Efficient Extraction of Skolem Functions from QRAT Proofs – Marijn Heule, Martina Seidl and Armin Biere **107**

Small Inductive Safe Invariants – Alexander Ivrii, Arie Gurfinkel and Anton Belov **115**

On Interpolants and Variable Assignments – Pavel Jancik, Jan Kofron, Simone Fulvio Rollini and Natasha Sharygina **123**

Post-silicon Timing Diagnosis Made Simple using Formal Technology – Daher Kaiss and Jonathan Kalechstain **131**

Leveraging Linear and Mixed Integer Programming for SMT – Timothy King, Clark Barrett and Cesare Tinelli **139**

A Program Transformation for Faster Goal-Directed Search – Akash Lal and Shaz Qadeer **147**

Infinite-State Backward Exploration of Boolean Broadcast Programs – Peizun Liu and Thomas Wahl **155**

Kuai: A Model Checker for Software-defined Networks – Rupak Majumdar, Sai Deep Tetali and Zilong Wang **163**

ILP Modulo Data – Panagiotis Manolios, Vasilis Papavasileiou and Mirek Riedewald **171**

Turbo-Charging Lemmas on Demand with Don't Care Reasoning – Aina Niemetz, Mathias Preiner and Armin Biere **179**

Reduction for Compositional Verification of Multi-Threaded Programs – Corneliu Popeea, Andrey Rybalchenko and Andreas Wilhelm **187**

Finding Conflicting Instances of Quantified Formulas in SMT – Andrew Reynolds, Cesare Tinelli and Leonardo De Moura **195**

Using Interval Constraint Propagation for Pseudo-Boolean Constraint Solving – Karsten Scheibler and Bernd Becker **203**

Patient-Specific Models from Inter-Patient Biological Models and Clinical Records – Enrico Tronci, Toni Mancini, Ivano Salvo, Stefano Sinisi, Federico Mari, Igor Melatti, Annalisa Massini, Francesco Davi, Thomas Dierkes, Rainald Ehrig, Susanna Röblitz, Brigitte Leeners, Tillmann Kruger, Marcel Egli and Fabian Ille **207**

Reducing CTL-live Model Checking to First-Order Logic Validity Checking – Amirhossein Vakili and Nancy A. Day **215**

Predicate Abstraction for Reactive Synthesis – Adam Walker and Leonid Ryzhyk **219**

Author Index **227**

Challenging Problems in Industrial Formal Verification

Ziyad Hanna

Cadence Design Systems

ABSTRACT OF TUTORIAL TALK

The electronic design industry has emerged in the recent years to adopt the system-on-chip (SoC) design methodology, where systems become a smart and complex integration of many configurable and reusable intellectual properties (IP) designs such as CPU, GPU, DSP, etc. SoC design methodologies have become common to a wide range of systems, starting from high-end servers, down to tablets, smartphones, Internet-of-things and wearable devices. The aggressive time-to-market and the hard competition add a major challenge to the electronic design companies to deliver high volume, and high quality products. Integration and validation of such designs has become the major challenge. The EDA industry and the academia has continued the innovation pipeline trying to cope with the complexity of such systems however major challenges are still ahead. Formal verification has emerged in the recent years to become a mainstream technology in SoC/IP design and verification methodologies. In the past, the usage of formal verification was limited to a small range of applications and it was mainly for verifying complex protocols, or some tricky logic functionality by formal experts. However in the recent years,we see a rapid adoption of formal, and we see a widespread of formal verification applications for low power design, security, SoC connectivity, configuration status register, and many more. In this talk, we provide an overview of the challenges that we see in designing SoC systems and configurable IPs, and provide some ideas to stimulate the academic research, aiming at increasing the research and innovation in such areas for keeping bridging the emerging gap that the electronic design industry is facing now and will face in the future.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Challenges in Bit-Precise Reasoning

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract of tutorial talk

Bit-precise reasoning (BPR) precisely captures the semantics of systems down to each individual bit and thus is essential to many verification and synthesis tasks for both hardware and software systems. As an instance of Satisfiabiliy Modulo Theories (SMT), BPR is in essence about word-level decision procedures for the theory of bit-vectors. In practice, quantiers and other theory extensions, such as reasoning about arrays, are important too. In the first part of the tutorial we gave a brief overview on basic techniques for bit-precise reasoning and then covered more recent theoretical results, including complexity classification results. We discussed challenges in developping an efficient SMT solver for bit-vectors, like our award winning SMT solver Boolector, and in particular presented examples, for which current techniques fail. Finally, we reviewed the state-of-the-art in word-level model checking, and argued why it is necessary to put more effort in this direction of research.

Funded by Austrian Science Fund (FWF) NFN Grant S11408-N23 (RiSE).

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Efficient symbolic execution for software testing

Johannes Kinder

Department of Computer Science at Royal Holloway, University of London
Email: johannes.kinder@rhul.ac.uk

ABSTRACT OF TUTORIAL TALK

Symbolic execution has proven to be a practical technique for building automated test case generation and bug finding tools. While the basic technique had been introduced already in the 70s, the advent of modern SAT and SMT solvers has lead to a surge of tools and techniques in the area over the last decade. This tutorial will introduce and compare the different approaches to using symbolic execution for testing and discuss the specific challenges and trade-offs.

A main challenge in symbolic execution is path explosion, and various proposals have been made to combat it. I will discuss how these techniques affect the number and type of solver queries that have to be made, and how this can lead to surprising effects on the efficiency of a symbolic execution engine. Going further, we will look at developments to increase the scope of symbolic execution to larger software systems. Specific topics covered include state merging, procedure summaries, abstraction, search strategies, and parallelization.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

A Tour of CVC4: How it works, and how to use it

Morgan Deters Andrew Reynolds Tim King Clark Barrett Cesare Tinelli

ABSTRACT OF TUTORIAL TALK

CVC4 is a solver for Satisfiability Modulo Theories (SMT). This tutorial aims to give participants an overview of SMT, describe the main features of CVC4, and walk through in-depth examples using CVC4 to demonstrate how to solve real problems with an SMT solver. We will provide a detailed description of various aspects of CVC4's internals, including its architecture, its capacity for dealing with quantifiers, its finite model finder, and the linear arithmetic solver. We will show examples of software and hardware verification problems, and how they are encoded and handled by these features in CVC4.

Participants are expected to have only a basic knowledge of what SMT is. This tutorial will give casual users a taste of encoding complex, real-world problems in SMT and effectively using CVC4 to solve them. Participants will be left with some knowledge of what goes on inside a modern SMT solver and some of the practical issues that arise in using them.

CVC4, jointly developed at New York University and the University of Iowa, is freely available for both research and commercial use under an open-source license. The organizers of this tutorial are all architects and implementors of CVC4 and have extensive expertise in the area of SMT.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Compiler verification for fun and profit

Xavier Leroy
Inria Paris–Rocquencourt,
Domaine de Voluceau, BP 105, 78153 Le Chesnay, France
Email: xavier.leroy@inria.fr

Abstract of invited talk

Formal verification of software or hardware systems — be it by model checking, deductive verification, abstract interpretation, type checking, or any other kind of static analysis — is generally conducted over high-level programming or description languages, quite remote from the actual machine code and circuits that execute in the system. To bridge this particular gap, we all rely on *compilers* and other code generators to automatically produce the executable artifact. Compilers are, however, vulnerable to *miscompilation*: bugs in the compiler that cause incorrect code to be generated from a correct source code, possibly invalidating the guarantees so painfully obtained by source-level formal verification. Recent experimental studies [1] show that many widely-used production-quality compilers suffer from miscompilation.

The formal verification of compilers and related code generators is a radical, mathematically-grounded answer to the miscompilation issue. By applying formal verification (typically, interactive theorem proving) to the compiler itself, it is possible to guarantee that the compiler preserves the semantics of the source programs it transforms, or at least preserves the properties of interest that were formally verified over the source programs. Proving the correctness of compilers is an old idea [2], [3] that took a long time to scale all the way to realistic compilers. In the talk, I give an overview of CompCert C [4], a moderately-optimizing compiler for almost all of the ISO C 99 language that has been formally verified using the Coq proof assistant [5].

The CompCert project is one point in a space of code generators whose verification deserves attention. For example, functional languages and object-oriented languages raise the issue of jointly verifying the compiler and the run-time system (memory management, exception handling, etc) that the generated code depends on. At the other end of the expressiveness spectrum, synchronous languages and hardware description languages also raise interesting verified generation issues, as exemplified by Pnueli's seminal work on translation validation for Signal [6] and Braibant and Chlipala's recent work on verified hardware synthesis [7].

Orthogonally, the integration of verification tools and compilers that are both verified against a shared formal semantics opens fascinating opportunities for "super-optimizations" that generate better code by exploiting the properties of the source code that were formally verified.

References

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI 2011: Programming Language Design and Implementation*. ACM, 2011, pp. 283–294.

[2] J. McCarthy and J. Painter, "Correctness of a compiler for arithmetical expressions," in *Mathematical Aspects of Computer Science*, ser. Proc. of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, 1967, pp. 33–41.

[3] R. Milner and R. Weyrauch, "Proving compiler correctness in a mechanized logic," in *Proc. 7th Annual Machine Intelligence Workshop*, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., vol. 7. Edinburgh University Press, 1972, pp. 51–72.

[4] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[5] Coq development team, "The Coq proof assistant," Software and documentation available at http://coq.inria.fr/, 1989–2014.

[6] A. Pnueli, O. Strichman, and M. Siegel, "Translation validation for synchronous languages," in *ICALP'98: Automata, Languages and Programming*, ser. LNCS, vol. 1443. Springer, 1998, pp. 235–246.

[7] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *CAV 2013: Computer Aided Verification*, ser. LNCS, vol. 8044. Springer, 2013, pp. 213–228.

Acknowledgments

This work was supported by the VERASCO project (ANR-11-INSE-003) of Agence Nationale de la Recherche.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Computer-Aided Verification Technology for Biology

Thomas A. Henzinger
IST Austria

Abstract of invited talk

We summarize some recent results on using computed-aided verification technology for understanding biological systems. This includes the use of reactive models for specifying cellular mechanisms, the use of symbolic state space exploration for analyzing molecular reaction networks, and the use of SMT solvers for studying the evolution of gene regulatory circuits.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

The FMCAD 2014 Graduate Student Forum

Ruzica Piskac, *Student Forum Chair*
Computer Science Department, Yale University
email: ruzica.piskac@yale.edu

The Graduate Student Forum was first introduced in 2013 to the FMCAD conference series. The goal of the Forum is to enable graduate students to attend the conference, even if they do not have a paper accepted at the main conference track. Students were attracted with an opportunity to present their on-going work to a broader scientific audience and receive valuable feedback about the research they are currently pursuing.

Following last year's success, FMCAD 2014 hosted the second edition of the Graduate Student Forum. The 2014 Forum is based on a similar premise and format as the first Forum. There was an open, widely-publicized call for papers. In addition, the Organizing Committee personally informed a number of renowned scientists about the Forum. In the call, students were asked to submit a 2-page summary of their research and on-going work. As expected, we received a wide-range of submissions, and at the end we accepted 10 submissions. Here is a list of the accepted submissions. If there are more authors on a submission, the main student author is marked with *.

- Petr Bauch: *Bit-Precise LTL Model Checking*
- Seyedhassan Daryanavard*, Thomas Marconi, Mohammad Eshghi: *Design of CAD Module for JIT Extensible Processor Customized for Placement and Routing*
- Marko Doko*, Viktor Vafeiadis: *Reasoning about Memory Fences in C11 Relaxed Memory Model*
- Usman Khalid: *Bayesian Networks based Probablistic Approach for Digital Circuits Reliability*
- Christian Krieg*, Michael Rathmair, Florian Schupfer: *Device Library Attack: Silently Compromising the FPGA Design Flow*
- Siddharth Krishna: *Learning Linear Invariants using Decision Trees*
- Andrey Kupriyanov*, Bernd Finkbeiner: *Causality-based LTL Model Checking without Automata*
- Michael Rathmair*, Florian Schupfer: *Structural System Analysis from Design Level down to Netlist Level*
- Thorsten Tarrach: *Using synthesis to fix concurrency bugs*
- Leander Tentrup: *Verifying Partial Designs with Partial Observability*

Es evident, the student submissions covered a broad spectrum of topics present in the FMCAD community. The Organizing Committee discounted submissions that were out of the scope of FMCAD.

The main purpose of the Student Forum is that the student authors of the accepted papers present their work in the poster session at the main conference. This way they can receive feedback from all conference participants. In addition, every student received a written review from an expert in their research area. Those reviews were also used to decide about the acceptance of the submitted papers. The experts were chosen from the FMCAD's Program Committee, or if there was no relevant expert in the PC, the Organizing Committee asked well-established scientists for help with reviewing. Hereby we would like to express our gratitude to all reviewers of the FMCAD Student Forum for their work.

Looking at the seniority of students, this Forum featured students at various stages of their PhD studies: there were students who just started their graduate studies, as well as students who were close to defending their thesis. While the junior students looked at the Forum as an opportunity to formulate their research goals better, for the senior students the Forum provided a chance to search for post-doctoral positions.

The heterogeneity of the students also resulted in different styles of submissions. The senior students mostly presented a summary of their results, and the submissions of the junior students were mainly surveys of existing work. However, common to all the accepted papers is that they outlined interesting and promising new research directions.

Every student author of an accepted paper received a travel grant covering all the costs to attend the FMCAD conference. The students in general did not have access to other travel funds, and these grants enabled them to benefit from attending the conference. We are deeply grateful to the sponsors of the FMCAD conference for their contributions: FMCAD, Inc. and the EPFL School of Computer and Communication Sciences, as well as (listed in alphabetical order): ARM, Atrenta Inc., Cadence, Centaur, IBM Corporation, Intel Corporation, Jasper Design Automation, Mentor Graphics, Microsoft Corporation, OneSpin Solutions, Oski Technology Inc., Real Intent, Synopsys. Their generous contributions made this Forum possible.

This Student Forum takes place because of the students and their submissions. It is their excellent work that is making this forum series a success. The Organizing Committee would like to thank all the students who submitted a proposals to this Forum, and wishes them success in their future research.

Finally, we express our gratitude for their input and guidance to Thomas Wahl, last year's Student Forum Chair, to Barbara Jobstmann, the Publication Chair of FMCAD 2014, and to Koen Claessen and Viktor Kuncak, General and Program Chairs of FMCAD 2014.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Response property checking via distributed state space exploration

Brad Bingham and Mark Greenstreet
Department of Computer Science, University of British Columbia
201-2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4
{binghamb, mrg}@cs.ubc.ca

***Abstract*—A response property is a simple liveness property that, given state predicates p and q, asserts "whenever a p-state is visited, a q-state will be visited in the future". This paper presents an efficient and scalable implementation for explicit-state model of checking response properties on systems with strongly- and weakly-fair actions, using a network of machines. Our approach is a novel twist on the One-Way-Catch-Them-Young (OWCTY) algorithm. Although OWCTY has a worst-case time complexity of $O(n^2m)$ where n is the number of states of the model, and m is the number of fair actions, we show that in practice, the run-time is a very small multiple of n. This allows our approach to handle large models with a large number of fairness constraints. Our implementation builds upon PREACH, a distributed, explicit-state model checking tool. We demonstrate the effectiveness of our approach by applying it to several standard benchmarks on some real-world, proprietary, architectural models. *Index Terms*—distributed model checking, explicit-state model checking, murphi, liveness, fairness**

I. INTRODUCTION

Response properties are liveness properties of the form "From any state in which proposition p is satisfied, execution will eventually reach a state in which proposition q is satisfied." In LTL such properties are expressed as $\Box(p \rightarrow \Diamond q)$; the corresponding CTL specification is $\mathsf{AG}\,(p \rightarrow \mathsf{AF}\,q)$. Specifications of cache protocols and high-level architectural models often include response properties – e.g. if a processor attempts to write to a memory location, the processor will eventually have an exclusive copy of that location in its cache; or, if an instruction is fetched, eventually either it will be executed and committed or that (speculative) path will be aborted.

The standard approach to explicit state model checking of LTL properties involves constructing a product automaton that is the synchronous cross product of the Büchi automaton that accepts the negation of the property in question, and the Büchi automaton for the system itself [1], [2]. If the language accepted by the product automaton is empty, then the LTL property holds; otherwise, a counterexample trace is found. All model checking approaches are vulnerable to state-explosion problems, and the product-automaton construction for LTL model checking exacerbates this problem. If the original system has n reachble states, and the LTL specification, ϕ, consists of $|\phi|$ symbols and operators, then constructing the product automaton takes $\mathcal{O}(n2^{|\phi|})$ time and space.

This research was funded in part by generous support from NSERC Canada and Intel through their CRD and URO grants.

Response properties can be expressed with a Büchi automaton with only 2 states, and thus the blowup from the formula size is curbed. Unfortunately, only contrived systems that contain no cycles along any path from a p-state to a q-state will satisfy response. In practice, response is verified subject to *fairness assumptions* that attempt to characterize realistic traces. Response may be verified under those fairness assumptions that can be written as the LTL formula $Fair$, by using LTL model checking to verify the formula $Fair \rightarrow \Box(p \rightarrow \Diamond q)$. The Büchi automaton for this formula will grow exponentially in $|Fair|$, which in turn causes the number of states of the product automaton to explode.

Instead of expressing fairness as an antecedent to the LTL property of interest, fairness can be expressed in terms of how the original system is defined or as a specially handled input to the model checking algorithm. Kesten *et al.* [3] compare expressing fairness as a property antecedent with a "fair-aware" approach and show that latter achieves better performance. Manna and Pnueli [4], [5] present a model-checking algorithm property checking for response properties that takes advantage of two notions of action-based fairness. The Divine distributed explicit-state model checking tool has a specific mode where all transitions are assumed to be weakly fair [6]. In this paper, we follow suit and employ an algorithm that directly utilizes fairness assumptions for Manna and Pneuli's notions of strong and weak fairness. In the worst-case, the algorithm could perform $O(n^2|Fair|)$ state expansions, where n is the number of reachable system states. In the typical scenario where $|Fair|$ is much smaller than $\log(n)$, this far exceeds the number of worst-case expansions of the Büchi automaton approach which is $O(n2^{|Fair|})$. However, our results show on benchmark models that the algorithm vastly outperforms the worst case, which is indeed achievable (see online Appendix [7]). In contrast, Section VII reports results for a tool that implements the Büchi automaton approach and uses time and memory as one would expect from the worst-case analysis.

Our contributions are as follows:

1) present a novel, efficient, parallel approach for model checking response properties;
2) an implementation of the algorithm built as an extension of the PREACH [8], [9] model checker. PREACH is a distributed, explicit-state model checker based on Stern and Dill's [10] algorithm;

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

3) demonstrate that verifying liveness in large, realistic systems augmented with both strong and weak fairness is tractable using a modest network of machines;
4) show that the time requirements for One-Way-Catch-Them-Young style algorithms are far better in practice than would be expected from the worst case analysis. In practice, we observe that each state is visited a small number of times (typically less than 30).

II. Overview

Stern and Dill's distributed model checking algorithm [10] partitions the state space among processes with a uniform random hash function. Processes are said to *own* states that hash to their process IDs. Once a state has been visited, its owner process is responsible for storing it locally. In PREACH this is done with the Murφ model checker's hash table [11] which uses a predetermined number of bits[1] to represent each state. The use of hash compaction and bloom filters in explicit-state model checking is a thoroughly studied area [12], [13] and lends itself to practical approaches. Hash table compression admits a small probability that some state will erroneously be viewed as visited when it actually hasn't been. In our experience this probability is tiny; for example, a very large model checking experiment with about 100 billion states had only a 0.03% chance of a missed state [8]. The experiments in this paper admit a much smaller probability than this; the `German6` model with over 316 million states had a probability of a missed state of less than 7.36×10^{-5}. If this probability were of practical concern, the user could simply re-run the tool using a different seed for randomization and reduce the probability of a missed state in *both* runs to less than $(7.36 \times 10^{-5})^2 < 5.42 \times 10^{-9}$.

Once a state has been checked in the hash table, **HT**, it is queued for expansion in the work queue, **WQ**, the other key data structure of the Stern-Dill algorithm. Unlike the **HT** which has static size and resides in memory, the **WQ** has dynamic size and stores full state descriptors. Typically only a small percentage of the **WQ** is in memory; the rest is delegated to disk. Because states can be read and written in large batches, using disk storage for the **WQ** does not create a bottleneck. A key feature to PREACH performance, particularly in a heterogeneous computing environment, is load balancing. Once a state enters **WQ**, it is irrelevant which process actually checks the invariants, computes the successor states and sends them off to their respective owners. Thus, processes that amass a longer **WQ** will offload a chunk of their states to another process with a shorter **WQ**.

Erlang's message passing system relies on nonblocking sends. When a message arrives for some process, it resides in a message inbox in memory until a matching RECEIVE is called. The dynamic nature of distributed state space exploration and the performance asymmetry introduced by heterogeneous machines, or any other performance irregularities, can lead to very long messages queues. This is especially problematic as we have observed that the time it takes the Erlang runtime to consume an inbox message increases with the number of messages in the process's inbox. To combat this issue, PREACH employs a crediting mechanism that bounds the size of each process' inbox. If process A has states to send to their owner, process B, but it does not have sufficient credits to do so, the states are simply queued in A's "outbox" for B. Outboxes that grow large are also written to disk.

To check response properties, we have implemented an algorithm inspired by the set-based *One-Way-Catch-Them-Young* algorithm described in [14], [15]. We focus on systems with both *strongly fair actions* (a.k.a. compassion), denoted $\mathcal{C}$ and *weakly fair actions* (a.k.a. justice), denoted $\mathcal{J}$.

A. Preliminaries

A fair transition system, FTS, is a tuple $(\mathcal{S}, I, T, \mathcal{J}, \mathcal{C})$ where

- $\mathcal{S}$ is a finite set of states;
- $I \subseteq \mathcal{S}$ is the set of initial states;
- transition relation $T \subseteq \mathcal{S} \times \mathcal{S}$;
- weakly fair actions $\mathcal{J} \subseteq 2^T$;
- strongly fair actions $\mathcal{C} \subseteq 2^T$.

An *action* is a subset of T. Function $En : \mathcal{S} \to 2^{\mathcal{C} \cup \mathcal{J}}$ gives the set of actions enabled at state s, i.e. $En(s) = \{a \in \mathcal{C} \cup \mathcal{J} : \exists s'.\ (s, s') \in a\}$. State s enables action a if $a \in En(s)$. Given state s we use the shorthand notations $\mathcal{C}_s$ and $\mathcal{J}_s$ to refer to the sets of enabled actions that are strongly and weakly fair, respectively. Formally, $\mathcal{J}_s = \mathcal{J} \cap En(s)$ and $\mathcal{C}_s = \mathcal{C} \cap En(s)$. For convenience we assume transitions that are not members of any element of $\mathcal{J} \cup \mathcal{C}$ are members of the *non-fair* set, i.e. $NF = T \setminus \left(\bigcup_{a \in \mathcal{J} \cup \mathcal{C}} a\right)$. For $A \subseteq \mathcal{S}$, $\langle A \rangle$ denotes the subgraph of the digraph $(\mathcal{S}, T)$ induced by A.

A *trace* is a finite sequence of states $s_0 \circ s_1 \circ \ldots \circ s_k$ where $s_o \in I$, and $(s_i, s_{i+1}) \in T$ for $0 \le i < k$. A *predecessor trace* for state s is any trace where $s_k = s$.

An *execution* is an infinite sequence of states, $s_0 \circ s_1 \circ \ldots$, where $s_0 \in I$, and $\forall i \ge 0.\ (s_i, s_{i+1}) \in T$. For a given trace, action a satisfies

- $\mathit{InfOftenTaken}(a)$, if $\forall i \ge 0.\ \exists j \ge i.\ (s_j, s_{j+1}) \in a$,
- $\mathit{InfOftenEn}(a)$, if $\forall i \ge 0.\ \exists j \ge i.\ a \in En(s_j)$, and
- $\mathit{InfOftenDisabled}(a)$, if $\forall i \ge 0.\ \exists j \ge i.\ a \notin En(s_j)$.

An execution is called *fair* if

$$\begin{array}{ll} & \forall a \in \mathcal{C}.\ \mathit{InfOftenEn}(a) \Rightarrow \mathit{InfOftenTaken}(a) \\ \wedge & \forall a \in \mathcal{J}.\ \mathit{InfOftenTaken}(a) \vee \mathit{InfOftenDisabled}(a). \end{array}$$

In other words, an execution is fair if all actions of $\mathcal{C}$ are taken infinitely often or are never enabled beyond some finite prefix of the execution, and all actions of $\mathcal{J}$ are taken infinitely often *or* are disabled infinitely often. A strongly connected component (SCC) is called *fair* (a FSCC) if all enabled strongly fair actions in the SCC's states are taken within the SCC, and all enabled weakly fair actions in the SCCs states are either taken within the SCC or disabled at some state. Section III presents an algorithm that detects FSCCs within

[1] This number is a configuration parameter. The results in this paper use the default value of 40 bits.

the subgraph of reachable states that can be reached on a path from some p-state without visiting a q-state along the way (this subset is referred to as *pending*; see Figure 1). Such SCCs are counterexamples to the response property $\Box(p \rightarrow \Diamond q)$; furthermore, every counterexample execution has an infinite suffix that only visits states in a FSCC. Note that p is a subset of $pending$, and q is disjoint with $pending$. The initial states are usually disjoint from both p and $pending$, but this need not be the case.

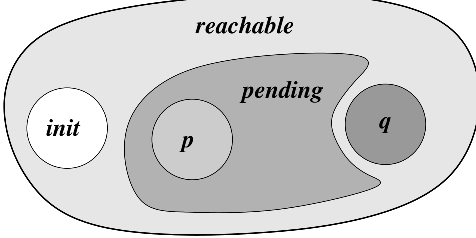


Fig. 1: Sets of interest when checking a system adheres to $\Box(p \rightarrow \Diamond q)$.

B. A note about stuttering

We note that fair systems may be defined with or without inherent stuttering, the former assuming that every state has a transition to itself and the latter does not. For simplicity in the following presentation, we assume that stuttering is allowed, thereby requiring a fair "reason" why indefinite stuttering cannot occur. This assumption requires that T is reflexive. Including stuttering simplifies the presentation; for example, it ensures that all traces can be extended to infinite executions.

III. ALGORITHM

Our distributed response checking algorithm is based on the One-Way-Catch-Them-Young (OWCTY) [14] approach. The key idea of the algorithm is to begin by initializing a set, $MaybeFair$, with the $pending$ states, and then iteratively remove states from $MaybeFair$ that cannot belong to a FSCC. A state, s, is removed when it is discovered that there is no predecessor trace of s in $\langle MaybeFair \rangle$ along which action $a \in \mathcal{C}$ is taken, where $a \in \mathcal{C}_s$. Similarly, s is removed if it is found that there is no predecessor trace of s in $\langle MaybeFair \rangle$ along which action $a \in \mathcal{J}_s$ is either taken *or* disabled at some state s' of the trace, where $a \in \mathcal{J}_s$. The response property holds iff $MaybeFair$ is empty when the algorithm terminates. To see this, note that any state that is removed from $MaybeFair$ cannot belong to a FSCC; thus, $\langle MaybeFair \rangle$ contains all of the FSCCs of $\langle pending \rangle$. The FSCCs of $\langle MaybeFair \rangle$ form a DAG. Let F be any FSCC of $\langle MaybeFair \rangle$ that has no predecessor FSCCs. It is straightforward to construct a cycle in F that satisfies all fairness constraints. By construction, this cycle is reachable from some initial state.

The description of OWCTY from [15] for model checking LTL formulas with strong and weak state-based fairness operates on sets of states performing union and disjunction operations, as well as deleting all members from a set which have

Algorithm 1 High level algorithm

```
 1 procedure FindFairCycle(S, I, T, C, J, p, q)
 2     ▷ Compute the pending states
 3     pending ← Reachability(S, I, T, p, q)
 4     ptfa ← new bit[pending][J ∪ C]          ▷ array of bit-strings
 5     Clear(ptfa)                              ▷ initialize to all 0s
 6     MaybeFair ← pending
 7     Prev ← ∅
 8     while MaybeFair ≠ Prev do
 9         Prev ← MaybeFair
10         ToExpand ← MaybeFair
11         while ToExpand ≠ ∅ do
12             s ← RemoveSomeElement(ToExpand)
13             ▷ Weakly fair actions not enabled at s
14             for all a ∈ J \ J_s do
15                 ptfa[s][a] ← 1
16             end for
17             Next ← Successors(s) \ q
18             for all s' ∈ Next do
19                 OldActions ← ptfa[s']
20                 a ← WhatActionTaken(s, s')
21                 if a ∈ J ∪ C then
22                     ptfa[s'][a] ← 1              ▷ Record action taken
23                 end if
24                 ▷ Actions preceeding s also preceed s'
25                 ptfa[s'] ← BitwiseOr(ptfa[s], ptfa[s'])
26                 if (ptfa[s'] ≠ OldActions) then
27                     ToExpand ← ToExpand ∪ {s'}
28                 end if
29             end for
30         end while
31         for all s ∈ MaybeFair do
32             if ∃a ∈ J_s ∪ C_s : ptfa[s][a] = 0 then
33                 MaybeFair ← MaybeFair − {s}
34             end if
35         end for
36         Clear(ptfa)
37     end while
38     return MaybeFair ≠ ∅
39 end procedure
```

no predecessor within the set until a fixed point is reached[2]. As described in Section II, PReach uses lossy compression when hashing states; thus, we cannot reconstruct states from hashtable entries. To retain the efficiency advantages of the Murφ hashtables, we avoid the explicit representation of large sets of states, and replace the union and intersection operations of OWCTY with tag bit manipulations, where each hash table entry includes one such tag bit per fair action. In Algorithm 1, these bits are stored in $ptfa$ *(predecessor trace fair actions)*, which is a two-dimensional array of bits initialized to all 0s. Bit $ptfa[s][a]$ is set for action $a \in \mathcal{J} \cup \mathcal{C}$ and state $s \in MaybeFair$ is set if a is taken in a predecessor trace of s in $\langle MaybeFair \rangle$, or if $b \in \mathcal{J}$ is disabled at some state of a predecessor trace of s in $\langle MaybeFair \rangle$. The set $pending$ stores the states of interest for response, those that can be reached on a path from a p-state without visiting a q-state.

Each iteration of the outer while-loop is called a *round*, and involves two *phases*.

Action Propagation Phase (AP):
This step is the while-loop from lines 11 to 30. Some state s is removed from $ToExpand$ and the tag bits are set for each

[2]To the best of our knowledge, the algorithm from [15] not been implemented.

weakly fair action that is disabled at s; this is because any eventual successor of s within $\langle pending \rangle$ *may* be part of an SCC with s. If so, this SCC is fair with respect to these weakly fair actions. Then, the successors of s within $\langle pending \rangle$ are computed. For each of these the current tag bits are saved in $OldActions$. If the transition that is taken from s to reach a successor s' is a member of some $a \in \mathcal{J} \cup \mathcal{C}$, the $ptfa[s'][a]$ is set (line 22). Then, the bit-string $ptfa[s]$ is ORed with the $ptfa[s']$, as any predecessor trace ρ for s implies a predecessor trace for s', namely $\rho \circ s'$. If any of these operations have set new bits for s', it must be added to $ToExpand$ so the bits are propagated along. Otherwise, the s' is discarded. This loop continues until a fixed point is reached for the contents of $ptfa$.

Figure 2 illustrates some operations of AP with an example. For this example, $\mathcal{J} = \{a_0, a_1, a_2, a_3\}$ and $\mathcal{C} = \{a_4, a_5, a_6, a_7\}$, and PTFAs are represented as $a_7 \dots a_0$, as seen below each state. Assume that $En(b) = \{a_0, a_2, a_3, a_4\}$, $En(c) = \{a_0, a_1, a_7\}$, and $En(d) = \{a_0, a_1, a_3, a_5\}$. When b is expanded, the PTFA on the arc is passed to state e which changes the PTFA for e and requires e to be expanded. Subsequently, c is expanded and the PTFA for e is again updated and another e expansion is needed to communication the new PTFA to successors. Finally, when d is expanded the PTFA sent to e contains no new actions, so e does not need another expansion.

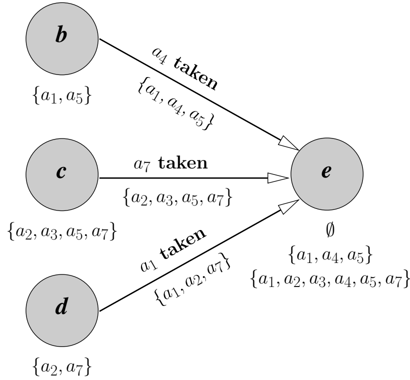


Fig. 2: Example of PTFA updates as states are expanded.

State Deletion Phase (SD):
This phase appears from lines 31 to 36. Any states that enabled a fair action a but with the corresponding tag bit cleared cannot be part of a FSCC and are removed from $MaybeFair$.

Soundness for the algorithm was described at the beginning of this section. To see that the algorithm terminates, we first note that the while loop at lines 10–28 must terminate because the flag bits in $ptfa$ are strictly increasing with successive iterations of the loop. The while-loop at lines 7–35 terminates because the loop adds no new elements to $MaybeFair$.

IV. Distributed Implementation

The distributed version of this algorithm starts with a Stern-Dill style reachability computation that identifies all p and $pending$ states. Each worker process stores its p states on disk, and $pending$ states are marked with tag bits in the hash-table. Initially, the PTFA for $pending$ states are set to all fair actions, $\mathcal{J} \cup \mathcal{C}$. The distributed algorithm then performs rounds that correspond to those of the sequential version, Algorithm 1. As described in more detail below, each round propagates PTFA tags according to the next state relation until a fix point is reached. At the boundary between rounds, states are identified whose PTFAs do not satisfy the fairness constraints for the state. Such states cannot be part of an FSCC and are marked as "dead" (i.e, removed from $MaybeFair$). The number of live, $MaybeFair$ states is non-increasing. The algorithm terminates when this number no-longer decreases. If at this point, all $MaybeFair$ states have been eliminated, then the response property is satisfied. Otherwise, counter-example is generated. The remainder of this section describes this algorithm in more detail.

Algorithm 2 shows pseudo-code for the root process. It initiates the initial reachability computation to identify p and $pending$ states. It then initiates rounds of propagating PTFA tags and eliminating $pending$ states until no further states can be eliminated. The termination detection algorithm from the original Stern and Dill approach is used to identify the end of each round and compute the total number of $pending$ states. This provides a barrier separating the computations of successive rounds. After the final round, the root process notifies the workers that the computation is complete and reports either that the response property has been verified or provides a counter-example.

Algorithm 3 shows pseudo-code for the worker processes. Like the reachability computation, each worker has two main activities: receiving incoming states and checking if they have been "seen" previously, and expanding states to send their successors to their owners. Algorithm 3 augments each of these activities to maintain the tags for PTFAs. At the beginning of each round, each process checks its subset of the p states to determine which ones satisfied their associated fairness constraints in the previous round. Those that don't are marked as dead. All p-states are added to the work-queue, $ToExpand$, even if they are dead to ensure that their successors are examined in this round. When a state is received, the algorithm first checks to see if this is the first time the state has been seen for the current round. If so, the state's PTFA is checked to see if the state should be marked as dead, and all states are entered into $ToExpand$ the first time they are visited in each round. If the state has been seen before, then if the new PTFA indicates incoming paths for fairness constraints that haven't already been satisfied, these constraints are added to the state's PTFA, and the state is enqueued in $ToExpand$ to propagate this information to its successors.

When a worker removes a state from its work queue, $ToExpand$, it computes all successor states as in the original reachability algorithm. Because the incoming paths to this state are prefixes of incoming paths for its successors, the PTFA of the successor must contain the PTFA for this state. Furthermore, if the transition to the new state corresponds to a fair action, then this action is added to the PTFA. These

updates are made to the PTFA for the successor, and the successor with this PTFA set is sent to the successor's owner.

Every operation either marks a state a dead or adds a fair action to some state's PTFA. Thus, the activities for updating fairness information eventually reach a fixpoint and the round terminates. Many optimizations are possible to improve the performance of this algorithm. These are described in the next section.

Algorithm 2 Root Process

```
 1 function ROOTSTART(I, p, q)
 2     ▷ Tags for initial states
 3     for all s ∈ I do
 4         SENDSTATE((s, ∅))
 5     end for
 6     CurMaybeFairCount ← TALLY(nstates)
 7     PrevMaybeFairCount ← CurMaybeFairCount + 1
 8     while CurMaybeFairCount ≠ PrevMaybeFairCount do
 9         BROADCAST(doRound)
10         PrevMaybeFairCount ← CurMaybeFairCount
11         CurMaybeFairCount ← TALLY(nstates)
12     end while
13     BROADCAST(stop)
14     if CurStates > 0 then
15         return GENERATECOUNTEREXAMPLETRACE(...)
16     else
17         return verified
18     end if
19 end function
```

Algorithm 3 Worker Process

```
 1 function WORKER(S, I, T, J, C, p, q, rootPid)
 2     PS ← COMPUTEPSTATES(S, I, T, J, C, p, q)
                        ▷ Global variable queue that stores p-states
 3     RoundCount ← 0
 4     while true do
 5         case RECEIVE() of                    ▷ Blocking receive
 6             doRound → ok
 7             stop → break while loop
 8         end case
 9         RoundCount ← RoundCount + 1
10         for all s ∈ PS do
11             WQ ← INITSTATEFORROUND(s, ∅, RoundCount)
12         end for
13         ▷ Stern and Dill's termination alg
14         while round not terminated do
15             while (s, thisPTFA) ← RECEIVE() do     ▷ Nonblk. recv
16                 T ← HT.GETTAGS(s)
17                 if T.round ≠ RoundCount then
18                     INITSTATEFORROUND(s, thisPTFA, RoundCount)
19                 else if ¬T.dead ∧ (thisPTFA ⊈ T.PTFA) then
20                     T.PTFA ← T.PTFA ∪ thisPTFA
21                     WQ.INSERT((s, T))
22                     HT.UPDATETAGS(s, T)
23                 end if
24             end while
25             EXPANDANDSEND(J, C)                 ▷ See Alg. 4
26         end while
27         send (nstates, MyMaybeFairCount) to rootPid
28     end while
29 end function
30
31 function INITSTATEFORROUND(s, thisPTFA, RoundCount)
32     T ← HT.GETTAGS(s)
33     if ENABLED(s) ⊈ T.PTFA then
34         T.dead ← true
35         thisPTFA ← ∅
36     end if
37     T.round ← RoundCount
38     T.PTFA ← thisPTFA
39     WQ.INSERT((s, T))
40     HT.UPDATETAGS(s, T)
41 end function
```

V. Optimizations

Early experiments with a prototype implementation revealed several opportunities to improve performance. We aim to address the average number of state expansions during a phase, the number of states visited during a phase, *and* the number of rounds. A key observation is that for many examples, the number of states in the *pending* set decreases rapidly with successive rounds. Thus, it is important to avoid touching "dead" states so that the work done in later rounds decreases with the smaller size of *pending*. This also means that most of the time is spent in the initial reachability computation and the first two or three rounds of the liveness computation. Thus, optimization should focus on these early rounds. Furthermore, the same state can be updated several times during a single round. Consolidating these updates was simple and led to significant performance gains. The remainder of this section presents three methods of reducing each of these metrics in turn. In addition, various optimizations are inherited from PREACH's state space exploration technique. Namely, load balancing of states offers modest speedups even in a homogeneous network of machines. Batching of states into messages containing hundreds or thousands is also of benefit. The reader may consult [8] for details.

A. Saved Expansions

The description in the algorithms and implementations presented so far have states paired with their tags, including PTFAs, when enqueued to the $\mathbf{WQ}$. When the $\mathbf{WQ}$ grows large, state s may arrive tagged with PTFA b_2 while the same state is waiting for expansion in the $\mathbf{WQ}$ while paired with PTFA b_1, which matches the PTFA at the $\mathbf{HT}$ entry for s. When b_2 has at least one bit set that b_1 does not, s is enqueued for expansion in $\mathbf{WQ}$ paired with PTFA $b_1 \cup b_2$. This renders the earlier $\mathbf{WQ}$ entry of (s, b_1) redundant and unnecessary.

To avoid this scenario, the $\mathbf{HT}$ is used to maintain PTFA information, and $\mathbf{WQ}$ entries do not contain a PTFA. When a state s is enqueued, a new $\mathbf{HT}$ tag bit $InWQ$ is set; when s is dequeued, $InWQ$ is cleared and the current $\mathbf{HT}$ value for PTFA is used when computing the PTFA for s's successors. If state s with PTFA b_2 arrives when the $\mathbf{HT}$ entry has $InWQ$ set, then $\mathbf{HT}$ PTFA $b_{\mathbf{HT}}$ is set to $b_{\mathbf{HT}} \cup b_2$ and the just-arrived state s is discarded. This approach reduces the number of state expansions at the cost of an additional bit in $\mathbf{HT}$ per state, and one additional $\mathbf{HT}$ lookup.

B. Dynamic Kernel

The algorithm implementation above uses the reachable p-states as the *kernel*, defined as follows.

Definition 1: Given a FTS, $K \subseteq \mathcal{S}$ is a *kernel* for $A \subseteq \mathcal{S}$ if A is a subset of the reachable states from K in the digraph

Algorithm 4 Dequeues a **WQ** state and sends next states with tags to their owners.

```
1  function ExpandAndSend(𝒥, 𝒞)
2      if IsEmpty(WQ) then
3          return done
4      end if
5      (X, Tags) ← Dequeue(WQ)
6      NextStates ← ComputeSuccessors(X)
7      if Tags.dead then
8          for all s′ ∈ NextStates do
9              SendState((s′, ∅))
10         end for
11         return
12     end if
13     PTFA ← Tags.PTFA
14     PTFA ← PTFA ∪ (𝒥 − Enabled(X))
15     for all s′ ∈ NextStates do
16         ActionTaken ← whatActionTaken(X, s′)
           ▷ Successor PTFA is current state PTFA with the fair action taken
17         if ActionTaken ∈ NF then
18             NextPTFA ← PTFA
19         else
20             NextPTFA ← PTFA ∪ ActionTaken
21         end if
22         SendState((s′, NextPTFA))
23     end for
24     return
25 end function
```

$(\mathcal{S}, T)$.

Note that the initial states I is a kernel for any subset of the reachable states. In the code presented in Section IV, we used the reachable p-states K_p as a kernel for $MaybeFair$ to initiate each phase because K_p is a kernel for every subset of $pending$. Our experiments showed that for typical examples, the number of states in $MaybeFair$ drops rapidly with each SD phase. The expansion of such deleted states can be avoided by modifying K after each SD phase, using an extra **HT** tag bit InK and additional disk space.

During the initial phase, only the p-states have InK set to true, and these states are saved to disk in the kernel-queue. When a state s is removed from $MaybeFair$ during SD that has InK set, this flag is cleared. When a process receives state s' tagged with mode delete_pred (signaling that a predecessor of s' has just been removed from $MaybeFair$), then if s' has its InK flag cleared, it is set to true and s' is added to the kernel-queue. Finally, at the start of an AP phase the kernel-queue is copied to the **WQ** to serve as the set of initial states, but any state encountered that has its InK flag cleared is ignored and removed from the kernel-queue.

While this approach does not necessarily maintain the smallest possible kernel for $MaybeFair$, its simple implementation and low overhead lead to large performance gains.

C. Deletion by Predecessor Counting

There are performance advantages when storing the number of predecessors each state has in $\langle MaybeFair \rangle$. Under the assumption of stuttering and ensuring the safety property that every state $s \in pending$ has $|\mathcal{J}_s \cup \mathcal{C}_s| \geq 1$, any state with 0 predecessors in $\langle MaybeFair \rangle$ will be deleted from $MaybeFair$ in the next SD phase. However, storing the number of predecessors in **HT** allows detection of this case in order to preemptively remove such states. We choose to add 8 bits to the **HT** tags to store the predecessor count. This additional bookkeeping complicates Algorithms 3 and 4 somewhat (details omitted). In particular, a state may be expanded more than once during an SD. This occurs when the first time a state is visited the condition on line 33 of Algorithm 3 holds, but subsequently all of its predecessors are deleted. However, this turns out to be a rare occurrence in the benchmarks, and this strategy can reduce the number of phases. Note that the impact of this optimization is omitted from the Results section as it was inherent to our early implementation versions.

VI. Results

We ran PReach on a variety of combinations of Murφ models with all optimizations of section V enabled, summarized in Table I. For each, we chose a suitable response property such as “requests for exclusive access to a cache line are eventually granted”, or “processes waiting to enter the critical section will eventually do so”. The Murφ models used are the German cache coherence protocol, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm, a snoopy protocol used as a benchmark in previous verification work [16] and an Intel proprietary protocol. Let `GermanX` denote the German model with `X` caches; `petersonY` is Peterson’s algorithm with `Y` processes and `mcslock5` is the MCS Lock algorithm with 5 processes; `snoop2` is the snoopy protocol with 2 L1 caches and 2 clusters. Models `saw`, `gbn` and `swp` are various sliding window communication protocols, with the response property that the sender can always eventually accept new data to transmit. All models and the PReach code is provided online [7]. Each Murφ “rule” (a.k.a. guarded command) is considered a separate action; we attached suitable fairness assumptions specific to the model. The network of machines used for experiments are as follows:

- UBC cluster: 40 PReach processes on a homogeneous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 8 GB of memory (non-`intel_*` models).
- Intel cluster: 16 PReach processes on a heterogeneous network of contemporary Intel® Xeon® machines, each with at least 8 GB of memory (`intel_*` models).

Not included in the table, but worth noting, is an Intel proprietary sliding window protocol model. With over 450 million states and tens of fairness (both strong and weak), we were able to verify response in about 5 and a half hours using 32 cores.

A few modifications were required when checking the `snoop` protocol. This model was created to represent a cache-coherence protocol in a realistic processor. The protocol appears to have been designed with an emphasis on safety, and liveness does not appear to have been primary concern. For example, requests for cache lines are clearly not responsive as they may be *negatively acknowledged* (Nackd) an arbitrary number of times. To avoid this, we changed the protocol so that Nacks of this type are simply ignored, and the request

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

model	runtime	states	p-states	$pending$-states	q-states	rounds	exp/state	no -ko	no -se	no opt.
German5_sf	189	15,836,445	3,699,486	4,858,596	5,103	1	3.48	0.98	2.42	2.86
German6_sf	4,253	316,542,087	74,465,244	95,266,520	18,225	1	3.33	1.01	3.30	3.52
peterson6_wf	820	13,817,679	2,947,800	12,111,713	45,209	14	12.91	1.65	1.30	1.95
peterson6_sf	423	13,817,679	2,947,800	12,111,713	45,209	5	9.03	1.36	1.73	2.12
peterson7_wf	26,957	380,268,668	79,029,594	340,549,743	775,138	17	14.19	1.65	1.66	2.16
peterson7_sf	14,613	380,268,668	79,029,594	340,549,743	775,138	6	10.11	1.27	2.26	-
mcslock5_wf	1415	59,318,541	27,785,789	51,474,427	2,780,517	3	5.09	1.17	1.10	1.25
snoop2_sf	160	2,648,763	670,689	1,313,100	1,335,663	3	12.71	1.07	4.57	5.00
saw20_sf	323	314,183	309,140	309,140	5,043	23	44.06	1.04	1.09	1.15
gbn3_2_sf	369	12,753,395	7,859,200	7,859,200	4894195	6	6.44	1.60	1.95	2.56
swp4_2_sf	503	18,595,425	11,715,440	11,715,440	6,879,985	6	6.58	1.59	1.63	2.22
intel_small_sf	285	476,778	268,078	268,078	164,057	4	6.36	-	-	-
intel_med_sf	1,015	2,696,059	1,944,360	1,944,360	635,672	4	8.59	-	-	-
intel_big_sf	13,872	51,791,350	29,899,694	29,899,694	19,855,989	8	11.92	-	-	-

TABLE I: Column "runtime" is given in seconds; "exp/state" is the average number of times each $pending$-state was expanded. Model peterson6_sf is peterson6 with all actions strongly fair, as opposed to peterson6_wf where some rules were weakly fair and the rest as not fair (for example, the rule that initiates the move from the noncritical section to requesting to enter the critical section needs no fairness assumption). These two models have the same number of states of each type but perform a different number of expansions, and illustrate the benefit of only using more fairness than required for the response property to hold. All other models require strong fairness.

persists. This turned out to also not be responsive, although less obviously so – the counterexample trace included 72 transitions. Therefore, not all of the pending states were deleted. Online Appendix [7] Figure 5 shows that about half of the the $pending$-states remained in the $MaybeFair$ set when the algorithm terminated. Additional plots for the experiments appear in the Appendix.

The rightmost three columns of Table I show the slowdown when benchmarks are run without the kernel optimization, without the saved expansions optimization and without either, respectively. The kernel optimization is of most benefit when the number of rounds is large[3]. In particular, it is of no benefit for those benchmarks that only require a single round, as the kernel states are only used during subsequent rounds. The saved expansion optimization offers large performance gains in many cases. Typically, only 5 to 10% of the total state expansions are explicitly avoided by the when a just-received state state is present in the **WQ**. However, avoiding these redundant expansions can in turn save many expansions of successor states which in turn saves expansions of states that are two transitions away. This cascading effect decreases the total number of expansions by a significant factor.

VII. Related Work

Divine is a parallel and distributed LTL model checker that is the closest tool to ours [17]. Divine constructs a product Büchi automaton to check liveness properties; thus, Divine's space requirement grows as the product of the number of states in the system model and the number in the system automaton. Applying Divine to the examples from Section VI, we observed that it ran out of memory for all examples except for those with no or a small number of strong fairness constraints. Divine provides a mode for models where all transitions are weakly fair. Using this feature, Divine performed well for the Peterson example for which weak fairness constraints are sufficient to ensure responsiveness. However, many problems require strong fairness; for example cache coherence protocols often include states where taking one action disables another. We found that for an encoding of the German protocol with 4 caches, the reachable state space of Divine's product automaton doubled with *each* additional fair action included. For only 6 fair rules, Divine on a multicore machine took 17 minutes to construct the system automata, 13 minutes to perform the model checking task and used over 16 GB of main memory. In our experiments, adding one more fair rule exhausted the main memory of our 32 GB machine and rendered the computation time infeasible.

Our algorithm has a worst-case time complexity that is at least $O(n^3)$ where n is the number of reachable states – it is straightforward to construct an example where the transition relation has $O(n^2)$ edges, and for which Algorithm 1 removes one state per iteration of the outer while-loop. In practice, we observe that the transition relation is sparse and Algorithm 1 converges in far fewer then n rounds – the most extreme case in Table I has 23 rounds. The *worst-case* time complexity of Divine is better, $O(n2^{|\phi|})$ – Divine replaces a factor of the system model size with the number of states for the checking Büchi automaton. However, our experiments show that the actual time and memory requirements for Divine's algorithm are fairly close to what one would expect from the worst-case bounds, while our approach, in practice, scales much more efficiently. We see this gap between worst-case and actual performance as a promising area for further investigation.

Using a sequential algorithm for accepting cycle detection such as Tarjan's [18], SCCs may be found in $\mathcal{O}(|V|+|E|)$ time. However, such DFS-based algorithms are unsuited to parallelization unless $P = NC$ [19]. Manna and Pnueli presented sequential algorithm for model checking response properties of fair transitions systems [5], but this is not easily parallelizable and so scalability is limited. Recently, Holzmann implemented some interesting liveness checking algorithms in a multicore version of SPIN [20]; however this approach will only find counterexamples of bounded length. Other work related to ours includes that of the authors of the LTSMin

[3]One exception is saw_20_sf where a large proportion of the runtime is spent coordinating threads between rounds.

model checking tool, most notably their algorithms for parallel SCC decomposition on multicore machines [21], [22].

VIII. Conclusions and Future Work

We have extended the PReach explicit-state, distributed model-checking tool to support verification of response properties under both strong and weak fairness of actions. Our approach uses multiple rounds of reachability computation to implement a variation of the OWCTY algorithm. For a model with n states, m fairness constraints, OWCTY could expand states $O(nm)$ times on average. This would be prohibitively expensive. Our implementation shows that for practical examples, the number of rounds is small – typically less than 30, with a maximum of about 44. Thus, OWCTY appears to provide a practical approach to checking response properties for real-world problems. For these examples, liveness checking is slower than safety checking, but not prohibitively so.

Implementing our algorithm on top of the PReach distributed model checker allows it to exploit the aggregate memory of large compute clusters. This enabled verification of response properties for a sliding-window protocol with over 450 million states in about $5\frac{1}{2}$ hours.

We compared our approach with a tool that uses the standard product-automaton formulation, with one automaton for the system model, and the other for the LTL liveness property. As predicted by the worst-case analysis, we observed that the size of the property automaton grew exponentially with the number of fairness constraints. The product-automaton approach was significantly faster than PReach for the problems that it could complete. However, it ran out of memory for all but the smallest examples.

This approach can be generalized in a number of directions. One is to handle other simple liveness properties such as *reactivity*, expressed in LTL as $\Box\Diamond p \vee \Diamond\Box q$, where p and q are past formulas. We hope to combine these model checking methods with the decompositional inference rules of Manna and Pnueli [4], [5]. Such decompositions establish that a response property is implied by a handful of safety properties and "smaller" response properties, i.e. depending on a smaller fraction of the state space. Adapting our algorithm to verify multiple such response properties in the same model checking run would leverage human insight to increase performance.

Acknowledgments

The authors extend their gratitude to Jiří Barnat for his help in understanding and running Divine. They also appreciate assistance from colleagues Jesse Bingham and Jim Grundy at Intel for providing examples of architectural models during the first author's internship, and Flemming Andersen for his vision and support for developing and demonstrating scalable verification methods and tools.

References

[1] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*. IEEE Comp. Soc. Press, Jun. 1986, pp. 332–344.

[2] R. Gerth, D. Peled, M. Y. Vardi, R. Gerth, D. D. Eindhoven, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *In Protocol Specification Testing and Verification*. Chapman & Hall, 1995, pp. 3–18.

[3] Y. Kesten, A. Pnueli, L.-O. Raviv, and E. Shahar, "Model checking with strong fairness," *Form. Methods Syst. Des.*, vol. 28, pp. 57–84, January 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1116046.1116050

[4] Z. Manna and A. Pnueli, "Completing the temporal picture," *Theor. Comput. Sci.*, vol. 83, pp. 97–130, June 1991. [Online]. Available: http://portal.acm.org/citation.cfm?id=111775.111780

[5] ——, "Temporal verification of reactive systems: Progress (draft)," http://theory.stanford.edu/~zm/tvors3.html, 1996.

[6] J. Barnat, J. Havlíček, and P. Ročkai, "Distributed LTL Model Checking with Hash Compaction," *Electr. Notes Theor. Comput. Sci.*, vol. 296, pp. 79–93, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2013.07.006

[7] B. Bingham, "Preach-response," https://bitbucket.org/binghamb/preach-response, 2013.

[8] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *Parallel and Distributed Model Checking*, 2010.

[9] J. Bingham, J. Erickson, B. Bingham, and F. M. de Paula, "Open-source PReach," http://bitbucket.org/jderick/preach, 2013.

[10] U. Stern and D. L. Dill, "Parallelizing the murphi verifier," *Formal Methods in System Design*, vol. 18, no. 2, pp. 117–129, 2001.

[11] ——, "Improved probabilistic verification by hash compaction," in *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, 1995, pp. 206–224.

[12] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *IN COMPUTER AIDED VERIFICATION. 5TH INTERNATIONAL CONFERENCE*. Springer-Verlag, 1993, pp. 59–70.

[13] P. C. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 367–381.

[14] Y. Kesten, A. Pnueli, and L. on Raviv, "Algorithmic verification of linear temporal logic specifications," in *Proc. 25th Int. Colloq. Aut. Lang. Prog., volume 1443 of Lect. Notes in Comp. Sci.* Springer-Verlag, 1998, pp. 1–16.

[15] I. Černá and R. Pelánek, "Distributed explicit fair cycle detection," in *Proc. SPIN workshop*, ser. LNCS, vol. 2648. Springer, 2003, pp. 49–74.

[16] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou, "Hierarchical cache coherence protocol verification one level at a time through assume guarantee," in *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, 2007, pp. 107–114.

[17] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[18] R. E. Tarjan, "Depth-first search and linear graph algorithms," *Siam Journal on Computing*, vol. 1, pp. 146–160, 1972.

[19] J. Barnat, L. Brim, and P. Ročkai, "A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties," in *Formal Methods and Software Engineering (ICFEM 2009)*, ser. LNCS, vol. 5885. Springer, 2009, pp. 407–425.

[20] G. J. Holzmann, "Parallelizing the spin model checker," in *Proceedings of the 19th international conference on Model Checking Software*, ser. SPIN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 155–171. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31759-0_12

[21] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs, "Multi-core nested depth-first search," in *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, ser. Lecture Notes in Computer Science, vol. 6996. London: Springer Verlag, July 2011, pp. 321–335.

[22] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. van de Pol, "Improved multi-core nested depth-first search," in *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis, ATVA 2012, Thiruvananthapuram (Trivandrum), Kerala*, ser. Lecture Notes in Computer Science, vol. 7561. London: Springer Verlag, October 2012, pp. 269–283.

Towards Pareto-Optimal Parameter Synthesis for Monotonic Cost Functions

B. Bittner, M. Bozzano, A. Cimatti, M. Gario, A. Griggio
Fondazione Bruno Kessler, Trento, Italy
Email: surname@fbk.eu

***Abstract*—Designers are often required to explore alternative solutions, trading off along different dimensions (e.g., power consumption, weight, cost, reliability, response time). Such exploration can be encoded as a problem of parameter synthesis, i.e., finding a parameter valuation (representing a design solution) such that the corresponding system satisfies a desired property. In this paper, we tackle the problem of parameter synthesis with multi-dimensional cost functions by finding solutions that are in the Pareto front: in the space of best trade-offs possible. We propose several algorithms, based on IC3, that interleave in various ways the search for parameter valuations that satisfy the property, and the optimization with respect to costs. The most effective one relies on the reuse of inductive invariants and on the extraction of unsatisfiable cores to accelerate convergence. Our experimental evaluation shows the feasibility of the approach on practical benchmarks from diagnosability synthesis and product-line engineering, and demonstrates the importance of a tight integration between model checking and cost optimization.**

I. Introduction

Many application domains can be described in terms of *parameterized* systems, where parameters are variables whose value is invariant over time, but is only partially constrained. Choosing an appropriate value of the parameters is a widely spread engineering problem, a form of design space exploration where the parameters can represent different design or deployment decisions. Examples of domains that require the analysis of various solutions include function allocation [1], [2], automated configuration of communication media (e.g., time-triggered ethernet protocols [3], flexray [4], [5]), product lines [6], dynamic memory allocation [7], schedulability analysis [8], and sensor placement [9], [10]. In fact, finding an appropriate valuation is rarely sufficient. Often designers are interested in finding *the most appropriate* valuation with respect to weight, latency, memory footprint, flexibility, reliability. Even more interestingly, there are cases where several of the above dimensions must be taken into account at the same time, and it may be necessary to trade off according to multiple cost functions.

In this paper we consider the problem of parameter synthesis when multiple cost functions cannot be easily combined into one. For example, it is possible that a configuration that is best in terms of reliability (e.g., due to redundancy) may not be optimal in terms of weight, cost, or response times. The solution is to build the so-called Pareto front [11], that is the set of parameter valuations that cannot be improved along one dimension without increasing the cost along the others[1].

We present several algorithms for the construction of the Pareto front on the space of parameter valuations that satisfy a parameterized model checking problem. We remark that we focus on *universal* parameter valuations, that guarantee the satisfaction of a property *for all* associated execution traces: this means that it is not sufficient to analyze a single trace (e.g., constructed by a bounded model checker) to have a guarantee that the parameter valuation is valid.

We tackle the problem under the assumption of monotonicity, that naturally occurs in several domains of practical interest, such as sensor placement [10], product lines engineering [6] and fault-tree analysis [12]. In particular, we require that (i) the space of parameters is upward-closed with respect to property satisfaction, and (ii) the cost functions are monotonic. We propose several algorithms of increasing strength. The first idea is to proceed by *valuations-first*, i.e. to identify the set of all valuations that satisfy the property, and then, within this set, represented as a formula in the parameter variables, to identify the ones on the Pareto front. The upfront computation of the set of valid parameter valuations, corresponding to the first phase, can be tackled in various ways. One way is to carry out a symbolic reachability in the parameterized transition system, e.g., by means of a BDD-based model checker [13]. The scalability of BDD-based techniques is however rather limited. An alternative approach is to solve the existential parameter valuation problem for the negation of the property and then complement. This can be easily encoded on top of a SAT-based engine, where the parameters are free. Once the set of valuations is found, we can independently optimize the complement set [14]. Unfortunately, this approach does not allow us to exploit the cost function for pruning.

The second approach, referred to as *one-cost slicing*, prioritizes the search according to one cost function. The first step is to identify a target value, and to collect all the valid parameter valuations. Then, the valuation with the best value along the other cost functions is selected and further optimized, so that one point in the Pareto front is found. The process is iterated until the limit target value is reached. The monotonicity assumption guarantees that the search can be suitably initialized. Compared to the previous one, this

[1]More formally, the Pareto front of a set of parameter valuations is the subset composed by those valuations associated with cost vectors that are not strictly dominated by any other solution. One valuation γ strictly dominates (or "is preferred to") a valuation γ' if each value of γ is not strictly greater than the corresponding value of γ', and at least one value is strictly less. That is, $\gamma_i \leq \gamma'_i$ for each i, and $\gamma_i < \gamma'_i$ for some i. This is written as $\gamma \prec \gamma'$ to mean that γ strictly dominates γ'. Then the Pareto frontier is the set of points from Γ that are not strictly dominated by any other point in Γ.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

approach needs not wait until all the valid parameter valuations are found; however, it still relies on the computation of the valuations for the selected slice.

The third approach, referred to as *costs-first*, is conceptually rather simple. It is based on a sampling of the space of cost values, and for each of them, on solving the associated (non-parameterized) ground model checking problem. This approach, apparently quite naïve, turns out to be extremely efficient once appropriately cast in the setting of IC3 [15]. Intuitively, rather than solving the ground problem, we solve the parameterized problem under assumptions. When IC3 successfully terminates, as a byproduct it produces a parameterized inductive invariant, possibly containing the assumptions, that is sufficiently strong to prove the property. From the validity proof, we extract an unsatisfiable core that allows us to reduce the candidate set of parameters. This step has a substantial effect in speeding up the search, by accelerating it towards (potentially) less expensive parameter configurations. Another advantage is in the fact that the approach works directly in the space of good parameters, and is thus providing an "any-time" algorithm, that can return a subset of the Pareto front if run only within a given resource bound.

We experimentally evaluated the approach, working on benchmarks from various sources [10], [16]. The results show a significant speed up with respect to methods based on enumeration of violations, both in terms of one cost function, and in the case of multiple cost functions. Incidentally, we also report substantial scalability improvements in significant practical cases, compared to a BDD-based approach previously used for single-cost optimization [10].

Structure of the Paper: This paper is structured as follows. In Section II we review some related work. In Section III we define the spectrum of problems. In Section IV we define the various solutions, and in Section V we discuss the impact of IC3 specific techniques. In Section VI we present two motivating domains. In Section VII we evaluate experimentally the three approaches. In Section VIII we draw some conclusions, and outline some directions for future work.

II. Related work

There are many works dealing with parameter synthesis and parameter optimization. The literature can be classified along various dimensions: discrete parameters versus continuous parameters; combinational (e.g., SMT) problems versus sequential (e.g., reachability) problems; number and quality of parameter valuations found (one vs all valuations vs the optimal ones).

a) MaxBMC: The work closest to ours is [17], where the Pareto front is synthesized in the case of circuit initialization. An initialization sequence is intended to take the circuit, starting from any configuration that it could assume at power-up, to a configuration where all flops are initialized. The work in [17] analyzes the trade-off between two dimensions, i.e., the length of the initialization sequence, and the number of flops initialized after the execution.

There is a key difference with our work: in [17] it is sufficient to find a suitable trace to have a valid parameter valuation (i.e., that satisfies the property), even though it may not be optimal with respect to costs. In this sense, the parameters are *existential* with respect to the traces of the transition system being analyzed. Thus, the framework of Bounded Model Checking can be directly used to find candidate valuations. In our work, however, the parameters are *universal* with respect to the traces: in order to prove the validity of a candidate parameters valuation, a full model checking problem needs to be solved. As a consequence, it is not possible to leverage the "trace finding" mechanism of BMC to generate valid candidate valuations. Other differences are in the fact that in [17] the problem does not fall within the hypothesis of monotonicity, and that our algorithms rely on an extension of with IC3, whereas [17] is based on a complete version of Bounded Model Checking.

b) Combinational Pareto front: Other works on the construction of the Pareto front in a formal setting are [18], [7]. The key difference between these works and the one presented here is in the fact that the problem is combinational (e.g., satisfiability) in nature, while we deal with a sequential problem, i.e., invariant checking for a parameterized transition system. The work in [18] tackles the computation of the Pareto front with respect to cost functions imposed on a combinatorial SMT problem. The work in [7] tackles the problem of Pareto-optimal solutions for the problem of dynamic memory allocation.

c) Real-values parameter synthesis: The work in [8] deals with the synthesis of parameters over continuous ranges, to identify the space of valuations that result in a schedulable set of tasks. The method is based on complement, i.e., the set of bad valuations is computed first, and then complemented. The work in [14] relies on IC3 to solve the same problem more generally and more efficiently. Other works [19], [20], [21], [22], [23] synthesize parameters for real-time and hybrid systems. The key difference with respect to the problem tackled here is that no cost functions are considered, i.e., the space of *all* valuations is considered.

d) Synthesis of Fault trees: The work in [13] proposes several approaches to automatically generate Fault Trees for parameterized transition systems. This can be seen as a sequential problem of discrete parameter synthesis under the hypothesis of monotonicity. The key difference is that in [13] costs are not taken into account - *all* parameter valuations, each representing fault combinations in which a property is falsified, are found. We also remark that the parameters are existential, i.e., a valuation is deemed valid by the existence of a trace.

e) Synthesis of Observability Requirements: Identifying sufficient sets of sensor that guarantee the diagnosability of properties of interest is tackled in [9] and in [10]. Optimizations are found with respect to a single cost function, so there is no notion of Pareto front. The work in [9] proposes an explicit-state exploration of the space of costs, to synthesize a minimal configuration that is a global minimum. Domain specific techniques for the analysis of the sensor placement problem are also discussed.

III. Problem Description

We consider transition systems of the form $S = (X, I, T)$, where X is the set of state variables, $I(X)$ is the initial condition, and $T(X, X')$ is the transition relation.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

We generalize transition systems to parametric transition system $S = (U, X, I, T)$, where U is the set of parameters, X is the set of state variables, $I(U, X)$ is the initial condition, and $T(U, X, X')$ is the transition relation. Intuitively, a parameter can be seen as a variable that does not change over time.

We assume that the parameters are Boolean, and valuations are in $\Gamma = \mathbb{B}^{|U|}$. The order relation $<$ over $\mathbb{B}$ induces a partial order $\preceq$ over the parameter valuations Γ.

A valuation to the parameters, γ, provides us with a transition system $S_\gamma = (X, I(\gamma, X), T(\gamma, X, X'))$, in which each parameter is replaced with the value assigned in γ.

We assume the usual symbolic representation: a state is represented as an assignment to the X variables, while a parameter valuation γ is an assignment to the U variables. Sets of states are expressed as formulae in X; sets of parameter valuations are expressed as formulae in U, with each satisfying assignment corresponding to a parameter valuation. Boolean connectives have the usual set theoretical connotation (e.g., complementation is represented by logical negation, and intersection by conjunctions), while projection is represented by existential quantification.

We write $\textsc{Reachable}_S(U, X)$ for the set of reachable states in S, where a state is a valuation to the variables X and the parameters U. We notice that $\textsc{Reachable}_S(X) \wedge \gamma = \textsc{Reachable}_{S_\gamma}(X)$, i.e., the reachable state space of a parameterized system S can be seen as an association between an parameter valuation γ and the set of reachable states in the corresponding (non-parameterized) transition system S_γ.

The techniques described in this paper apply both to finite-state and to infinite-state systems. In the case of finite-state systems, termination is guaranteed; in the infinite case, convergence depends on the termination of the calls to the underlying model checking engine.

Let a property $\varphi(X)$ (φ for short) be a formula describing a set of states. A transition system S satisfies φ ($S \models \varphi$) iff $\textsc{Reachable}_S(X) \subseteq \varphi(X)$. The set of parameter valuations valid for φ for a parametric transition system S is defined as $\textsc{ValidPars}_{S,\varphi}(U) = \{\gamma \in \Gamma \mid S_\gamma \models \varphi\}$. A valid parameter valuation γ guarantees that the property φ holds in S_γ.

We consider cost functions $\textsc{Cost} : \Gamma \rightarrow \mathbb{N}$ as integer-valued functions over parameter valuations. A multi-dimensional cost function is defined as a vector of cost functions; for brevity we write $\textsc{Cost} : \Gamma \rightarrow \mathbb{N}^m$. We call $\mathbb{N}^m$ the space of costs. Notice that a cost function can be symbolically represented as a term. Given two cost vectors $(v_1, \cdots, v_m)$ and $(w_1, \cdots, w_m)$, we define the partial order relation $\preceq$ as $(v_1, \cdots, v_m) \preceq (w_1, \cdots, w_m)$ iff $\forall i.\ v_i \leq w_i$.

Given S, φ and $\textsc{Cost}$, we say that an assignment $\gamma \in \Gamma$ is Pareto-Optimal iff:

1) $S_\gamma \models \varphi$, and
2) if there is γ' s.t. $S_{\gamma'} \models \varphi$ and $\textsc{Cost}(\gamma') \preceq \textsc{Cost}(\gamma)$ then $\gamma = \gamma'$.

Pareto-optimality boils down to optimality with respect to a single cost function when $m = 1$. The cost function can be represented symbolically as a term $\textsc{Cost}(U)$; a set of assignments is then simply identified by a formula $\textsc{Cost}(U) \bowtie v$ where v is a natural number and $\bowtie$ is a relation operator.

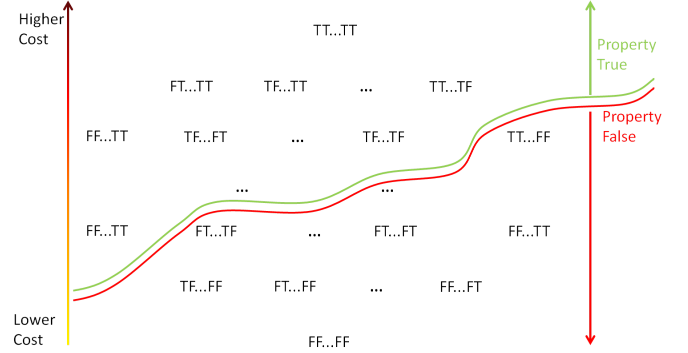


Fig. 1. Monotonicity with respect to Property and Cost function

In this paper, we make two assumptions of *monotonicity*. The first one is monotonicity of the "property holds" relation, and the second is monotonicity of the cost function.

We say that $S \models \varphi$ is monotonic w.r.t. Γ iff

$$\forall \gamma,\ \text{If } S_\gamma \not\models \varphi \text{ then } \forall \gamma'.\ \gamma' \preceq \gamma \Rightarrow S_{\gamma'} \not\models \varphi$$

Intuitively, if the property does not hold under a given parameter valuation, then it does not hold for any of its predecessors. Conversely, if the property holds under a given valuation, then it also holds for all the successors.

We say that $\textsc{Cost}$ is monotonic w.r.t. Γ iff

$$\forall \gamma, \gamma'.\ \text{ If } \gamma \preceq \gamma' \text{ then } \textsc{Cost}(\gamma) \preceq \textsc{Cost}(\gamma')$$

The Pareto-Frontier $PF(\textsc{Cost}, \varphi) \subseteq \Gamma$ is the set of parameter assignments that are valid for φ and that are Pareto-optimal with respect to $\textsc{Cost}$.

The space explored in this paper is depicted in Figure 1. Above the line are the valuations that are valid for φ; below the line are the ones for which the property does not hold, i.e., the ones which are associated to at least one counterexample trace. The vertical arrow on the left denotes a cost function that is upwards monotonic along each path.

IV. Solution Description

In the following we present several algorithms for the computation of the Pareto frontier, for a given S, φ, and $\textsc{Cost}$. We assume that both the property satisfaction relation ($S \models \varphi$) and $\textsc{Cost}$ are monotonic with respect to Γ. For the sake of presentation, we assume also that there is at least one parameter valuation γ s.t. $S_\gamma \models \varphi$.

The algorithms that we present define a spectrum based on how much of the set of $\textsc{ValidPars}$ we compute up-front. In Section IV-A, we compute the whole set of good valuations up-front, and then proceed to the computation of the Pareto-Frontier. In Section IV-B, we "slice" the space $\textsc{ValidPars}$ by one dimension and compute one of the slices at the time; once a slice has been computed, we minimize w.r.t. to the other costs. By doing so, we are able to skip slices (using the monotonicity assumption), so that we end up computing a subset of $\textsc{ValidPars}$. Finally, in Section IV-C we do not compute $\textsc{ValidPars}$ directly, but navigate through

the valuations lattice driven by the cost functions and test on-the-fly membership of points to ValidPars.

For the sake of presentation, we describe most algorithms assuming that we have a two-dimensional cost function.

A. The valuations-first approach

```
function ValuationsFirst(S, Cost, φ)
    VP := ValidPars(S, φ)
    return ParetoFront(Cost, VP)
end function
```

Fig. 2. Valuations First pseudo-code

```
function ValidPars(S, φ)
    Bad := ⊥
    S = (U, X, I, T)
    while S ⊭ φ do
        γ' := project counter-example on U
        Bad := Bad ∨ γ'
        I := I ∧ ¬Bad
    end while
    return ¬Bad
end function
```

Fig. 3. ValidPars computation

The first algorithm we present is an eager, two-stage approach. Figure 2 provides a high-level description of the algorithm.

The first stage constructs the set of parameter valuations that are valid for the φ property. This gives a closed-form representation of the solution space, regardless of cost considerations, represented as a formula ValidPars. The second phase carries out an analysis of the solution space taking the costs into account, selecting the assignments that are Pareto-optimal, thus building the Pareto front.

Each of the phases can be in turn refined. The computation of ValidPars can be carried out directly, by performing a reachability analysis on S, thus obtaining $\textsc{Reachable}(U, X)$, and then considering only the valuations for which the states always satisfy the property. This idea has been explored with a BDD-based implementation in [13], where it was applied in the computation of Fault-Trees. In many cases, however, the computation of the reachable states can be over-killing. In Figure 3 an alternative approach is presented, based on the algorithm proposed in [14], that constructs the set $\textsc{ValidPars} = \{\gamma_i \mid S \models \gamma_i \rightarrow \varphi\}$ of valid parameter valuations. The idea is to rely on a model-checking routine to compute the set $InvalidPars = \{\gamma_i \mid S \not\models \gamma_i \rightarrow \varphi\}$, i.e., a representation of the "lower part" of the lattice in Figure 1. At a very high level, this is done by enumerating counterexample traces to the negation of φ. Each trace is associated with an invalid parameter valuation, which is then accumulated in the result, and removed from the initial states, thus preventing the model checker from re-discovering it. Once $InvalidPars$ is computed, the space of valid parameter valuations is simply obtained by complement. This algorithm can thus rely on a model-checker as a black-box, therefore leveraging recent advancements in SAT-based model-checking techniques (e.g., IC3).

The second phase carries out the optimization of the combinatorial space defined by ValidPars with respect to Cost. This can be done, for example, by enumerating all the elements in ValidPars and comparing the associated costs, or by considering the symbolical characterization of the Pareto front:

$$\textsc{ParetoFront}(U) = VP(U) \wedge \nexists U'.((U' \prec_{\textsc{Cost}} U) \wedge VP(U'))$$

A simple way of computing $\textsc{ParetoFront}(U)$ is given by the possibility of encoding the cost functions into SAT (e.g., using SMT over bit-vectors), and applying an iterative approach that tightens the constraints on the cost along each dimension.

There are two big disadvantages in the valuations-first algorithm. First, in order to compute ValidPars, we need to enumerate all the elements of $InvalidPars$. This means that the first phase may be in some cases inherently expensive. Second, the first phase proceeds by under-approximating the complement of the valid space, regardless of the cost information. This means that virtually no information on what is a valid (let alone optimal) solution is found until convergence in the accumulation has been reached, i.e., until the whole $InvalidPars$ set has been computed.

B. The one-cost slicing approach

```
function Slicing(S, Cost, φ)
    PF := ∅
    γ = ⊤;
    c1 := Cost1(γ)
    S' := FixCost(S, Cost1 = c1)
    VP_Cost1 := ValidPars(S', φ)
    while VP_Cost1 ≠ ∅ do
        (γ, c2) = Minimize (Cost2, VP_Cost1)
        (γ, c1) := Reduce_Cost1(S, γ, φ, c2)
        PF.add(γ, c1, c2)
        c1 := c1 − 1
        S' := FixCost(S, Cost1 = c1)
        VP_Cost1 := ValidPars(S', φ)
    end while
    return PF
end function

function FixCost(S, CostExpr))
    S = (U, X, I, T)
    S' := (U, X, I ∧ CostExpr, T) return S'
end function
```

Fig. 4. Slicing algorithm

The second algorithm (Figure 4) we propose interleaves the analysis of the cost information with checks on the validity of the parameters. This is done by slicing the space of valid parameters along the different dimensions (i.e., cost functions).

We first initialize c_1 to the highest possible value, and the Pareto frontier to the empty set. We iterate as follows. First, we compute all the candidate solutions on the parametric system S' in which we fixed the cost $\textsc{Cost}_1$ to the value c_1. We then search in the set of candidates ($VP_{\textsc{Cost}_1}$) for the best valuation and cost for $\textsc{Cost}_2$. Once an optimal cost c_2 has been found, we fix it and try to find a smaller valuation w.r.t. $\textsc{Cost}_1$, and add the solution to the Pareto front. This is done

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

by calling a function $\text{REDUCE}_{\text{COST}_1}$ which, given a solution γ of cost (c_1, c_2), returns another solution γ' of cost (c'_1, c_2) with $c'_1 \leq c_1$. For now, REDUCE is simply a function that tries to improve a candidate solution γ. We shall describe its actual implementation in the next Section.

In order to find the other points of the Pareto frontier, we decrease c_1 and test whether any solution (independently of COST_2) exists. If it does, we repeat the process, otherwise we terminate.

Note that in the function MINIMIZE we operate on the set of the solutions, while in REDUCE, we generate a candidate $\gamma' \preceq \gamma$ and test whether it is still a solution (i.e. $S_{\gamma'} \models \varphi$). Due to the monotonicity assumption, REDUCE cannot skip solutions. However, REDUCE can drastically accelerate the search by avoiding the enumeration of all costs in c_1.

In the pseudo-code, the addition of solutions to the Pareto front is slightly simplified. In practice, we cannot add a solution γ_1 immediately in the Pareto front, but we need to wait for the next solution γ_2 to be added ($PF.add$). If the costs of γ_1 and γ_2 are incomparable, then γ_1 is Pareto-optimal and gets added to the frontier. If γ_2 is smaller than γ_1, then γ_1 is not optimal and is discarded. This pair-wise operation guarantees that only Pareto optimal solutions are considered.

This approach only computes slices of VALIDPARS when needed. Although in the worst-case we can end-up computing it all by slices, when calling the REDUCE function, it is generally possible to accelerate the search and skip intermediate slices.

C. The costs-first approach

```
function CostsFirst(S, Cost, φ)
    PF := ∅
    γ := ⊤;
    c1 = Cost1(γ); c̄2 = Cost2(γ)
    repeat
        c2 = c̄2
        for γi ∈ MaxSmallerCandidate_Cost2(c1, c2) do
            if S_γi ⊨ φ then
                (γ, c2) := Reduce_Cost2(S, γ, φ, c1)
            end if
        end for
        (γ, c1) := Reduce_Cost1(S, γ, φ, c2)
        PF.add(γ, c1, c2)
        c1 := c1 − 1
    until No solution exists for FixCost(S, Cost1 = c1)
    return PF
end function
```

Fig. 5. CostsFirst pseudo-code

One of the key insights of the slicing algorithm is that big parts of VALIDPARS might not be necessary in order to compute the Pareto front. In the costs-first approach we take this idea to the extreme: we do no compute VALIDPARS anymore. Instead, we explore the lattice of valuations induced by the cost functions. Every time we find a promising valuation γ, we test whether it is actually a solution (i.e., $S_\gamma \models \varphi$). Due to the monotonicity assumption, whenever we find a valuation that is not a solution, we can prune all of its predecessors in the lattice (since they cannot be solutions either).

An overview of the algorithm is provided in Figure 5. We start by getting an upper-bound on both costs by considering the cost of the top valuation. In the outer-loop we decrease the value of COST_1, similarly to the slicing approach. Within the inner-loop, however, we proceed by enumerating the solutions that have smaller value w.r.t. COST_2. In particular, MAXSMALLERCANDIDATE returns the maximal solution(s) with the same cost c_1 but with smaller c_2.

The process terminates whenever no solution can be found for a given value of COST_1. Note how the structure of this algorithm is similar to the one of the slicing approach. The main difference is that we never need to compute VALIDPARS.

This algorithm allows us to find subsets of the Pareto front, since it can be interrupted at any point and is guaranteed to provide an under-approximation of the Pareto-Frontier. This is in contrast with the approaches described in Section II for parameter synthesis, that require termination of the procedure in order to provide the solution space of the parameters.

V. IC3-BASED IMPLEMENTATION

We implemented the approaches described in the previous section using IC3-based techniques.

In particular, there are two key ideas that we can leverage in order to have an efficient algorithm using IC3. First, we notice that $S_\gamma \models \varphi$ holds iff $S \models \gamma \rightarrow \varphi$. This observation makes it possible to reason always on the same system, and moves the choice of the valuations within the property. This leads us to the second fundamental observation. If $S \models \gamma \rightarrow \varphi$, we can extract from the IC3 model-checker the inductive invariant ψ. By definition of inductive invariant we know that $\psi \models \gamma \rightarrow \varphi$; moreover, it might be the case that we can reuse the same invariant to check whether another valuation γ' is a solution: i.e., $\psi \models \gamma' \rightarrow \varphi$. We will use this idea when trying to reduce the valuation, since this makes it possible to reason locally on a (relatively small) formula, and does not require unrolling or computing reachable states. The efficiency of the procedure will then largely depend on how well the reduction step works.

Figure 6 presents the adaptation of the costs-first algorithm when using the inductive invariant to perform the REDUCE step. The same idea for the REDUCE can be applied in the slicing algorithm.

We navigate the lattice by picking the maximal candidate(s) of smaller cost w.r.t. COST_2 (MAXSMALLERCANDIDATE). This fact guarantees that the algorithm will terminate, since we are always picking a solution of smaller dimension. We then check that the property still holds for the new valuation γ_i, by using IC3. If this is the case, we are provided with an inductive invariant ψ, s.t., $\psi \models \gamma_i \rightarrow \varphi$.

The operation of picking a cost-predecessor could be, in principle, delegated to a pseudo-boolean constraint solver, or to other reasoning engines that are able to deal with costs natively. For our simple implementation, we use an SMT solver with the theory of bit-vectors.

When considering the parameters as a set of elements, we can try to minimize the set by implementing the REDUCE procedure using unsat-cores. Namely, we check the unsatisfiability of $\psi \wedge \neg\varphi$ under the assumption of γ_i and use standard features

```
function CostsFirstIC3(S, Cost, φ)
    PF := ∅
    γ := ⊤;
    c1 = Cost1(γ); c̄2 = Cost2(γ)
    repeat
        c2 := c̄2
        for γi ∈ MaxSmallerCandidate_Cost2(c1, c2) do
            (res, ψ) := IC3(S, γi → φ)
            if res == Safe then
                # ψ is an inductive invariant s.t. ψ |= γi → φ
                (γi, c1, c2) := Reduce_Cost2(ψ, γi, φ)
            end if
        end for
        (γi, c1, c2) := Reduce_Cost1(ψ, γi, φ)
        PF.add(γ, c1, c2)
        c1 := c1 − 1
    until No solution exists for FixCost(S, Cost1 = c1)
    return PF
end function
```

Fig. 6. IC3-based CostsFirst pseudo-code

from modern SAT solvers to minimize the unsat-core that, in turn, translates in picking a subset of the parameters that makes the formula unsatisfiable. By doing so, we are able to "jump" and quickly reduce the valuation γ. For integer parameters, instead, we use a REDUCE procedure that performs a linear or binary search, using the inductive invariant.

In general, we could use the identity function as REDUCE, and this would still guarantee the correctness and termination of the algorithm. However, this would end-up requiring an explicit state search of the lattice. Having a smart REDUCE procedure makes it possible to jump and terminate faster.

Since the inductive invariant does not depend on the costs, it is possible to reuse the invariant from previous calls in an incremental way. Intuitively, this provides us with stronger invariants that are more likely to allow us to reduce the parameters aggressively.

VI. MOTIVATING APPLICATION DOMAINS

We describe now two motivating application domains: sensor placement for diagnosability and product line engineering.

Sensor Placement: The problem is of practical relevance, and substantial interest has been devoted to it in the setting of autonomous systems. Typical architectures integrate components for Fault Detection and Identification (FDI) that are used to detect, during operation, whether some (and which) faults may have occurred [24], [10]. The information produced by FDI is then used for Fault Isolation and Recovery, i.e., to respond to the effects of faults, e.g., by reconfiguration.

Intuitively, the problem is to identify a suitable set of sensors that will allow the FDI subsystem to have enough information to carry out, within a given delay, its diagnosis task. In this setting, a parameter represents whether a sensor is present in the design, and a parameter valuation identifies a subset of all available sensors. There is a trade off between the observation power of the available sensors, and the delay required to diagnose a certain condition of interest. Intuitively, a reduction in the set of sensors may lead to an increase in the delay.

The property φ that we want to show is *diagnosability* with respect to a given delay d, i.e., the ability to detect an event of interest within at most d steps. In the case of a given set of sensors, this is reduced to the model checking problem on the so-called *twin plant*, a construction based on the composition of two replicas of the plant under observation [10]. The twin plant encodes the existence of a critical pair, i.e., two indistinguishable traces satisfying a pair of conditions of interest (e.g., the occurrence of two faults of different type that must be identified).

The problem is generalized by considering a parameter set U, defined as $\{s_1, \ldots, s_k, d\}$, where a valuation to the vector $(s_1, \ldots, s_k)$ of k sensor parameters identifies a configuration in the design space. The delay d is an integer valued parameter. The space of assignments is then $\mathbb{B}^k \times \mathbb{N}$

The diagnosability property φ is defined as the invariant property:

$$\neg((delay_\psi \geq d) \wedge obs_{eq})$$

where $delay_\psi$ counts the steps since the occurrence of the condition of interest ψ. This formula is satisfied in the twin-plant iff there is no critical pair. In general, we assume that an upper-bound on the delay for the model is known. In addition to the theoretical bound that always exists for state transition systems, in practice there usually is an application specific time-frame after which the diagnosis is not useful anymore (e.g., mission life-span, propagation time). Several interesting COST functions are possible. For the sensor placement problem we use one based on weights/delay pairs:

$$\textsc{Cost}_{weighted}(s_1, \ldots, s_k, d) = (\sum_i w_i s_i,\ d)$$

Product Lines: When designing a product, engineers are often faced with a high degree of variability in terms of possible features. Such variability is usually captured in product line models (sometimes referred to as feature models). For instance, in [16] the authors model variability in a controller design, and the authors of [6] consider software product lines. Here we are specifically interested in the analysis of dynamic systems as opposed to static contexts which are usually addressed with constraint programming techniques.

The goal of product line engineering is usually to identify which combinations of features satisfy a certain property. Here however we specifically address the Pareto-optimal trade-off problem. In various works there are different assumptions on the monotonicity of features, that is whether by adding features the possible behaviors increase monotonically or whether some behaviors can be overridden. In our work we only assume the monotonicity of the property of interest in terms of feature additions.

VII. EXPERIMENTAL EVALUATION

A. Experimental Setup

We have implemented the algorithms described above on top of the NUXMV model checker [25]. Although our framework can in principle support any number of cost functions, our current implementation only supports two of them. The executables and benchmark instances used in the evaluation

are available at https://es-static.fbk.eu/people/griggio/papers/fmcad14pareto.tar.bz2.

We evaluate our approach on a series of benchmarks coming from the domains of sensor placement [10] and product lines [16]. The sensor placement examples were obtained from realistic case studies on a simpler problem, i.e. finding a good set of sensors for a fixed delay. These simpler benchmarks were challenging and in some case not solvable for our previous techniques [10]. The properties for the product lines benchmarks that were derived for our invariant checking framework are artificial but tailored specifically for these examples. For both cases we are unfortunately not aware of other publicly available industrial benchmarks.

ORBITER, ROVERSMALL, and ROVERBIG are models of an orbiter and of a planetary rover, both developed in the OMCARE project (see [26], [27]). The models describe the functional level, with various relevant subsystems including failure modes. CASSINI models the propulsion system of the Cassini spacecraft (see [13]). It is composed of two engines fed by redundant propellant/gas circuit lines, which contain several valves and pyro-valves. Leakage failures are attached to all components. ELEVATOR models an elevator controller, parameterized by the number of floors. The modeled properties are cabin and door movement, request and reset operations at each floor, and the controller logic. C432 is a boolean circuit used as a benchmark in the DX Competition [28], whose gates can permanently fail (inverted output). The observables are the inputs and output values for the gates of the circuit. X34 is a benchmark describing a simplified version of the main propulsion system of a spacecraft [29]. All models also contain faults based on which a sensor placement problem is formulated. PRODUCT LINES are benchmark instances derived from [16], describing a railway switch controller. The product-line features correspond to possible communication strategies used by the controller. We explore a design trade-off along two dimensions. The first is the upper bound on message sequence lengths. The second one is a cost associated to dropping a particular feature, specified by a random cost function. Our aim is to minimize both the message sequence bound and the cost of removing features in order to guarantee it.

For each example, we generated multiple benchmarks by varying both the number of parameters considered and the (randomly-generated) costs of the individual parameters. Overall, our benchmark set consists of 81 instances. The number of Pareto-optimal solutions varies between 0 and 5.

B. Results

Family	#Instances	valuations-first	one-cost slicing	costs-first
c432	32	11	13	**32**
cassini	21	6	12	**21**
elevator	4	4	4	4
orbiter	4	4	4	4
roversmall	4	4	4	4
roverbig	4	4	4	4
x34	4	4	4	4
product lines	8	6	4	**8**
TOTAL	81	43	49	**81**

Fig. 7. Number of solved instances by each approach

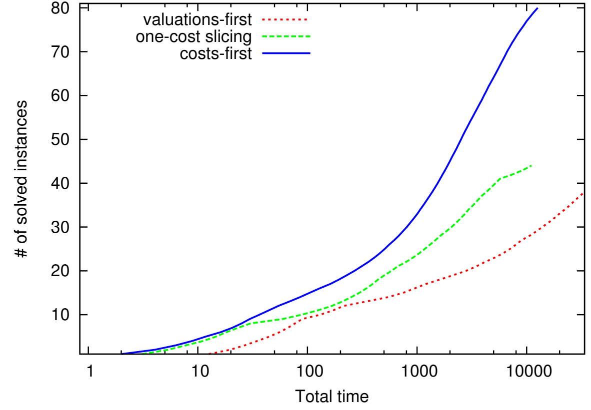


Fig. 8. Accumulated-time plot showing the number of solved instances (x-axis) in a given total time (y-axis) for the various algorithms.

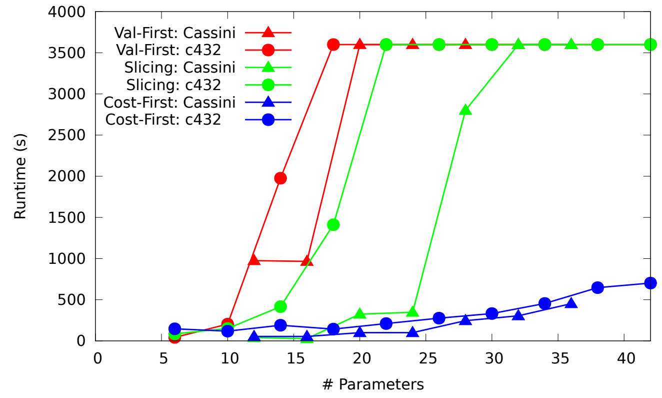


Fig. 9. Runtime for different number of parameters

We executed the experiments on a Linux cluster equipped with 2.5Ghz Intel Xeon CPUs with 96Gb of RAM. We used a time limit of 1 hour and a memory limit of 6Gb.

In Figure 7 we present the number of instances solved for each problem family. For the C432, CASSINI and PRODUCT LINES benchmarks, we can see how the costs-first approach finds all the solutions within the timeout, whereas the other two approaches fail on several instances. Figure 8 shows the accumulated-time plots for the different algorithms, plotting the number of solved instances (y-axis) in the given total amount of time (x-axis) in logarithmic scale.

For the C432 and CASSINI benchmark, we show in Figure 9 the runtime as a function of the parameters. As expected, on the same model, the number of parameters has a big impact on the runtime. Indeed, for the valuations-first and the one-cost slicing approaches this has an exponential tendency.

Finally, in order to evaluate the impact of the REDUCE procedure in the costs-first algorithm, we performed an experiment in which we ran costs-first without applying REDUCE. From the scatter plot of Figure 10, we can see that REDUCE is crucial for performance: without it, costs-first solves only 48 instances, and the runtimes increase by orders of magnitude.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

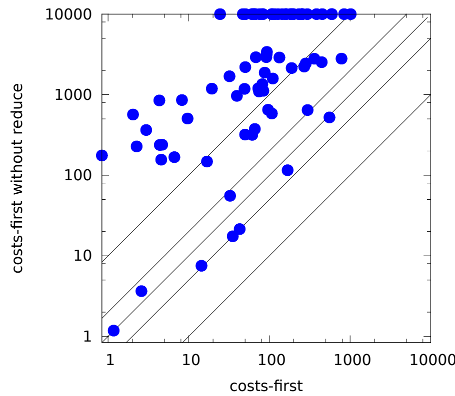


Fig. 10. Impact of REDUCE in the costs-first algorithm.

VIII. CONCLUSIONS

In this paper we have proposed a new method for the synthesis of optimal parameter valuations for multi-dimensional monotonic cost functions, enabling the construction of the Pareto front with respect to the cost function.

We analyzed three algorithms of increasing efficiency, that interleave in various ways the search for parameter valuations that satisfy the property and the optimization with respect to costs, and we showed how to implement them on top of IC3, exploiting its features for efficiency.

Our experimental evaluation shows the feasibility of the approach on benchmarks from important practical domains, and demonstrates the importance of a tight integration between model checking and cost optimization.

In the future we will generalize the approach to deal with real-valued parameters; in particular, we will investigate the generalization of the notion of monotonicity. From a practical point of view, it would be important to find an effective way of automatically testing the monotonicity assumptions. We will also generalize our implementation to support more than two cost functions, and devise strategies to handle multiple cost functions in an effective way. Further interesting directions are the investigation of specialized techniques for specific patterns of properties (e.g., response time), thus enabling the approach to be applied beyond safety properties, and techniques for relaxing the assumptions of monotonicity currently required.

REFERENCES

[1] P. Manolios, D. Vroon, and G. Subramanian, "Automating component-based system assembly," in *ISSTA*, 2007, pp. 61–72.

[2] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing cyber-physical architectural models with real-time constraints," in *CAV*, 2011, pp. 441–456.

[3] W. Steiner and B. Dutertre, "Layered diagnosis and clock-rate correction for the ttethernet clock synchronization protocol," in *PRDC*, 2011, pp. 244–253.

[4] S. Samii, P. Eles, Z. Peng, and A. Cervin, "Design optimization and synthesis of flexray parameters for embedded control applications," in *DELTA*, 2011, pp. 66–71.

[5] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty, "Constraint-driven synthesis and tool-support for flexray-based automotive control systems," in *CODES+ISSS*, 2011, pp. 139–148.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE*, 2011, pp. 321–330.

[7] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, D. Soudris, and J. Mendias, "Automated exploration of pareto-optimal configurations in parameterized dynamic memory allocation for embedded systems," in *DATE*, 2006, pp. 874–875.

[8] A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic computation of schedulability regions using parametric timed automata," in *RTSS*. IEEE Computer Society, 2008.

[9] A. Grastien, "Symbolic testing of diagnosability," in *Twentieth International Workshop on Principles of Diagnosis (DX-09)*, 2009.

[10] B. Bittner, M. Bozzano, A. Cimatti, and X. Olive, "Symbolic Synthesis of Observability Requirements for Diagnosability," in *AAAI*, 2012.

[11] V. Pareto, *Manuale di economia politica*, ser. Collezione saggi & documenti. Edizioni Studio Tesi, 1994.

[12] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. M. III, and J. Railsback, "Fault tree handbook with aerospace applications," *Technical report, NASA*, 2002.

[13] M. Bozzano, A. Cimatti, and F. Tapparo, "Symbolic fault tree analysis for reactive systems," in *ATVA*. Springer, 2007, pp. 162–176.

[14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with ic3," in *FMCAD*. IEEE, 2013, pp. 165–168.

[15] A. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

[16] J. Greenyer, A. Sharifloo, M. Cordy, and P. Heymans, "Efficient consistency checking of scenario-based product-line specifications," in *Requirements Engineering Conference (RE)*, 2012, pp. 161–170.

[17] S. Reimer, M. Sauer, T. Schubert, and B. Becker, "Using maxbmc for pareto-optimal circuit initialization," in *DATE*, 2014, pp. 1–6.

[18] J. Legriel, C. L. Guernic, S. Cotton, and O. Maler, "Approximating the pareto front of multi-criteria optimization problems," in *TACAS*, 2010, pp. 69–83.

[19] T. A. Henzinger and P.-H. Ho, "Hytech: The cornell hybrid technology tool," in *Hybrid Systems*, 1994, pp. 265–293.

[20] F. Wang, "Symbolic parametric safety analysis of linear hybrid systems with bdd-like data-structures," *IEEE Trans. Soft. Eng.*, vol. 31, no. 1, pp. 38–51, 2005.

[21] G.Frehse, S. Jha, and B. Krogh, "A counterexample-guided approach to parameter synthesis for linear hybrid automata," in *HSCC*, 2008, pp. 187–200.

[22] É. André, L. Fribourg, U. Kühne, and R. Soulat, "IMITATOR 2.5: A tool for analyzing robustness in scheduling problems," in *FM*, 2012, pp. 33–36.

[23] É. André and U. Kühne, "Parametric analysis of hybrid systems using HyMITATOR," in *iFM*, 2012, pp. 16–19.

[24] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta, "Formal design of fault detection and identification components using temporal epistemic logic," in *TACAS*. Springer, 2014, pp. 326–340.

[25] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NUXMV symbolic model checker," in *CAV*, ser. LNCS. Springer, 2014.

[26] M. Bozzano, A. Cimatti, A. Guiotto, A. Martelli, M. Roveri, A. Tchaltsev, and Y. Yushtein, "On-board autonomy via symbolic model-based reasoning," in *Proceedings of the 10th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, 2008.

[27] M. Bozzano, A. Cimatti, M. Roveri, and A. Tchaltsev, "A comprehensive approach to on-board autonomy verification and validation," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[28] A. Feldman, T. Kurtoglu, S. Narasimhan, S. Poll, D. Garcia, J. de Kleer, L. Kuhn, and A. van Gemund, "Empirical evaluation of diagnostic algorithm performance using a generic framework," *International Journal of Prognostics and Health Management*, Sep 2010.

[29] A. Cimatti, C. Pecheur, and R. Cavada, "Formal verification of diagnosability via symbolic model checking," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

SAT-Based Methods for Circuit Synthesis

Roderick Bloem[1], Uwe Egly[2], Patrick Klampfl[1], Robert Könighofer[1], and Florian Lonsing[2]

[1]Institute for Applied Information Processing and Communications, Graz University of Technology, Austria
[2]Knowledge-Based Systems Group, Institute of Information Systems, Vienna University of Technology, Austria

***Abstract*—Reactive synthesis supports designers by automatically constructing correct hardware from declarative specifications. Synthesis algorithms usually compute a strategy, and then construct a circuit that implements it. In this work, we study SAT- and QBF-based methods for the second step, i.e., computing circuits from strategies. This includes methods based on QBF-certification, interpolation, and computational learning. We present optimizations, efficient implementations, and experimental results for synthesis from safety specifications, where we outperform BDDs both regarding execution time and circuit size.**

I. Introduction

Synthesis is an ambitious design approach: Instead of checking whether an already constructed system satisfies its specification, a correct implementation is derived *automatically* from the specification [3]. Synthesis is also used in rapid prototyping, automatic repair [9], and program sketching [14].

Existing work often focuses on finding strategies to satisfy the specification, or only on deciding realizability. However, computing circuits from strategies is computationally demanding as well. System quality (e.g., circuit size and depth) imposes additional challenges. Synthesized strategies usually allow for much implementation freedom, which needs to be exploited cleverly. Algorithms must also be symbolic (operate on formulas rather than enumerating states) to achieve scalability. These symbolic algorithms are usually implemented with BDDs because they offer existential *and* universal quantification. Recently, SAT-based synthesis algorithms have been proposed [12], [4] as alternative to BDDs and their scalability issues. However, these works do not address circuit extraction.

We thus present and compare several SAT- and QBF-based circuit synthesis algorithms. The basic algorithms are not new, but we present novel optimizations, combinations, efficient implementations for safety synthesis problems, and extensive experiments. This includes methods based on QBF-certification, computational learning (including the first application of incremental QBF solving in synthesis), and interpolation. We achieve the best results by combining ideas from interpolation [8] with learning [7], thereby outperforming BDDs both regarding computation time and circuit size.

Related work. It is argued [7] that many circuit synthesis methods are still outperformed by the simple BDD-based cofactor approach [3]. The same work [7] also proposes learning-based techniques, which are implemented with BDDs. This yields smaller circuits, but is slower. We show how learning can be efficiently realized with SAT- and QBF-solvers, and that this realization can outperform the cofactor approach both regarding circuit size and computation time. SAT-based learning is also used in [4]. However, this work only addresses strategy computation and not circuit synthesis. Jiang et al. [8] propose interpolation for circuit extraction, and show how quantifier alternations can be avoided by temporarily treating outputs as inputs. We combine this idea with learning to compute interpolants, thereby achieving a speedup of several orders of magnitude. QBF certification [13] can derive circuits from a completeness proof of the strategy formula. We show how this method can be applied efficiently for safety synthesis.

This work was supported in part by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23, S11409-N23) and the project QUAINT (I774-N23), as well as by the European Commission through project STANCE (317753).

II. Preliminaries

We assume familiarity with propositional logic, SAT- and QBF-solving (cf. [1]) but repeat the most important concepts.

Basic Notation. A *literal* is a Boolean variable or its negation. A *clause* (*cube*) is a disjunction (conjunction) of literals, and a *Conjunctive Normal Form (CNF)* formula is a conjunction of clauses. We denote variables vectors with overlines, corresponding cubes in bold, and propositional formulas with capital letters. E.g., $\mathbf{x}$ is a cube over the variable vector $\overline{x} = (x_1, \ldots, x_n)$, and $F(\overline{x})$ is a formula over $\overline{x}$. If the variables are irrelevant, we simply write F instead of $F(\overline{x})$.

Decision Procedures. A *SAT-solver* checks if a CNF is satisfiable. We write $(\mathsf{sat}, \mathbf{x}) := \textsc{PSat}(F(\overline{x}))$ for a SAT-solver call, where sat is assigned $\mathtt{true}$ iff the CNF F is satisfiable, and $\mathbf{x}$ is a satisfying assignment given as cube over $\overline{x}$. Let $\mathbf{x}$ be a cube. We write $\mathbf{y} := \textsc{PCore}(\mathbf{x}, F)$ to denote the extraction of an unsatisfiable core: Given that $\mathbf{x} \wedge F$ is unsatisfiable, $\mathbf{y}$ will be a sub-cube of $\mathbf{x}$ such that $\mathbf{y} \wedge F$ is still unsatisfiable. Let $A(\overline{x}, \overline{y})$ and $B(\overline{x}, \overline{z})$ be two CNFs such that $A \wedge B$ is unsatisfiable, and $\overline{y}$ and $\overline{z}$ are disjoint. An *interpolant* is a formula $I(\overline{x})$ such that $A \Rightarrow I \Rightarrow \neg B$. Interpolants can be computed from the unsatisfiability proof of $A \wedge B$ [6]. We denote this computation by $I := \textsc{Int}(A, B)$. A *Quantified Boolean Formula (QBF)* is a formula $Q_1\overline{x} \,.\, Q_2\overline{y} \ldots F(\overline{x}, \overline{y}, \ldots)$, where $Q_i \in \{\forall, \exists\}$ and F is a CNF. The quantifiers have their expected semantics. A *QBF-solver* checks if a QBF is satisfiable. We write $(\mathsf{sat}, \mathbf{a}) := \textsc{QSat}(\exists\overline{a} \,.\, Q_1\overline{x} \,.\, Q_2\overline{y} \ldots F(\overline{a}, \overline{x}, \overline{y}, \ldots))$ for QBF-solver calls. The satisfying assignment $\mathbf{a}$ can only be extracted for variables that are quantified existentially on the outermost level. Finally, we write $\mathbf{b} := \textsc{QCore}(\mathbf{a}, \exists\overline{a} \,.\, Q_1\overline{x} \,.\, Q_2\overline{y} \ldots F(\overline{a}, \overline{x}, \overline{y}, \ldots))$ to denote the extraction of an unsatisfiable core.

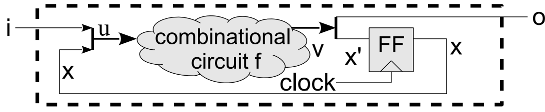


Fig. 1. Implementation of a strategy. (FF = flip-flops).

Circuit Synthesis. A *strategy* is a formula $S(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$ such that $\forall \overline{x}, \overline{i} \,.\, \exists \overline{o}, \overline{x}' \,.\, S$, where $\overline{x}, \overline{i}, \overline{o}$ are state-, input-, and output-bits, respectively, and $\overline{x}'$ is the next-state copy of $\overline{x}$. Intuitively, for a given state $\mathbf{x}$ and input $\mathbf{i}$, S defines allowed output-values $\mathbf{o}$ and next states $\mathbf{x}'$: $\mathbf{o}, \mathbf{x}'$ is *allowed* iff $\mathbf{x} \wedge \mathbf{i} \wedge \mathbf{o} \wedge \mathbf{x}'$ satisfies S. Let $\overline{u} = \overline{x} \cup \overline{i}$ and $\overline{v} = \overline{o} \cup \overline{x}'$. An *implementation* of $S(\overline{u}, \overline{v})$ is a function $f : 2^{|\overline{u}|} \rightarrow 2^{|\overline{v}|}$ such that $\forall \overline{u} \,.\, S(\overline{u}, f(\overline{u}))$. This function can be implemented in hardware as shown in Fig. 1.

Strategies for safety specifications are particularly simple: given a *winning region* $W(\overline{x})$ from which the specification can be enforced, and a complete[1] and deterministic[2] transition relation $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$ defining the state transitions, the strategy must pick values for $\overline{o}$ such that the next state is in W again, i.e., $S = \big(\neg W(\overline{x})\big) \vee \big(T(\overline{x}, \overline{i}, \overline{o}, \overline{x}') \wedge W(\overline{x}')\big)$. We only need to synthesize circuits for $\overline{o}$, and define $\overline{x}'$ using T.

III. Circuit Synthesis Algorithms

A. QBF-Certification

An implementation can be computed as Skolem functions[3] for the signals $\overline{o}$ and $\overline{x}'$ in the QBF $\forall \overline{x}, \overline{i} \,.\, \exists \overline{o}, \overline{x}' \,.\, S(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$. QBFCert [13] computes such functions using DepQBF [10].

Optimizations for Safety Specifications. We need to find Skolem functions for $\overline{o}$ in $\forall \overline{x}, \overline{i} \,.\, \exists \overline{o}, \overline{x}' \,.(\neg W) \vee (T \wedge W')$. Yet, we achieve better results with QBFCert by computing Herbrand functions[4] in the unsatisfiable QBF $\exists \overline{x}, \overline{i} \,.\, \forall \overline{o} \,.\, \exists \overline{x}' \,.\, W \wedge T \wedge \neg W'$. This works because T is deterministic and complete. In our setting, W is in CNF, so the conjunctions in the latter formulation are simpler to realize in CNF. Also, the clause resolution proofs required for unsatisfiable QBFs are usually less expensive than the cube resolution proofs for satisfiable ones. Still, the intermediate files produced by QBFCert can grow large (hundreds of GB). One reason is that a straightforward CNF encoding of $\neg W'$ requires many auxiliary variables and clauses. We could reduce the size of the files by up to a factor of 30 by learning a CNF representation of $\neg W'$ without introducing auxiliary variables using the following algorithm:

```
1: procedure NEGLEARN(W'),        returns: ¬W'
2:   N' := true
3:   while sat, with (sat, x) := PSAT(W' ∧ N') do
4:     N' := N' ∧ ¬PCORE(x, ¬W')
5:   return N'
```

[1] I.e., $\forall \overline{x}, \overline{i}, \overline{o} \,.\, \exists \overline{x}' \,.\, T(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$. T can always be made complete: if some input is not allowed by the original specification, T can allow for arbitrary outputs; if some output is not allowed originally, T can visit an unsafe state.

[2] I.e., $\forall \overline{x}, \overline{i}, \overline{o}, \overline{x_1}', \overline{x_2}' \,.(T(\overline{x}, \overline{i}, \overline{o}, \overline{x_1}') \wedge T(\overline{x}, \overline{i}, \overline{o}, \overline{x_2}')) \Rightarrow (\overline{x_1}' = \overline{x_2}')$.

[3] Skolem functions define existentially quantified variables as a function over the universally quantified ones such that the QBF becomes $\mathbf{true}$.

[4] Herbrand functions define universally quantified variables as a function over the existentially quantified ones such that the QBF becomes $\mathbf{false}$.

As long as N' is not yet $\neg W'$, i.e., $W' \wedge N'$ is still satisfiable, we refine N' with a clause that excludes the cube $\mathbf{x}$ witnessing this insufficiency. By taking the unsatisfiable core, the clause eliminates also other counterexamples. Since clauses are only added, NegLearn is suitable for incremental SAT solving.

Using incremental SAT solving, we also simplify W by removing literals and clauses as long as W does not change semantically. This is applied to all following methods as well.

B. QBF-Based Learning

In [7], several learning-based circuit synthesis algorithms are presented and implemented using BDDs. Here, we discuss an efficient implementation of the CNF-learning algorithm using a QBF-solver. Since QBF-solvers operate on CNFs, this algorithm is particularly suitable. It can be defined as follows.

```
1: procedure SYLEARNQBF(S(x̄, ī, ō, x̄'))
2:   ū := x̄ ∪ ī,   v̄_a := v̄ := ō ∪ x̄'
3:   for v ∈ v̄ do
4:     v̄_a := v̄_a\{v},  v̄_e := v̄\v̄_a,  f_v := true,  R := v∧¬S
5:     while sat, with (sat, u):=QSAT(∃ū . ∀v̄_a . ∃v̄_e . R) do
6:       u_2 := QCORE(u, ∃ū . ∀ō_a . ∃ō_e, x̄' . ¬v ∧ ¬S)
7:       f_v := f_v ∧ ¬u_2,   R := R ∧ ¬u_2
8:     DUMPCIRCUIT(v, f_v),   S := S ∧ (v ↔ f_v)
```

SyLearnQBF builds up circuits in f_v for one $v \in \overline{v}$ after the other. Initially, $f_v = \mathbf{true}$, i.e., the circuit always outputs $\mathbf{true}$. While there exists an input $\mathbf{u}$ for which v must be **false** (the QBF in line 5 is satisfiable), f_v is refined with a clause that maps $\mathbf{u}$ to **false**. By taking the core in line 6, other inputs are also mapped to **false** as long as **false** is allowed by S. The final solution f_v is dumped, and S is refined with the implementation of v before the next circuit is computed. The final f_v are in CNF, so the circuits have a depth of only two. Even after optimizations and mapping to standard cells, the depth usually remains low [7], which enables fast clock rates.

Once $\neg S$ is available in CNF, the algorithm only adds clauses to existing CNFs (i.e., to R and f_v). Only for the resubstitution in line 8, a CNF encoding of $\neg f_v$ is needed.

Optimizations for Safety Specifications. As in Sect. III-A, $\neg S$ is defined as $W \wedge T \wedge \neg W'$. This requires a CNF encoding of $\neg W'$. While computing $\neg W'$ with NegLearn is beneficial for QBFCert, it does not pay off for SyLearnQBF. Hence, we build a CNF for $\neg W'$ with one auxiliary variable per clause of W'. Recently, the QBF solver DepQBF was equipped with incremental solving capabilities [11]. SyLearnQBF is well suited for incremental solving. We use two solver instances for line 5 and 6 respectively. For each $v \in \overline{v}$, a new incremental session is started. During the inner loop, we only add clauses to the former solver. The QBF of the latter even stays the same. DepQBF supports unsatisfiable cores natively. The resulting cores are small but not necessarily minimal, so we iterate over the remaining literals to obtain even smaller cores because (slightly) smaller cores typically mean (much) less iterations.

C. Interpolation

Jiang et al. [8] present two interpolation-based approaches to synthesize circuits for one $v \in \overline{v}$ after the other. The first one

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

expands S over $\overline{v}$. We consider this intractable in our setting. The second approach circumvents the quantifier alternation and expansion by temporarily treating output signals as inputs:

1: **procedure** SYINT($S(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$)
2: $\overline{d} := \overline{x} \cup \overline{i} \cup \overline{o} \cup \overline{x}'$, $\overline{r} := \emptyset$
3: **for** $v \in \overline{v}$ **do**
4: $\overline{d} := \overline{d} \setminus \{v\}$, $\overline{r} := \overline{r} \cup \{v\}$
5: $\overline{r}_1, \overline{r}_2, \overline{r}_3, \overline{r}_4 :=$ create4FreshCopies($\overline{r}$)
6: $M_1(\overline{d}, \overline{r}_1, \overline{r}_2) := (S \wedge v)[\overline{r} \leftarrow \overline{r}_1] \wedge (\neg S \wedge \neg v)[\overline{r} \leftarrow \overline{r}_2]$
7: $M_0(\overline{d}, \overline{r}_3, \overline{r}_4) := (S \wedge \neg v)[\overline{r} \leftarrow \overline{r}_3] \wedge (\neg S \wedge v)[\overline{r} \leftarrow \overline{r}_4]$
8: $f_v(\overline{d}) := \text{INT}(M_1(\overline{d}, \overline{r}_1, \overline{r}_2), M_0(\overline{d}, \overline{r}_3, \overline{r}_4))$
9: DUMPCIRCUIT(v, f_v), $S := S \wedge (v \leftrightarrow f_v)$

In each iteration, $\overline{d}$ contains all variables on which the implementation of the current $v \in \overline{v}$ can depend, and $\overline{r}$ contains the rest. For $\overline{v} = (v_1, \ldots, v_n)$, v_1 can depend not only on $\overline{u}$ but also on $(v_2, \ldots, v_n)$, v_2 can depend on $\overline{u}$ and $(v_3, \ldots, v_n)$, etc. Yet, when the circuits for all $v \in \overline{v}$ are built together, the signals $\overline{v}$ effectively depend on $\overline{u}$ only. The formulas M_1 and M_0 characterize the $\overline{d}$-vectors for which v must be set to $\mathbf{true}$ and $\mathbf{false}$ respectively. The syntax $[\overline{r} \leftarrow \overline{r}_i]$ means that the variables $\overline{r}$ are renamed by fresh copies $\overline{r}_i$. Line 8 computes an interpolant between M_1 and M_0. The property $M_1 \Rightarrow f_v \Rightarrow \neg M_0$ of the interpolant ensures that (a) f_v is $\mathbf{true}$ whenever it must be $\mathbf{true}$, and (b) whenever f_v is $\mathbf{true}$ then it does not have to be $\mathbf{false}$. The renaming of the variables $\overline{r}$ has the effect that f_v can only depend on the shared signals $\overline{d}$.

Optimizations for Safety Specifications. In order to avoid double-negations of W in S by negating S, we compute

$$M_1 := (T \wedge W' \wedge v)[\overline{r} \leftarrow \overline{r}_1] \wedge (T \wedge \neg v \wedge W \wedge \neg W')[\overline{r} \leftarrow \overline{r}_2]$$
$$M_0 := (T \wedge W' \wedge \neg v)[\overline{r} \leftarrow \overline{r}_3] \wedge (T \wedge v \wedge W \wedge \neg W')[\overline{r} \leftarrow \overline{r}_4]$$

Note the difference to a plain substitution of $S = T \wedge (\neg W \vee W')$ and $\neg S = T \wedge W \wedge \neg W'$ in SYINT: $(\neg W \vee W')$ reduces to W' due to the conjunction with W from $\neg S$. This is fortunate because disjunctions are expensive in CNF. Since SYINT allows v_i to depend on other v_j with $j > i$, it is sensitive to the variable order, both regarding execution time and circuit size. We exploit this insight with the following optimization. Once v_i has been synthesized, we analyze on which v_j it actually depends. If v_i does not depend on a particular v_j, then v_j is allowed to depend on v_i. This gives an increased flexibility without introducing circular dependencies. We simplify all computed interpolants with ABC[5] [5].

D. SAT-Based Learning

Here, we use SYINT but with a special interpolation procedure (called in line 8) that applies computational learning:

1: **procedure** INTLEARN($M_1(\overline{d}, \overline{r}_1, \overline{r}_2)$, $M_0(\overline{d}, \overline{r}_3, \overline{r}_4)$)
2: $f := \mathbf{true}$
3: **while** sat, with $(\mathsf{sat}, \mathbf{d}) := \text{PSAT}(M_0 \wedge f)$ **do**
4: $f := f \wedge \neg\text{PCORE}(\mathbf{d}, M_1)$
5: **return** f

[5]We use the command sequence `strash; refactor -zl; rewrite -zl;` up to 3 times, followed by `dfraig; rewrite -zl; dfraig;`.

As long as there exists some $\mathbf{d}$ for which f is $\mathbf{true}$ but must be $\mathbf{false}$, i.e., $M_0 \wedge f$ is satisfiable, we refine f with an additional clause that excludes the cube $\mathbf{d}$ witnessing this insufficiency. By taking the unsatisfiable core, other inputs are also mapped to $\mathbf{false}$ as long as $\mathbf{false}$ is allowed.

Optimizations. We use two SAT solver instances, one for line 3 and one for line 4. A new incremental session is started upon each call of INTLEARN. Using activation variables to set $\overline{v}$-variables to $\mathbf{true}$, $\mathbf{false}$, or equal to their renamed copy, we can even use one incremental session throughout the entire SYINT procedure. However, this did not result in significant improvements in our experiments. All optimizations discussed in Sect. III-C can be applied. We also extended the variable dependency optimization further: The CNF T often contains many auxiliary variables that are defined uniquely by other signals of $\overline{x}$, $\overline{i}$, $\overline{o}$. If some of these auxiliary variables depend only on $\overline{d}$, then we allow f to depend on them as well by including them into $\overline{d}$. This can be beneficial because these auxiliary variables often capture the important connections between the variables $\overline{x}$, $\overline{i}$, $\overline{o}$. When dumping the circuits, we add additional gates that define the referenced auxiliary variables as done by T. We also implemented a second minimization pass that tries to remove every clause and literal from every CNF f after SYINT is done. However, this only results in minor circuit size improvements (around 20%).

IV. EXPERIMENTAL RESULTS

A. Implementation

We implemented the discussed methods and optimizations in the SAT-based synthesis tool Demiurge[6] [4]. Demiurge synthesizes AIGER[7] circuits from safety specifications and complies with the SyntComp[8] competition rules. The archive of version 1.1.0 contains way more experiments than reported here. E.g., for the SAT-based learning approach alone we implemented 24 variants. Here, we only compare interesting versions, summarized in the following table.

Name	Engine	Algorithm
BDD	CuDD 2.4.2	Cofactor-Based [3]
QC	QBFCert 1.0	QBF-Certification (Sect III-A)
QL	DepQBF 3.02	SYLEARNQBF (Sect III-B)
SI	MathSAT 5	SYINT (Sect III-C)
SL	Lingeling ats	SYINT+INTLEARN (Sect III-D)
SLN	Lingeling ats	SL without dependency opt.

BDD serves as baseline for our comparison. It was created by students and won a competition held in a synthesis lecture. It implements a cofactor-based approach [3], uses dynamic variable reordering, and forced reorderings at certain points. QC, QL, SI, and SL implement the methods from the previous section with all optimizations. SLN is used to highlight the benefits of the dependency optimization. All our methods use ABC[5] [5] to minimize the final circuits further. SI uses

[6]http://www.iaik.tugraz.at/content/research/design_verification/demiurge/.
[7]http://fmv.jku.at/aiger/
[8]http://www.syntcomp.org/

MathSAT, which supports several interpolation schemes. We use McMillan's scheme (see [6]), but the performance is similar with other schemes. We also implemented our own interpolation engine by processing proofs produced by PicoSat. However, the proof files grew prohibitively large.

The limitations of our implementation are that it can only handle safety specifications in AIGER format, it can produce circuit only in AIGER format, and it runs under Linux only.

B. Benchmarks

We use the same benchmarks as [4], but report here only results for the interesting ones. The benchmarks `amba`ij specify an arbiter for ARM's AMBA AHB bus [3], where i is the number of masters, and $j \in \{\mathtt{c}, \mathtt{b}, \mathtt{f}\}$ indicates the method used to transform the original benchmark [3] into our input format [4]. The benchmarks `genbuf`ij, again with $j \in \{\mathtt{c}, \mathtt{b}, \mathtt{f}\}$, define a generalized buffer [3] connecting i senders to two receivers. The specifications `add`i and `mult`i denote i-bit combinational adders and multipliers.

C. Results and Discussion

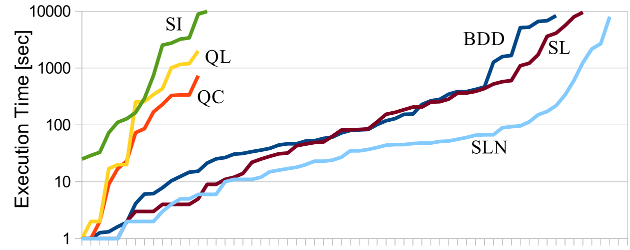
Fig. 2 summarizes our results with cactus plots. The y-axis gives the execution time or circuit size, and the x-axis gives the number of benchmarks that can be solved within this time or size limit. Concrete numbers and more plots can be found in an extended version [2] of this paper and in the downloadable archive. All experiments were performed on an Intel Xeon E5430 CPU running a 64 bit Linux at 2.66 GHz. The memory limit was set to 8 GB, the time-out to 10 000 seconds. All circuits have been successfully model checked.

Method SL achieves the best results both regarding execution time and circuit size. The dependency optimization (SL vs. SLN) is very beneficial for `add` and `mult`, but slower for `amba` and `genbuf`. QC, QL, and SI do not perform so well. Using incremental QBF solving in QL gives an average speedup of factor 3.5. The speedup factor compared to using the QBF preprocessor Bloqqer is even 21. Still, QL is not very competitive. BDD is much better, but still outperformed by SL. In particular, SL outperforms SI by many orders of magnitude. Hence, our idea of implementing the interpolation procedure with computational learning is very beneficial. Execution time and circuit size are not in conflict but rather correlate. The time for optimization with ABC is usually insignificant, but only yields moderate size reductions (around 25 % for SL). Using method SLN, Demiurge won a track of SyntComp 2014. One reason was the small circuit size compared to other tools.

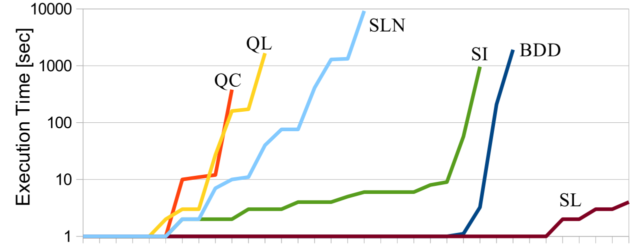
V. Conclusion

We compared several SAT- and QBF-based methods to synthesize circuits from strategies, and presented optimizations and efficient implementations for safety specifications. Our SAT-based learning method combines the quantifier elimination approach by Jiang et al. [8] with computational learning as proposed by Ehlers et al. [7], and outperforms BDDs both regarding execution time and circuit size in our experiments.

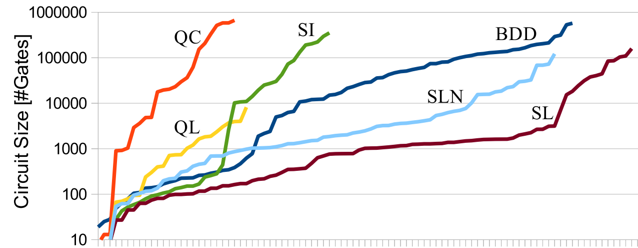
Future research includes preprocessing for incremental QBF solving, exploiting unreachable states, and parallelization.



(a) Execution time for `amba` and `genbuf`.



(b) Execution time for `add` and `mult`.



(c) Circuit size for all benchmarks.

Fig. 2. Cactus plots summarizing our performance evaluation.

References

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[2] R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing. SAT-based methods for circuit synthesis. *CoRR*, abs/1408.2333, 2014.

[3] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[4] R. Bloem, R. Könighofer, and M. Seidl. SAT-based synthesis methods for safety specs. In *VMCAI'14*. Springer, 2014.

[5] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV'10*. Springer, 2010.

[6] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI'10*. Springer, 2010.

[7] R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *FMCAD'12*. IEEE, 2012.

[8] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung. Interpolating functions from large boolean relations. In *ICCAD'09*. IEEE, 2009.

[9] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.

[10] F. Lonsing and A. Biere. DepQBF: A dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.

[11] F. Lonsing and U. Egly. Incremental QBF solving. In *CP'14*. Springer, 2014. To appear.

[12] A. Morgenstern, M. Gesell, and K. Schneider. Solving games using incremental induction. In *IFM'13*. Springer, 2013.

[13] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere. Resolution-based certificate extraction for QBF. In *SAT'12*. Springer, 2012.

[14] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS'09*. Springer, 2009.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Synthesis of Synchronization using Uninterpreted Functions

Roderick Bloem, Georg Hofferek, Bettina Könighofer,
Robert Könighofer, Simon Außerlechner, and Raphael Spörk
Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.

Abstract—**Correctness of a program with respect to concurrency is often hard to achieve, but easy to specify: the concurrent program should produce the same results as a sequential reference version. We show how to automatically insert small atomic sections into a program to ensure correctness with respect to this implicit specification. Using techniques from bounded software model checking, we transform the program into an SMT formula that becomes unsatisfiable when we add correct atomic sections. By using uninterpreted functions to abstract data-related computational details, we make our approach applicable to programs with very complex computations, e.g., cryptographic algorithms. Our method starts with an empty set of atomic sections, and, based on counterexamples obtained from the SMT solver, refines the program by adding new atomic sections until correctness is achieved. We compare two different such refinement methods and provide experimental results, including Linux kernel modules where we successfully fix race conditions.**

I. Introduction

Concurrency-related bugs form a serious problem in software development. First, concurrent programs are hard to get right due to the large number of possible interleavings of threads. Second, concurrency issues are difficult to detect and to reproduce: faults may only appear in rare cases that are never hit by tests but only in operation. Third, even if detected and reproducible, concurrency errors are difficult to fix. There is the danger of fixing only some but not all symptoms, or even introducing new errors. At the same time, the desired behavior of a concurrent program is typically easy to specify: it should behave as if executed sequentially. This important property is called *serializability*, meaning that any concurrent execution must behave as if all threads were executed one after the other (in some order). In this paper, we present methods to synthesize efficient synchronization in form of atomic sections to ensure serializability. Assertions can be used as an additional (or alternative) specification. Thus, on a high abstraction level, we address the same problem as [7, 24].

Adequate *abstraction* is a key factor in making synthesis of synchronization tractable. Our intuition is that synchronization usually should not depend on the semantics of data operations. Thus, we propose to use abstract data operations by means of *uninterpreted functions*. This is done by replacing all arithmetic operations as well as calls to functions without side-effects by uninterpreted functions during program analysis. This speeds up the synthesis process significantly. However, abstraction may induce spurious counterexamples, which may lead to more and larger atomic sections than actually necessary. One way to address this issue is to allow the user to refine (some) uninterpreted function symbols with fundamental properties like commutativity and associativity. Such properties are important in the context of concurrent programs because different interleavings often apply the same operations in different order (e.g., $(3+4)+5$ vs. $4+(5+3)$).

Building on abstraction by means of uninterpreted functions, we present and compare two synthesis methods. They repeatedly check for counterexamples (executions violating the specification) and add atomic sections until no more counterexamples exist. Counterexamples are computed by a Satisfiability Modulo Theories (SMT) solver, using a Bounded Model Checking (BMC) approach [21]. We unroll loops in the program and guarantee correctness only up to the unrolling depth. First, we present a novel method that we named FixSwitches. It analyzes counterexamples with a heuristic to guess the context switch that causes the problem, and forbids this switch with an atomic section. It does not guarantee minimality of the atomic sections, nevertheless it always produced a minimal solution in all our experiments. The second method, named AtomConstr, is based on [24] and collects constraints for the atomic sections based on the counterexamples: at least one context switch of every counterexample must be forbidden. These constraints are then solved to obtain a global minimum of atomic sections. We implemented our synchronization synthesis approach in a prototype tool called Atoss and present first experimental results. We also compared our methods with several set minimization algorithms (e.g. the QuickXplain algorithm [17]), trying to find a (locally) minimal set of atomic sections that is sufficient to make the program correct. It turns out that FixSwitches and the AtomConstr algorithm scale best, so we do not present these experiments in detail.

Related Work. A lot of work has been done to *verify* concurrent programs [14, 10, 8]. Verification is an important building block in our synthesis method: we use a BMC approach [21] to search for counterexamples. Automatic synthesis of synchronization was first considered in 1981 by Clarke and Emerson [7]. In the last few years, this topic was taken up again, e.g. in [23], [24], [5], [18], and [6]. Vechev et al. [24] abstract the program state using a finite domain and compute counterexamples by explicitly searching

This research was supported by the Austrian Science Fund (FWF) through projects RiSE (S11406-N23) and QUAINT (I774-N23).

through the abstract transition system graph. Then, heuristics decide whether to refine the abstraction or insert an atomic section. The user has to provide a characterization of the good states as specification. In contrast, our approach can take the sequential behavior as implicit specification, it searches for counterexamples symbolically, using an SMT solver, and uses uninterpreted functions for abstraction. Counterexample-guided synthesis is also considered in [5]. Counterexamples are generalized to so-called partial-order traces that represent all counterexamples that lead to the same error. Partial-order traces are eliminated by lock insertion, but also by other semantics-preserving program transformations like instruction reordering. In contrast, we consider counterexamples with an increasing number of context switches, thus we can skip the generalization step. Kahlon [18] considers the problem of fixing concurrency errors once they are detected. Given a set of mutually atomic segments, the algorithm inserts locks around the segments to fix the atomicity violation without introducing new deadlocks. In contrast, our approach does not assume that mutually atomic sections are already given.

Uninterpreted Functions are often used as an adequate mean of abstraction in verification, e.g., in translation validation [20], where a compiler is verified by checking its input and output program for sequential correctness. Another example is proving equivalence between a pipelined and a non-pipelined version of the same processor [4, 3], where the complex datapath elements such as the ALU are abstracted. Abstraction by uninterpreted functions has also been used for synthesizing controllers that avoid concurrency-related problems in pipelined processors [15, 16]. The main difference is that [15, 16] synthesizes controllers whose actions may depend on the current inputs of the system. This amounts to solving formulas of the form $\forall \mathsf{inputs}.\exists \mathsf{control}.\forall \mathsf{outcomes}.\phi$, where ϕ is a correctness criterion. In this paper, we effectively solve problems of the structure $\exists \mathsf{control}.\forall \mathsf{inputs}.\phi$, because in software it is customary to have static synchronization mechanisms that do not depend on the current inputs of a program. This quantifier structure also makes the problem easier and allows us to deal with larger numbers of existentially quantified variables, whereas the approach of [15, 16] scales exponentially w.r.t. this number.

Contributions. In summary, the main contributions of this work are as follows.

- We relieve the user from writing a specification by taking the sequential behavior of the concurrent program as implicit specification.
- To the best of our knowledge, we are the first to use uninterpreted functions as abstraction for synthesis of synchronization. We show that this allows us to handle programs that cannot be handled with finite-domain abstractions.
- We present and compare two methods to infer atomic sections from counterexamples. One is novel and specifically tailored towards our synthesis algorithm, the other one is based on ideas from [24].

Outline. The rest of this paper is structured as follows. Section II discusses preliminaries and establishes notation. Section III presents an illustrating example. Section IV presents the synchronization synthesis algorithms and introduces our abstraction method based on uninterpreted functions. Experimental results are shown in Section V. Section VI concludes and discusses directions for future work.

II. Preliminaries

Concurrent Programs. A concurrent program P is a set of threads $T = \{t_1, \dots t_n\}$. Each thread t_i is represented as a control flow graph $t_i = (b_i, e_i, V_i, E_i)$, where $V_i = \{s_{i1}, \dots, s_{im}\}$ is the set of nodes, $b_i \in V_i$ is a unique start node, $e_i \in V_i$ is a unique end node, and $E_i \subseteq V_i \times L_i \times V_i$ is a set of directed and labeled edges between the nodes. The set of labels L_i is comprised of Boolean expressions ($\mathbb{B}$-expr), defined below. If the control flow graph is cyclic, which means that the program contains loops, we unroll them up to a certain depth to make it acyclic. Each node s_{ij} is labeled by a program statement. For simplicity, we assume that each statement of the concurrent program corresponds to a different node in the graph. Thus, different nodes can be labeled with the same instruction. Edge labels express conditions. An edge $(s, l, s') \in E_i$ means that s' is the successor statement of s if condition l holds. The node e_i does not have a successor. We denote with $\mathcal{G}$ the set of global variables shared between all threads. Furthermore, each thread t_i has a set of local variables $\mathcal{L}_i$. To simplify the presentation, we assume that all program variables range over the same domain $\mathbb{D}$.

We will model concurrent programs as formulas in the quantifier-free fragment of the *Theory of Uninterpreted Functions and Equality* $\mathcal{T}_U$ (`QF_UF`). To do so, we make the following more formal definition of statements and conditions. Let $\mathcal{F}$ be a set of (uninterpreted) functions $f : \mathbb{D}^+ \mapsto \mathbb{D}$, let $\mathcal{P}$ be a set of (uninterpreted) predicates $p : \mathbb{D}^+ \mapsto \mathbb{B}$ with $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, let $v \in \mathcal{L}_i \cup \mathcal{G}$ be a variable, $f \in \mathcal{F}$ be an uninterpreted function, let $p \in \mathcal{P}$ be an uninterpreted predicate, and let $=$ be the (interpreted) equality predicate. The set of $\mathbb{D}$-expressions and $\mathbb{B}$-expressions is defined as follows.

$$\begin{aligned} \mathbb{D}\text{-expr} &::= v \mid f(\mathbb{D}\text{-expr}^+) \\ \mathbb{B}\text{-expr} &::= p(\mathbb{D}\text{-expr}^+) \mid \mathbb{D}\text{-expr} = \mathbb{D}\text{-expr} \mid \\ &\qquad \neg\mathbb{B}\text{-expr} \mid \mathbb{B}\text{-expr} \vee \mathbb{B}\text{-expr} \end{aligned}$$

A statement is of the form $v := r$, where $v \in \mathcal{L}_i \cup \mathcal{G}$ and $r \in \mathbb{D}$-expr. That is, all statements are assignments; we assume that all function calls have been inlined and do not allow recursion. An edge label $l \in L_i$ is a $\mathbb{B}$-expression. The semantics of statements and conditions on edges are as expected. The labeled edges are such that all statement nodes $s \in V_i \setminus \{e_i\}$ have exactly one successor for every variable valuation (i.e., for a given scheduling, the program is deterministic). We will write $V = \bigcup_i V_i$ for the set of all graph nodes, $V' = \bigcup_i (V_i \setminus \{e_i\})$ for all but the end nodes, and $\mathsf{thread}(s_{ij})$ for the thread t_i to which the statement s_{ij} belongs.

Listing 1 RSA decryption using the Chinese Remainder Theorem (CRT)

Input: large primes p, q; ciphertext c; private exponent d;
Output: plaintext in m_p

1: bool fin_1=false, fin_2=false;
2: int merged=0, m_p=0, m_q=0;

3: **procedure** THREAD$_1$	10: **procedure** THREAD$_2$
4: m_p=$c^d \mod p$;	11: m_q=$c^d \mod q$;
5: fin_1 = true;	12: fin_2 = true;
6: **if** merged=0 && fin_2	13: **if** merged=0 && fin_1
7: merged=1;	14: merged=2;
8: **if** merged=1	15: **if** merged=2
9: m_p=crt(m_p,m_q);	16: m_p=crt(m_p,m_q);

Concurrent Executions and Correctness. An execution of program P is a sequence of statements $\overline{s} = s_1, s_2, \ldots \in V^*$ respecting the program semantics. An atomic section set is a set $A \subseteq V'$. A program execution $\overline{s} = s_1, s_2, \ldots$ respects the atomic section set A if $s_i \in A$ implies $\mathsf{thread}(s_i) = \mathsf{thread}(s_{i+1})$ for all i. That is, if statement s_i is protected by an atomic section, then no thread switch is allowed immediately after the statement. An execution is *sequential* if it respects the atomic section set $A = V'$. In order to define a notion of correctness for concurrent programs, we introduce a function $\mathsf{eval} : V^* \to \mathbb{D}^{|\mathcal{G}|}$, which — given an execution $\overline{s} = s_1, s_2, \ldots$ of P — returns the values of the global variables after the execution. We say that an execution $\overline{s}$ is *correct* if there exists a sequential execution $\overline{s}'$ such that $\mathsf{eval}(\overline{s}) = \mathsf{eval}(\overline{s}')$. A *counterexample* is an incorrect execution. We define a procedure $\mathsf{ce}(A)$ which returns a counterexample that respects an atomic section set $A \subseteq V'$, or the constant None if no such counterexample exists. An atomic section set A is *sufficient* if $\mathsf{ce}(A) = \mathsf{None}$. An atomic section set A is a *local minimum* if it is sufficient and all $A' \subset A$ are not sufficient. An atomic section set A is a *global minimum* if it is sufficient and all A' with $|A'| < |A|$ are not sufficient. Given an execution $\overline{s} = s_1, s_2, \ldots$ of P, we say that a thread switch after statement s_i (with $\mathsf{thread}(s_i) \neq \mathsf{thread}(s_{i+1})$) is *mandatory* if $s_i \notin V'$, i.e., s_i is an end node of some control flow graph. Otherwise, the thread-switch is *non-mandatory*.

III. Illustrating Example

We give an example to demonstrate our approach, in particular the benefits of abstraction with uninterpreted functions. Consider the problem of decrypting an RSA-encrypted message (cf. Listing 1). For efficiency, many cryptographic libraries employ the Chinese Remainder Theorem (CRT) during RSA decryption [19]. As usual, p and q are two large prime numbers, c represents the ciphertext and d is the private decryption exponent. In standard RSA, the message m is obtained by computing $m = c^d \mod p \cdot q$. To speed up the decryption process, Thread 1 computes $m_p = c^d \mod p$ and Thread 2 computes $m_q = c^d \mod q$. After m_p and m_q are found, one of these threads uses the function `crt` to compute the final message (modulo $p \cdot q$) and stores it in m_p. The concurrent program is correct if the final message m_p equals the message obtained by a sequential run of the two threads (in either order).

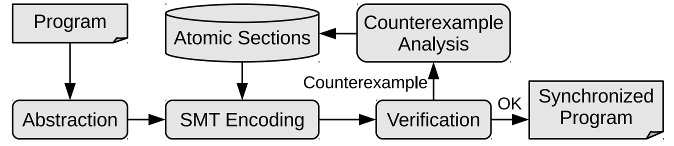


Fig. 1. Overview of our synthesis approach.

Without any atomic sections, the following problem could occur. If Thread 1 is interrupted between lines 6 and 7, and Thread 2 executes lines 13–16 in the meantime, Thread 1 will subsequently set `merged` to 1, and execute line 9. However, the merge has already been performed by Thread 2, and doing it a second time results in erroneous output. The problem could be prevented by making lines 6–7 and lines 13–14 atomic sections.

The RSA algorithm uses complex arithmetic functions (modular reductions, exponentiations, etc.) on very large numbers. Modeling this program with linear integer arithmetic is not possible, due to the complex operations involved. On top of that, modeling it with bitvectors is also not feasible, due to the large bitwidths involved. However, when using an abstraction with uninterpreted functions, the resulting SMT formula is rather simple. The line $m_p = c^d \mod p$, for example, reduces to one simple equality between a domain variable and an uninterpreted function instance: $m_p = f_{modexp}(c, d, p)$. Using abstraction with uninterpreted functions, our tool was able to find the minimal set of atomic section in a few seconds (atomic sections spanning lines 6–7 and lines 13–14). Without any abstraction, it would not be possible to verify this program.

Note that the finite-domain abstraction approach presented in [24] cannot deal with this example. One problem is that finite-domain abstractions are not *equality preserving*. They only track properties like the parity of variable values, or whether certain values are in a particular interval. This is usually too coarse to prove the equality of values (without refining the abstraction until all bits of the relevant variables are tracked). Note that this problem also occurs for simple functions such as addition or multiplication.

IV. Synthesis Approach

The working principle of our synthesis approach is outlined in Figure 1. The main input is a concurrent program P without any synchronization. First, the program is abstracted using uninterpreted functions. This step is explained in Section IV-A. Next a counterexample-guided synchronization refinement loop is entered. There is a database of (candidates for) atomic sections, which is initially empty. Considering these already known atomic sections, we next encode the concurrent verification problem into an SMT formula. Satisfying assignments of this formula correspond to counterexamples, i.e., executions of the concurrent program which violate the specification. The

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

SMT encoding is discussed in Section IV-B. In the verification step, an SMT solver searches for a counterexample in form of a satisfying assignment of the constructed formula. If a counterexample is found, it is analyzed in order to infer new atomic sections that prevent (at least) this particular counterexample, and we loop back to checking whether the program is correct now. Two different methods for analyzing counterexamples and refining the atomic sections will be presented in Section IV-C and Section IV-D, respectively. If no more counterexamples exist, we have found a set of atomic sections that are sufficient to prevent erroneous executions and the algorithm terminates.

A. Abstraction using Uninterpreted Functions

A program statement updates a variable with a new value that is the result of some computation. The computation can be as simple as an increment, or an inlined addition, but it can also be a call to a complex n-ary function. We observe that in many cases, correctness of a program does not depend on the actual semantics of the functions involved in the computations. For example, if you replace all additions in a correct concurrent program by multiplications, the resulting program still should not depend on the scheduling. The only thing that is relevant to correctness is *functional consistency*, i.e., given the same inputs, a particular statement should always produce the same result.

It might be obvious to use logics based on the theories of linear integer arithmetic, linear real arithmetic, or bitvector arithmetic, which include interpreted and axiomatized symbols encoding addition, multiplication, etc. In fact, loop-free programs can be modeled perfectly using bitvector logic [9]. However, by doing so we burden the SMT solver unnecessarily, because it now has to look for solutions that satisfy all the axioms of the interpreted symbols. In addition to that, more complex operations might not easily (or even not at all) be expressible in terms of the available interpreted functions.

Thus, we suggest to "forget" all the semantics of a statement, and abstract it using uninterpreted functions only. E.g. a statement `a = b + c` becomes $a = f_{plus}(b,c)$, where $f_{plus} \in \mathcal{F}$ is an uninterpreted symbol. In the example presented in Section III, there are two uninterpreted functions that we would need to introduce: $f_{modexp}(\cdot,\cdot,\cdot)$ and $f_{crt}(\cdot,\cdot)$.

However, even though the functions we use are uninterpreted, there are two important properties that are of particular interest in the setting of concurrency: *commutativity* and *associativity*. The reason for that is that different interleavings of threads will lead to a different order of operations. However, knowing that some functions are commutative and associative, it is still possible to prove that the final outcome is the same. One possible way to achieve this is to add those concrete instances of the commutativity and associativity axioms that are actually relevant to a particular example: i.e., state for every pair of variables a, b that $f_{plus}(a,b) = f_{plus}(b,a)$, and similar for associativity. A potentially more efficient way is to add support for commutativity and associativity directly in the congruence closure module of the underlying SMT solver.

Listing 2 C Code

```
1: int g;
2: procedure THREAD1            7:  procedure THREAD2
3:    int x = g;                8:     int y = g;
4:    x = x + 1;                9:     y = y + 2;
5:    g = x;                    10:    g = y;
6:    x = x + 1;                11:    y = y + 2;
```

Listing 3 SSA Constraints

```
2: procedure THREAD1            7:  procedure THREAD2
3:    t1x1 = t1g1               8:     t2y1 = t2g1
4:    t1x2 = t1x1 + 1           9:     t2y2 = t2y1 + 2
5:    t1g2 = t1x2               10:    t2g2 = t2y2
6:    t1x3 = t1x2 + 1           11:    t2y3 = t2y2 + 2
```

The theory of how to do this has been outlined in [1]; we are currently working on adding this feature to the Z3 SMT solver [12].

B. SMT Encoding of the Concurrent Verification Problem

This section explains how we encode the concurrent verification problem into an SMT formula such that satisfying assignments correspond to counterexamples. SMT encoding of programs has been addressed before, e.g. in [14] and [11]. We use an encoding called TCBMC [21], with small modifications. The main idea is to limit the maximum number of thread switches while allowing them to be anywhere in the code. This has the advantage that we are able to analyze counterexamples with an increasing number of thread switches. Most concurrency errors appear with only a few thread switches [13]. By first eliminating these counterexamples, we forbid many other execution paths representing the same bug. TCBMC consists of four steps.

Step 1: Preprocessing. Complex program statements are not always executed atomically. However, if there is at most one occurrence of a global variable in a statement, context switches during the execution of the complex statement obviously cannot introduce concurrency-related errors. In contrast to this, context switches in statements that have more than one occurrence of global variables can introduce concurrency bugs. To model such context switches, we split statements with more than one reference to a global variable. This is done like in a compiler, where complex statements are broken down into simple instructions. For example, consider the statement $g_3 = g_1 + g_2$;, where g_1, g_2, g_3 are global variables. The statement is translated into $l_1 = g_1; \quad l_2 = g_2; \quad g_3 = l_1 + l_2$;, where l_1, l_2, l_3 are fresh local variables.

Step 2: Applying CBMC Separately on Each Thread. The next step is to unroll all loops, inline all function calls, and transform the code into *static single assignment form* (SSA form), where each variable is assigned only once. Hence, for each assignment to a variable, a new copy of this variable is created. Additionally, all variable names are prefixed with a thread identifier. E.g., for a global variable g (cf. Listing 2), copies "$t_1 g_1$", "$t_1 g_2$", etc. are created (cf. Listing 3). This

second step is performed for each thread in isolation, as done in CBMC [9]. It yields a separate formula for each thread, not taking into account that an execution of a thread can be interrupted by another one, which may change global variables. This is dealt with by Step 4, where additional concurrency-related constraints are added. To illustrate Step 2, consider the simple program shown in Listing 2. After applying CBMC separately on each thread, we get a formula representing the two threads as illustrated in Listing 3.

Step 3: Generating Block Variables and Atomic Sections. During an execution, sequentially executed lines of code from one thread form a so-called *context switch block*. For each line l of thread t, a so-called *block variable* $block_t(l)$ is introduced. The value of the block variable encodes to which context switch block the line belongs. Lines with the same values for their block variables belong to the same context switch block, and the blocks are executed in increasing order. So, by choosing values for the block variables, the SMT solver establishes the scheduling of the threads. Potential values of the block variables for our example from Listing 2 are illustrated in Figure 2. The block variables have to satisfy the following constraints:

1) The first block value of each thread should be positive, i.e., $\forall t \in T : block_t(1) \geq 1$.
2) For all threads, the block variable values must increase monotonically w.r.t. line numbers within a thread, i.e., $\forall t \in T, l \in t : block_t(l) \leq block_t(l+1)$.
3) The values of the block variables are not allowed to change by one (at least one thread should be running in between), i.e., $\forall t \in T, l \in t : block_t(l) + 1 \neq block_t(l+1)$.
4) No block variable value must exceed a given bound n. This is enforced by $\forall t \in T : block_t(m) \leq n$, where m is the last line number of the respective thread.
5) Each block variable value can only occur in one thread, i.e., $\forall t \in T, l \in t : \forall t' \in T \setminus t, l' \in t' : block_t(l) \neq block_{t'}(l')$.

Note that these rules for the block variables differ from [21]. The authors in [21] only give a detailed description of how to encode the block variables for two threads. For extending this to the general case, they suggested to enforce a round robin scheme among the threads, or to introduce new variables that represent which thread runs in which context switch block. We tried both methods, but found out that our definition of the block variables is much more efficient.

To model an *atomic section* between two consecutive lines of code, it is enough to require that the block variables for these lines must be equal. For instance, to model an atomic section in thread 1 between line 2 and 3, we add the constraint $block_1(2) = block_1(3)$. By adding the constraint $t_1a_{2,3} \rightarrow block_1(2) = block_1(3)$, where $t_1a_{2,3}$ is a boolean variable, we can easily enable or disable atomic sections in our synthesis algorithm by setting $t_1a_{2,3}$ to true or false.

Step 4: Generating Constraints for Concurrency. We have to adjust the SSA statements of each thread, as constructed in Step 2, to capture context switches. A statement

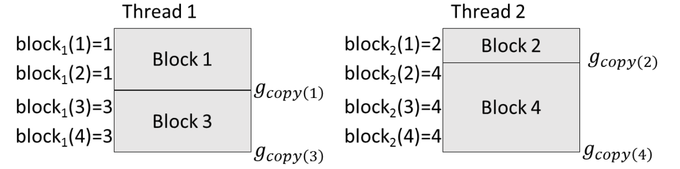


Fig. 2. Context Switch Blocks and Copy Variables.

Listing 4 SSA Constraints

```
1: procedure THREAD1
2:   if block(t1x1) = block(t1g1)
3:     t1x1 = t1g1;
4:   else
5:     b = block(t1g1) - 1;
6:     t1x1 = g_copy(b);
7:   t1x2 = t1x1 + 1;
8:   t1g2 = t1x2;
9:   t1x3 = t1x2 + 1;
10: procedure THREAD2
11:   if block(t2y1) = block(t2g1)
12:     t2y1 = t2g1;
13:   else
14:     b = block(t2y1) - 1;
15:     t2y1 = g_copy(b);
16:   t2y2 = t2y1 + 2;
17:   t2g2 = t2y2;
18:   t2y3 = t2y2 + 2;
```

that reads a global variable has to distinguish if the global variable was last assigned in its own context switch block or in a previous one. In the former case, the local value of the global variable is up to date and can be used. In the latter case, another thread may have altered the global variable, and we need to take the value as assigned by the other thread. Hence, we create copies of the global variables for each block, storing the values of the global variables at the end of the block. The SSA statement can access the copies of the global variables when needed. This is illustrated in Fig. 2. In this example we have four context switch blocks, so we create four additional copies $g_{copy(1)}$ to $g_{copy(4)}$ for each global variable. At the end of each block, we store the value of the last assignment of the global variable in the respective copy.

Let us continue our example. After applying Step 4 to our SSA constraints from Listing 3 we get the final concurrency constraints shown in Listing 4, where $block(x)$ gives the context switch block in which the variable x was assigned. Note that we only have to change an SSA statement if it reads a global variable. In this case, we have to check if the local value is up to date, or if we must use the copy of the global variable from the previous context switch block.

Modeling Assumptions and Assertions: We extended the SMT encoding to also support assertions and assumptions, which are Boolean conditions in the input program P. A counterexample must satisfy all assumptions, but violate one assertion or sequential correctness. Hence, modeling assumptions and assertions in the SMT encoding is straightforward: For computing counterexamples, we add the constraints $\bigwedge_i \mathsf{assumption}_i \wedge \neg\big(\mathsf{seqSpec} \wedge \bigwedge_j \mathsf{assert}_j\big)$. When searching for valid runs, the negation ($\neg$) is omitted. Assumptions can, for example, be used to model `wait` statements.

C. Finding Atomic Sections with the FixSwitches Algorithm

We now turn to the first method to analyze counterexamples in order to infer a small but sufficient set of atomic sections.

Listing 5 FixSwitches Algorithm

```
1: procedure FixSwitches
2:   A := ∅
3:   while ce(A) ≠ None do
4:     s̄ := (s_1, s_2, ... s_m) := ce(A)
5:     (k_1, ..., k_n) := findSwitches(s̄)
6:     for i = n...1 do
7:       if existsValidRun((s_1, s_2, ... s_{k_i})) then
8:         A := A ∪ {s_{k_i}}
9:         break
10:  return A
```

Listing 5 presents a method to compute atomic sections based on a heuristic to analyze counterexamples. As outlined in Fig. 1, it starts with an empty set of atomic sections A. In a loop, a new counterexample $\overline{s} = s_1, s_2, \ldots s_m$ is computed that respects the atomic sections A that have already been found so far. If no such counterexample exists, then A must be sufficient and the algorithm terminates. Otherwise, the procedure `findSwitches` computes all non-mandatory context switches of the counterexample $\overline{s}$ in form of a sequence of indices k such that $\mathsf{thread}(s_k) \neq \mathsf{thread}(s_{k+1})$ and $s_k \in V'$. Next, the algorithm analyzes the context switches of the counterexample in reverse order, i.e. starting with the last non-mandatory context switch k_n. The procedure `existsValidRun` now checks whether it is possible to extend the incomplete execution $s_1, s_2, \ldots s_{k_n}$ to a complete one that is correct and does not have a context switch at k_n. If this is not the case, the program cannot be fixed just by forbidding the context switch k_n; a concurrency problem must already exist in an earlier stage of the execution $\overline{s}$. Thus, we continue to analyze the previous context switch k_{n-1}. Eventually, we must find an index i such that `existsValidRun`$(s_1, \ldots, s_{k_i})$ returns true, because if there are no more switches left in the prefix, then a sequential execution is possible. If `existsValidRun` returns true, we add an atomic section that forbids the context switch k_i (thus making the current counterexample infeasible), and look for a new counterexample.

The procedure `existsValidRun` can be implemented similar to ce, based on an SMT-solver call. In the SMT-solver query of `existsValidRun`, we cannot only assert the execution path of the prefix but also all variable values (taken from the satisfying assignment of the SMT-call in ce) in the different execution steps of the prefix. This renders `existsValidRun`-calls typically much cheaper than ce-calls in terms of computation time. The performance of the entire algorithm increases if we consider counterexamples with a small number of context switches first, and increase the maximum number of (non-mandatory) context switches incrementally. That is, only if no more counterexamples with one context switch exists, we search for counterexamples with two context switches, and so on.

Listing 6 illustrates how this algorithm works. According to the implicit sequential execution, the global variable h should be 2 after executing Thread1 and Thread2 in parallel. Suppose, we get the following counterexample: $\overline{s} = s_1, s_2, s_3$, where $s_1 = \{3, 8\}$, $s_2 = \{9, 4\}$, and $s_3 = \{6, 10\}$. The last switch s_3 is a mandatory context switch. So in order to get rid of the counterexample, we can either forbid s_1 or s_2. First we investigate, whether switch s_2 is the bad switch. Therefore, we fix the execution until s_2. So first one line of thread 1 has to be executed (block(3)=1) and then two lines of thread 2 (block(8)=block(9)=2), see Listing 7. Now the algorithm checks whether it is possible to extend this incomplete execution to a complete correct one. Since this is not the case, s_2 is innocent, and the real problem lies in s_1. In the next step, the algorithm forbids s_1 by inserting an atomic section between line 3 and line 4.

Listing 6 Fix Switches Example

```
1: int g; int h = 0;
2: procedure THREAD1              7:  procedure THREAD2
3:   g = 0;          --s1-->      8:    g = 1;
4:   if g = 0 then   <--s2--      9:    if g = 1 then
5:     int tmp = h;
6:     h = tmp + 1;  --s3-->      10:     int tmp = h;
                                  11:     h = tmp + 1;
```

Listing 7 Fix Switches Example (continued)

```
1: int g; int h = 0;
2: procedure THREAD1                         7:  procedure THREAD2
3:   g = 0; block(3)=1   --s1-->             8:    g = 1;           block(8)=2
                                             9:    if g = 1 then    block(9)=2
-------------------------------------------------------------------------------
4:   if g = 0 then                           10:     int tmp = h;
5:     int tmp = h;                          11:     h = tmp + 1;
6:     h = tmp + 1;
```

D. *Finding Atomic Sections with the* AtomConstr *Algorithm*

The *Atomicity Constraint Algorithm* (AtomConstr), shown in Listing 8, is inspired by [24]. While FixSwitches added atomic sections to the set A in each iteration, AtomConstr only adds candidates for atomic sections to a set of sets $\mathcal{A}$. Initially, $\mathcal{A}$ is empty. The algorithm searches in a loop for counterexamples $\overline{s} = s_1, s_2, \ldots s_m$ that respect $\mathcal{A}$ and computes all thread switches $K = \{k_1, k_2, \ldots k_n\}$ of $\overline{s}$. The set K represents all possible ways to eliminate the

Listing 8 AtomConstr Algorithm

```
1: procedure AtomicityConstraint
2:   𝒜 := ∅
3:   while ce'(𝒜) ≠ None do
4:     s̄ := (s_1, s_2, ... s_m) := ce'(A)
5:     K := {k_1, ..., k_n} := findSwitches(s̄)
6:     𝒜 := 𝒜 ∪ {K}
7:   return hittingSet(𝒜)
```

TABLE I

EXPERIMENTAL RESULTS. *The column #Lines gives the lines of code after preprocessing with FoREnSiC. All other columns give total execution times in seconds. We used a timeout (t.o.) of 5400 seconds. The numbers in brackets give the number of iterations in the refinement loop of Fig. 1.*

		integer arithmetic		uninterpreted func.	
	#Lines	AtomConstr	FixSwitches	AtomConstr	FixSwitches
RSA	23	–	–	1.5(2)	1.26(4)
linEq _2t 1	38	0.7(2)	0.9(2)	0.6(2)	0.6(2)
linEq _2t 2	55	43(4)	42(4)	1.7(4)	1.8(4)
linEq _2t 3	70	550(6)	623(6)	3.2(6)	4.8(6)
linEq _2t 4	87	4882(8)	5320(8)	6.9(8)	8.7(8)
linEq _2t 6	121	t.o.	t.o.	21(12)	17(12)
linEq _2t 8	155	t.o.	t.o.	44(16)	42(16)
linEq _2t 10	189	t.o.	t.o.	71(20)	86(20)
linEq _2t 12	223	t.o.	t.o.	117(24)	129(24)
linEq _2t 14	257	t.o.	t.o.	186(28)	169(28)
linEq _3t 1	52	25(3)	26(3)	2.3(3)	2.1(3)
linEq _3t 2	76	t.o.	t.o.	8.2(6)	7.8(6)
linEq _3t 3	97	t.o.	t.o.	18(9)	18(9)
linEq _3t 4	121	t.o.	t.o.	42(12)	38(12)
linEq _3t 6	169	t.o.	t.o.	113(18)	106(18)
linEq _3t 8	218	t.o.	t.o.	247(24)	258(24)
linEq _3t 10	265	t.o.	t.o.	398(30)	378(30)
linEq _4t 1	66	t.o.	t.o.	7(4)	3.9(4)
linEq _4t 2	97	t.o.	t.o.	28(8)	38(8)
linEq _4t 3	124	t.o.	t.o.	89(12)	90(12)
linEq _4t 4	155	t.o.	t.o.	150(16)	169(16)
linEq _4t 6	217	t.o.	t.o.	485(24)	506(24)
VecPrime 2	157	173(836)	53(108)	2.9(16)	3.1(16)
VecPrime 3	221	471(942)	190(162)	11(24)	12(24)
VecPrime 4	290	2018(1018)	519(2016)	66(32)	69(32)
VecPrime 5	359	t.o.	1356(2070)	627(40)	514(40)
IIO	60	1.1(9)	1.3(9)	0.9(9)	1.1(9)
CVE	150	11(21)	13(21)	4.1(12)	5.8(12)
TG3	133	17(74)	21(74)	9.8(74)	13(74)

counterexample $\overline{s}$: At least one of the switches from K must be forbidden by an atomic section to make $\overline{s}$ unfeasible. In the next step, we add K to $\mathcal{A}$. The set $\mathcal{A}$ consists of sets of atomic sections candidates and from each set, at least one of the atomic section must be active to forbid the corresponding counterexample. So $\mathcal{A}$ represents a CNF formula.

A hitting set for $\mathcal{A}$ is a set A that shares at least one common element with every set in $\mathcal{A}$. If no more counterexample exists, the minimal hitting set of $\mathcal{A}$ represents a global minimum of atomic sections. One efficient way to compute a minimal hitting set is described in [22].

V. EXPERIMENTAL RESULTS

We have evaluated our approach experimentally, using a prototype implementation for concurrent C programs, called Atoss. It uses the front-end of the FoREnSiC [2] tool, which in turn uses gcc to parse the input C files. We have added a new back-end to FoREnSiC to create the SMT queries that we submit to the Z3 solver. The models returned by Z3 are the counterexamples that Atoss analyzes to create a refined SMT query, until Z3 returns UNSAT and we have found a solution. In addition to the illustrative example presented in Section III, we used two parameterized benchmarks called `linEq` and `VecPrime`, which can also be solved with integer arithmetic. This enables us to rate the effects of our abstraction with uninterpreted functions. To show that our approach is also applicable to real-world problems, we also ran Atoss on three bugs in Linux kernel modules. Our prototype implementation, all benchmarks, as well as scripts to run them are available for evaluation at http://www.iaik.tugraz.at/content/research/design_verification/atoss/.

Our experimental results are summarized in Table I. We show execution times to synthesize synchronization for each of the benchmarks, using our two different algorithms (FixSwitches and AtomConstr), comparing abstraction with uninterpreted functions and normal integer arithmetic.

The RSA example has already been explained in Section III. This benchmark can only be solved by abstraction with uninterpreted functions, as the complex arithmetic functions involved are beyond the capabilities of state-of-the-art `QF_LIA` solvers. FixSwitches finds the atomic sections one would naturally expect (lines 6–7 and 13–14; see Section III). Interestingly, AtomConstr computes a different solution of the same size: it suggests to make the lines 5–6 and 12–13 atomic. This is also correct because if each thread decides on the merge right after being finished, only the second thread to finish can enter the `if` to do the merge.

The `linEq` benchmark is based on the idea of checking whether a given n-tuple satisfies a given linear equation with n variables. Multiplications of the equation's coefficients with the elements of the n-tuple is distributed over multiple threads. This example is scalable w.r.t. two different parameters: the number of threads, and the size of n. The naming convention in Table I is as follows: "`linEq` _3t 4" has 3 threads and $n = 4$. We can see from Table I that using uninterpreted functions for abstraction significantly speeds up the synthesis process, and even enables synthesis for many benchmarks that would timeout otherwise. Concerning scalability, it should be noted that each of these benchmarks contains n times the number of threads potential race conditions. In real-world examples, we usually expect a much lower number of potential concurrency issues. These benchmarks were specifically designed to challenge the scalability of our approach.

The idea of `VecPrime` benchmark is that there is a vector of numbers, and we want to count the contained prime numbers. One thread starts counting from the "left" side of the vector, the other one starts from the "right" side. Every number that has been taken into account is set to 0. This way it is ensured that no element is counted twice.[1]

We also applied our tool to three real world examples. The first one (`linux_iio`) is based on a bug[2] found in the industrial I/O subsystem (IIO) of the linux kernel. IIO

[1] We assume that the check `isPrime(0)` is significantly faster than other calls to `isPrime`. Thus, it hardly matters for efficiency that both threads go through the entire vector, for simplicity.

[2] http://git.io/JjCEXg

polls hardware-devices for triggers, to notify consumers of events, e.g. that new data is available. A global variable counts the number of running threads. The race condition occurs, if several trigger-consumer modify this variable simultaneously.

The second example, the `CVE-2014-0196` benchmark is based on a bug[3] in a Linux kernel module, which has only been discovered very recently. Slightly simplified, a race condition could lead to an erroneous value in a variable that counts how much space remains in a buffer, which can subsequently result in a buffer overflow. This can lead to memory corruption, and exploits have been published that allow crashing the system or gaining root access. We have fed the relevant part of the kernel module's code (150 lines of code) to Atoss.

Finally, the last example (`linux_tg3`) is based on a bug[4] found in the Broadcom Tigon3 (TG3) ethernet driver. In the retrieval function for hardware statistics, the driver retrieves the statistics from the device and stores it into a buffer. Since the tg3 driver updates the hardware statistics in a non-atomic way, the state of the statistics can get inconsistent.

For all three real-world examples, we did not have to add any specification, but just relied on the implicit specification given by serializability. Within just a few seconds, Atoss was able to find a fix. For CVE and TG3, the computed fix matches the "official" fix that has been made by the kernel community. For IIO our tool found a slightly different fix.

When comparing FixSwitches with AtomConstr, there is no clear winner. Each algorithm is faster for some examples. It should be noted that both algorithms always found a globally minimal set of atomic sections for all our benchmarks.

VI. Conclusion

We have presented a new approach to synthesis of synchronization for concurrent programs. Using uninterpreted functions, we are able to efficiently abstract details of the program that are irrelevant for concurrency issues. We have shown experimentally that this abstraction is more efficient than just using integer arithmetic without any abstraction. Also, we are able to handle benchmarks that would not have been feasible at all, using integer arithmetic.

Moreover, we have demonstrated that this approach can be applied to real-world concurrency issues, such as race conditions in kernel modules. In particular, the applicability of our approach is supported even further by a very low entry barrier. We do not require designers to write a formal specification. They can simply run our tool on their code as it is, and still detect and fix concurrency issues.

Due to this encouraging results, we plan to do future work on several improvements and optimizations. First, we would like to add support for commutative and associative (yet still uninterpreted) functions, to improve the abstraction/expressibility trade-off. This should lead to a performance boost for benchmarks where the order of operations is not relevant for the final result. Second, we note that in practical examples, concurrency bugs are usually limited to a few lines of code only. Thus, we would like to be able to automatically disregard program parts that do not contain any concurrency bugs, by abstracting them with uninterpreted functions. This should improve scalability so that we could deal more easily with even larger real-world examples.

[3]https://bugzilla.redhat.com/show_bug.cgi?id=1094232

[4]http://git.io/7wWrKw

References

[1] L. Bachmair, I. V. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In *FroCoS'00*, LNCS 1794, 2000.

[2] R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC - an automatic debugging environment for C programs. In *HVC'12*, LNCS 7857. Springer, 2012.

[3] R. E. Bryant, S. M. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Log.*, 2(1):93–134, 2001.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, LNCS 818. Springer, 1994.

[5] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV'13*, LNCS 8044. Springer, 2013.

[6] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI'08*. ACM, 2008.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*. Springer, 1981.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[9] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS'04*, LNCS 2988. Springer, 2004.

[10] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, LNCS 5674. Springer, 2009.

[11] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE'11*. ACM, 2011.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, 2008.

[13] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS'03*. IEEE, 2003.

[14] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN'08*, LNCS 5156. Springer, 2008.

[15] G. Hofferek and R. Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *MEMOCODE'11*. IEEE, 2011.

[16] G. Hofferek, A. Gupta, B. Könighofer, J.-H. R. Jiang, and R. Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *FMCAD'13*. IEEE, 2013.

[17] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI'04*. AAAI Press / The MIT Press, 2004.

[18] V. Kahlon. Automatic lock insertion in concurrent programs. In *FMCAD'12*. IEEE, 2012.

[19] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[20] A. Pnueli, O. Strichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *STTT*, 2(2):192–201, 1998.

[21] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV'05*, LNCS 3576. Springer, 2005.

[22] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1), 1987.

[23] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI'08*. ACM, 2008.

[24] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL'10*. ACM, 2010.

Interpolation with Guided Refinement: revisiting incrementality in SAT-based Unbounded Model Checking

G. Cabodi, M. Palena, P. Pasini
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
Email: {gianpiero.cabodi, marco.palena, paolo.pasini}@polito.it

***Abstract*—This paper addresses model checking based on SAT solvers and Craig interpolants. We tackle major scalability problems of state-of-the-art interpolation-based approaches, and we achieve two main results: (1) a novel model checking algorithm; (2) a new and flexible way to handle an incremental representation of (over-approximated) forward reachable states. The new model checking algorithm (IGR: Interpolation with Guided Refinement), partially takes inspiration from IC3 and interpolation sequences. It bases its robustness and scalability on incremental refinement of state sets, and guided unwinding/simplification of transition relation unrollings. State sets, the central data structure of our algorithm, are incrementally refined, and they represent a valuable information to be shared among related problems, either in concurrent or sequential (multiple-engine or multiple property) execution schemes. We provide experimental data, showing that IGR extends the capability of a state-of-the-art model checker, with a specific focus on hard-to-prove properties.**

I. Introduction

Craig interpolants (ITPs for short) [1], [2], introduced by McMillan [3] in the Unbounded Model Checking (UMC) field, have shown to be effective on difficult verification instances. Though recently challenged by new techniques (IC3, Incremental Construction of Inductive Clauses for Indubitable Correctness [4]), our experience within the field of HWMCC competitions [5] and industrial co-operations shows that interpolation-based approaches still play an important role within a portfolio-based tool.

From a high-level Model-Checking perspective, Craig interpolation is an operator able to compute over-approximated images. The approach can be viewed as an iterative refinement of proof-based abstractions, to narrow down a proof to relevant facts. Over-approximations of the reachable states are computed from refutation proofs of unsatisfied BMC-like runs, in terms of AND/OR circuits, generated in linear time and space, w.r.t. the proof.

Craig interpolants most interesting features are their completeness and the automated abstraction mechanism. Whereas one of their major challenges is the inherent redundancy of interpolant circuits, as well as the need for fast and scalable techniques to compact them. Improvements over the base method [3] were proposed in [6], [7], [8], [9], [10] and [11], in order to push forward applicability and scalability of the technique.

Interpolant compaction is a potential approach that we have specifically addressed in [12]. We follow here a second track of research: alternative ITP-based traversal schemes for model checking algorithms, under the underlying purpose of incrementally computing state sets and reducing the complexity of their computation. We also follow the idea of incrementality in order to support optimal data structures for the verification of multiple properties, and for a tighter integration with counterexample- and/or proof-based [13], [14] abstraction/refinement approaches.

A. Contributions

The main contributions of this work are: (1) A novel model checking algorithm based on interpolation and characterized by: incremental computation of state sets, guided deployment and simplification of transition relation unrollings; (2) Internal optimizations to image computation, exploiting the incremental state representation; (3) A new and flexible way to compute and refine state set representations.

B. Related works

Our work is related to many recent papers on SAT-based Model Checking. Among others, let us mention that the idea of guided search and refinement is clearly present in some past BDD-based works (see for instance [15]), in IC3 [4], as well as in interpolation sequences (ITPSEQ [16], [17]). More recently, Vizel et al. [18] have proposed *Dual Approximated Reachability* (DAR), an evolution of interpolation sequences that considers mixing forward and backward reachability. Our approach takes ideas from all above works, it is based on interpolation, it computes just forward approximations of state sets, which allows us to potentially reuse them for multiple properties (or sub-properties) of the same model.

Our scheme of incremental refinement of state sets takes equal inspiration from IC3 and ITPSEQ. Compared to IC3, we represent state sets by circuits instead of clauses, and our state sets relax inductiveness constraints. Compared to interpolation sequences, though our refinement scheme is similar, we never compute an interpolation sequence from a single SAT run (and

[1]This work was supported in part by SRC contract 2012-TJ-2328.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

proof), but we activate sequences of standard interpolation and/or approximate image calls.

Many other internal details, at the level of SAT and circuit-based reasoning, take inspiration from the above, as well as other existing works. Let us mention for instance clause propagation by *pushing*, redundancy removal by subsumption, that we brought from IC3 and re-implemented on circuit-based (AIG) representations.

C. Outline

Section II introduces background notions and notation about BMC and UMC, SAT-based Craig interpolant Model Checking, IC3. Sections III, IV, V introduce our contributions. Section III discusses incremental state sets in interpolation, section IV introduces base concept on guiding cones through state sets, section V presents the overall IGR algorithm. Section VII discusses the experiments we performed. Section VIII concludes with some summarizing remarks.

II. Background

A. Model and Notation

We address systems modeled by labeled state transition structures and represented implicitly by Boolean formulas. From our standpoint, a system M is a triplet $M = (S, S_0, T)$, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is a total transition relation. The system state space is encoded with an indexed set of Boolean variables $X = \{x_1, \ldots, x_n\}$, so that a state $s \in S$ corresponds to a valuation of the variables in X, and a set of states can be represented with a Boolean formula over X. We use the primed notation (X') for the *next state* of a variable (so a transition relation is $T(X, X')$). Whenever more time frames are involved, we use a superscript notation: e.g., in circuit unrollings, we use X^i for the X variables instantiated at the i-th time frame. Support variables will be omitted for simplicity when easily guessed from the context. A literal is a Boolean variable or its negation. A clause is a disjunction of literals. A CNF formula is a conjunction of clauses. Most modern SAT solvers [19], [20] adopt clauses as their main representation and manipulation formalism for Boolean functions. Given a Boolean formula $\mathcal{F}$, whenever we need to explicitly indicate its *before/after* version, w.r.t. an evaluation (e.g., a refinement step), we use a -1 superscript for the *before* version: $\mathcal{F}^{-1}$. We will use overlined letters for arrays of functions: e.g., $\overline{\mathcal{F}} = (\mathcal{F}_0, \mathcal{F}_1, ...)$.

B. Bounded and Unbounded Model Checking

Given a sequential system M and an invariant property p, SAT-based BMC [21] is an iterative process to check the validity of p up to a given bound. To perform this task, the transition relation T is unrolled k times

$$T^k(X^{0..k}) \quad = \quad \bigwedge_{i=0}^{k-1} T(X^i, X^{i+1})$$

in order to implicitly represent all state paths of length k. BMC tools use SAT checks such as:

$$bmc^k(X^{0..k}) \quad = \quad S_0(X^0) \wedge T^k(X^{0..k}) \wedge \bigvee_{i=0}^{k} \neg p(X^i)$$

to look for counterexamples (of length $\leq k$) that start from the initial states S_0 and falsify p. The same formula can be rewritten, in a simpler form, by omitting support variables, as follows:

$$bmc^k \quad = \quad S_0 \wedge T^k \wedge \bigvee_{i=0}^{k} \neg p$$

Though BMC tools are effective at finding bugs, their verification method is not complete. Therefore, specific techniques are required in order to support Unbounded Model Checking (UMC). The ability to check reachability fix-points and/or to find inductive invariants, is thus the main difference, and additional complication, between BMC and UMC.

C. Craig Interpolants

Let A and B be two inconsistent Boolean formulas, i.e., such that $A \wedge B \equiv \bot$. An ITP I for (A, B) is a formula such that: (1) $A \Rightarrow I$, (2) $I \wedge B \equiv \bot$, and (3) $supp(I) \subseteq supp(A) \cap supp(B)$.

```
INTERPOLANTMC (S0, T, ¬p)
  k = 0
  do
    Cone^k = CONEUNROLL(¬p, T, k)
    res = FINITERUN (S0, T, Cone^k)
    k = k + 1
  while (res = undecided)
  return (res)

FINITERUN (S0, T, Cone)
  if SAT(S0 ∧ T ∧ Cone) return (reachable)
  R = S0
  while (⊤)
    Image = ITP(R ∧ T, Cone)
    if (Image = undefined)
      return (undecided)
    if (Image ⇒ R) return (unreachable)
    R = R ∨ Image
```

Fig. 1. Interpolant-based Verification.

An interpolant $I = \text{ITP}(A, B)$ can be derived, as an AND/OR circuit, from the refutation proof of $A \wedge B$. McMillan [3] proposed an effective fully SAT-based Unbounded Model Checking algorithm, exploiting interpolants, as sketched in Figure 1.

Routine FINITERUN operates a forward traversal, where interpolation is used as an over-approximate image operator. The degree of accuracy or abstraction of the operation is tied to the bound K of the $Cone^{0..k}$ transition relation unrolling. Whenever the product $(S_0 \wedge T \wedge Cone)$ is UNSAT, we say that S_0 and $Cone$ are mutually *adequate*. The function may end up with three possible results:

- *reachable*, if it proves $\neg p$ reachable in k steps, hence the property has been disproved;
- *unreachable*, if the approximate traversal using the IMG^{+}_{Adq} image computation reaches a fix-point. In this case the property is proved;
- *undecided*, if $\neg p$ intersects the over-approximate state sets. Then, k in increased for a new FINITERUN call.

Routine INTERPOLANTMC, on top of FINITERUN, loops through increasing k bound values. The previous algorithm is sound and complete [3].

D. IC3

IC3 [4] is based on incrementally refining and extending a sequence $\mathcal{F}_1, ..., \mathcal{F}_k$ of sets of reachable states (*bounded invariants*) represented by sets of clauses, under the following rules:

$$\begin{array}{lll} \mathcal{F}_0 & = & S_0 \\ \mathcal{F}_i & \Rightarrow & \mathcal{F}_{i+1},\ for 1 \leq i \leq k-1 \\ \mathcal{F}_i & \supseteq & \mathcal{F}_{i+1} \text{ as sets of clauses, for} 1 \leq i \leq k-1 \\ \mathcal{F}_i \wedge T & \Rightarrow & \mathcal{F}_{i+1}, \text{ for } 1 \leq i \leq k-1 \\ \mathcal{F}_i & \Rightarrow & p. \end{array}$$

The introduction of IC3 [4] suggested a different way to compute information about reachable states, as (unlike other ITP-based approaches) IC3 requires no unrolling of the transition relation. One of the major contributions of IC3 is an inductive reasoning, where induction is exploited under stepwise assumptions-assertions. IC3 is incremental in that it finds inductive subclauses of the negations of states. The main limitation of IC3 is the potential clause-based state set enumeration. Some interesting ideas of IC3, that partially influenced our work, are:

- the incremental representation of state sets;
- the *push* operation, that possibly re-uses clauses from inner state sets to outer ones;
- redundancy removal by subsumption.

III. INCREMENTAL STATE SETS IN ITP

In this section we describe our model of incremental state sets. Instead of directly introducing the overall IGR algorithm (see section V), we just propose here some modifications to the standard interpolation algorithm of [3], that would allow reusing and refining previously computed interpolants.

As already pointed out, incremental state sets are present in ITPSEQ [16], [17] and DAR [18]. Compared to those works, our approach, as described in the sequel, is much closer to standard interpolation. More in detail:

- we just work on approximations of forward reachable states, with no attempt to mix forward and backward state sets (as in DAR);
- we keep the standard interpolation scheme, extended by saving and reusing previously computed state sets;
- we always refine (i.e., strengthen) state sets, which does not prevent us from possibly simplifying their representation by using ad–hoc redundancy removal.

We use for state sets a notation taken from IC3 and ITPSEQ. $\overline{\mathcal{F}} = \mathcal{F}_1, ..., \mathcal{F}_k$ is a sequence of sets of reachable states represented by circuits (AIGs) instead of sets of clauses. Let $\mathcal{R}_i^E$ represent the set of states reachable in *exactly* i steps, and $\mathcal{R}_i = \cup_{j=0..i} \mathcal{R}_j^E$ the sets of all states reachable in at most i steps. $\mathcal{R}_i$ includes all previous state sets, whereas $\mathcal{R}_i^E$ does not necessarily.

Our implementation supports both versions:

$$\begin{array}{lll} \mathcal{R}_i^E & \Rightarrow & \mathcal{F}_i \\ \mathcal{R}_i & \Rightarrow & \mathcal{F}_i \end{array}$$

the choice being a user selected option[1]. On the one hand, the fully inclusive ($\mathcal{R}_i$) representation has nice properties, which are at the base of the IC3 inductive reasoning. On the other hand, state set strengthening is generally more powerful using $\mathcal{R}_i^E$. In the sequel we will assume the first (non inclusive) model. So our assumptions for the $\mathcal{F}_i$ sets are the following:

$$\begin{array}{lll} \mathcal{F}_0 & = & S_0 \\ \mathcal{F}_i(X) \wedge T(X, X') & \Rightarrow & \mathcal{F}_{i+1}(X'), \text{ for } 1 \leq i \leq k-1 \end{array}$$

In order to represent an incremental refinement of $\mathcal{F}_i$ sets, we use notation $\mathcal{F}_i^{-1}$ for denoting the version of $\mathcal{F}_i$ prior to refinement. A refinement of $\mathcal{F}_i^{-1}$ is thus the result of a strengthening step, such that: $\mathcal{F}_i \Rightarrow \mathcal{F}_i^{-1}$.

```
INCRITPMC (S_0, T, ¬p)
  k = 0
  F̄ = (S_0)
  do
    Cone^k = CONEUNROLL(¬p, T, k)
    res = INCREMENTALFINITERUN (F̄, T, Cone^k)
    k = k + 1
  while (res = undecided)
  return (res)

INCRFINITERUN (F̄, T, Cone)
  if SAT(F_0 ∧ T ∧ Cone) return (reachable)
  R = F_0
  i = 0
  while (⊤)
    if (F_{i+1} = void) F_{i+1} = ⊤
    Image = IMGREF(F_i, T, Cone, F_{i+1})
    if (Image = undefined)
      return (undecided)
    F_{i+1} = Image
    if (F_{i+1} ⇒ R) return (unreachable)
    R = R ∨ F_{i+1}
    i = i + 1

IMGREF (F_i, T, Cone, F_{i+1}^{-1})
  C = SIMPLIFY (Cone, F_{i+1}^{-1})
  Image = ITP(F_i ∧ T, C)
  if (Image = undefined) return (Image)
  return (Image ∧ SIMPLIFY (F_{i+1}^{-1}, Image)
```

Fig. 2. Interpolant-based Image with refinement.

Figure 2 shows a variant of the algorithm in Figure 1. We explicitly use $\overline{\mathcal{F}}$ to represent state sets. $\overline{\mathcal{F}}$ is initialized to an empty array, with the exception of $\mathcal{F}_0 = S_0$. The standard interpolation operator is replaced here by IMGREF. In this new operator interpolation is preceded by cone simplification, based on previously available state sets, and followed by a refinement step. Refinement is a strenghtening step, done by

[1]The $\mathcal{R}_i$ option is internally handled by properly transforming the transition relation.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

conjoining the previous set with a new term. This is done by embedding a simplification step. SIMPLIFY is based on the general notions of redundancy removal under *external* or *observability don't cares*. Strictly speaking, whenever two functions are conjoined, either one could be pre-simplified using the other one as *care*:

$$A \wedge B \quad = \quad A \wedge \text{SIMPLIFY}(B, A) \qquad (1)$$

BDDs offered nice operators (cofactor, constrain, restrict [22]) for function simplification, that have no counterpart in gate-based rapresentations. Though many redundancy removal operators have been proposed, our experience shows that most of them are too expensive (and poorly scalable). As we need a fast operator, we limit ourselves to equivalences involving state variables, exploited for simplifications based on circuit merging.

We now prove that incrementality is guaranteeing the correctness of the Model Checking procedure. The proof is based on **Theorem 1**, stating that IMGREF (including the refinement step) is returning a correct over-approximated image.

Theorem 1 IMGREF is correct $\mathcal{F}_i \wedge T \Rightarrow$ IMGREF ($\mathcal{F}_i$, T, $Cone$, $\mathcal{F}_{i+1}^{-1}$)

Proof. Let us start by the observation that both the previous $\mathcal{F}_{i+1}^{-1}$ (assumed as $\top$ if not yet available) and $Image$ in IMGREF are implied by the exact image.

$$\begin{array}{lccc} (a) & \mathcal{F}_i \wedge T & \Rightarrow & \mathcal{F}_{i+1}^{-1} \\ (b) & \mathcal{F}_i \wedge T & \Rightarrow & Image \end{array}$$

(a) is true because $\mathcal{F}_i$ is a strengthening of its previous version $\mathcal{F}_i^{-1}$ ($\mathcal{F}_i \Rightarrow \mathcal{F}_i^{-1}$), so its image implies the image of $\mathcal{F}_i^{-1}$. (b) comes from $Image$ being an interpolant. By conjoining (a) and (b), we can derive:

$$(c) \quad \mathcal{F}_i \wedge T \quad \Rightarrow \quad \mathcal{F}_{i+1}^{-1} \wedge Image$$

From the definition of the SIMPLIFY operator 1:

$$(d) \quad Image \wedge \text{SIMPLIFY}(\mathcal{F}_{i+1}^{-1}, Image) \equiv Image \wedge \mathcal{F}_{i+1}^{-1}$$

The thesis comes from combining (c) and (d).

IV. GUIDED CONE

Let us identify a refinement step as a strengthening of a state set $\mathcal{F}_{i+1}^{-1}$, such that the new version implies the previous one: ($\mathcal{F}_{i+1} \rightarrow \mathcal{F}_{i+1}^{-1}$). We describe here how incrementality can exploit the fact that any subset of adequate backward cones can be used for refinement, based on two observations: (A) convergence of the approach is guaranteed by the fact that at worst a full cone of bound equal to the diameter is eventually used (see [3]), or the full enumeration of used cone subsets could completely cover the space backward reachable from the target ($\neg p$) (see [4]); (B) performance issues require a good balance between the opposite needs, to (1) keep small cones, for easier BMC-like SAT checks, and (2) to avoid activating too many refinement steps. We also need to avoid using cones that do not help refining previously computed state sets.

Let us thus start from the observation that any (subset of a) backward cone is acceptable by a refinement step (a call to IMGREF), as the cone is not required by the proof of **Theorem 1**. Of course, no refinement ($\mathcal{F}_{i+1} = \mathcal{F}_{i+1}^{-1}$) could come from a wrongly chosen cone, leading to explosion in the number of iterations. As an extreme option, any state cube backward reachable from the target (or known not to be forward reachable) could be used, as in IC3. Though cone subsetting is an option in view of scalability, it is not a primary focus of this work. Cone partitioning and/or subsetting would obviously reduce the size and depth of BMC-like checks, whose number whould increase. In this paper limit ourselves to cone simplification and guided rewinding/unwinding, see IV-B, exploiting previously computed $\overline{\mathcal{F}^{-1}}$ whenever available in order to:

- simplify $Cone$, using available $\overline{\mathcal{F}^{-1}}$ sets (in other words, restricting cones to *go into* the known state set rings);
- drive $Cone^k$ computation to a proper k depth, i.e., the minimum required in order to produce a strengthening.

A. Cone simplification

Whenever we are computing the image of $\mathcal{F}_i$, exploiting previously computed $\mathcal{F}_j^{-1}$ ($j > i$), we can use all available $\mathcal{F}_j^{-1}$ as *care sets* for $Cone$ simplification, based on the fact that the image will be conjoined with $\mathcal{F}_{i+1}^{-1}$.

So, $Cone^k$ in IMGREF can be replaced by FSIMPLIFY($Cone^k$, $\overline{\mathcal{F}}$, $i+1$), under the constraint that:

$$\text{FSIMPLIFY}(Cone^k, \overline{\mathcal{F}^{-1}}, j) \wedge \mathcal{F}_j^{-1} \quad \equiv \quad Cone^k \wedge \mathcal{F}_j^{-1}$$

A straightforward application of the previous formula is based on the so called *latch correspondences*, i.e., couples of latches that are known to be equivalent in $\mathcal{F}_j^{-1}$. For all of them, latches can be merged in $\mathcal{F}_j^{-1}$. More formally, for each couple of state variables (x_p, x_q) such that $\mathcal{F}_j^{-1} \Rightarrow (x_p \leftrightarrow x_q)$, the substitution $x_p \rightarrow x_q$ can be done in $Cone^k$. A similar operation can be done for all latch correspondences at intermediate transition relation boundaries in $Cone$. So for any known $\mathcal{F}_l^{-1}$ ($j < l < j + k$), implied equivalences can be used to simplify $Cone$.

A proof of correctness of the above steps is omitted for conciseness.

B. Guiding cones through state sets

Whenever INCREMENTALFINITERUN hits $Cone^k$ (a possibly false counterexample) at step i, standard interpolation would expand the cone by incrementing k, possibly by more than 1, and restart a new run from the initial state $\mathcal{F}_0$. Different ideas are followed in ITPSEQ and DAR, where refinements can be triggered based on BMC-like runs with growing depth. IC3, instead, drives refinements based on a prioritized selection of backward reachable cubes. In IGR, we follow two directions, that share the common goal of potentially expanding, by adding new frames, and refining, by strengthening, $\overline{\mathcal{F}}$:

- resuming forward traversal (and state refinements) with a *smaller* cone;
- restarting a new traversal at an intermediate step, such that a strengthening of the current $\overline{\mathcal{F}}$ is guaranteed.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

1) Cone Rewinding: We call refinement sequence an iterated tail of INCREMENTALFINITERUN, that optionally resumes, after hitting $Cone^k$ at iteration i, by iteratively using cones with decreasing bounds. The operation is inspired by interpolation sequences, with a specific reference to [17].

More formally, let us assume that:

$$\mathcal{F}_i \wedge T \wedge Cone^k \not\equiv \bot$$

We could resume the forward iteration using $Cone^{k-1}$, as it is guaranteed that:

$$\mathcal{F}_i \wedge T \wedge Cone^{k-1} \equiv \bot \quad (2)$$

based on the observation that $\mathcal{F}_i \wedge Cone^k \equiv \bot$ and that $Cone^k = T \wedge Cone^{k-1}$. We could thus operate up to k iterations, until $Cone^0$, and generate or refine state sets $\mathcal{F}_{i+1}..\mathcal{F}_{i+k}$. As an alternative, one could compute an interpolation sequence, directly from a single BMC call on problem 2. As observed in [17], we prefer an iterative computation of interpolants starting from previously computed ones.

Any sub-setting of $Cone^k$, that guarantees unsatisfiability, could be selected (instead of decrementing k). E.g., if the cone is generated by a set of properties/targets, one could remove the satisfied ones, and just keep the unsat subset.

2) Cone Unwinding: Given an abstract counterexample (a cone hit) at iteration i, the *cone rewinding* strategy has the effect of refining state sets from $\mathcal{F}_{i+1}$ to $\mathcal{F}_{i+k}$. Let us now find a (minimal) unwinding of $Cone^k$ that insures to refine $\mathcal{F}_i$ and other ones at lower i values.

Starting from the observation that $bmc^{i+k} \not\equiv \bot$ (i.e., the BMC problem of depth $i+k$ is SAT) would confirm the abstract counterexample as a concrete one, and that $bmc^{i+k} = 0$ would refute it, we can iteratively produce BMC problems of increasing bound, starting from k, until we obtain UNSAT.

Let ν ($0 < \nu < i$) be the minimum cone unwinding, from $Cone^k$ to $Cone^{\nu+k}$, such that, with $j = i - \nu - 1$:

$$\mathcal{F}_j \wedge T \wedge Cone^{\nu+k} \equiv \bot \quad (3)$$

We can restart the next INCREMENTALFINITERUN from $\mathcal{F}_j$, using cone $Cone^{\nu+k}$. From a more practical point of view, we are unwinding $Cone$ in a guided way through $\mathcal{F}_j$ state sets, in order to fine the outermost one able to provide an UNSAT BMC problem (against the unwinded $Cone$).

Alternative options, for the choices of j and ν, include going to larger ν values, combined with j values such that $j < i-\nu$, and that Equation 3 is still UNSAT.

Overall, guided cone unwinding/rewinding allows us to dynamically tune unrollings. In this respect, standard interpolation is too rigid, as refinement is always done by expanding cones and using them for newly restarted traversals. ITPSEQ introduces incrementality, but with a fixed and rigid scheme. Much more flexibility is present in DAR where local and global strengthening techniques introduce the notion of refinement just when and where needed. Although backward refinement in DAR has similarities with our approach, it is based on the idea of using over-approximated backward reachable states when refining forward reachable ones. Our approach, instead, is fully based on backward cones (i.e., T unrollings), in order to represent the backward exact behavior.

V. IGR: INTERPOLATION WITH GUIDED REFINEMENT

We now describe the overall IGR model checking procedure which combines the techniques mentioned in the previous sections. Figure 3 shows the top level function IGRMC, that iteratively chooses the bound k for an unwinded cone and activates IGRFINITERUN. The latter is a variant of INCRFINITERUN, that receives as additional parameters the index i of the $\mathcal{F}_i$ state set where to start a forward traversal, and the bound k, to be used for cone unwinding. The function returns the index i_{hit} of the state set where reachability hits a cone. At each iteration, i and k are properly computed by SEEKBESTUNSAT, starting from i_{hit} and k_{hit} (related to the previous abstract counterexample) Following the *cone unwinding* strategy described in section IV-B, the cone bound k is extended, and i is decremented, until an UNSAT BMC check is obtained. As a side effect, function SEEKBESTUNSAT also detects true failures whenever the unwinded cone hits $\mathcal{F}_0$ (this check has been removed from IGRFINITERUN). The overall task of IGRMC can thus be summarized as:

- Iteratively choose a starting $\mathcal{F}_i$ set and a cone $Cone^k$, unwinded in a guided manner throughout the (abstract) $\overline{\mathcal{F}}$ sets. This is done by function SEEKBESTUNSAT;
- Start a new forward traversal (INCRFINITERUN), that is expected to refine $\overline{\mathcal{F}}$ and filter out the last (abstract) counterexemple found within the $\mathcal{F}_{i_{hit}}$ state set.

INCRFINITERUN, though heavily based on the skeleton of FINITERUN and INCRFINITERUN (its variant supporting incremental state sets), is more flexible in selecting the starting point for a traversal and the backward cone:

- Traversals start at $\mathcal{F}_i$, with i received as parameter (see IGRMC), and reachable states are initialized as the union of all $\mathcal{F}_0..\mathcal{F}_i$ state sets;
- The backward cone is not kept constant as in FINITERUN. As in INCRFINITERUN, it is simplified exploiting $\mathcal{F}$ sets at outer indexes. It is kept until an abstract counterexample is generated, or a maximum number of iterations is reached. After that, $Cone$ is rewinded by one time frame at each iteration (see section IV-B).

Convergence is tested as in all interpolation-based approaches, based on set containment. The value of variable $ForceRewind$ is assigned as a set-up parameter that heuristically controls activation of cone rewinding. Whenever $ForceRewind = 0$, rewinding is always active, so the approach obtains a minimal refinement, and it mimics the effect of interpolation sequences. High values of $ForceRewind$ keep the k value constant until a hit, a scheme much closer to standard interpolation. We empirically observed that small values are better at small sequential depths, as they can produce more light-cost refinement steps.

Figures 4 and 5 report experimental data on a case study, circuit INTEL015 from [5], that we selected among the ones

IgrMC (S_0, T, $\neg p$)
  i_{hit} = k_{hit} = 0
  $\overline{\mathcal{F}} = (S_0)$
  do
    (res, i, k) = SeekBestUnsat($\neg p$, T, i_{hit}, k_{hit})
    if (res = `reachable`)
      return (res)
    (res, i_{hit}, k_{hit}) = IgrFiniteRun ($\neg p$, T, $\overline{\mathcal{F}}$, i, k)
  while (res = `undecided`)
  return (res)

IgrFiniteRun ($\neg p$, T, $\overline{\mathcal{F}}$, i, k)
  R = $\bigcup_{l=0..i} \mathcal{F}_l$
  $rewindEnabled = \bot$
  while ($\top$)
    if ($\mathcal{F}_{i+1}$ = {}) $\mathcal{F}_{i+1}$ = $\top$
    if ($rewindEnabled \wedge k > 0$) k = k-1
    $Cone^k$ = ConeUnroll($\neg p, T, k$)
    $Cone^{\mathcal{F}}$ =FSimplify($Cone^k$, $\overline{\mathcal{F}}$, $i+1$)
    $Image$ = ImgRef($\mathcal{F}_i \wedge T, Cone^{\mathcal{F}}, \mathcal{F}_{i+1}$)
    if ($Image$ = `undefined`)
      if ($rewindEnabled$) **return** (`undecided`, i, k)
      $rewindEnabled = \top$
    else
      $\mathcal{F}_{i+1} = Image$
      if ($\mathcal{F}_{i+1} \Rightarrow$ R) **return** (`unreachable`, -, -)
      R = R $\vee$ $\mathcal{F}_{i+1}$
      $i = i + 1$
      if ($i > ForceRewind$) $rewindEnabled = \top$

SeekBestUnsat($\neg p$, T, i_{hit}, k_{hit})
  $i = i_{hit}$
  $k = k_{hit}$
  while ($i \geq 0 \wedge$ Sat($\mathcal{F}_i \wedge T \wedge$ ConeUnroll($\neg p, T, k$)))
    $i = i - 1$
    $k = k + 1$
  if ($i < 0$) **return** (`reachable`, -, -)
  return (`undecided`, i, k)

Fig. 3. Interpolation with Guided Refinement.

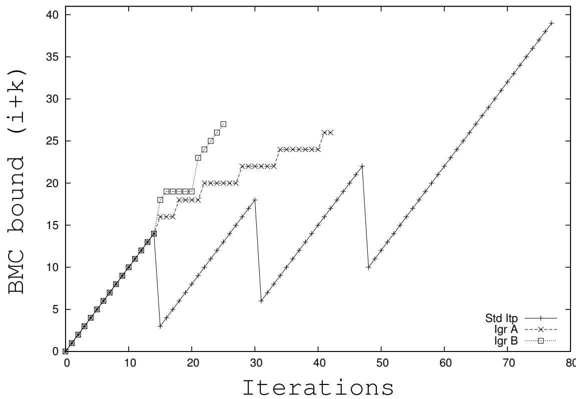


Fig. 4. BMC bound comparison in intel015, between standard interpolation and IGR in two versions: Igr A (rewind always enabled), Igr B (rewind disabled until hit).

where standard interpolation could be compared with IGR. Figure 4 plots $i + k$, the sum of state set indexes (i) and cone bounds (k). This is usually logged as an equivalent BMC bound. Iterations (on the X axis) indicate algorithm iterations (with image computation). The standard interpolation line clearly shows that BMC bounds grow linearly within FiniteRun, and they restart from the newly adjusted cone bound[2] at new FiniteRun calls. The IGR A line plots a run of IGR with cone rewinding always enabled: this means that the iterative decrease of k compensates the increase of i, keeping the BMC bound constant within IgrFiniteRun (except when we reach $k = 0$). The IGR B line plots a run of IGR with cone rewinding disabled until a BMC hit. In this case we observe an initial increase of BMC bounds, followed by a phase with constant BMC bound. Overall, IGR exploits its ability to avoid restarting from low bounds and seeking for optimal restarts, which can provide convergence at lower iteration indexes.

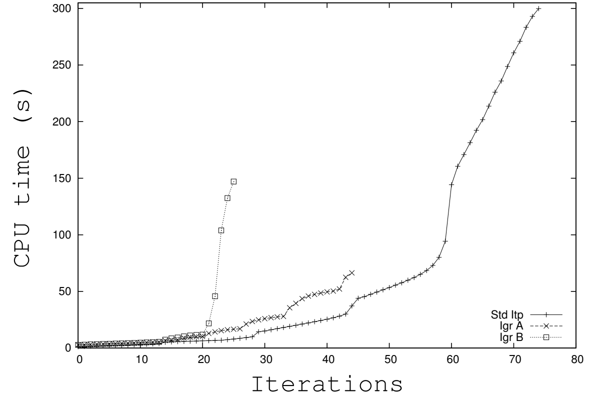


Fig. 5. CPU time comparison in intel015, between standard interpolation, Igr A and Igr B.

A comparison between IGR A and B shows that the latter can converge in fewer iterations, due to its ability to increase BMC bounds. However figure 5, that plots cumulative CPU times, shows that IGR A can be faster.

Intuitively, guided and simplified cones in IGR can produce cheaper BMC problems, as compared to standard interpolation. IGR A benefits from triggering more, but possibly simpler, refinement steps (SAT calls). Although this is a good way to avoid highly expensive BMC problems, IGR B often performs better in case of models with higher diameters (e.g., in the range of hundredths).

3) Other Implementation Issues: A few more points are worth being noticed, as having an impact on performance:

- We implemented a light weight redundancy removal procedure used for Simplify when applied to state sets, inspired by clause subsumption. Whenever a set is a conjunction of several terms, the procedure iteratively finds redundant ones through an incremental SAT formulation;
- We implemented a SAT-based procedure able to partially reuse and *push* forward components of $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$, whenever $\mathcal{F}_i$ is a conjunction. This process, which is similar to *clause pushing* in IC3, relies on an efficient incremental SAT formulation.

[2]Following [7], we heuristically increment cone bounds by more that 1, based on the depth of the previous FiniteRun run.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

VI. Laxy Abstraction and Multiple Properties

Due to the fully incremental representation chosen for reachable states, IGR can be tightly embedded in lazy abstraction, as well as multiple property verification loops.

Typical abstraction-refinement loops [13], [14] are based on the idea of looping through incremental model refinements, and restarting a new model checking problem after each new refinement step. Recent work [23] has explored a tighter integration of a model checking algorithm (IC3) within a lazy abstraction algorithm. As IGR is based on a similar data structure (the $\mathcal{F}_i$ over-approximations of forward reachable state sets), its integration within a lazy abstraction loop is straightforward: $\mathcal{F}_i$ state sets can be inherited by all refined models, as refinements can be considered as model strengthening steps. Let M^j and M^{j+1} be two abstract models (after refinement steps j and $j+1$). Let R_i^j and R_i^{j+1} be the states reachable by them in i steps. As refinement strengthens a model, $R_i^{j+1} \subseteq R_i^j$, so state set overapproximations for M^j also overapproximate states in M^{j+1}.

A similar framework can be adopted in multiple property verification, where $\mathcal{F}_i$ can be inherited and reused by all properties under check on the same model. Reusability of state sets is guaranteed here by sharing the same model over different property checks.

Though we already implemented both the above mentioned frameworks, their detailed description goes beyond the scope of this paper.

VII. Experimental Results

We implemented a prototype version of our methodology on top of the PdTRAV tool [24], a state-of-the-art verification framework. The experimental data in this section provide an evaluation of the techniques proposed, as well as a comparison with standard interpolation. Our experiments ran on a Quad-core workstation, with 2.5 GHz CPU frequency and 16 GB of main memory. We set time and memory limits to 1200 seconds and 2 GB, respectively.

We performed an extensive experimentation on a selected sub-set of publicly available benchmarks from the HWMCC'12 and HWMCC'13[5] suites. We selected them by excluding problems that PdTRAV could originally solve in less than 1 minute, and those that we could not solve with any technique (including the one presented here). It is worth noticing that all of the selected benchmarks are from industrial origin (IBM, Intel). In most cases, we operated a pre-processing using the ABC tool for combinational and/or sequential light weight optimizations, i.e., latch and signal correspondence, rewriting and refactoring. For the intel benchmarks, we also operated implicit invariant extraction and phase abstraction.

Table I provides detailed data, showing (column Best ITP) the best results we could obtain through standard interpolation, without the techniques described in this paper. Column Best IGR shows the best we could obtain with Igr, whereas column Best HWMCC shows best results attained during past HWMCCs. To this respect, it is worth noticing that time statistics from competitions were measured on a different machine, with a time limit of 900 seconds, by portfolio based (concurrent) model checkers. In the Best ITP and IGR experiments we used a single engine and we increased our time limit to 7200 seconds (2 hours), in an attempt to observe potentially difficult problems.

Table I highlights IGR as a clear winner with respect to standard interpolation, in most challenging problems. Higher runnign times in some of the easier examples simply witness some overhead for state set handling and cone winding/unwinding phases. Overall, IGR proves more scalable. The comparison with other engines is not as easy. To this respect it is worth noticing that the best model checkers at HWMCCs highly rely on aggressive transformational techniques, that seek to pre-simplify problems under various equivalence-preserving notions, before getting to Model Checking engines.

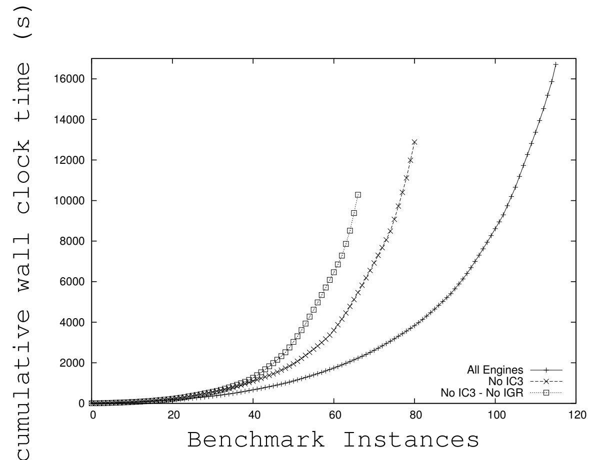


Fig. 6. Wall clock cumulative time comparison on hwmcc'12 instances solved by PdTrav (concurrent multi-engine version), with all engines active, without IC3, and without IC3 and IGR. Time limit 900 seconds per instance.

In order to gather more data, we did a second experimental evaluation of IGR, extended to the full set of HWMCC'12 (single property track, including more and easier benchmarks than HWMCC'13) benchmark instances. We repeated a competition run with 900 seconds time limit, using our multi-engine portfolio in three different setups: with the full set of engines, excluding IC3 and excluding both IC3 and IGR.

The results are plotted in figure 6, which clearly confirms IC3 as the most powerful engine. But it also shows a good impact of IGR, as a relevant contribution to the portfolio. The run with the full set of engines solved 116 problems, of which 47 were covered by IC3, and 10 by IGR. When disabling $IC3$, the overall result decreased to 81, with IGR solving 18 problems. Data also show that IGR is still not oriented to fast runs (within minutes). As seen in table I, a 2 hours timeout better shows the gain of IGR over ITP.

VIII. Conclusions

We addressed the problem of optimizing interpolants for SAT-based Unbounded Model Checking. Our main contribution is to provide a new approach, that improves over standard interpolation, by exploiting the ideas of incremental

Model				Best IGR	Best ITP	Best HWMCC	
Name	#PI	#FF	#AIG	Time	Time	Time	# of Solver
6s8	86	396	3016	835.40	-	147.82	4
6s34	77	1565	11098	2002.76	-	87.18	9
6s35	77	1571	11504	525.54	-	-	0
6s38	343	1931	10847	392.22	-	606.89	2
6s102	72	1108	7700	488.47	726.62	10.58	8
6s144	480	3337	45470	291.48	160.62	155.98	6
6s148	480	3337	45470	2011.54	1713.52	-	0
6s189	479	2436	39830	214.46	282.66	110.48	3
6s194	532	2131	13617	423.45	852.17	54.38	7
6s366r	86	1998	20560	612.28	-	-	0
6s428rb093	410	3790	29084	746.75	-	273.34	2
intel010	1111	280	10156	200.91	265.70	96.37	3
intel011	1024	273	9362	190.73	899.89	440.09	4
intel015	1024	273	9362	130.30	-	272.22	3
6s160(*)	149	559	8716	97700.21	-	-	0

TABLE I

RESULTS ON SELECTED HWMCC BENCHMARKS. COMPARING OUR BASIC VS. OPTIMIZED INTERPOLATION VERSIONS. (*) 6S160 WAS SOLVED WITHOUT TIME LIMIT, USING LAZY ABSTRACTION (STANDARD INTERPOLATION WENT OUT OF MEMORY). THE # of Solver COLUMN REPORTS HOW MANY MODEL CHECKERS SOLVED THE PROBLEM, OUT OF 21 (17) IN HWMCC'12 (HWMCC'13).

refinement and guidance through state sets. We experimentally observed that the proposed optimizations have improved both performance and scalability of our existing UMC approaches. Albeit we need to put some extra effort in a better engineering and overall integration of the proposed techniques, as well as more experimental work, we deem that current experimental data clearly witness the improvements attained.

REFERENCES

[1] W. Craig, "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.

[2] R. C. Lyndon, "An Interpolation Theorem in the Predicate Calculus," *Pacific Journal of Mathematics*, pp. 155–164, 1959.

[3] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[4] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, Austin, Texas, Jan. 2011, pp. 70–87.

[5] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc."

[6] K. L. McMillan and R. Jhala, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 39–51.

[7] J. Marques-Silva, "Improvements to the implementation of Interpolant–Based Model Checking," in *Proc. Correct Hardware Design and Verification Methods*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 367–370.

[8] V. D'Silva, M. Purandare, and D. Kroening, "Approximation Refinement for Interpolation-Based Model Checking," in *Verification, Model Checking and Abstract Interpretation*, ser. Lecture Notes in Computer Science, vol. 4905. Springer, 2008, pp. 68–82.

[9] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Boosting Interpolation with Dynamic Localized Abstraction and Redundancy Removal," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 309–340, Jan. 2008.

[10] G. Cabodi, P. Camurati, and M. Murciano, "Automated Abstraction by Incremental Refinement in Interpolant-based Model Checking," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: ACM Press, Nov. 2008, pp. 129–136.

[11] B. Li and F. Somenzi, "Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3920, 2006, pp. 227–241.

[12] G. Cabodi, C. Loiacono, and D. Vendraminetto, "Optimization techniques for Craig Interpolant compaction in Unbounded Model Checking," in *Proc. Design Automation & Test in Europe Conf.* Grenoble, France: IEEE Computer Society, Mar. 2013, pp. 1417–1422.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000, pp. 154–169.

[14] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative Abstraction using SAT-based BMC with Proof Analysis," in *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, Nov. 2003, pp. 416–423.

[15] G. Cabodi, S. Nocco, and S. Quer, "Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification," in *Proc. Computer Aided Verification*, ser. LNCS, E. Brinksma and K. G. Larsen, Eds., vol. 2102. Copenhagen, Denmark: Springer-Verlag, Jul. 2002, pp. 471–484.

[16] Y. Vizel and O. Grumberg, "Interpolation-Sequence based Model Checking," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 2517. Austin, Texas, USA: Springer, Nov. 2009, pp. 1–8.

[17] G. Cabodi, S. Nocco, and S. Quer, "Interpolation Sequences Revisited," in *Proc. Design Automation & Test in Europe Conf.* Grenoble, France: IEEE Computer Society, Mar. 2011, pp. 316–322.

[18] Y. Vizel, O. Grumberg, and S. Shoham, "Intertwined Forward-Backward Reachability Analysis Using Interpolants," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 7795. Rome, Italy: Springer, Mar. 2013, pp. 308–323.

[19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. 38th Design Automation Conf.* Las Vegas, Nevada: IEEE Computer Society, Jun. 2001.

[20] N. Eén and N. Sörensson, "The Minisat SAT Solver, http://minisat.se," Apr. 2009.

[21] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 317–320.

[22] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Sequential Machines Based on Symbolic Execution," in *Lecture Notes in Computer Science 407*, Berlin, Germany, 1989, pp. 365–373.

[23] Y. Vizel and S. S. O. Grumberg, "Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking," in *Proc. Formal Methods in Computer-Aided Design*. Cambridge, UK: IEEE, Oct. 2012, pp. 173–181.

[24] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots

Sagar Chaki
Software Engineering Institute
Email: chaki@sei.cmu.edu

Arie Gurfinkel
Software Engineering Institute
Email: arie@cmu.edu

Nishant Sinha
IBM Research
Email: nishant.sinha@in.ibm.com

***Abstract*—We verify safety properties of periodic programs, consisting of periodically activated threads scheduled preemptively based on their priorities. We develop an approach based on generating, and solving, a provably correct verification condition (VC). The VC is generated by adapting Lamport's sequential consistency to the semantics of periodic programs. Our approach is able to handle periodic programs that synchronize via two commonly used types of locks – priority ceiling protocol (PCP) locks, and CPU locks. To improve the scalability of our approach, we develop a strategy called snapshotting, which leads to VCs containing fewer redundant sub-formulas, and are therefore more easily solved by current SMT engines. We develop two types of snapshotting – SS-ALL snapshots all shared variables aggressively, while SS-MOD snapshots only modified variables. We have implemented our approach in a tool. Experiments on a benchmark of robot controllers indicate that SS-MOD is the best overall strategy, and even outperforms significantly the state-of-the art periodic program verifier prior to this work.**

I. Introduction

Periodic programs (PPs) are used frequently to control safety-critical systems. Thus, verifying safety (i.e., reachability) properties of PPs is an important problem [1]. They are inherently concurrent, and model checking them is difficult to scale. In recent years, a number of projects [2], [3], [4], [5], [6] have explored symbolic bounded model checking of multi-threaded programs (MTPs), i.e., concurrent programs with shared memory communication. Specifically, given a MTP $\mathcal{P}$ and a safety property ϕ, the approach is to verify $\mathcal{P} \models \phi$ using two steps: (i) VCGEN: generate a verification condition (VC), a formula $VC(P, \phi)$ that is satisfiable iff $\mathcal{P} \not\models \phi$; (ii) SAT: check if $VC(P, \phi)$ is satisfiable using an SMT solver. We call this approach "memory consistency based BMC" (BMC-MC), since the construction of $VC(P, \phi)$ is based on a specific memory consistency model.

A PP consists of a finite set of *tasks*, each executing in its own thread. However, a PP differs from a MTP in several verification-relevant ways. *First*, each task consists of an infinite sequence of *jobs*, activated periodically. A task's thread remains inactive between the completion of a job and the activation of the next one. *Second*, each task has a priority, that is inherited by its thread. Among all active threads, the one with the highest priority is scheduled – thus, scheduling is deterministic. Scheduling is also preemptive, a newly activated thread with higher priority preempts the currently executing one. *Third*, each task has a worst-case-execution-time (or, WCET) i.e., the maximum time between the arrival and completion of any job of the task, assuming it is not preempted. *Finally*, each task has an arrival time, i.e., the activation time of its first job.

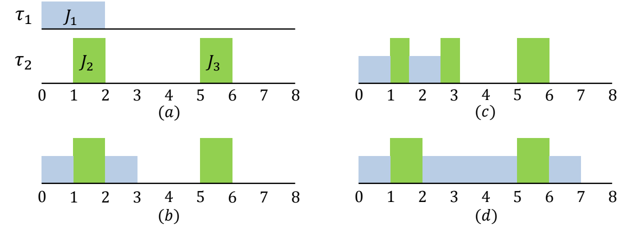


Fig. 1. (a) Example periodic program; (b) legal execution; (c,d) illegal executions; x-axis denotes time; y-axis denotes priority of executing job.

Note that, even though scheduling of a PP is deterministic, its overall behavior is non-deterministic, for two reasons. First, WCET is only an *upper-bound* on execution time. Whether a job J is preempted or not by another job J', depends on the *actual execution time* of J, which is non-deterministic. Second, we abstract away individual statement execution times, and only require that the job's WCET is not exceeded. Therefore, statements execute for a non-deterministic amount of time, and the exact *preemption location* in the control flow of J at which it is preempted by J' is non-deterministic.

We focus on "time-bounded verification" of PPs, i.e., verifying a safety property of a PP assuming it executes for time T. The time-bound fixes the number of jobs for each task, and makes the verification amenable to BMC-MC. Assuming a bound on the execution time is a useful restriction since it occurs naturally in safety-critical systems. For example, once a crash is perceived, an air bag must deploy within a time bound. Figure 1(a) shows a time-bounded PP $\mathcal{P}$ with two tasks – τ_1 and τ_2 – with priorities 1 and 2, periods 8 and 4, WCETs 2 and 1, and arrival times 0 and 1, respectively, and a time bound $T = 8$. Figure 1(b) shows a legal execution of $\mathcal{P}$. In this paper, we develop a BMC-MC approach for time-bounded verification of PPs. We address two challenges – correctness and efficiency – and perform an empirical evaluation, as discussed next.

Correctness of VCs. In current BMC-MC approaches, the construction of $VC(P, \phi)$ is based on Lamport's notion of sequential consistency [7], which we call SC-MT. However, SC-MT is *imprecise* for PPs, and cannot be used for VC generation. This imprecision arises from the combination of priority-based scheduling, WCETs, and arrival times. Consider

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

the PP $\mathcal{P}$ shown in Figure 1(a). Note that if J_2 preempts J_1, then J_2 must complete before J_1 can resume. Recall that SC-MT assumes a non-deterministic scheduler, i.e., any active non-blocked thread is allowed to execute. Thus the execution in Figure 1(c) is impossible for the $\mathcal{P}$, while it is allowed by SC-MT. Similarly, due the arrival and WCETs of τ_1 and τ_2, it is impossible for J_3 to preempt J_1. Therefore, the execution in Figure 1(d) is illegal for $\mathcal{P}$, while it is allowed by SC-MT.

Our **first contribution** is a new method to construct $VC(P, \phi)$ based on a PP-specific notion of sequential consistency. A satisfying assignment to $VC(P, \phi)$ induces an event order corresponding to a legal execution of P. Previous works [3], [5], [6] on memory-consistency based VC generation for MTPs leverage the concept of Lamport clocks [8], which are symbolic integer-valued *timestamps* associated with each program event (i.e., an access to a shared variable). These timestamps order program events in a sequentially consistent logical timeline. However, they are not sufficient to capture all legal executions of PPs. To solve this problem, we propose *hierarchical* timestamps, which not only capture the program order and the write-read ordering as before, but also take into account the priority-based preemption semantics of PPs.

Like MTPs, PPs protect access to shared variables via locking. However, unlike MTPs, locks in PPs are implemented by *altering* thread priorities. Our **second contribution** to deal with two variants of such locks – Priority Ceiling Protocol (PCP) locks [9], and CPU locks (another variant, the Priority Inheritance lock [9], is beyond the scope of this paper). When a thread acquires such a lock, its priority is raised, which disables scheduling of other threads from which the shared resource must be protected. When a thread releases a lock, its priority is reduced correspondingly. To encode such locks, we introduce priority-test-and-set (PTAS) operations, which atomically test and update the set of acquired locks. We formalize the semantics of PTAS operations, and show how to implement PCP and CPU locks using them. We also update $VC(P, \phi)$ to handle PTAS operations in a provably correct manner. Further details are presented in Section IV.

Efficiency of Encoding. As observed in the BMC-MC literature [3], [4], [5], [10], verification conditions, if constructed naively, are intractable for even state-of-the-art SMT solvers. An effective strategy for generating tractable VCs is to reduce the set of writes to a shared variable g that could be "observed" by a read of g, where a read r observes a write w if w is the most recent write to g prior to r. For PPs, we note that the observable write sets for reads in successive jobs contain many common write events from previous job instances, which leads to a severely redundant encoding. Our **third contribution** is an efficient encoding scheme for PPs which reduces the size of observation sets via the idea of *snapshots*.

A snapshot ss of g, at a location l inside a task τ, reads the current value of g in τ and then writes the same value back atomically. Thus, by introducing a new atomic *read/write pair* for g at l, ss prevents the reads in τ following l from directly observing the writes to g prior to l. Snapshotting is useful if multiple reads following l may observe the same (or largely similar) set of prior writes: multiple write events prior to l are effectively *merged* into a single write event at l. This reduces the large (quadratic) number of write-read data flows into a small (linear) number of flows, improving efficiency of the encoding. To be beneficial, snapshots must be performed for *selective* shared variables and locations. We explore two snapshotting strategies: (i) SS-ALL: all shared variables are snapshotted at the end of every job; (ii) SS-MOD: only shared variables that could be modified by a job are snapshotted at its end. Further details are presented in Section V.

Empirical Evaluation. Our **final contribution** is an implementation of our approach in a tool called LLREK, and empirical evaluation on a benchmark comprising of PPs that implement controllers for LEGO Mindstorms robots. Our results indicate that both SS-MOD and SS-ALL outperform SS-NONE, with SS-MOD being the best overall strategy. In some cases, SS-MOD is five times faster than SS-NONE. In other cases, SS-MOD completes verification successfully while SS-ALL and SS-NONE run out of memory. This work is part of an ongoing project on developing efficient software model checkers for periodic programs. We also compared LLREK with REKH [11], the most advanced PP verifier developed by the project prior to LLREK. On our benchmark, LLREK outperforms REKH significantly (in some cases by a factor of seven), and also solves many instances for which REKH runs out of memory. Further details are presented in Section VI.

Related Work. There is a large body of work in verification of logical properties of both sequential and concurrent software (see [12] for a survey). However, these techniques abstract away time completely, by assuming a non-deterministic scheduler model. In contrast, we focus on periodic programs where scheduling is non-deterministic, and influenced by both thread priorities and timing.

A number of projects [13], [14] verify timed properties of systems using discrete-time [15] or real-time [16] semantics by abstracting away data- and control-flow completely. Instead, we focus on the verification of real implementations of periodic programs, and do not abstract data- and control-flow.

Verification of multi-threaded programs via BMC-MC [3], [4], [5], [6] has also been studied by several researchers. However, previous methods focus on constructing VCs for MTPs. These methods are incorrect for PPs, as argued earlier. The purpose of snapshotting is orthogonal to that of interference abstraction (IA) [5], commonly used in BMC-MC. IA assigns symbolic values to existing reads to decouple them from writes, while snapshotting introduces new symbolic reads to *merge* data flows arising from multiple writes on a shared variable into a single read/write unit. Merging allows the reads in the following program fragment to observe a single data source as opposed to a large number, thus improving the efficiency of the symbolic encoding significantly.

Florian et al. [1] extend the explicit-state model checker SPIN to verify periodic programs written in PROMELA. Our focus is on the verification of periodic programs at the source code level using BMC-MC, which is a symbolic approach.

Time-bounded verification of PPs via sequentialization was

proposed by Chaki et al. [17], and later extended to be compositional [11]. However, sequentialization-based methods for MTPs [18], [19], [20] typically rely on modeling context switches (preemptions) for thread interleavings instead of exploiting memory consistency of read/writes. Sequentialization has also been applied iteratively to verify PPs with priority inheritance locks [21]. It is possible to extend the approach in this paper in a similar manner, but this requires a non-trivial modification to the encoding. Kidd et al. [22] have applied sequentialization to verify PPs, by using function calls to model preemptions. Our encoding relies on memory consistency, and does not model preemptions explicitly. Finally, applying naive concurrency (i.e., MTP) verification to PPs result in virtually 100% of false positives, as explored in prior work [17], [11].

The rest of the paper is organized as follows. In Section II, we present basic concepts and notation. In Section III we present our basic construction of $VC(P, \phi)$. In Section IV, we show how to augment $VC(P, \phi)$ to encode PCP and CPU locks. In Section V we present snapshotting, and its two variants. In Section VI we present our empirical evaluation, and in Section VII, we conclude.

II. Preliminaries

We assume an universe bounded by time T. A task τ is a 5-tuple $(\mathtt{J}, \pi, P, C, A)$ where: (i) π is its priority; (ii) P is its period; (iii) $\mathtt{J}$ is a sequence of $\frac{T}{P}$ jobs; (iv) $C > 0$ is its WCET; and (v) $A \geq 0$ is its arrival time. A periodic program $\mathcal{P}$ is a finite sequence of tasks. Consider a PP $\mathcal{P} = \langle \tau_1, \dots, \tau_n \rangle$ such that $\tau_i = (\mathtt{J}_i, \pi_i, P_i, C_i, A_i)$. We write $J_{i,j}$ to mean the the j-th job of the i-th task, i.e., $\mathtt{J}_i = \langle J_{i,1}, \dots, J_{i,|\mathtt{J}_i|} \rangle$. We assume that tasks have: (i) distinct and mutually disjoint jobs, i.e., $(i,j) \neq (i',j') \implies J_{i,j} \neq J_{i',j'}$; and (ii) distinct priorities $i \neq i' \implies \pi_i \neq \pi_{i'}$. Let RT_i be the response time of τ_i, i.e., the time required by any job of τ_i to complete, assuming maximal preemption by other tasks. Note that RT_i is statically computable via Rate-Monotonic Analysis [23]. We assume that the first job of τ_i always completes before time P_i, i.e., $A_i + RT_i \leq P_i$. It can be shown that $RT_i \geq C_i$, which implies that $RT_i > 0$ and $P_i > 0$.

Job Orderings. Let $\mathcal{J}$ be the set of all jobs. We define two relations $\sqsubset$ (*finishes-before*) and $\uparrow$ (*may preempt*) over $\mathcal{J}$ to characterize the order between jobs. Each job $J_{i,j}$ has a priority $\pi(J_{i,j}) = \pi_i$, arrival time $A(J_{i,j}) = A_i + (j-1) \times P_i$, and departure time $D(J_{i,j}) = A(J_{i,j}) + RT_i$. Then:

$$J_1 \sqsubset J_2 \iff \begin{array}{c}(\pi(J_1) \leq \pi(J_2) \wedge D(J_1) \leq A(J_2)) \vee \\ (\pi(J_1) > \pi(J_2) \wedge A(J_1) \leq A(J_2))\end{array} \quad (1)$$

$$J_1 \uparrow J_2 \iff \pi(J_1) < \pi(J_2) \wedge A(J_1) < A(J_2) < D(J_1) \quad (2)$$

Note that $J_1 \sqsubset J_2$ means that J_1 always completes before J_2 starts, and $J_1 \uparrow J_2$ means that it is possible for J_1 to be preempted by J_2. Since $RT_i \leq P_i$, earlier jobs of a task always finish before later jobs of the same task, i.e., $\forall i \in [1,n] \centerdot \forall 1 \leq j < j' \leq |\mathtt{J}_i| \centerdot J_{i,j} \sqsubset J_{i,j'}$. Also $A(J) < D(J)$.

States and Events of PPs. We assume a denumerable set G of $\mathbb{D}$-valued shared variables; $\mathbb{D}$ contains a distinguished value $\perp$. Function $\mathcal{I} : G \mapsto \mathbb{D}$ maps shared variables to their initial values. Let $\mathbb{Z}$ be the set of integers. An action α is a 4-tuple (J, pc, η, g) and an event ϵ is a pair (α, v) such that $J \in \mathcal{J}$, $pc \in \mathbb{Z}$, $\eta \in \{r, w\}$, $g \in G$ and $v \in \mathbb{D}$. Let $\mathtt{J}(\alpha) = \mathtt{J}(\epsilon) = J$, $\pi(\alpha) = \pi(\epsilon) = \pi(J)$, $\eta(\alpha) = \eta(\epsilon) = \eta$, $g(\alpha) = g(\epsilon) = g$, and $v(\epsilon) = v$. Events $((J, pc, r, g), v)$ and $((J, pc, w, g), v)$ denote, respectively, that value v is read from and written to variable g by job J at location pc.

Action $(J, \triangleright)$ and event $((J, \triangleright), \perp)$ denote start of job J. For $\alpha = (J, \triangleright)$, and $\epsilon = ((J, \triangleright), \perp)$, $\mathtt{J}(\alpha) = \mathtt{J}(\epsilon) = J$, $\pi(\alpha) = \pi(\epsilon) = \pi(J)$, $\eta(\alpha) = \eta(\epsilon) = \triangleright$. Similarly, action $(J, \triangleleft)$ and event $((J, \triangleleft), \perp)$ denote termination of job J. For $\alpha = (J, \triangleleft)$, and $\epsilon = ((J, \triangleleft), \perp)$, $\mathtt{J}(\alpha) = \mathtt{J}(\epsilon) = J$, $\pi(\alpha) = \pi(\epsilon) = \pi(J)$, $\eta(\alpha) = \eta(\epsilon) = \triangleleft$.

Note that we use different fonts for J to denote different things. In general, J (or J_x) denotes a specific job, $\mathtt{J}$ (or $\mathtt{J}_x$) denotes a set of jobs, while $\mathtt{J}(\cdot)$ is a function that maps actions and events to their corresponding jobs.

Job Alphabet and Program Order. Each job J has an alphabet of read actions $\Sigma_r(J) \subseteq \{J\} \times \mathbb{Z} \times \{r\} \times G$, and write actions $\Sigma_w(J) \subseteq \{J\} \times \mathbb{Z} \times \{w\} \times G$. Let $\Sigma(J) = \Sigma_r(J) \cup \Sigma_w(J) \cup \{(J, \triangleright), (J, \triangleleft)\}$. Let $PO(J)$ be a partial order over $\Sigma(J)$, representing control flow. We write $\alpha \xrightarrow{J} \alpha'$ to mean $(\alpha, \alpha') \in PO(J)$. Thus, $\forall \alpha \in \Sigma_r(J) \cup \Sigma_w(J) \centerdot (J, \triangleright) \xrightarrow{J} \alpha \xrightarrow{J} (J, \triangleleft)$. Let $\mathbb{J}$ be a linearization of $\Sigma(J)$ consistent with $PO(J)$, and $\iota(\alpha)$ be the index of α in $\mathbb{J}$. In particular, $\iota(J, \triangleright) = 1$, and $\iota(J, \triangleleft) = |\Sigma(J)|$.

Timed Event Sequences. The valid executions of periodic programs are characterized by timed event sequences (TES). Formally, a TES is a sequence $\langle (\epsilon_1, t_1), \dots, (\epsilon_k, t_k) \rangle$ where ϵ_i is an event, and t_i is a real-valued timestamp. For TESs e_1 and e_2, $e_1 \oplus e_2$ is the set of TESs obtained via their arbitrary interleaving, and $e_1 \odot e_2$ is their concatenation. Operations $\oplus$ and $\odot$ extend naturally to sets of TESs. Let $PriorWr(e, i)$ be the indices of events in e prior to ϵ_i that write to $g(\epsilon_i)$, i.e., $PriorWr(e, i) = \{j \in [1, i) \mid \eta(\epsilon_j) = w \wedge g(\epsilon_j) = g(\epsilon_i)\}$. Then, $LastWr(e, i)$ is last value written to $g(\epsilon_i)$ prior to ϵ_i, or $\mathcal{I}(g(\epsilon_i))$ if no such write exists, i.e., if $PriorWr(e, i) = \emptyset$ then $LastWr(e, i) = \mathcal{I}(g(\epsilon_i))$ else $LastWr(e, i) = v(\epsilon_m)$ where $m = \max(PriorWr(e, i))$.

Job Semantics. The semantics of J, denoted $[\![J]\!]$, is a set of TESs. Formally, $\langle (\epsilon_1, t_1), \dots, (\epsilon_k, t_k) \rangle \in [\![J]\!]$ if: (i) $\forall i \in [1, k] \centerdot \mathtt{J}(\epsilon_i) = J$; (ii) $A(J) \leq t_1 < t_2 < \dots < t_k \leq D(J)$; and (iii) if $\forall i \in [1, k] \centerdot \epsilon_i = (\alpha_i, v_i)$, then the sequence of actions $\langle \alpha_1, \dots, \alpha_k \rangle$ respects the program order $PO(J)$, i.e., $\alpha_1 = (J, \triangleright)$, $\alpha_k = (J, \triangleleft)$, and $\forall i \in [1, k) \centerdot \alpha_i \xrightarrow{J} \alpha_{i+1}$. For example, suppose the body of job J_2 from our running example is described by the control-flow-graph shown in Figure 2(c). Then $[\![J_2]\!]$ contains all TESs of the form $\langle (((J_2, \triangleright), \perp), t_1), (((J_2, 1, r, g), v_1), t_2), (((J_2, pc, w, g), v_2), t_3), (((J_2, \triangleleft), \perp), t_4) \rangle$ such that: (i) $1 \leq t_1 < t_2 < t_3 < t_4 \leq 2$; (ii) $(v_1 < 0 \wedge pc = 3 \wedge v_2 = v_1 + 7) \vee (v_1 \geq 0 \wedge pc = 2 \wedge v_2 = v_1 \times 5)$.

Task Semantics. The semantics of τ_i, denoted $[\![\tau_i]\!]$, is the set of TESs: $\bigodot_{j=1}^{|\mathtt{J}_i|} [\![J_{i,j}]\!]$. Thus, each execution of τ_i is a concatenation of an execution from each of its jobs. The

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

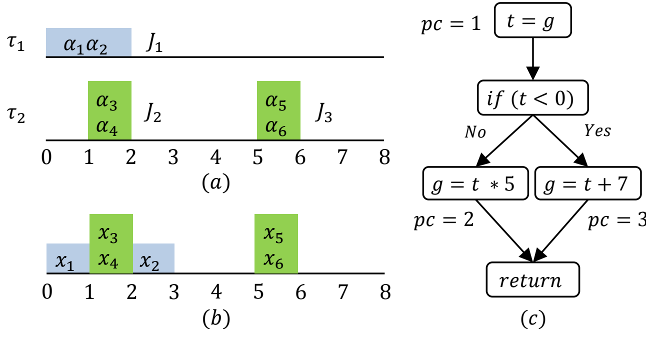


Fig. 2. (a) Periodic program; (b) Execution; (c) Control-Flow Graph.

semantics of $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]$, is also a set of TESs. Formally, $e = \langle(\epsilon_1, t_1), \ldots, (\epsilon_k, t_k)\rangle \in [\![\mathcal{P}]\!]$ if:

$$(a)\ e \in \bigoplus_{i=1}^{n} [\![\tau_i]\!] \qquad (b)\ \forall i \in [1,k) \centerdot t_i < t_{i+1} \tag{3}$$

$$\forall 1 \le i < j \le k \centerdot \neg(\mathsf{J}(\epsilon_j) \sqsubset \mathsf{J}(\epsilon_i)) \tag{4}$$

$$\forall 1 \le i \le j \le h \le k \centerdot \mathsf{J}(\epsilon_i) = \mathsf{J}(\epsilon_h) \implies \pi(\epsilon_i) \le \pi(\epsilon_j) \tag{5}$$

$$\forall i \in [1,k] \centerdot \eta(\epsilon_i) = r \implies v(\epsilon_i) = \mathit{LastWr}(e,i) \tag{6}$$

Informally, (3) states that e is an interleaving of executions of tasks in $\mathcal{P}$ with non-decreasing timestamps; (4) enforces job ordering; (5) enforces priority based preemptive scheduling; and (6) states that the last written value is always read.

III. VC Generation for Periodic Programs

Hierarchical Clock. The concept of a hierarchical Lamport clock is fundamental to our VCGen algorithm. To understand this idea, consider the PP shown in Figure 2(a). It is the same as in Figure 1(a), except that we have added actions, with program ordering, to the jobs. Specifically $\Sigma(J_1) = \{\alpha_1, \alpha_2\}$, $\Sigma(J_2) = \{\alpha_3, \alpha_4\}$, and $\Sigma(J_3) = \{\alpha_5, \alpha_6\}$, with program order $\alpha_1 \overset{J_1}{\to} \alpha_2$, $\alpha_3 \overset{J_2}{\to} \alpha_4$, and $\alpha_5 \overset{J_3}{\to} \alpha_6$. Now consider a legal execution of the PP shown in Figure 2(b), where $\forall i \in [1,6] \centerdot x_i = ((\alpha_i, v_i), t_i)$. Let $\mathsf{R}(e,i)$, be the number of jobs ending before x_i. Let $\bowtie \in \{<,>\}$. Then, we observe for each (x_i, x_j):

1) If $\mathsf{R}(e,i) \bowtie \mathsf{R}(e,j)$, then $t_i \bowtie t_j$. Example pairs are (x_4, x_2) and (x_2, x_5).
2) If $\mathsf{R}(e,i) = \mathsf{R}(e,j) \wedge \pi(\alpha_i) \bowtie \pi(\alpha_j)$, then $t_i \bowtie t_j$. An example is (x_1, x_3).
3) If $\mathsf{R}(e,i) = \mathsf{R}(e,j) \wedge \pi(\alpha_i) = \pi(\alpha_j)$ (note this implies $\mathsf{J}(\alpha_i) = \mathsf{J}(\alpha_j)$), but $\iota(\alpha_i) \bowtie \iota(\alpha_j)$, then $t_i \bowtie t_j$. Example pairs are (x_3, x_4) and (x_5, x_6).

The above observations imply that, for the TES in Figure 2(b), the ordering of x_i's by their timestamps t_i's equals their lexicographic ordering by the tuple $(\mathsf{R}(e,i), \pi(\alpha_i), \iota(\alpha_i))$. Thus, $(\mathsf{R}(e,i), \pi(\alpha_i), \iota(\alpha_i))$ is a logical representation of the timestamp t_i of event (α_i, v_i). Our key insight is that this holds for arbitrary PPs and their legal executions. In the rest of this section, we formalize this insight, use it to construct the VC for an arbitrary PP, and prove its correctness.

VCGen for Jobs. We assume that for any job J, there exists a bit-vector logic formula $VC(J)$ over the set of predicates $En(J) = \{En(\alpha) \mid \alpha \in \Sigma(J)\}$, and terms $V(J) = \{V(\alpha) \mid \alpha \in \Sigma_r(J) \cup \Sigma_w(J)\}$ such that the following holds.

Fact 1 (Job Verification Condition). *For any* $\{\alpha_1, \ldots, \alpha_k\} \subseteq \Sigma(J)$*, and sequence* $\langle v_1, \ldots, v_k\rangle \in \mathbb{D}^k$*, the formula* $VC(J) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ *is satisfiable iff* $\forall A(J) \le t_1 < \cdots < t_k \le D(J) \centerdot \langle((\alpha_1, v_1), t_1), \ldots, ((\alpha_k, v_k), t_k)\rangle \in [\![J]\!]$.

Thus, every satisfying assignment of $VC(J) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ corresponds to a legal execution of J. If J is a C function – without unbounded loops, recursion and dynamic memory – $VC(J)$ can be constructed polynomially [24]. The VC of $\mathcal{P}$ is also a bit-vector formula, and consists of three sub-VCs: (i) VC_{seq} captures the thread local behavior of each task; (ii) VC_{clk} orders events into a total order along a logical timeline; and (iii) VC_{obs} relates the read and write events on shared variables so that they are sequentially consistent. Formally,

$$VC(\mathcal{P}) = VC_{seq} \wedge VC_{clk} \wedge VC_{obs} \quad \text{, where} \tag{7}$$

$$VC_{seq} = \bigwedge_{J \in \mathcal{J}} VC(J) \tag{8}$$

and VC_{clk} and VC_{obs} are presented below. In the following, Σ_r denotes $\bigcup_{J\in\mathcal{J}} \Sigma_r(J)$, Σ_w denotes $\bigcup_{J\in\mathcal{J}} \Sigma_w(J)$, and Σ denotes $\bigcup_{J\in\mathcal{J}} \Sigma(J)$. All terms have bit-vector type.

The Clock VC: VC_{clk}**.** For each $\alpha \in \Sigma$, let term $R(\alpha)$ denote the round of α. Following our intuition, we write $\kappa(\alpha)$ to mean $(R(\alpha), \pi(\alpha), \iota(\alpha))$, i.e., the symbolic timestamp of α. During VC construction, we can now use the predicate $\kappa(\cdot)$ to order events in a periodic program, akin to the way *happens-before* predicate is used for non-periodic, multi-threaded programs [7].

For each job J, we introduce two terms: $SR(J)$ and $ER(J)$, to represent, respectively, the earliest (i.e., start) and latest (i.e., end) round of J's execution, in which any action in $\Sigma(J)$ may occur. Then, VC_{clk} is a conjunction of the following:

$$\begin{array}{ll}(a) & \bigwedge_{J\in\mathcal{J}} \bigwedge_{\alpha\in\Sigma(J)} (SR(J) \le R(\alpha) \le ER(J)) \\ (b) & \bigwedge_{J_1 \sqsubset J_2} ER(J_1) < SR(J_2)\end{array} \tag{9}$$

$$\bigwedge_{J_1 \uparrow J_2} \bigwedge_{\alpha\in\Sigma(J_1)} R(\alpha) \le SR(J_2) \vee R(\alpha) > ER(J_2) \tag{10}$$

Informally, (9)(a) asserts that actions respect starting and ending rounds; (9)(b) asserts that if J_1 finishes before J_2 starts then the ending round of J_1 must be less than the starting round of J_2; (10) asserts that if J_1 could be preempted by J_2, then it cannot execute while J_2 is active.

The Observation VC: VC_{obs}**.** For a read action $\alpha_r \in \Sigma_r$, let $\mathcal{W}(\alpha_r)$ be the set of write actions that α_r may observe, i.e., the set of writes to variable $g(\alpha_r)$ belonging to jobs that do not start after $\mathsf{J}(\alpha_r)$ ends. Formally:

$$\mathcal{W}(\alpha_r) = \{\alpha_w \in \Sigma_w \mid g(\alpha_w) = g(\alpha_r) \wedge \neg(\mathsf{J}(\alpha_r) \sqsubset \mathsf{J}(\alpha_w))\} \tag{11}$$

For each $\alpha_r \in \Sigma_r$, we introduce three additional variables $\tilde{R}(\alpha_r)$, $\tilde{\pi}(\alpha_r)$, and $\tilde{\iota}(\alpha_r)$. In essence, $(\tilde{R}(\alpha_r), \tilde{\pi}(\alpha_r), \tilde{\iota}(\alpha_r))$

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

denotes the symbolic clock of the write action observed by α_r, and is denoted by $\tilde{\kappa}(\alpha_r)$. Let $\alpha \prec \alpha'$ denote α *happens before* α', i.e., $\alpha \prec \alpha' = En(\alpha) \wedge \kappa(\alpha) < \kappa(\alpha')$. Then VC_{obs} is a conjunction of the following for each read action $\alpha_r \in \Sigma_r$:

$$En(\alpha_r) \Rightarrow \left(\bigwedge_{\alpha_w \in \mathcal{W}(\alpha_r)} \alpha_w \prec \alpha_r \Rightarrow \kappa(\alpha_w) \leq \tilde{\kappa}(\alpha_r) \right) \quad (12)$$

$$En(\alpha_r) \Rightarrow \left(VC^1_{obs} \vee \bigvee_{\alpha_w \in \mathcal{W}(\alpha_r)} VC^2_{obs}(\alpha_w) \right), \text{ where} \quad (13)$$

$$VC^1_{obs} = \left(\bigwedge_{\alpha \in \mathcal{W}(\alpha_r)} \alpha \not\prec \alpha_r \right) \wedge (\mathcal{I}(g(\alpha_r)) = V(\alpha_r)) \quad (14)$$

$$VC^2_{obs}(\alpha) = \alpha \prec \alpha_r \wedge \kappa(\alpha) = \tilde{\kappa}(\alpha_r) \wedge V(\alpha) = V(\alpha_r) \quad (15)$$

Note that (12) asserts that write action observed by α_r must have executed prior to α_r and no later than any write action to the same shared variable; (13)–(15) asserts that α_r reads the value written by the write action its observes. Thus, VC_{obs} is essentially an encoding of (6).

Correctness. The correctness of $VC(\mathcal{P})$ is expressed by Theorem 1, which states essentially that every satisfying assignment of $VC(\mathcal{P}) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ corresponds to a legal execution of $\mathcal{P}$. For brevity, we defer the proof to the extended version [25] of the paper.

Theorem 1. *For any set of actions* $\{\alpha_1, \ldots, \alpha_k\} \subseteq \Sigma$*, and sequence of values* $\langle v_1, \ldots, v_k \rangle \in \mathbb{D}^k$*, the formula* $VC(\mathcal{P}) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ *is satisfiable iff* $\exists t_1, \ldots, t_k \centerdot \langle ((\alpha_1, v_1), t_1), \ldots, ((\alpha_k, v_k), t_k) \rangle \in [\![\mathcal{P}]\!]$.

Constructing $VC(\mathcal{P}, \phi)$. To check a property ϕ for $\mathcal{P}$, let us assume that $\mathcal{P}$ is augmented with an action $\alpha(\phi)$ such that $\mathcal{P} \models \phi$ iff no TES in $[\![\mathcal{P}]\!]$ contains the event $(\alpha(\phi), v)$ for some value v. Then, from Theorem 1, $\mathcal{P} \models \phi \iff VC(\mathcal{P}) \wedge En(\alpha(\phi))$ is unsatisfiable. Thus, $VC(\mathcal{P}, \phi) = VC(\mathcal{P}) \wedge En(\alpha(\phi))$.

IV. HANDLING LOCKS

In this section, we extend VC generation to handle acquiring and releasing of locks. We consider PPs with two kinds of locks – priority ceiling protocol (PCP) locks and CPU locks. Each PCP lock l is associated with a priority level $\pi(l)$. Acquiring l disables scheduling any task whose priority is less than $\pi(l)$. Thus, a job is executed iff it is active and its priority is higher than all other active jobs, as well as those of all PCP locks held. A CPU lock disables scheduling altogether. In the rest, we only deal with PCP locks since a CPU lock is equivalent to a PCP lock l such that $\pi(l)$ is greater than the largest task priority.

To formalize PCP locks, we introduce atomic *priority-test-and-set* (PTAS) actions. Let $\mathcal{L}$ be the set of all PCP locks. For $L \subseteq \mathcal{L}$, let $\pi(L) = \{\pi(l) \mid l \in L\}$. Formally, a PTAS action is a 5-tuple (J, pc, π_t, L_r, L_a) such that $J \in \mathcal{J}$, $pc \in \mathbb{Z}$, π_t is a priority value, $L_r \subseteq \mathcal{L}$, and $L_a \subseteq \mathcal{L}$. A PTAS event ϵ is a pair (α, L_h) such that α is a PTAS action, and $L_h \subseteq \mathcal{L}$. Informally, L_h denotes the set of locks held after ϵ occurs. PTAS actions restrict the set of legal executions of a PP. Specifically, whenever, a PTAS action (J, pc, π_t, L_r, L_a) appears on an execution, the following holds: (i) *test*: all currently held PCP locks have priority less than π_t; and (ii) *set*: locks in L_r are released, locks in L_a are acquired.

Modeling Locks. Let $\Sigma_p(J)$ be the set of PTAS actions in $\Sigma(J)$. Formally, $\Sigma_p(J) = \{sched(J)\} \cup \bigcup_{l \in \mathcal{L}}(lock(J, l) \cup unlock(J, l))$, where: $sched(J) = (J, 0, \pi(J), \emptyset, \emptyset)$, $lock(J, l) \subseteq \{(J, pc, \max(\pi(\mathcal{L})) + 1, \emptyset, \{l\}) \mid pc \in \mathbb{Z}\}$, and $unlock(J, l) \subseteq \{(J, pc, \max(\pi(\mathcal{L})) + 1, \{l\}, \emptyset) \mid pc \in \mathbb{Z}\}$. Action $sched(J)$ denotes the scheduling of J for the first time. Actions in $lock(J, l)$ and $unlock(J, l)$ are used, respectively, to acquire and release lock l. Program order $PO(J)$ satisfies:

$$\forall \alpha \in \Sigma(J) \setminus \{sched(J), (J, \rhd)\} \centerdot (J, \rhd) \xrightarrow{J} sched(J) \xrightarrow{J} \alpha \quad (16)$$

Note that this means on any execution of J, $sched(J)$ appears before every other action in $\Sigma(J)$, except for $(J, \rhd)$. Every TES $e \in [\![\mathcal{P}]\!]$ also satisfies the following condition. Let there be k PTAS events in e, and $\tilde{\epsilon}_i = ((J^i, pc^i, \pi^i_t, L^i_r, L^i_a), L^i_h)$ be the i-th PTAS event in e. Then:

$$L^1_h = L^1_a \quad \bigwedge \quad \forall i \in (1, k] \centerdot L^i_h = L^{i-1}_h \setminus L^i_r \cup L^i_a \quad (17)$$

$$\forall i \in (1, k] \centerdot \max(\pi(L^{i-1}_h)) < \pi^i_t \quad (18)$$

Note that (16)–(18) imply that J is scheduled only if the priority of J is higher than all PCP locks held. The CPU lock has priority $\max(\{\pi(J) \mid J \in \mathcal{J}\}) + 1$.

Updated Construction of $VC(\mathcal{P})$. Let $\Sigma_p = \bigcup_{J \in \mathcal{J}} \Sigma_p(J)$. When constructing VC_{seq}, we treat each $\alpha \in \Sigma_p$ as a *NOP*. The construction of VC_{clk} uses the augmented $\Sigma(J)$ containing the additional PTAS actions. The construction of VC_{obs} is updated as follows. For each $\alpha \in \Sigma_p$, we add the following terms: $R(\alpha)$, $\tilde{R}(\alpha)$, $\tilde{\pi}(\alpha)$, $\tilde{\iota}(\alpha)$, and $V(\alpha)$. Their meaning is the same as for other events, except that $V(\alpha)$ now represents the set of PCP locks held after α occurs. Also, we define $\mathcal{W}(\alpha)$, i.e., the set of actions that α may observe, to contain all other PTAS actions belonging to jobs that do not start after $\mathtt{J}(\alpha)$ finishes. Formally:

$$\mathcal{W}(\alpha) = \{\alpha' \in \Sigma_p \mid \alpha' \neq \alpha \wedge \neg(\mathtt{J}(\alpha) \sqsubset \mathtt{J}(\alpha'))\} \quad (19)$$

Then VC_{obs} contains the following additional constraints for each $\alpha = (J, pc, \pi_t, L_r, L_a) \in \Sigma_p$:

$$En(\alpha) \implies \left(\bigwedge_{\alpha' \in \mathcal{W}(\alpha)} \alpha' \prec \alpha \implies \kappa(\alpha') \leq \tilde{\kappa}(\alpha) \right) \quad (20)$$

$$En(\alpha) \implies \left(VC^3_{obs} \vee \bigvee_{\alpha' \in \mathcal{W}(\alpha)} VC^4_{obs}(\alpha') \right), \text{ where} \quad (21)$$

$$VC^3_{obs} = \left(\bigwedge_{\alpha' \in \mathcal{W}(\alpha)} \alpha' \not\prec \alpha \right) \wedge (V(\alpha) = \pi(L_a)) \quad (22)$$

$$VC^4_{obs}(\alpha') = \left(\begin{array}{c} \alpha' \prec \alpha \wedge \kappa(\alpha') = \tilde{\kappa}(\alpha) \wedge \\ \max(\pi(V(\alpha'))) < \pi_t \wedge \\ V(\alpha) = V(\alpha') \setminus L_r \cup L_a \end{array} \right) \quad (23)$$

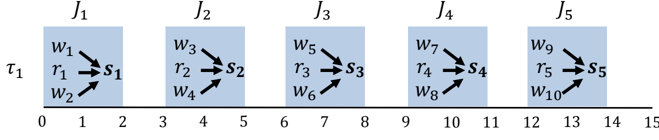


Fig. 3. Example periodic program to illustrate snapshotting.

Note that (20)–(23) assert that the PTAS action observed by α must be the last PTAS action that executed prior to α; (21)–(23) further asserts the semantics of PTAS actions is respected. Thus, (20)–(23) encode (18). We claim that Theorem 1 is valid even for the new $VC(\mathcal{P})$. The proof of this claim is in the extended version [25] of the paper.

V. Snapshotting Shared Variables

In this section, we present snapshotting of shared variables. To understand what snapshotting is, and why it is important, consider the PP in Figure 3. It consists of 1 task τ_1 with 5 jobs $J_1, \dots, J_5$. Consider initially only the read and write actions $r_1, \dots, r_5, w_1, \dots, w_{10}$, and for each read, the set of writes it may observe. Then, we have: $\mathcal{W}(r_1) = \{w_1, w_2\}, \mathcal{W}(r_2) = \{w_1, \dots, w_4\}, \dots, \mathcal{W}(r_5) = \{w_1, \dots, w_{10}\}$. In general, $\mathcal{W}(r_i) = \{w_1, \dots, w_{2\times i}\}$. Recall – from (12)–(15) – that VC_{obs} encodes, for each r_i, the most recent write in $\mathcal{W}(r_i)$ prior to r_i. However, since $\mathcal{W}(r_{i-1}) \subseteq \mathcal{W}(r_i)$, the problem for r_{i-1} (and indeed for all $j < i$) is re-encoded (and resolved by the SMT solver) as part of the problem for r_i.

Snapshotting eliminates much of this redundant encoding and solving. Semantically, a snapshot of shared variable g in job J appears after every write to g in the program order of J, and *atomically* reads the value of g and writes the same value back to g. In Figure 3, these actions are shown as $s_1, \dots, s_5$[1]. A snapshot *dominates* every other write to g in its job, and therefore eliminates them from being observed by future reads. At the same time, it may observe these writes, and snapshots in other jobs. With the snapshots added to Figure 3, we now have: $\mathcal{W}(s_1) = \mathcal{W}(r_1) = \{w_1, w_2\}, \mathcal{W}(s_2) = \mathcal{W}(r_2) = \{s_1, w_3, w_4\}, \dots, \mathcal{W}(s_5) = \mathcal{W}(r_5) = \{s_4, w_9, w_{10}\}$. Note how the problem for r_i is solved only once (for s_i), and then the solution for s_i is reused for all $j > i$. Empirically, snapshotting leads to significantly improved (see Section VI) verification time.

Formalism. We define a function $\mathit{Snaps} : \mathcal{J} \mapsto 2^G$. Informally, $\mathit{Snaps}(J)$ is the set of shared variables snapshotted by job J. The alphabet $\Sigma(J)$ of J is augmented with snapshot actions: $\Sigma_s(J) = \{(J, s, g) \mid g \in \mathit{Snaps}(J)\}$. Let $\Sigma_s(J) = \langle \alpha_s^1, \dots, \alpha_s^k \rangle$. The program order $PO(J)$ is augmented with:

$$\forall \alpha \in \Sigma(J) \setminus (\Sigma_s(J) \cup \{(J, \lhd)\}) \centerdot \alpha \xrightarrow{J} \alpha_s^1 \dots \xrightarrow{J} \dots \alpha_s^k \xrightarrow{J} (J, \lhd) \tag{24}$$

Thus, every execution in $[\![J]\!]$ snapshots all variables in $\mathit{Snaps}(J)$, and snapshot events appear after all other events, except for $(J, \lhd)$. Both reads and snapshots observe the last written values. Formally (6) is replaced by:

$$\forall i \in [1, k] \centerdot \eta(\epsilon_i) \in \{r, s\} \implies v(\epsilon_i) = \mathit{LastWr}(e, i) \tag{25}$$

Updated Construction of $VC(\mathcal{P})$. Let Σ_s be the set of snapshot actions, i.e., $\Sigma_s = \bigcup_{J \in \mathcal{J}} \Sigma_s(J)$. When constructing VC_{seq}, we treat each $\alpha \in \Sigma_s$ as a *NOP*. The construction of VC_{clk} uses the augmented $\Sigma(J)$ containing the additional snapshot actions. The construction of VC_{obs} is updated as follows. For every action $\alpha_r \in \Sigma_r \cup \Sigma_s$, we define $\mathcal{W}(\alpha_r)$, i.e., the set of actions that α_r may observe, as follows. For every job J, and shared variable g, let $\Psi_\sqsubset(J, g)$ be the maximal set of g-snapshotting jobs less than J according to the $\sqsubset$ order, i.e.,

$$\begin{aligned}\Psi_\sqsubset(J, g) = \{J' \in \mathcal{J} \mid g \in \mathit{Snaps}(J') \wedge J' \sqsubset J \bigwedge \\ \forall J'' \in \mathcal{J} \centerdot g \in \mathit{Snaps}(J'') \wedge J'' \sqsubset J \implies \neg(J' \sqsubset J'')\}\end{aligned} \tag{26}$$

Let $\Psi_\uparrow(J, g)$ be the set of jobs that can preempt J and also snapshot g, and $\Psi_\downarrow(J)$ be the set of jobs that can be preempted by J, and J itself, i.e.,

$$\Psi_\uparrow(J, g) = \{J' \in \mathcal{J} \mid g \in \mathit{Snaps}(J') \wedge J \uparrow J'\} \tag{27}$$

$$\Psi_\downarrow(J) = \{J' \in \mathcal{J} \mid J' = J \vee J' \uparrow J\} \tag{28}$$

Let $\alpha_r = (J, \eta, g)$. Then $\mathcal{W}(\alpha_r)$ consists of: (i) snapshots by jobs in $\Psi_\sqsubset(J, g)$ and $\Psi_\uparrow(J, g)$; and (ii) writes by jobs in $\Psi_\downarrow(J)$. Formally:

$$\begin{aligned}\mathcal{W}(\alpha_r) = \{(J', s, g) \mid J' \in \Psi_\sqsubset(J, g) \cup \Psi_\uparrow(J, g)\} \bigcup \\ \{(J', w, g) \mid J' \in \Psi_\downarrow(J)\}\end{aligned} \tag{29}$$

Finally, VC_{obs} contains the constraints defined in (12)–(15) for each $\alpha_r \in \Sigma_r \cup \Sigma_s$. Note that this means that a read or snapshot action α_r observes the last write or snapshot action to $g(\alpha_r)$ that executed prior to α_r. We claim that Theorem 1 also holds for the new $VC(\mathcal{P})$. The proof of this claim is in the extended version [25] of the paper.

We have implemented two variants of snapshotting – SS-ALL and SS-MOD – which differ in the set of variables snapshotted. For SS-ALL, all shared variables are snapshotted at the end of each job, i.e., $\mathit{Snaps}(J) = G$. For SS-MOD, only shared variables that are written by a job are snapshotted by it, i.e., $\mathit{Snaps}(J) = \{g \mid (J, w, g) \in \Sigma(J)\}$. We denote by SS-NONE the strategy of no snapshotting, presented in earlier sections. Next, we evaluate snapshotting empirically.

VI. Empirical Validation

We implemented our approach in a tool called LLREK, on top of UFO [26] and LLVM [27]. The input to LLREK is a PP $\mathcal{P}$ written in C, with jobs implemented via C functions, and periods, priorities etc. specified via macros. The safety property ϕ is expressed as an assertion in the job code. LLREK constructs the verification condition $VC(\mathcal{P}, \phi)$, as described earlier, and solves it using STP [28]. All experiments were performed on a machine running at 2.9GHz with a memory limit of 2GB and a time limit of 60 minutes. Our tools and benchmark are available at andrew.cmu.edu/~schaki/misc/llrek.tgz.

[1]For simplicity, we view a snapshot as either a read or a write, based on the context.

Name	Time (in seconds)				SAT Vars (in 1000s)				SAT Clauses (in 1000s)				AvgObs($\mathcal{P}$)			$\lvert W(\mathcal{P})\rvert$		
	NONE	ALL	MOD	REKH	NONE	ALL	MOD	REKH	NONE	ALL	MOD	REKH	NONE	ALL	MOD	NONE	ALL	MOD
nxt.bug1:H1	33	9	**7**	18	612	234	223	698	3252	1029	985	2642	25.6	2.9	2.9	298	455	416
nxt.bug2:H1	32	10	**7**	31	642	250	235	710	3394	1091	1030	2684	26.5	3.1	3.2	310	492	429
nxt.ok1:H1	19	**7**	8	17	612	234	223	698	3252	1030	986	2642	25.6	2.9	2.9	298	455	416
nxt.ok2:H1	20	7	**6**	29	611	234	223	699	3246	1029	985	2645	25.4	3.0	2.9	298	454	415
nxt.ok3:H1	30	8	**6**	31	642	250	235	709	3394	1091	1030	2675	26.5	3.1	3.2	310	492	429
aso.bug1:H1	29	**9**	**9**	34	636	274	249	737	3346	1198	1090	2796	26.0	3.6	3.6	304	512	427
aso.bug2:H1	28	10	**9**	32	646	277	251	734	3399	1211	1100	2780	26.4	3.7	3.7	308	516	431
aso.bug3:H1	29	13	**11**	80	690	305	270	958	3608	1324	1171	3660	25.5	3.6	3.5	355	615	504
aso.bug4:H1	32	17	**9**	66	649	306	265	891	3412	1357	1168	3396	26.5	4.6	4.4	309	543	434
aso.ok1:H1	32	11	**10**	32	658	286	261	726	3458	1255	1148	2746	27.1	4.1	4.2	311	519	434
aso.ok2:H1	38	29	**17**	67	651	307	265	893	3421	1360	1170	3406	26.5	4.6	4.4	311	545	436
nxt.bug1:H4	*	119	**74**	*	*	1096	1046	*	*	4897	4681	10696	99.5	3.0	3.0	1192	1835	1676
nxt.bug2:H4	*	172	**92**	*	*	1177	1105	*	*	5214	4916	10877	102.9	3.1	3.2	1240	1989	1731
nxt.ok1:H4	*	89	**49**	*	*	1096	1046	*	*	4898	4682	10696	99.5	3.0	3.0	1192	1835	1676
nxt.ok2:H4	*	125	**49**	*	*	1096	1046	*	*	4897	4682	10708	99.3	3.0	3.0	1192	1834	1675
nxt.ok3:H4	*	358	**133**	*	*	1177	1105	*	*	5213	4916	10830	102.9	3.1	3.2	1240	1989	1731
aso.bug1:H4	*	128	**92**	*	*	1301	1177	*	*	5773	5231	11394	99.9	3.6	3.6	1216	2072	1723
aso.bug2:H4	*	147	**74**	*	*	1316	1189	*	*	5840	5283	11316	101.6	3.7	3.7	1232	2088	1739
aso.bug3:H4	*	209	**136**	*	*	1452	1280	*	*	6408	5647	*	98.3	3.6	3.5	1420	2490	2034
aso.bug4:H4	*	329	**152**	*	*	1465	1261	*	*	6579	5645	*	100.4	4.6	4.4	1236	2199	1751
aso.ok1:H4	*	270	**210**	*	*	1359	1237	*	*	6061	5523	11151	103.2	4.1	4.2	1244	2100	1751
aso.ok2:H4	*	*	**1312**	*	*	1469	1264	*	*	6597	5659	*	100.1	4.6	4.4	1244	2207	1759
ctm.bug2	36	29	**21**	105	656	429	336	719	3253	1822	1448	2801	17.9	4.1	4.5	512	1052	683
ctm.bug3	*	124	**59**	258	*	705	554	1098	*	3066	2439	4389	26.6	4.1	4.5	768	1588	1033
ctm.ok1	23	37	**21**	122	668	434	341	730	3309	1839	1466	2845	18.6	4.1	4.6	512	1052	684
ctm.ok2	28	26	**17**	111	657	431	338	724	3261	1829	1455	2823	18.1	4.1	4.5	512	1052	683
ctm.ok3	*	116	**53**	275	*	714	567	1124	*	3106	2497	4485	27.9	4.1	4.5	780	1600	1057
ctm.ok4	*	320	**143**	395	*	959	760	1410	*	4184	3356	5713	36.4	4.2	4.7	1040	2140	1400

TABLE I
EXPERIMENTAL RESULTS; * = MEMORYOUT OR TIMEOUT; VARS = # OF SAT VARIABLES; CLAUSES = # OF SAT CLAUSES; BEST NUMBERS ARE IN BOLD.

Benchmark. Our benchmark consist of a set of PPs for controlling two LEGO Mindstorms robots – a two-wheel self-balancing robot (http://lejos-osek.sourceforge.net/nxtway_gs.htm), and a metal-stamping robot (http://www.cs.cmu.edu/~soonhok/blog/building-a-lego-turing-machine). The self-balancing robot controllers come in two variants. Some – named `nxt.*` in our tables – have three periodic tasks: a `Balancer`, with period of 4ms, that keeps the robot upright and monitors the bluetooth link for user commands, an `Obstacle`, with a period of 48ms, that monitors a sonar sensor for obstacles, and a 96ms `Background` task that prints debug information on an LCD screen. Others – named `aso.*` – have the functionality for monitoring bluetooth refactored out into the `Background` task.

The Turing Machine examples are named `ctm.*` and have four periodic tasks – `Controller`, `TapeMover`, `Reader`, and `Writer` in order of ascending priority. The `Controller` task has 500ms period and 440ms WCET. The other three tasks each have 250ms period and 10ms WCET respectively. The `Controller` task looks up a transition table, determines next operations to execute, and gives commands to the other tasks. The `TapeMover` task moves the tape to the left (or right). The `Reader` task moves the read head back and forth by rotating the read motor and reads the current bit of the tape. The `Writer` task rotates the write lever to flip a bit. In each case, we have safety properties (whose violations lead to potential collisions between the robot and an obstacle, or between different arms of the robot etc.) encoded as assertions, and both buggy and safe versions – named `*.bug*` and `*.ok*` – of the controller w.r.t. these assertions.

Evaluation of Snapshotting. Our first set of experiments were aimed at evaluating the three snapshotting strategies – SS-NONE, SS-ALL, SS-MOD. Our results are show in Table I. The first column shows the experiment name. For the `nxt.*` and `aso.*` example, `Hk` indicates that the time-bound T was set to equal `k` hyper-periods (i.e., $T = \mathtt{k} \times 96$) of the PP. The next three columns show the verification time $Time$, and the number of variables *Vars* and clauses *Clauses* of the final SAT formula solved by STP after simplifying and bit-blasting the SMT formula – for each snapshotting strategy.

These results indicate that SS-MOD is the best overall strategy. In all but one instance, it is the fastest. Sometimes it is more than twice as fast as the next best strategy SS-ALL. The worst choice is SS-NONE which runs out of memory in many instances, while both SS-ALL and SS-MOD complete successfully. These trends are mirrored when we consider *Vars* and *Clauses*, suggesting that snapshotting effectively eliminates a lot of redundancy in the SMT formulas generated by SS-NONE, with SS-MOD producing the most compact SAT formula overall. Next, we present a more direct quantitative evaluation of the effectiveness of snapshotting.

Observation Set Redundancy. Let $W(\mathcal{P})$ be the set of output (write or snapshot) actions in a PP $\mathcal{P}$. For each $w \in W(\mathcal{P})$, let $Obs(w)$ be the set of input (read or snapshot) actions that may observe w. Thus, $Obs(w) = \{\alpha \mid w \in \mathcal{W}(\alpha)\}$. Let AvgObs($\mathcal{P}$) be the mean of the set $\{|Obs(w)| \mid w \in W(\mathcal{P})\}$. A smaller value of AvgObs($\mathcal{P}$) indicates lower redundancy in the observation sets of $\mathcal{P}$. Here, redundancy means that a single output action may be observed by multiple input actions. Table I shows the values of AvgObs($\mathcal{P}$) and $|W(\mathcal{P})|$ for each $\mathcal{P}$ in our benchmark and for each snapshotting strategy. As expected, AvgObs($\mathcal{P}$) is much smaller for SS-MOD and SS-ALL compared to SS-NONE (sometimes by a factor of

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

over 30), indicating that snapshotting reduces redundancy in observation sets significantly. Both SS-MOD and SS-ALL have similar values of $\text{AvgObs}(\mathcal{P})$. However, $|W(\mathcal{P})|$ is smaller for SS-MOD since it snapshots more selectively. This leads to better overall performance of SS-MOD compared to SS-ALL.

Comparison with REKH. We also compare LLREK with REKH [11]. REKH constructs a sequential (but non-deterministic) C program that is semantically equivalent to $\mathcal{P}$, and verifies it using CBMC [29] 4.5. Internally, CBMC constructs a verification condition and solves it using a SAT solver. Thus, REKH and LLREK are similar – both generate and solve verification conditions. However, they construct VCs differently: LLREK generates it directly based on sequential consistency and snapshotting, while REKH generates a C program using rounds and prophecy variables (following Lal and Reps [30]), from which the VC is constructed by CBMC. The results for REKH are also presented in Table I. They indicate that SS-ALL and SS-MOD perform better than REKH, sometimes by a factor of over seven, and often complete verification when REKH runs out of memory. Thus, LLREK is a clear and significant improvement over REKH.

VII. Conclusion

We addressed the problem of verifying safety properties of PPs. Our solution is based on the BMC-MC paradigm and consists of two steps: (i) generate a provably correct VC; (ii) solve the VC using a SMT engine. We generate the VC by adapting Lamport's sequential consistency to the semantics of PPs. Moreover, we handle PPs that synchronize via two commonly used types of locks – PCP locks, and CPU locks. To improve scalability, we develop a strategy called snapshotting, aimed at generating VCs with fewer redundant sub-formulas. We develop two snapshotting strategies – SS-ALL snapshots all shared variables, while SS-MOD only snapshots modified variables. We have implemented our approach in a tool called LLREK. Experiments indicate that snapshotting improves effectiveness of verification significantly. In particular, SS-MOD is the best strategy, and it even outperforms the state-of-art verifier for PPs. An important direction for future work is to handle additional synchronization primitives, such as priority-inheritance locks [9], and to relax the restriction of a time bound.

Acknowledgment

Copyright 2014 Carnegie Mellon University and FMCAD, Inc. [2]

References

[1] M. Florian, E. Gamble, and G. Holzmann, "Logic Model Checking of Time-Periodic Real-Time Systems," in *Proc. of Infotect@Aerospace*, 2012.

[2] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *Proc. of CAV*, 2005.

[3] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: checking consistency of concurrent data types on relaxed memory models," in *Proc. of PLDI*, 2007.

[4] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta, "Symbolic Predictive Analysis for Concurrent Programs," in *Proc. of FM*, 2009.

[5] N. Sinha and C. Wang, "Staged concurrent program analysis," in *Proc. of FSE*, 2010.

[6] J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *Proc. of CAV*, 2013.

[7] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, 1979.

[8] ——, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM (CACM)*, vol. 21, no. 7, 1978.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Comp.*, vol. 39, no. 9, 1990.

[10] N. Sinha and C. Wang, "On interference abstractions," in *Proc. of POPL*, 2011.

[11] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, "Compositional Sequentialization of Periodic Programs," in *Proc. of VMCAI*, 2013.

[12] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Trans. on Comp. Aided Des.*, vol. 27, no. 7, 2008.

[13] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *STTT*, vol. 1, no. 1-2, 1997.

[14] V. A. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," in *Proc. of FSE*, 1999.

[15] F. Laroussinie, N. Markey, and P. Schnoebelen, "Efficient timed model checking for discrete-time systems," *Theoretical Computer Science (TCS)*, vol. 353, no. 1-3, 2006.

[16] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theo. Comp. Sc.*, vol. 126, no. 2, 1994.

[17] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-Bounded Analysis of Real-Time Systems," in *Proc. of FMCAD*, 2011.

[18] S. L. Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *Proc. of CAV*, 2009.

[19] N. Ghafari, A. J. Hu, and Z. Rakamaric, "Context-Bounded Translations for Concurrent Software: An Empirical Evaluation," in *Proc. of SPIN*, 2010.

[20] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-Bounded Scheduling," in *Proc. of POPL*, 2011.

[21] S. Chaki, A. Gurfinkel, and O. Strichman, "Verifying Periodic Programs with Priority Inheritance Locks," in *Proc. of FMCAD*, 2013.

[22] N. Kidd, S. Jagannathan, and J. Vitek, "One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling," in *Proc. of SPIN*, 2010.

[23] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973.

[24] A. Gurfinkel, S. Chaki, and S. Sapra, "Efficient Predicate Abstraction of Program Summaries," in *Proc. of NFM*, 2011.

[25] S. Chaki, A. Gurfinkel, and N. Sinha, "Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots," Extended version of paper published in the proceedings of FMCAD'14, andrew.cmu.edu/~schaki/publications/FMCAD-2014-Extended.pdf.

[26] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification," in *Proc. of CAV*, 2012.

[27] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO*, 2004.

[28] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Proc. of CAV*, 2007.

[29] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS*, 2004.

[30] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *Proc. of CAV*, 2008.

[2] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN AS-IS BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. DM-0000981

Under-approximate Flowpipes for Non-linear Continuous Systems

Xin Chen
RWTH Aachen University, Germany
xin.chen@cs.rwth-aachen.de

Sriram Sankaranarayanan
University of Colorado, Boulder, CO
srirams@colorado.edu

Erika Ábrahám
RWTH Aachen University, Germany
abraham@cs.rwth-aachen.de

***Abstract*—We propose an approach for computing under- as well as over-approximations for the reachable sets of continuous systems which are defined by non-linear Ordinary Differential Equations (ODEs). Given a compact and connected initial set of states, described by a system of polynomial inequalities, we compute under-approximations of the set of states reachable over time. Our approach is based on a simple yet elegant technique to obtain an accurate Taylor model over-approximation for a backward flowmap based on well-known techniques to over-approximate the forward map. Next, we show that this over-approximation can be used to yield both over- and under-approximations for the forward reachable sets. Based on the result, we are able to conclude "may" as well as "must" reachability to prove properties or conclude the existence of counterexamples. A prototype of the approach is implemented and its performance is evaluated over a reasonable number of benchmarks.**

I. Introduction

In this paper, we present an approach for computing under-approximations of the reachable sets of continuous systems described by ODEs. Continuous systems arise in a variety of domains including biological systems, control systems and aggregate mean field models of parameterized systems. Computing over-approximations of the reachable set of discrete, continuous and hybrid systems is a fundamental primitive for verifying safety properties. There has been much progress towards reachable set over-approximations for linear as well as non-linear continuous or hybrid systems through the use of invariant computation [1], [2], [3], conservative abstraction on dynamics [4], [5], [6], flowpipe construction [7], [8], [9], [10], [11], level sets [12], [13], [14], and advanced interval arithmetic techniques [15], [16], [17], [18], [19]. However, less attention has been given to the problem of finding reachable set under-approximations.

Whereas over-approximations represent states that "may" be reachable, under-approximations characterize states that "must" be reachable. As a result, under-approximations can be used to show that the system must reach a given target (or unsafe) set. The presence of under- as well as over-approximations can help us prove "reach-while-avoid" properties that are common in many control systems: the system must reach a specified target set of states, while avoiding a set of unsafe states. Under-approximations also help us judge the quality of related over-approximations by comparing the states that "may" be reachable with the states that "must" be reachable. Besides, under-approximation techniques are also crucial in finding counterexamples for continuous and hybrid systems, and could be extended to carry out Counterexample-Guided Abstraction Refinement (CEGAR) for these systems [20].

Our approach in this paper is based on the use of Taylor Model-based techniques that have been used for over-approximations [16], [19]. Starting from a given ODE $\dot{\vec{x}} = f(\vec{x})$ and an initial set X_0 defined by polynomial inequalities, we seek to derive an under-approximation Ω_t of the reachable set X_t at time $t > 0$. The basic idea for deriving an under-approximation starts by first deriving an *over-approximate backward flowmap* Φ that maps a state $\vec{x}_t \in \Omega_t$ potentially reachable at time t to a set of possible initial states $\Phi(\vec{x}_t)$. This can be seen (roughly) as an over-approximate pre-condition of the state $\vec{x}_t$. Next, we prove that a topologically connected set Ω_t which does not intersect the boundary of X_t is an under-approximation of it, if $\Phi(\vec{x}) \subseteq X_0$ for some $\vec{x} \in \Omega_t$. The condition of topological connctedness is an important technicality that must be checked for a set Ω_t before it can be identified as an under-approximation. Our approach integrates interval arithmetic approaches using higher order Taylor models [16], [19] with techniques from computational topology for proving connectedness [21]. The contributions of this paper are summarized as follows:

1) We show how Taylor model arithmetic can be used to over-approximate a backward flowmap (in addition to the forward flowmap). A key feature of our approach is that we mostly reuse the calculations for the forward map to also derive the backward map, using the structure of the Lagrange remainder in the Taylor series expansion.

2) We use the *Taylor model backward flowmap* to construct under-approximations. In doing so, it becomes necessary to prove that a set implicitly defined by polynomial inequalities is connected. We prove the property of *star-connectedness* through repeated satisfiability checks.

3) Finally, we have implemented our approach based on the computational library of FLOW* [22]. We provide experimental evaluation on a set of interesting and challenging benchmarks.

A. Related Work

As mentioned earlier, a significant volume of work has been devoted to the problem of finding over-approximations of the reachable states of continuous systems. Surprisingly, very little work has been focused on under-approximations. The main reason is the hardness of the task.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Several techniques of under-approximating reachable sets are introduced in [23], [24] for the systems defined by linear ODEs. However, they can not be easily extended to handle non-linear systems which are most often found in applications.

The idea of using over-approximations of backward flowmaps to compute reachable set under-approximations has been discussed elsewhere [25], [26]. Nevertheless, very few existing methods or tools can handle the job efficiently. We propose a more applicable method based on TM representations and does not require splitting the state space too often.

Under-approximation techniques have also received attention from the interval analysis community. The technique of modal (Kaucher) intervals provides a framework for under-approximations using intervals [27]. This was used to provide under-approximations of reachable sets for programs by Goubault et al. [28]. The recent work of Goubault et al. uses modal intervals with affine forms to provide under-approximations for the reachable sets of continuous systems [29]. In contrast, our approach relies on Taylor model based over-approximations, but of the backward flowmap rather than the forward map. Therefore, we are able to provide a higher-order technique for generating under-approximations in contrast to the first order approach of Goubault et al. using affine forms. Given the very recent nature of Goubault et al.'s contribution, we are unable to provide an experimental comparison of our techniques. However, a detailed comparison is planned as part of our extended version, in the future.

The work of Bai Xue and Zhikun She is yet another important contribution to the problem of under-approximating reachable sets of continuous systems, that inspired our approach in this paper [30]. Their approach is similar to ours in the use of backward flowmaps to compute under-approximations. A key difference, however is that Xue and She use an over-approximation of the boundary of the reachable set to find under-approximations. In our experience, the boundary of these sets if often complex and requires a fine subdivision of the state-space. Our approach, in comparison, avoids gridding the boundary. Instead, we are left with the problem of proving topological connectedness of a set, which is also hard in practice. Furthermore, the modification of Taylor models to compute backward flowmap over-approximations is a unique contribution of this paper.

Recently, Gao et al. presented a relaxed notion of δ-satisfiability to build constraint solvers for non-linear real arithmetic [31]. δ-satisfiability argues that a formula is unsatisfiable, or a δ-perturbation of it is satisfiable. By adjusting δ, the approach handles complex formulae involving real functions such as the flowmaps of ODEs. It has been implemented in the constraint solver dReal [32], and the tool dReach focusing on the analysis of non-linear systems [33]. Our approach has many fundamental differences: dReach attempts to answer a single reachability query using constraint solving, whereas our approach builds representations for reachable set segments that can be used to answer more complex queries. Our approach finds guaranteed over- and under-approximations, but does not reason about perturbations. Finally, the approach presented here can be a primitive inside a tool such as dReach, providing a more powerful approach to reachability analysis.

II. Preliminaries

Let $\mathbb{R}$ denote the set of real values. A set of variables $x_1, \ldots, x_n$, is collectively written as a vector $\vec{x}$. For a vector $\vec{x}$, we use x_i to denote its i-th component. Let $\mathbb{I}$ denote the set of all intervals $I = [a, b] \subseteq \mathbb{R}$ with $a, b \in \mathbb{R}$ and $a \leq b$. Multi-dimensional intervals are Cartesian products of intervals, and we continue to call them *intervals* in the paper. Given a variable or function $x(t)$ of time, we use $\dot{x}$ to denote the time derivative of x. Given a set S, we use $\texttt{Int}(S)$ to denote the smallest interval enclosure of S.

Definition 1 (Continuous system)**.** *An n-dimensional* continuous system $\mathcal{S}$ *is defined by an ODE $\dot{\vec{x}} = f(\vec{x})$, wherein $\vec{x}$ is a $n \times 1$ vector of* state variables *and the function f denotes the* vector field *which associates each state $\vec{c} \in \mathbb{R}^n$ a derivative vector $f(\vec{c}) \in \mathbb{R}^n$.*

Executions of a continuous system $\mathcal{S}$ correspond to the time trajectories of the ODE. We assume that the function f defining the ODE is (locally) Lipschitz continuous in $\mathbb{R}^n$. This guarantees that for each $\vec{x}_0 \in \mathbb{R}^n$, there exists a unique solution $\vec{x}(t)$ defined over some interval of existence $(-T(\vec{x}_0), T(\vec{x}_0))$, with initial condition $\vec{x}(0) = \vec{x}_0 \in \mathbb{R}^n$ [34]. Here $(-T(\vec{x}_0), T(\vec{x}_0))$ is the interval of existence and depends, in general, on the initial condition $\vec{x}_0$. We denote the value $\vec{x}(t)$ for any time $t \in (-T(\vec{x}_0), T(\vec{x}_0))$ by $\varphi_f(\vec{x}_0, t)$. We assume that for the models considered in this paper, the solutions exist for $T(\vec{x}_0) > T$, where T is a time horizon of interest. The function $\varphi_f(\vec{x}_0, t)$ is also called the *flowmap* which is *forward* if $t \geq 0$, and *backward* otherwise. In the rest of the paper, we assume that the dynamics $f(\vec{x})$ are given by a multivariate polynomial over $\vec{x}$.

The reachable set of a continuous system defined by $\dot{\vec{x}} = f(\vec{x})$ from an initial set $X_0 \subseteq \mathbb{R}^n$ is the set of flows $\{\varphi_f(\vec{x}_0, t) \,|\, \vec{x}_0 \in X_0\}$. For simplicity, we denote it by $\varphi_f(X_0, t)$ if $\varphi_f(\vec{x}_0, t)$ exists for all $\vec{x}_0 \in X_0$ in the time interval of interest. Given a time interval $\Delta \in \mathbb{I}$, the image of the map $\varphi_f(X_0, t)$ with $t \in \Delta$, is called a *flowpipe*.

Since we assume Lipschitz continuous ODEs, the map from $\vec{x}_0 \in X_0$ to $\varphi_f(\vec{x}_0, t)$ is *bijective*. Ideally, we wish to compute the map φ_f by solving the given ODE analytically. However, this cannot be done exactly, since most of the ODEs do not have closed form solutions. A typical approach is to approximate a solution by a Taylor polynomial which can be computed based on the higher-order Lie derivatives of the vector field. We will address it in Sect. IV.

Definition 2 (Lie derivative)**.** *Given an ODE $\dot{\vec{x}} = f(\vec{x})$ with n variables, the* Lie derivative *of a differentiable function $g(\vec{x}, t)$ w.r.t. f is defined by*

$$\mathcal{L}_f(g) = \sum_{i=1}^{n} \left(\frac{\partial g}{\partial x_i} \cdot f_i \right) + \frac{\partial g}{\partial t}$$

wherein f_i denotes the i-th component of f. If g is k times differentiable, the higher-order Lie derivatives of it are defined

recursively by

$$\mathcal{L}_f^{m+1}(g) = \mathcal{L}_f(\mathcal{L}_f^m(g)) \;\; for \;\; m = 1, 2, \ldots, k-1$$

In the rest of the section, we give a brief introduction of Taylor models (TMs). TMs were introduced by Berz and Makino to provide a framework for constructing high-order over-approximations of continuous functions as well as common operations over them. They are described in detail elsewhere [35]. A *Taylor model (TM)* is denoted by a pair (p, I) such that p is a polynomial over a closed and bounded domain and I is an interval which represents a remainder. Given a function $f(\vec{x})$ over D, we say that it is over-approximated by the TM $(p(\vec{x}), I)$ if $f(\vec{x}) \in p(\vec{x}) + I$ for all $\vec{x} \in D$. Intuitively, the TM maps any $\vec{x} \in D$ to an interval which contains $f(\vec{x})$. In the paper, we always use TM to mean a TM over-approximation.

TMs are closed under arithmetic operations of addition, scaling, multiplication and integration. The arithmetic over TMs can be viewed as a higher-order *interval arithmetic* [36]. Given two intervals $[a_1, b_1], [a_2, b_2] \in \mathbb{I}$, their sum and product are defined by $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$ and $[a_1, b_1] \cdot [a_2, b_2] = [\min\{a_1 \cdot a_2, a_1 \cdot b_2, b_1 \cdot a_2, b_1 \cdot b_2\}, \max\{a_1 \cdot a_2, a_1 \cdot b_2, b_1 \cdot a_2, b_1 \cdot b_2\}$ respectively. Then for two functions f, g over the same domain, if their TMs are given by (p_1, I_1), (p_2, I_2) respectively, a TM for $f + g$ can be computed by directly adding the polynomial and remainder part respectively, i.e., $(p_1 + p_2, I_1 + I_2)$, while an order k TM for their product $f \cdot g$ can be computed by

$$(\, p_1 \cdot p_2 - r_k \,,\, I_1 \cdot B(P_2) + I_2 \cdot B(P_1) + I_1 {\cdot} I_2 + B(r_k)\,)$$

wherein $B(p)$ denotes an interval enclosure of the range of p, and the *truncated part* r_k consists of the terms in $p_1 \cdot p_2$ of degrees $> k$. By TM arithmetic, we may compute an over-approximation for a complex function based on the TMs of its components.

TMs can be applied to provide over-approximations for flowpipes. They serve a dual purpose: they are used to conservatively approximate the *flowmap* $\varphi_f(\vec{x}_0, t)$ by a TM (p, I) for some $\vec{x}_0 \in X_0$ and $t \in \Delta \in \mathbb{I}$, such that

$$\forall \vec{x}_0 \in X_0,\ \forall t \in \Delta,\ \varphi_f(\vec{x}_0, t) \in p(\vec{x}_0, t) + I$$

They also serve as implicit definition of the flowpipe that over-approximates the image of φ_f over the set $\vec{x}_0 \in X_0$ and $t \in \Delta$. That is, a flowpipe $\varphi_f(X_0, t)$ for some $X_0 \subseteq \mathbb{R}^n$ and $t \in \Delta \in \mathbb{I}$ can be over-approximated by a TM $(p(\vec{x}_0, t), I)$ with $\vec{x}_0 \in X_0$ and $t \in \Delta$. Such a TM is also called a *TM flowpipe*, its computation is presented in Sect. IV.

III. Under-approximation technique at a glance

In this section, we present a brief sketch of our over- and under-approximate flowpipe computation technique. This section will serve to motivate the description of our approach through the rest of this paper.

Given a Lipschitz continuous ODE $\dot{\vec{x}} = f(\vec{x})$ and a compact and connected initial set X_0. We want to compute an under-approximation for the flowpipe $X_t : \varphi_f(X_0, t)$ with $t \in \Delta$ for some small time interval Δ. To do so, we seek to compute a set $\mathfrak{F}_t$ which *strictly* contains ∂X_t, i.e., the boundary of X_t. Since X_t is still compact and connected (see [34]), we may conclude that a connected set Ω_t which does not intersect ∂X_t is an under-approximation of X_t if Ω_t contains some state in X_t. To ensure these properties, we could (i) prove that Ω_t does not intersect $\mathfrak{F}_t$, and (ii) find a state in $\Omega_t \cap X_t$. An illustration is presented in Figure 1.

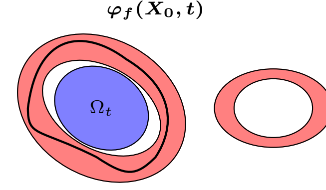


Fig. 1. Illustration of the main idea. The red region denotes the boundary over-approximation $\mathfrak{F}_t$, which is computed as a system of polynomial inequalities and could be disconnected.

It will be shown that a backward flowmap over-approximation plays a key role in achieving both (i) and (ii). In Sect. IV, we show how such an over-approximation can be effectively derived as a TM (p_b, I_b). The computation of $\mathfrak{F}_t$ based on (p_b, I_b) is described in Sect. V, where we also give a method to verify a reachable state by using (p_b, I_b).

IV. TMs for forward and backward flowmaps

In the section, we introduce a modified TM flowpipe construction approach which is an extention of our previous work [19]. A key feature of it is the derivation of a TM that approximates the backward flowmap by reusing the calculations for the forward map.

A. Modified TM flowpipe construction

Given an n-dimensional continuous system defined by $\dot{\vec{x}} = f(\vec{x})$, and a time step δ, the reachable set for a bounded time horizon $[0, T]$ and an initial set $X_0 \subseteq \mathbb{R}^n$ is over-approximated by a finite sequence of TMs $\mathcal{F}_1, \ldots, \mathcal{F}_N$, wherein $N = \lceil \frac{T}{\delta} \rceil$. For all $1 \leq i \leq N$, $\mathcal{F}_i$ over-approximates the image $\varphi_f(X_0, t)$ with $t \in [(i-1)\delta, i\delta]$. The TMs are computed iteratively, such that the segment $\mathcal{F}_i$ is used to compute the initial set for the subsequent TM. In the i-th iteration, we assume that the local initial set is given by a TM X_l. The i-th TM flowpipe $\mathcal{F}_i$ is computed by the following two steps.

Step 1: Compute a Taylor polynomial p_f for the forward flow $\varphi_f(X_l, t)$ up to order k in t. The polynomial p_f can be derived as the following Taylor polynomial of $\varphi_f(X_l, t)$,

$$p_f(\vec{x}_l, t) = \vec{x}_l + \mathcal{L}_f(\vec{x}_l) \cdot t + \cdots + \mathcal{L}_f^k(\vec{x}_l) \cdot \frac{t^k}{k!} \qquad (1)$$

wherein $\vec{x}_l \in X_l$ and we simply denote $\mathcal{L}_f^j(\vec{x})|_{\vec{x}=\vec{x}_l}$ by $\mathcal{L}_f^j(\vec{x}_l)$ for $1 \leq j \leq k$. Unlike our previous work, the degrees of $\vec{x}_l$ in p_f are not limited.

Step 2: Evaluate a safe remainder interval I_f for p_f over $t \in [0, \delta]$. The purpose is to find an interval I_f such that

the TM $(p_f(\vec{x}_l,t),I_f)$ is an over-approximation of $\varphi_f(\vec{x}_l,t)$ over $\vec{x}_l\in X_l$ and $t\in[0,\delta]$. As we have the assumption that f is at least locally Lipschitz continuous, then an interval I_f is sufficient (or safe) if the *Picard operator*

$$\mathbb{P}_f(g)(\vec{x}_l,t)=\vec{x}_l+\int_0^t f(g(\vec{x}_l,s))\ ds \tag{2}$$

is *contractive* over (p_f,I_f) (see [16], [19]). To find such an interval, we may start with an estimation I_e which could be incorrect, and then conservatively check the contractiveness of the Picard operation by means of TM arithmetic. If it can not be verified, we enlarge the interval I_e until we obtain a contractive interval. The resulting interval I_e may be further refined by repeatedly performing the Picard operation on (p_f,I_e). We then set $I_f=I_e$. Unlike previous work, we only truncate the polynomial terms whose degrees of t are larger than k. Afterwards, if $i>1$, the TM $\mathcal{F}_i$ can be derived by substituting X_l in the place of $\vec{x}_l$ in (p_f,I_f) by TM arithmetic, otherwise it is the first iteration and we simply rename $\vec{x}_l$ by $\vec{x}_0$.

B. Compute over-approximations for backward flowmaps

The flowpipe construction presented thus far only produces a TM that over-approximates the forward flowmap from X_0 to X_t : $\varphi_f(X_0,t)$ for $t\in[0,T]$, and the under-approximation approach requires over-approximations for the backward flowmaps.

Even though the backward flowmap is conceptually obtained by negating the time variable, a TM over-approximation for the backward flowmap is not easy to obtain. A simple way to do that is performing a backward flowpipe computation from an over-approximation of X_t which is obtained by a forward one. However, it is not only time consuming but also inaccurate, since the overestimation generated in the forward computation is also considered in estimating the remainder intervals for the backward flowmaps by the Picard operation. Thus, we need a method to obtain backward over-approximations without using flowpipe construction.

We introduce a novel method to generate accurate backward over-approximations by reusing the calculation of the forward modified TM flowpipe construction. Let us fix a time $t\geq 0$ and consider the initial set X_l for the i-th step of the forward flowpipe construction. Let us denote $Y_l(t)=\varphi_f(X_l,t)$, as the image of the forward flowmap for any $t\in[0,\delta]$. We assume that φ_f is over-approximated by a TM $(p_f(\vec{x}_l,t),I_f)$, wherein $\vec{x}_l\in X_l$. Our goal is to construct a TM (p_b,I_b) that over-approximates the flowmap from Y_l back to X_l.

Constructing p_b It is easy to see that while $\varphi_f(\vec{x}_0,t)$ for $\vec{x}_0\in X_0, t\geq 0$ represents the forward map, the backward map is represented by $\varphi(\vec{y}_0,-t)$ where $\vec{y}_0\in\varphi_f(X_0,t), t\geq 0$. Therefore, its Taylor expansion is related to that of $\varphi(\vec{x}_0,t)$ when $t\geq 0$. Using this observation, the polynomial p_b is derived from p_f by syntactically replacing $\vec{x}_l$, the variables denoting the starting state, by $\vec{y}_l$, the variables denoting the ending state. Likewise, we replace the time variable t by $-t$. The renaming of $\vec{x}_l$ is not technically necessary, we do it to distinguish the domains of the TMs for forward and backward flowmaps. The challenge remains to construct the remainder interval I_b. In doing so, we wish to avoid computing Picard operation by TM arithmetic which could potentially introduce a large overestimation.

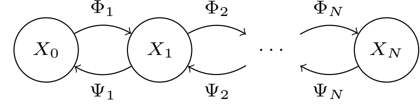


Fig. 2. Flowmap automaton

The Lagrange remainder term of p_f at some $\vec{x}_l\in X_l$ and $t\in[0,\delta]$ is

$$\varepsilon(\vec{x}_l,t)=\frac{1}{(k+1)!}\mathcal{L}_f^{k+1}(\varphi_f(\vec{x}_l,\xi))\cdot t^{k+1} \tag{3}$$

wherein ξ is between 0 and t. Then an interval enclosure $\mathcal{E}(X_l,[0,\delta])$ of all $\varepsilon(\vec{x}_l,t)$ over $\vec{x}_l\in X_l$ and $t\in[0,\delta]$ can be evaluated as

$$\frac{1}{(k+1)!}\mathcal{L}_f^{k+1}(\mathtt{Int}(\{\varphi_f(\vec{x}_l,\xi)|\vec{x}_l\in X_l,\xi\in[0,\delta]\}))\cdot([0,\delta])^{k+1}$$

Similarly, since the remainder term for p_b at some $\vec{y}_l\in Y_l(t)$ and $t\in[0,\delta]$ can be expressed by $\varepsilon(\vec{y}_l,-t)$ such that ξ is between 0 and $-t$. An interval enclosure of those remainders over $\vec{y}_l\in Y_l(t)$ and $t\in[0,\delta]$ could be obtained as $\mathcal{E}(Y_l(\delta),[-\delta,0])$. By Lemma 3, we have that $\mathcal{E}(Y_l(\delta),[-\delta,0])=(-1)^{k+1}\cdot\mathcal{E}(X_l,[0,\delta])$. In other words, I_b can be computed as an interval enclosure of $(-1)^{k+1}\cdot\mathcal{E}(X_l,[0,\delta])$.

Lemma 3. *For an order $k\geq 0$ and a time interval $[0,\delta]$, we have that*

$$\mathcal{E}(Y_l(\delta),[-\delta,0])=(-1)^{k+1}\cdot\mathcal{E}(X_l,[0,\delta])$$

Although the interval $\mathtt{Int}(\{\varphi_f(\vec{x}_l,\xi)\,|\,\vec{x}_l\in X_l,\xi\in[0,\delta]\})$ is hard to compute, we may obtain an interval enclosure $I_{\vec{x}}$ for it from an interval evaluation of $\mathcal{F}_i$, and hence

$$I_\varepsilon=\frac{1}{(k+1)!}\mathcal{L}_f^{k+1}(I_{\vec{x}})\cdot([0,\delta])^{k+1} \tag{4}$$

is an interval enclosure of $\mathcal{E}(X_l,[0,\delta])$. At last, we have the safe remainder interval $I_b=(-1)^{k+1}\cdot I_\varepsilon$.

Notice that I_b is sufficiently large for any point in $(p_f(\vec{x}_l,t),I_f)$ with $\vec{x}_l\in X_l$, $t\in[0,\delta]$, i.e., $\mathcal{F}_i$. In other words, for any point $\vec{y}_l\in(p_f(\vec{x}_l,t),I_f)$, $(p_b(\vec{y}_l,t),I_b)$ defines an over-approximation for the backward map $\varphi_f((p_f(\vec{x}_l,t),I_f),-t)$. The reason is that I_ε is computed based on the over-approximation $\mathcal{F}_i$.

The TMs of the forward and backward flowmaps computed in all time steps can be organized as an automaton shown in Fig.2. For $1\leq i\leq N$, the state X_i denotes the exact reachable set $\varphi_f(X_0,i\delta)$. The forward edge $\Phi_i(\vec{x}_l,t)$ denotes the forward TM $(p_f(\vec{x}_l,t),I_f)$ in the i-th time step, while the backward edge $\Psi_i(\vec{y}_l,t)$ is the backward TM $(p_b(\vec{y}_l,t),I_b)$ there. When we take $t=\delta$, they are over-approximations of the maps between the states. Then for any $\tau\in[0,T]$, an order k TM for the backward map from $\varphi_f(X_0,\tau)$ to X_0

can be obtained by composing the TMs along the path from X_i to X_0 such that $\tau \in [(i-1)\delta, i\delta]$. It can be done by Algorithm 1. In the TM computation, we take the TM flowpipe $\mathcal{F}_i$ with $t = \tau - (i-1)\delta$ as the range of $\vec{y}_l$. To achieve a good accuracy, some preconditioning techniques proposed for intervals [37] and TMs [38] can be applied. Additionally, we may also consider the case that τ ranges in a time interval by taking an additional variable t.

Algorithm 1 Composing TMs for backward flows

```
Π ← Ψ_i(y_l, τ − (i − 1)δ);        # by TM arithmetic
for all j = i − 1 to 1 do
  Π ← Ψ_j(Π, δ);                   # by TM arithmetic
end for
return Π;
```

V. UNDER-APPROXIMATION GENERATION

In the section, we show how flowpipe under-approximations can be generated based on the TMs of backward flowmaps.

A. Main theorem

Given an n-dimensional continuous system defined by $\dot{\vec{x}} = f(\vec{x})$. If the initial set is defined by $X_0 = \{\vec{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{m}(p_i(\vec{x}) \leq 0)\}$ which is *compact* and *connected*, then the reachable set at time $t \geq 0$ can be characterized by

$$\varphi_f(X_0, t) = \{\vec{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{m}(p_i(\varphi_f(\vec{x}, -t)) \leq 0)\} \quad (5)$$

which is also compact and connected (see [34]). Intuitively, a state $\vec{x}$ is in $\varphi_f(X_0, t)$ iff the backward flow maps it to a state in X_0 at time $-t$. We present an example in Fig. 3 to show such evolution of a constraint. Given a time point $t = \tau$, if $(p_b(\vec{x}), I_b)$ is a TM for the backward flowmap from $\varphi_f(X_0, \tau)$ to X_0, then we may compute an order k TM $(\phi_i(\vec{x}), [\ell_i, v_i])$ for $p_i(\varphi_f(\vec{x}, -\tau))$ from evaluating $p_i((p_b(\vec{x}), I_b))$ by TM arithmetic for all $1 \leq i \leq m$. Such a TM of the backward flowmap as well as a TM $\mathcal{F}$ of $\varphi_f(X_0, \tau)$ can be obtained using the forward as well as backward flowmap computation presented in Sect. IV by taking a TM of X_0. Then the constrained flowpipe $\mathcal{F}_o = \{\vec{x} \in \mathcal{F} \mid \bigwedge_{i=1}^{m}(\phi_i(\vec{x}) + \ell_i \leq 0)\}$ defines a refined over-approximation of the reachable set $\varphi_f(X_0, \tau)$ since $\mathcal{F}$ is derived based on a TM of X_0, while an under-approximation of $\varphi_f(X_0, \tau)$ can be computed as a *connected* subset Ω of $\mathcal{F}_u = \{\vec{x} \in I_{\mathcal{F}} \mid \bigwedge_{i=1}^{m}(\phi_i(\vec{x}) + u_i \leq 0)\}$ wherein $I_{\mathcal{F}}$ is an interval enclosure of $\mathcal{F}$ and $u_i = v_i + \epsilon$ for some $\epsilon > 0$, if $\Omega \cap \varphi_f(X_0, \tau) \neq \emptyset$. The purpose to raise those upper bounds is to ensure that $\mathcal{F}_u$ has no intersection with the boundary of $\varphi_f(X_0, \tau)$ which is strictly over-approximated by $\mathfrak{F}_\tau = \mathcal{F}_o \backslash \mathcal{F}_u$. The detail is explained in the proof of Theorem 4.

Theorem 4. *The constrained flowpipe $\mathcal{F}_o$ is an over-approximation of $\varphi_f(X_0, \tau)$. For any* connected *subset Ω of $\mathcal{F}_u$, if $\varphi_f(X_0, \tau) \cap \Omega \neq \emptyset$, then Ω is an under-approximation of $\varphi_f(X_0, \tau)$.*

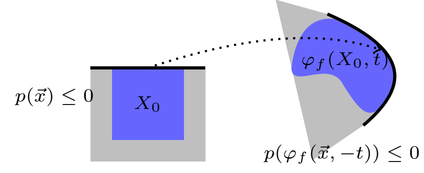


Fig. 3. Evolution of a constraint $p(\vec{x}) \leq 0$

Proof. We first prove the over-approximation. Since the TM $(\phi_i(\vec{x}), [\ell_i, u_i])$ is an over-approximation of $p_i(\varphi_f(\vec{x}, -\tau))$ for $1 \leq i \leq m$, more precisely, we have that

$$\phi_i(\vec{x}) + \ell_i \leq p_i(\varphi_f(\vec{x}, -\tau)) < \phi_i(\vec{x}) + u_i \quad (6)$$

for all $\vec{x} \in \varphi_f(X_0, \tau)$. Then for any $\vec{x} \in \varphi_f(X_0, \tau)$, the implication $p_i(\varphi_f(\vec{x}, -\tau)) \leq 0 \rightarrow \phi_i(\vec{x}) + \ell_i \leq 0$ holds, and hence $\varphi_f(X_0, \tau) \subseteq \{\vec{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{m}(\phi_i(\vec{x}) + \ell_i \leq 0)\}$. Since $\varphi_f(X_0, \tau) \subseteq \mathcal{F}$, we conclude that $\varphi_f(X_0, \tau) \subseteq \mathcal{F}_o$.

We turn to the under-approximation. The boundary of $\varphi_f(X_0, \tau)$ is given by

$$\partial\varphi_f(X_0, \tau) = \left(\bigcup_{i=1}^{m}\{\vec{x} \in \mathbb{R}^n \mid p_i(\varphi_f(\vec{x}, -\tau)) = 0\}\right) \cap \varphi_f(X_0, \tau)$$

Then the set $S = \{\vec{x} \in \mathbb{R}^n \mid \phi_i(\vec{x}) + u_i \leq 0\}$ does not intersect $\partial\varphi_f(X_0, \tau)$. The reason is that for any $\vec{x} \in S$, if $\vec{x} \in \varphi_f(X_0, \tau)$ there is $p_i(\varphi_f(\vec{x}, -\tau)) < 0$ for all $1 \leq i \leq m$ by the inequality (6), otherwise $p_i(\varphi_f(\vec{x}, -\tau)) > 0$ for all $1 \leq i \leq m$. It is also the case for all subsets of S. Therefore, any *connected* subset of $S(t)$ either is entirely contained in $\varphi_f(X_0, \tau)$ or has no intersection with $\varphi_f(X_0, \tau)$. Since $\mathcal{F}_u \subseteq S$, we conclude that $\Omega \subset \varphi_f(X_0, \tau)$ for any connected set $\Omega \subseteq \mathcal{F}_u$ if $\varphi_f(X_0, \tau) \cap \Omega \neq \emptyset$. □

By taking t as an additional variable over a small time interval Δ, Theorem 4 can be extended to produce under- as well as over-approximation for the reachable set over Δ.

B. Methodologies to find an under-approximation

From Theorem 4, we need three steps to compute an under-approximation of the TM $\mathcal{F}$ for the reachable set $\varphi_f(X_0, \tau)$. The first step is to obtain a subset Ω of $\mathcal{F}_u$. It can be done by taking Ω as $\mathcal{F}_u$ or a subset of it. Then in the second step, we need to prove that Ω is connected, and ensure that the intersection $\Omega \cap \varphi_f(X_0, \tau)$ is nonempty in the third step. There are various ways to achieve this, we present some methods based on interval arithmetic. Again, the following methods can be extended to handle the reachable set over a time interval by taking an additional variable t.

Taking $\Omega = \mathcal{F}_u$. To limit the underestimation, we mainly consider the case that $\Omega = \mathcal{F}_u$. Then it requires to verify that $\mathcal{F}_u$ is a connected set. Since it is defined by a system of polynomial inequalities, to verify its connectedness is at least as hard as solving the same problem on a *basic closed semialgebraic set*, and it is intractable in general (see [39]). Fortunately, we could use the sufficient condition given in [21] on which the connectedness may possibly be proved efficiently. The idea is to find a *star point* in $\mathcal{F}_u$.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

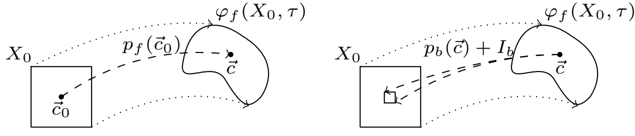


Fig. 4. **(L)** Compute a candidate star point $\vec{c}$, and **(R)** verify $\vec{c}$ is reachable

Given a set S, a point $s^* \in S$ is a *star point* if for any $s \in S$ the line segment connecting s^*, s is contained in S. Furthermore, if S has a star point then it is connected. To find a star point in $\mathcal{F}_u$, we may first compute a candidate point $\vec{c} \in \mathcal{F}_u$. Assume that the forward flowmap from X_0 to $\varphi_f(X_0, \tau)$ is over-approximated by $(p_f(\vec{x}), I_f)$. The point $\vec{c}$ can be computed as $p_f(\vec{c}_0)$ wherein $\vec{c}_0$ is an approximation of the geometric center of X_0. Fig. 4(**right**) shows the idea. When the TM order is sufficiently high, the inclusion $\vec{c} \in \mathcal{F}_u$ can be ensured. To verify that $\vec{c}$ is a star point in $\mathcal{F}_u$, as stated by Theorem 5 and Corollary 6, we may prove the unsatisfiability of the constraints

$$\phi_i(\vec{x}) + u_i = 0 \ \wedge \ \sum_{j=1}^{n} \left(\frac{\partial \phi_i}{\partial x_j} \cdot (x_j - c_j) \right) \leq 0$$

over $\vec{x} \in I_{\mathcal{F}}$ for all $1 \leq i \leq m$. This may be efficiently done by using *Interval Constraint Propagation (ICP)* [40].

Theorem 5 ([21]). *Given a set $S = \{\vec{x} \in D \subset \mathbb{R}^n \,|\, \psi(x) \leq 0\}$ wherein D is a convex set and ψ has continuous derivatives in D. For any $\vec{c} \in S$, if the constraint*

$$\psi(\vec{x}) = 0 \ \wedge \ \sum_{i=1}^{n} \left(\frac{\partial \psi}{\partial x_i} \cdot (x_i - c_i) \right) \leq 0$$

is unsatisfiable *for $\vec{x} \in D$, then $\vec{c}$ is a star point in S.*

Corollary 6. *Given a set $S = \{\vec{x} \in D \subset \mathbb{R}^n \,|\, \bigwedge_{i=1}^{m}(\psi_i(x) \leq 0)\}$ wherein D is a convex set and $\psi_1, \ldots, \psi_m$ have continuous derivatives in D. If $\vec{c}$ is a star point in $S_i = \{\vec{x} \in D \,|\, \psi_i(x) \leq 0\}$ for all $1 \leq i \leq m$, then it is also a star point in S.*

In the last step, we should prove that the intersection $\mathcal{F}_u \cap \varphi_f(X_0, \tau)$ is nonempty. To do so, we assume that the backward flowmap from X_0 to $\varphi_f(X_0, \tau)$ is over-approximated by the TM $(p_b(\vec{x}), I_b)$ based on the method in Sect. IV-B. Then, as we pointed out, I_b is safe for all states in (p_f, I_f). Hence we may check whether the interval $p_b(\vec{c}) + I_b$ is included by X_0. If so, then $\vec{c}$ is in $\varphi_f(X_0, \tau)$, and $\mathcal{F}_u$ is an under-approximation of $\varphi_f(X_0, \tau)$. The idea is illustrated in Fig. 4 (**Left**). It will be shown in Sect. VI that the three steps succeed in most of our experiments. A simple example is given as below.

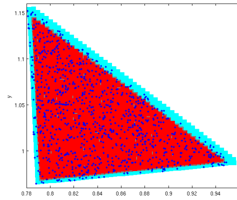


Fig. 5. Sets $\mathcal{F}_o$, $\mathcal{F}_u$

Example 7. *We consider the Moore-Greitzer model of a jet engine described in [41]. It is the continuous system defined by the following ODE.*

$$\begin{cases} \dot{x} &= -y - 1.5 \cdot x^2 - 0.5 \cdot x^3 - 0.5 \\ \dot{y} &= 3 \cdot x - y \end{cases}$$

The initial set is given by the simplex

$$X_0 = \{(x, y) \in \mathbb{R}^2 \,|\, -x \leq -0.9 \wedge -y \leq -0.9 \wedge x + y - 2 \leq 0\}$$

We try to compute the under-approximation $\mathcal{F}_u$ as well as the over-approximation $\mathcal{F}_o$ at $t = 0.04$ based on the TMs of the forward and backward flowmaps. Those TMs are computed on the interval enclosure $I_{X_0} = \{(x, y) \,|\, x \in [0.9, 1.1], y \in [0.9, 1.1]\}$ of X_0. An interval enclosure of the TM flowpipe $\mathcal{F}$ at time 0.04 is

$$I_{\mathcal{F}} = \left\{ (x, y) \;\middle|\; \begin{array}{l} x \in [0.78063344, 0.95902894], \\ y \in [0.96380802, 1.1772562] \end{array} \right\}$$

By transferring the constraints defining X_0 to the time 0.04, we obtain the polynomials ϕ_1, ϕ_2, ϕ_3 and constant bounds $\ell_1, \ell_2, \ell_3, u_1, u_2, u_3$ in the definition of $\mathcal{F}_u$, $\mathcal{F}_o$:

$$\begin{aligned} \phi_1 = & -4.0810848e\text{-}2 \cdot y - 9.9877519e\text{-}1 \cdot x - 3.3480961e\text{-}5 \cdot y^2 \\ & -2.4637920e\text{-}3 \cdot x \cdot y - 6.0550400e\text{-}2 \cdot x^2 - 3.6608001e\text{-}7 \cdot y^3 \\ & -3.7006081e\text{-}5 \cdot x \cdot y^2 - 1.4139012e\text{-}3 \cdot x^2 \cdot y - 2.3644942e\text{-}2 \cdot x^3 \\ & -1.1520000e\text{-}7 \cdot x \cdot y^3 - 7.8739201e\text{-}6 \cdot x^2 \cdot y^2 \\ & -2.4417472e\text{-}4 \cdot x^3 \cdot y - 3.2465277e\text{-}3 \cdot x^4 \end{aligned}$$

$$\begin{aligned} \phi_2 = & -1.0383459 \cdot y + 1.2238309e\text{-}1 \cdot x + 1.0022399e\text{-}6 \cdot y^2 \\ & +9.8899199e\text{-}5 \cdot x \cdot y + 3.6712367e\text{-}3 \cdot x^2 + 7.6799999e\text{-}9 \cdot y^3 \\ & +1.0828799e\text{-}6 \cdot x \cdot y^2 + 5.4915839e\text{-}5 \cdot x^2 \cdot y + 1.3629942e\text{-}3 \cdot x^3 \\ & +1.7280000e\text{-}7 \cdot x^2 \cdot y^2 + 7.2460800e\text{-}6 \cdot x^3 \cdot y + 1.2865439e\text{-}4 \cdot x^4 \end{aligned}$$

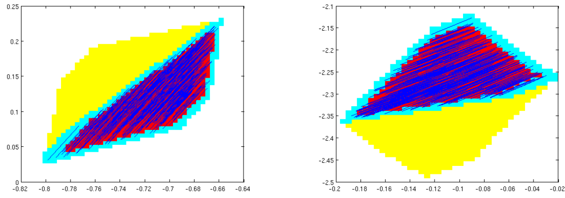
$$\begin{aligned} \phi_3 = & \, 1.0791566 \cdot y + 8.7639208e\text{-}1 \cdot x + 3.2478719e\text{-}5 \cdot y^2 \\ & +2.3648927e\text{-}3 \cdot x \cdot y + 5.6879162e\text{-}2 \cdot x^2 + 3.5840000e\text{-}7 \cdot y^3 \\ & +3.5923200e\text{-}5 \cdot x \cdot y^2 + 1.3589852e\text{-}3 \cdot x^2 \cdot y \\ & +2.2281947e\text{-}2 \cdot x^3 + 1.1519999e\text{-}7 \cdot x \cdot y^3 + 7.7011199e\text{-}6 \cdot x^2 \cdot y^2 \\ & +2.3692863e\text{-}4 \cdot x^3 \cdot y + 3.1178732e\text{-}3 \cdot x^4 \end{aligned}$$

$$\begin{array}{lll} \ell_1 = 0.88000760, & \ell_2 = 0.90121569, & \ell_3 = -1.9812255 \\ u_1 = 0.88000946, & u_2 = 0.90121597, & u_3 = -1.9812233 \end{array}$$

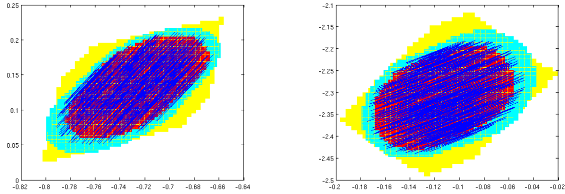
We choose the point $\vec{c}_0 = (0.95, 0.95) \in X_0$ and its image under the forward flowmap approximation p_f is $\vec{c} = (0.82910752, 1.0171865)$ which can be easily verified by iSAT [42] as a star point in $\mathcal{F}_u$. Therefore $\mathcal{F}_u$ is connected. To ensure that the intersection of $\mathcal{F}_u$ and the reachable set at $t = 0.04$ is nonempty, we compute the interval image of $\vec{c}$ under the TM of the backward flowmap and it is contained in X_0. Hence, $\mathcal{F}_u$ is an under-approximation of the reachable set at $t = 0.04$. To visualize the sets $\mathcal{F}_u$ and $\mathcal{F}_o$, we plot the grids with a specified size that intersect $\mathcal{F}_o$ in cyan, and the grids that are covered by $\mathcal{F}_u$ in red. They are shown in Fig. 5. Besides, we also give the simulations [1] *in blue.*

To further investigate the performance of our method, we consider to under- as well as over-approximate a flowpipe over a time step. We set the step-size $\delta = 0.02$ and compute the TMs of forward and backward flowmaps for the time horizon $[0, 3]$. In Fig. 6(a) and 6(b) respectively, we plot the set $\mathcal{F}_o$ in cyan, the set $\mathcal{F}_u$ in red and the unconstrained TM flowpipe $\mathcal{F}$ in yellow for t ranges in a time step. Additionally, we also plot the similar approximation sets in Fig. 6(c) and 6(d) for the ellipsoidal initial set $\{(x, y) \in \mathbb{R}^2 \,|\, (x-1)^2 + (y-1)^2 \leq 0.01\}$.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.



(a) $t \in [0.98, 1]$ from the simplex (b) $t \in [2.98, 3]$ from the simplex



(c) $t \in [0.98, 1]$ from the ellipsoid (d) $t \in [2.98, 3]$ from the ellipsoid

Fig. 6. Reachable set under- and over-approximations for the jet engine model. Numerical simulations are given in blue.

Other methods. The under-approximate set Ω may also be defined as a geometric object, such as a set of connected boxes or polytopes. To do so, we may follow the methods presented in [43] and [44]. The main idea is to first randomly generate a set of points in $\mathcal{F}_u$, and then successively bloat each point to a set which is made as large as possible but still contained in $\mathcal{F}_u$. Then Ω is the union of those sets which are connected to the others. To verify that Ω intersects the exact reachable set $\varphi_f(X_0, \tau)$, we may just compute the images of those random points under the backward TM map, if at least one of them is in X_0, then Ω is an under-approximation of $\varphi_f(X_0, \tau)$.

VI. Experiments

We implemented our approach based on the TM library of FLOW* [22]. The experiments are described as follows.

System models. We select 9 non-linear continuous systems whose dimensions range from 2 to 7. In order to evaluate our method on tough examples, some chaotic systems are also included. They are Lorenz system, Rössler attractor and Shimizu-Morioka system [45].

Initial sets. We want to handle the initial sets defined by polynomial constraints. Such a set is usually not TM definable but the TM forward and backward flowmaps can be computed on a TM of it. In our experiments, we consider two initial sets for each system: a *simplex* defined by $S_0 = \{\vec{x} \in \mathbb{R}^n \,|\, (\bigwedge_{i=1}^n(-x_i + a_i - r \le 0)) \wedge (\sum_{i=1}^n x_i - \sum_{i=1}^n a_i) \le 0\}$, and an *ellipsoid* defined by $E_0 = \{\vec{x} \in \mathbb{R}^n \,|\, \sum_{i=1}^n (x_i - a_i)^2 \le r^2\}$. The constants $\vec{a}$ and r for the systems are listed as follows: *jet engine*: $\vec{a} = (1, 1)$, $r = 0.1$; *Brusselator*: $\vec{a} = (0.95, 0.05)$, $r = 0.05$; *Rössler attractor*: $\vec{a} = (0, -8.4, 0)$, $r = 0.1$; *Lorentz system*: $\vec{a} = (15, 15, 36)$, $r = 0.01$; *Shimizu-Morioka system*: $\vec{a} = (15, 15, 36)$, $r = 0.01$; *Lotka-Volterra system [46]*: $\vec{a} = (0.5, 0.5, 0.5, 0.5)$, $r = 0.1$; *coupled Van-der-Pol system*: $\vec{a} = (1, 1, 1, 1)$, $r = 0.1$; *Watt steam governor [47]*: $\vec{a} = (0, 0, 0, 0, 0)$, $r = 0.1$; *biological system [48]*: $\vec{a} = (0.1025, \dots, 0.1025)$, $r = 0.0025$. Notice that these initial sets are at least in the same scale as those typically used in evaluating verified integration methods. Also, we evaluate the accuracy of an approximation at the end of the time horizon.

[1]A numerical simulation is only an approximation whose error bound is not guaranteed. However, it usually can be made very accurate.

TABLE I
EVALUATION OF THE APPROXIMATIONS FOR $\vec{x}(T)$ WITH INITIAL SETS AS *simplices*. **VAR**: # VARIABLES, $\boldsymbol{\delta}$: STEP-SIZES, $\boldsymbol{k}$: TM ORDERS, **TIME**: TOTAL RUNNING TIME.

#	systems	var	T	δ	k	time (s)	$\gamma_{\min}$
1	jet engine	2	4	0.02	4	56	∼0.8
2	jet engine	2	5	0.02	4	71	∼0.75
3	Brusselator	2	3	0.02	4	55	∼0.7
4	Brusselator	2	4	0.02	4	89	∼0.55
5	Rössler	3	1.5	0.01	5	165	∼0.5
6	Rössler	3	1.6	0.01	5	178	Fail
7	Lorenz	3	0.5	0.01	5	35	∼0.65
8	Lorenz	3	0.6	0.01	5	45	∼0.35
9	Shimizu-Morioka	3	1	0.01	5	58	∼0.7
10	Shimizu-Morioka	3	1.2	0.01	5	69	∼0.3
11	Lotka-Volterra	4	1	0.01	4	297	∼0.4
12	coupled Van-der-Pol	4	4	0.01	4	118	∼0.45
13	steam governor	5	2.5	0.01	5	16	∼0.35
14	biological system	7	0.2	0.002	3	632	∼0.25

Results. Since the exact accuracy evaluation is very hard, we intuitively only measure the widths w.r.t. a set of directions. Given an over-approximation S_o, an under-approximation S_u and a set of vectors V, we conservatively compute the widths of S_o, S_u w.r.t. each $\vec{v} \in V$: $\gamma_o(\vec{v}) \ge |\max\{\vec{v}^T \cdot \vec{x} \,|\, \vec{x} \in S_o\} + \max\{-\vec{v}^T \cdot \vec{x} \,|\, \vec{x} \in S_o\}|$ and $\gamma_u(\vec{v}) \le |\max\{\vec{v}^T \cdot \vec{x} \,|\, \vec{x} \in S_u\} + \max\{-\vec{v}^T \cdot \vec{x} \,|\, \vec{x} \in S_u\}|$. Then we compute the minimum width ratio $\gamma_{\min} = \min\{\gamma_u(\vec{v})/\gamma_o(\vec{v}) \,|\, \vec{v} \in V\}$ which gives an intuitive evaluation on the accuracy, i.e., the larger the value the better the approximation. In Table I and II, we present the experimental results on our benchmarks. The over- and under-approximations are the sets $\mathcal{F}_o$ and $\mathcal{F}_u$ respectively at time T. The vectors are selected along the dimensions (axis-aligned). It can be seen that our method found a valid under-approximation in most cases, and even could handle chaotic behaviors in reasonably long time horizons.

On one hand, our prototype produces interesting results on most of the benchmark examples. Since interval (as well as TM) based integration methods are very sensitive to the size of the initial set and the length of the time horizon, our under-approximation method underperforms on hard case studies, such as the test #6. However, there is still a lot of room for engineering improvements to our prototype implementations.

Acknowledgments. This work was supported by the DFG project HyPro, and in part, by the US National Science Foundation (NSF) under award # CNS-0953941. All opinions expressed are those of the authors and not necessarily of DFG or NSF.

References

[1] S. Sankaranarayanan, H. Sipma, and Z. Manna, "Constructing invariants for hybrid systems," in *Proc. HSCC'04*, ser. LNCS, vol. 2993. Springer, 2004, pp. 539–554.

[2] S. Gulwani and A. Tiwari, "Constraint-based approach for analysis of hybrid systems," in *Proc. CAV'08*, ser. LNCS, vol. 5123. Springer, 2008, pp. 190–203.

TABLE II

EVALUATION OF THE UNDER-APPROXIMATIONS FOR $\vec{x}(T)$ WITH INITIAL SETS AS *ellipsoids*. LEGENDS: SEE TABLE I.

#	systems	var	T	δ	k	time (s)	$\gamma_{\min}$
1	jet engine	2	4	0.02	4	45	∼0.8
2	jet engine	2	5	0.02	5	185	∼0.7
3	Brusselator	2	3	0.02	4	43	∼0.65
4	Brusselator	2	4	0.01	4	138	∼0.5
5	Rössler	3	1.5	0.01	5	153	∼0.4
6	Rössler	3	1.6	0.01	5	167	Fail
7	Lorenz	3	0.5	0.01	5	31	∼0.5
8	Lorenz	3	0.6	0.01	5	44	∼0.2
9	Shimizu-Morioka	3	1	0.01	5	55	∼0.5
10	Shimizu-Morioka	3	1.2	0.01	5	67	∼0.1
11	Lotka-Volterra	4	1	0.01	4	261	∼0.25
12	coupled Van-der-Pol	4	4	0.01	4	105	∼0.3
13	steam governor	5	2.5	0.01	5	16	∼0.2
14	biological system	7	0.2	0.002	3	581	∼0.15

[3] A. Platzer and E. Clarke, "Computing differential invariants of hybrid systems as fixedpoints," *FMSD*, vol. 35, no. 1, pp. 98–120, 2009.

[4] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi, "Beyond HYTECH: Hybrid systems analysis using interval numerical methods," in *Proc. HSCC'00*, ser. LNCS, vol. 1790. Springer, 2000, pp. 130–144.

[5] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *HSCC*, ser. LNCS, vol. 2289. Springer, 2005, pp. 258–273.

[6] T. Dang, O. Maler, and R. Testylier, "Accurate hybridization of nonlinear systems," in *Proc. HSCC '10*. ACM, 2010, pp. 11–20.

[7] A. Chutinan and B. Krogh, "Computing polyhedral approximations to flow pipes for dynamic systems," in *Proc. CDC'98*. IEEE, 1998.

[8] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *CAV*, ser. LNCS, vol. 2404. springer, 2002, pp. 365–370.

[9] A. Girard, "Reachability of uncertain linear systems using zonotopes," in *Proc. HSCC'05*, ser. LNCS, vol. 3414. Springer, 2005, pp. 291–305.

[10] C. Le Guernic and A. Girard, "Reachability analysis of hybrid systems using support functions," in *Proc. CAV'09*, ser. LNCS, vol. 5643. Springer, 2009, pp. 540–554.

[11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. CAV'11*, ser. LNCS, vol. 6806. Springer, 2011, pp. 379–395.

[12] I. M. Mitchell and C. Tomlin, "Level set methods for computation in hybrid systems," in *Proc. HSCC'00*, ser. LNCS, vol. 1790. Springer, 2000, pp. 310–323.

[13] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[14] I. M. Mitchell and J. A. Templeton, "A toolbox of hamilton-jacobi solvers for analysis of nondeterministic continuous and hybrid systems," in *Proc. HSCC'05*, ser. LNCS, vol. 3414. Springer, 2005, pp. 480–494.

[15] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss, "Validated solutions of initial value problems for ordinary differential equations," *Applied Mathematics and Computation*, vol. 105, no. 1, pp. 21–68, 1999.

[16] M. Berz and K. Makino, "Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models," *Reliable Computing*, vol. 4, pp. 361–369, 1998.

[17] N. Ramdani and N. S. Nedialkov, "Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques," *Nonlinear Analysis: Hybrid Systems*, vol. 5, no. 2, pp. 149–162, 2011.

[18] P. Prabhakar and M. Viswanathan, "A dynamic algorithm for approximate flow computations," in *Proc. HSCC'11*. ACM, 2011, pp. 133–142.

[19] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Taylor model flowpipe construction for non-linear hybrid systems," in *Proc. RTSS'12*. IEEE, 2012, pp. 183–192.

[20] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," in *Proc. VMCAI'13*, ser. LNCS, vol. 7737. Springer, 2013, pp. 48–67.

[21] N. Delanoue, L. Jaulin, and B. Cottenceau, "Using interval arithmetic to prove that a set is path-connected," *Theoretical Computer Science*, vol. 351, no. 1, pp. 119 – 128, 2006.

[22] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Proc. of CAV'13*, ser. LNCS, vol. 8044. Springer, 2013, pp. 258–263.

[23] A. Girard, C. Le Guernic, and O. Maler, "Efficient computation of reachable sets of linear time-invariant systems with inputs," in *Proc. HSCC'06*, ser. LNCS, vol. 3927. Springer, 2006, pp. 257–271.

[24] C. Le Guernic, "Reachability analysis of hybrid systems with linear continuous dynamics," Ph.D. dissertation, Université Joseph Fourier, 2009.

[25] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *Proc. DATE'06*, 2006, pp. 257–262.

[26] I. M. Mitchell, "Comparing forward and backward reachability as tools for safety analysis," in *HSCC*, ser. LNCS, vol. 4416. Springer, 2007, pp. 428–443.

[27] E. Gardeñes, M. Sainz, L. Jorba, R. Calm, R. Estela, H. Mielgo, and A. Trepat, "Modal intervals," *Reliable Computing*, vol. 7, no. 2, pp. 77–111, 2001.

[28] E. Goubault and S. Putot, "Under-approximations of computations in real numbers based on generalized affine arithmetic," in *Proc. SAS'07*, ser. LNCS, vol. 4634. Springer, 2007, pp. 137–152.

[29] E. Goubault, O. Mullier, and S. P. amd M. Kieffer, "Inner approximated reachability analysis," 2014, to Appear (April 2014).

[30] B. Xue, "Computing rigor quadratic lyapunov functions and under-approximate reachable sets for ordinary differential equations," Ph.D. dissertation, Beihang University, 2013.

[31] S. Gao, J. Avigad, and E. Clarke, "δ-complete decision procedures for satisfiability over the reals," in *IJCAR*, ser. LNCS, vol. 7365, 2012, pp. 286–300.

[32] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *CADE*, ser. LNCS, vol. 7898. Springer, 2013, pp. 208–214.

[33] S. Gao, S. Kong, and E. Clarke, "Satisfiability modulo ODEs," in *FMCAD*, Oct 2013, pp. 105–112.

[34] J. D. Meiss, *Differential Dynamical Systems*. SIAM publishers, 2007.

[35] K. Makino and M. Berz, "Taylor models and other validated functional inclusion methods," *J. Pure and Applied Mathematics*, vol. 4, no. 4, pp. 379–456, 2003.

[36] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. SIAM, 2009.

[37] R. J. Lohner, "Computation of guaranteed enclosures for the solutions of ordinary initial and boundary value problems," in *Computational ordinary differential equations*, 1992, pp. 425–435.

[38] K. Makino and M. Berz, "Suppression of the wrapping effect by taylor model-based verified integrators: Long-term stabilization by preconditioning," *International Journal of Differential Equations and Applications*, vol. 10, no. 4, pp. 353 – 384, 2005.

[39] S. Basu, R. Pollack, and M.-F. Roy, *Algorithms in Real Algebraic Geometry*. Springer, 2006.

[40] F. Benhamou and L. Granvilliers, "Continuous and interval constraints," in *Handbook of Constraint Programming*. Elsevier, 2006, pp. 571–590.

[41] E. M. Aylward, P. A. Parrilo, and J.-J. E. Slotine, "Stability and robustness analysis of nonlinear systems via contraction metrics and SOS programming," *Automatica*, vol. 44, no. 8, pp. 2163 – 2170, 2008.

[42] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 1, pp. 209–236, 2007.

[43] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths," in *Proc. PLDI'13*, vol. 48. ACM, 2013, pp. 447–458.

[44] A. Kanade, R. Alur, F. Ivancic, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar, "Generating and analyzing symbolic traces of simulink/stateflow models," in *Proc. CAV'09*, ser. LNCS, vol. 5643. Springer, 2009, pp. 430–445.

[45] A. L. Shil'nikov, "On bifurcations of the Lorenz attractor in the Shimizu-Morioka model," *Phys. D*, vol. 62, no. 1-4, pp. 338–346, 1993.

[46] J. A. Vano, J. C. Wildenberg, M. B. Anderson, J. K. Noel, and J. C. Sprott, "Chaos in low-dimensional Lotka-Volterra models of competition," *Nonlinearity*, vol. 19, no. 10, pp. 2391–2404, 2006.

[47] J. Sotomayor, L. F. Mello, and D. de Carvalho Braga, "Bifurcation analysis of the Watt governor system," *Comput. Appl. Math.*, vol. 26, no. 1, 2007.

[48] E. Klipp, R. Herwig, A. Kowald, C. Wierling, and H. Lehrach, *Systems Biology in Practice: Concepts, Implementation and Application*. Wiley-Blackwell, 2005.

Disproving termination with overapproximation

Byron Cook*†, Carsten Fuhs†, Kaustubh Nimkar† and Peter O'Hearn†
*Microsoft Research
†University College London

***Abstract*—When disproving termination using known techniques (*e.g.* recurrence sets), abstractions that overapproximate the program's transition relation are unsound. In this paper we introduce *live abstractions*, a natural class of abstractions that can be combined with the recent concept of *closed recurrence sets* to soundly disprove termination. To demonstrate the practical usefulness of this new approach we show how programs with nonlinear, nondeterministic, and heap-based commands can be shown nonterminating using linear overapproximations.**

1. Introduction

A program is terminating *iff* its transition relation (when restricted to reachable states) is well-founded. Because every subrelation of a well-founded relation is itself well-founded, if we prove an abstraction that overapproximates the program to be terminating, then we have proved the concrete program terminating. The reverse, unfortunately, is not true: the existence of a nonterminating overapproximating abstraction does not imply that the original concrete program is nonterminating. Thus, when proving nontermination, we currently cannot make use of the many techniques from program analysis that overapproximate programs.

In this paper we revisit a recently introduced concept called a *closed recurrence set* [8]. The existence of a closed recurrence set for a program implies that the program does not terminate. Curiously, the existence of a closed recurrence set for an overapproximating abstraction (meeting certain restrictions, which we formalize as *live abstractions*) also implies nontermination of the original concrete program. Thus, when combined with our technique, we can now use overapproximating abstractions when attempting to prove nontermination.

To demonstrate the usefulness of our approach we describe an experimental evaluation where nonlinear, nondeterministic, and heap-based programs are proved to be nonterminating using off-the-shelf overapproximating linear abstractions.

Limitations. As discussed in detail in the paper: not all overapproximating abstractions are compatible with our approach. We address this problem by describing the conditions on abstractions that make the abstraction sound for our approach, as the notion of *live abstractions*. Many of the known abstractions indeed meet these conditions. Additionally, closed recurrence sets are not complete, *i.e.* in some cases a closed recurrence set will not exist for nonterminating programs. In these situations our approach can still help in combination with previous techniques to disprove termination (*e.g.* underapproximation) in cases where existing techniques alone could not.

In our automation, counterexamples to termination are expressed as *simple while loops*, *a.k.a.* lasso paths, which are used extensively in the termination and nontermination proving literature. Unfortunately, not all counterexamples to termination can be expressed as lassos (see *e.g.* [8, Section 4] for a program where only *aperiodic* nonterminating runs exist). Furthermore, as done in TNT [18], when disproving termination of real programs with complex control-flow graphs, we must first search for candidate lassos before applying our approach. Like TNT, our tool also exhaustively searches program's control flow graph for candidate lassos. Alternatively, candidate lassos can be obtained from a termination prover when it fails to prove termination. Thus our technique can be efficiently combined with a termination prover.

Related work. Termination proving tools are now well-known, *e.g.* [5], [6], [11], [12], [14], [15], [22], *etc.* The difference here is that we are *dis*proving, rather than proving termination. While in some trivial cases termination provers can easily disprove termination (*e.g.* when variables are not modified in an infinite loop), in practice this is not the focus for these tools. Failure to find a termination proof does not imply a proof of nontermination. Thus dedicated techniques for nontermination proving are essential.

Since termination is not a safety property, its falsification cannot always be witnessed by a finite trace; thus testing cannot reliably be used to identify termination bugs.

In recent work, Chen *et al.* [8] introduce the notion of closed recurrence sets, upon which we build in this paper. Chen *et al.* combine closed recurrence sets with counterexample-guided underapproximation to harness safety provers for proving nontermination. The method hinges on the availability of suitable safety provers for the regarded class of programs, which currently makes an application of their method to nonlinear or heap-based programs difficult. We go beyond this limitation.

Closed recurrence sets were inspired by TNT [18], which uses a characterization of nontermination by (open) recurrence sets. Note that closed recurrence sets are stronger than recurrence sets: a recurrence set exists *iff* a program is nonterminating, whereas closed recurrence sets only *imply* nontermination; that is why they are useful for approximation. We show that additional techniques can be used to mitigate the relative strength of the condition. In contrast to us, TNT is restricted to programs using linear arithmetic. Our approach supports unbounded nondeterminism in the program's transition relation, whereas TNT is restricted to deterministic commands. As discussed later, this is due to a happy interaction between the definition of closed recurrence sets and Farkas' lemma.

Larraz *et al.* [23] prove non-termination via Max-SMT solving. The method explores all strongly connected subgraphs

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

of a program's CFG and thus can find witnesses for nontermination that need not be lasso paths. However this approach is limited to linear arithmetic as well.

Termination analysis tools for constraint-logic programs (*e.g.* [30]) can in cases be used to prove nontermination of imperative programs (*e.g.* JULIA [31] can show nontermination for Java bytecode programs if the abstraction to constraint-logic programs is exact, but provides no witness like a recurrence set to the user). The main difficulty here is in the application of the tools to imperative programs, as overapproximating abstractions are typically used for converting languages such as Java and C to constraint-logic programs. These abstractions are in general unsound for directly proving nontermination. Our work may in fact have application in this domain.

APROVE [15] uses SMT solving to prove nontermination of Java programs [7]. First nontermination of a loop regardless of its context is proved, then reachability of the loop with suitable values. The drawback of their technique is that they require either that (after program slicing to the variables that influence the loop control flow) the values of the program variables are always the same at the loop header or that the loop conditions themselves must be loop invariants.

The tool INVEL [34] analyzes nontermination of Java programs using a combination of theorem proving and invariant generation. Like Brockschmidt *et al.* [7], we were unable to obtain a working version of INVEL. Note that in the empirical evaluation by Brockschmidt *et al.* [7], the APROVE tool (which we have compared against) subsumed INVEL on INVEL's data set. Finally, INVEL is only applicable to deterministic integer programs, whereas our approach allows nondeterminism and heap-based data structures as well.

Gulwani *et al.* [17] can prove nontermination in some cases by proving the exit points of the program unreachable, but use a restriction to linear arithmetic. Their technique is fairly imprecise in the presence of nondeterminism in the input.

Atig *et al.* [2] reduce nontermination of multithreaded programs to nontermination reasoning for sequential programs. Our work complements Atig *et al.*, as we improve the underlying sequential tools that future multithreaded tools can use.

Previous works (*e.g.* [10], [19], [32]) describe techniques for proving properties expressed in branching-time temporal logic of infinite-state programs. Nontermination can be encoded in these logics (*e.g.* in CTL, nontermination is $\mathsf{EG}\ pc \neq \mathsf{END}$). Our work complements these previous works. Here we facilitate the use of overapproximation.

Finally, several automatic tools exist for proving nontermination of term rewrite systems (*e.g.* [13], [16], [29]). However, in nontermination analysis for term rewriting the *entire* state space is considered as legitimate initial states for a (possibly infinite) evaluation sequence, whereas our setting also factors in reachability from the initial states.

2. ILLUSTRATING EXAMPLE

Before formally introducing our approach, we first describe the idea informally using an example. Imagine that we want to show nontermination of the toy program in Fig. 1(a). Here

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    skip; // location ℓ
done
```

(a)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    assume(i ≥ 1);
done
```

(b)

Fig. 1. Nonlinear program **(a)**, and its linear abstraction **(b)**. The command **assume** [27] only allows executions to continue when the condition holds, **nondet** represents nondeterministic choice.

we are using an **assume** statement [27], which does not allow executions to pass unless the condition is valid.[1]

We are looking to find initial values for i, j and k from which an infinite run is possible. Indeed, such a run is possible: from the state $(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1)$ the program can perform a sequence of loop iterations via the states $(\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2)$, $(\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3)$, $(\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4)$, $\ldots$ leading to an infinite run. This set of states $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2)\}, (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ldots\}$ meets a criterion that defines a recurrence set [18]: during the execution of the **while** loop the program can get into the set of states $\mathcal{G}$, and when in $\mathcal{G}$ it is possible to stay in $\mathcal{G}$ during an iteration of the loop. Finding a valid recurrence set such as this is a complete method of proving nontermination.

Now the question is, how can we *automatically* find such a proof of nontermination? The difficulty here is the nonlinear assignment $\mathsf{i} := \mathsf{j} \times \mathsf{k}$: most automatic formal verification techniques struggle to support nonlinear arithmetic in a scalable fashion. An arbitrary overapproximation of this program will not help in this context. The problem is that if we prove nontermination of the overapproximation we still have not proved nontermination of the original concrete program. The reason is that—due to the nature of overapproximation—a nonterminating execution in the overapproximation need not correspond to any execution in the concrete program.

To avoid this problem we can use an overapproximating abstraction of our program such that the abstraction satisfies certain conditions. We call such an abstraction a live abstraction. See Section 3. Such an abstraction is shown in Fig. 1(b). This abstraction uses nondeterministic choice (*i.e.* **nondet**) to abstract away the nonlinear command and also uses a linear location invariant at location ℓ from the original program $(\mathsf{i} \geq 1)$. Note that in Fig. 1(b) we do not alter the loop condition from the original program but only overapproximate the transitions that can take place inside the loop. This abstraction is a live abstraction and is thus a safe abstraction for our approach. Later in Section 3 we give the necessary conditions for an abstraction to be a live abstraction. Most of the abstractions used in the termination literature satisfy the properties of a live abstraction.

Our approach is based on the following insight: if we can

[1]For termination, we can encode **assume**$(e) \equiv$ **if** $\neg e$ **then exit**(); **fi**

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    m := nondet();
done
```

(a)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    m := nondet();
    assume(m ≥ 0);
done
```

(b)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    m := nondet();
    assume(m ≥ 0);
    assume(i ≥ 1);
done
```

(c)

Fig. 2. Nonlinear program **(a)**, its underapproximation **(b)**, and the resulting linear abstraction **(c)**.

prove existence of a set of states $\mathcal{G}$ at the loop head in the live abstraction meeting the following conditions then we know that both the abstraction and the original concrete program are nonterminating: **a)** $\mathcal{G}$ is nonempty and at least one state in $\mathcal{G}$ is reachable, **b)** every state in $\mathcal{G}$ has at least one transition, and **c)** all transitions from $\mathcal{G}$ in the abstraction only lead to $\mathcal{G}$. If these conditions hold then $\mathcal{G}$ is a closed recurrence set. This now allows us to use tools on the overapproximating abstraction rather than the original program to establish nontermination. Here such a set could be given by $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1\}$.

Combining over- and underapproximation. Sometimes closed recurrence sets are alone not enough: we may still require the use of underapproximation. However, even then, our approach facilitates the *mixture* of over- and underapproximation to make more powerful nontermination proving tools.

Consider the program in Fig. 2(a). Here it is difficult to find a useful linear overapproximation directly because of the nondeterministic assignment to the variable m. However if an underapproximation of a program is nonterminating, then the original program itself is nonterminating as well. Here we can use known techniques to automatically find an underapproximation that rules out the unwanted transitions. Consider the program in Fig. 2(b), an underapproximation of the program in Fig. 2(a) restricting the choice for nondeterministic assignment to the variable m. Using our approach we can now easily find a useful linear overapproximation that is a live abstraction for this program. The program in Fig. 2(c) is a linear *over*approximation of the *under*approximation in Fig. 2(b). Here, we can find a closed recurrence set $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1 \wedge \mathsf{m} \geq 0\}$ for the program in Fig. 2(c), which proves nontermination of the program in Fig. 2(b), which in turn proves nontermination of the program in Fig. 2(a). Note that it is unsound to first overapproximate and then underapproximate: as in this example we must first underapproximate and then overapproximate. Also note that for overapproximations we only consider live abstractions.

3. Closed recurrence sets and overapproximation

In this section we discuss *closed recurrence sets* and their relationship to overapproximation.

Transition Systems. A *transition system* (S, R, I, F) is defined by a set of states S, a *transition relation* $R \subseteq S \times S$, a set of *initial states* $I \subseteq S$ and a set of *final states* $F \subseteq S$. For a state s with $R(s, s')$, we say that s' is a *post-state* of s and that s is a *pre-state* of s'. We also call s' a *successor of s under R*. Execution of a transition system can only halt in a final state, so every state $s \notin F$ must have a successor under R, and any final state $f \in F$ has no successors under R.

Example. Consider the example in Fig. 1(a). We can describe the loop and its initial condition as a transition system (S, R, I, F) where any state s is basically a tuple $(\mathsf{i}, \mathsf{j}, \mathsf{k})$ of values of variables and $S = \mathbb{Z}^3$, $R = \{(s, s') \mid s, s' \models \mathsf{i} \geq 0 \wedge \mathsf{i}' = \mathsf{j} \times \mathsf{k} \wedge \mathsf{j}' = \mathsf{j}+1 \wedge \mathsf{k}' = \mathsf{k}+1\}$, $I = \{s \mid s \models \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\}$, $F = \{s \mid s \models \mathsf{i} < 0\}$.

A. Closed recurrence sets.

A transition system (S, R, I, F) is nonterminating *iff* there exists an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ with $s_0 \in I$. Gupta *et al.* [18] characterize nontermination of a relation R by the existence of a *recurrence set*, *viz.* a nonempty set of states $\mathcal{G}$ such that for each $s \in \mathcal{G}$ there exists a transition to some $s' \in \mathcal{G}$. Here we extend the notion of a recurrence set to transition systems. A transition system (S, R, I, F) has a *recurrence set* (or *open recurrence set*) of states $\mathcal{G}(s)$ *iff*

$$\exists s. \mathcal{G}(s) \wedge I(s), \tag{1}$$

$$\forall s \exists s'. \mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s'). \tag{2}$$

A transition system (S, R, I, F) is nonterminating *iff* it has a recurrence set of states.

Quantifier alternation as in Condition (2) can be a headache for automation. To avoid this problem Gupta *et al.* [18] restrict the transition relation to deterministic programs only. In this case we can represent the post-state s' using a unique expression in terms of the pre-state s. Thus the existential quantifier can be eliminated by instantiating it with this expression.

Example. For the loop from Fig. 1(a), we can have a recurrence set $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2), (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ \ldots\ \}$.

Definition (Closed Recurrence Set [8])
A set $\mathcal{G}$ is a closed recurrence set *for a transition system (S, R, I, F)* iff *the following three conditions hold:*

$$\exists s. \mathcal{G}(s) \wedge I(s) \tag{3}$$

$$\forall s \exists s'. \mathcal{G}(s) \rightarrow R(s, s') \tag{4}$$

$$\forall s \forall s'. \mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s') \tag{5}$$

In contrast to standard (open) recurrence sets, we now require a purely universal property: for each $s \in \mathcal{G}$ and *for each* of its successors s', also s' must be in the recurrence set (Condition (5)). So instead of requiring that we *can* stay in the recurrence set, we now demand that we *must* stay in the recurrence set. This has several advantages. First, without quantifier alternation, Farkas' lemma can now be applied directly. This now helps us to incorporate nondeterministic transition systems too. Secondly, the interaction with overapproximation is improved. The downside is that the condition can be too strong.

There is an additional problem: what if a state s in our recurrence set $\mathcal{G}$ has no successor s' at all? This would bring our alleged infinite transition sequence to a sudden halt, yet our universal formula would trivially hold. To deal with this issue, we must impose that each $s \in \mathcal{G}$ has *some* successor s' (Condition (4)). But this existential statement need not mention that s' must be in $\mathcal{G}$ again—our previous *universal* statement already takes care of this. In this way, we have gained something: the existential quantifier in Condition (4) refers only to the (known) transition relation R and, as we shall see in the Section 5 on automation, the condition can be easily automated in spite of quantifier alternation when we search for a closed recurrence set $\mathcal{G}$.

Example. For the loop from Fig. 1(b), we can have a closed recurrence set $\mathcal{G} = \{s \mid s \vDash \mathsf{i} \geq 1\}$. $\mathcal{G}$ satisfies all the conditions of a closed recurrence set.

Theorem (Closed Recurrence Sets are Recurrence Sets [8]) *Let $\mathcal{G}$ be a closed recurrence set for (S, R, I, F). Then $\mathcal{G}$ is also a standard (open) recurrence set for (S, R, I, F).*

If our transition system is *deterministic*, every recurrence set is also a closed recurrence set. In particular, closed recurrence sets characterize nontermination in the setting of Gupta *et al.* [18], which assumes deterministic programs.

Corollary (Recurrence Sets are Closed Recurrence Sets for Deterministic Transition Systems)
Let $\mathcal{G}$ be a recurrence set for (S, R, I, F) such that for every state s there exists at most one state s' with $R(s, s')$. Then $\mathcal{G}$ is also a closed recurrence set for (S, R, I, F).

B. Live abstractions

We now describe generic conditions on abstractions that are sufficient to establish soundness for nontermination proving using our approach, in the form of *live abstractions*.

Live Abstractions. We assume that an abstraction of $T = (S, R, I, F)$ is a system $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$, with a concretion (or meaning) function $[\![\cdot]\!] : S^\alpha \to \mathcal{P}(S)$.

Definition (Live Abstraction)
An abstraction $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ is live *iff*

$\forall s \forall s' \forall a.\, R(s, s') \wedge s \in [\![a]\!] \to \exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!]$
(Simulation)

$\forall f \forall g.\, f \in F \wedge f \in [\![g]\!] \to g \in F^\alpha$
(Upward Termination)

The *Simulation* (or, 'up simulation') condition is a standard one for overapproximation: it says that any steps you can take in the concrete transition system can be overapproximated in the abstract transition system. The *Upward Termination* condition says that for every final state in the concrete transition system, any corresponding abstract state is also a final state in the abstract transition system. Together *Simulation* and *Upward Termination* imply that for every terminating run in the concrete transition system, also any corresponding run in the abstract transition system is terminating.

The connection of these conditions to disproving termination then is: if there is an initial state a_0 from which all computations in the abstract program are nonterminating and there is an initial state s_0 in the concrete program such that $s_0 \in [\![a_0]\!]$, then all computations in the concrete program starting from s_0 are nonterminating (*i.e.*, for live abstractions, *closed* recurrence carries over from the abstract to the concrete).

Theorem (Soundness)
Consider a live abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ for a transition system (S, R, I, F). Suppose $\mathcal{G}^\alpha$ is a closed *recurrence set for $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ and for some a_0 we have $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Then there also exists a closed recurrence set $\mathcal{G} = \{s \mid \exists a.\, \mathcal{G}^\alpha(a) \wedge s \in [\![a]\!]\}$ for (S, R, I, F).*

Proof. We need to prove Conditions (3), (4), and (5) for $\mathcal{G}$.

For Condition (3) for $\mathcal{G}$: We have for some a_0, $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Thus for such s_0 we have $I(s_0)$ and the definition of $\mathcal{G}$ implies $\mathcal{G}(s_0)$. Thus we have Condition (3) for $\mathcal{G}$.

For Condition (4) for $\mathcal{G}$: Let s such that $\mathcal{G}(s)$. We now prove that $s \notin F$ by contradiction. Suppose $s \in F$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Condition (4) for $\mathcal{G}^\alpha$ implies $\exists a'.R^\alpha(a, a')$. However *Upward Termination* implies $a \in F^\alpha$, which implies $\neg \exists a' R^\alpha(a, a')$. Thus we have a contradiction. Thus we must have $s \notin F$. This gives Condition (4) for $\mathcal{G}$.

For Condition (5) for $\mathcal{G}$: Let s, s' such that $\mathcal{G}(s) \wedge R(s, s')$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Moreover, the *Simulation* condition gives $\exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!]$. Condition (5) for $\mathcal{G}^\alpha$ implies $\mathcal{G}^\alpha(a')$. The definition of $\mathcal{G}$ gives $\mathcal{G}(s')$ and thus we have Condition (5) for $\mathcal{G}$. □

Note that similar to what many abstractions do, a live abstraction can overapproximate the concrete initial states. For a live abstraction to be useful for proving nontermination using closed recurrence sets, we only need $a_0 \in S^\alpha$ and $s_0 \in S$ that satisfy the conditions of the soundness theorem.

Example. Recall Fig. 1(a) and its abstraction in Fig. 1(b). We can represent the abstraction as a transition system:

$$I^\alpha = \{a \mid a \vDash \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\} \qquad F^\alpha = \{a \mid a \vDash \mathsf{i} < 0\}$$
$$S^\alpha = \mathbb{Z}^3 \qquad R^\alpha = \{(a, a') \mid (a, a') \vDash \mathsf{i} \geq 0 \wedge \mathsf{i}' \geq 1 \wedge \mathsf{j}' = \mathsf{j} + 1 \wedge \mathsf{k}' = \mathsf{k} + 1\}$$

The abstraction contains $\mathsf{i}' \geq 1$ in the transition relation of the loop instead of the nonlinear update $\mathsf{i}' = \mathsf{j} \times \mathsf{k}$. Here the abstraction has not changed the state space, the set of initial states and the set of final states, but it has weakened

the transition relation of the loop. Note that this abstraction fulfills all criteria for a live abstraction.

Example. Consider again the examples from Fig. 1(a) and (b). Here we have the closed recurrence set $\mathcal{G}^\alpha = \{s \mid s \models \mathsf{i} \geq 1\}$ for the loop in our abstraction in Fig. 1(b). This implies existence of a closed recurrence set $\mathcal{G}$ for the loop in the concrete program in Fig. 1(a) and hence its nontermination.

Example. To see why we need the *Upward Termination* condition for the abstraction, consider the following transition system (S, R, I, F) and its abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$:

$$S = \{s_0, s_1, s_2, s_3\} \quad I = \{s_0\} \quad F = \{s_1\}$$

$$R = \{(s_0, s_1), (s_2, s_3), (s_3, s_0)\}$$

$$S^\alpha = \{\{s_0\}, \{s_1, s_2\}, \{s_3\}\} \quad I^\alpha = \{\{s_0\}\} \quad F^\alpha = \emptyset$$

$$R^\alpha = \{(\{s_0\}, \{s_1, s_2\}), (\{s_1, s_2\}, \{s_3\}), (\{s_3\}, \{s_0\})\}$$

Here an abstract state is a subset of the set of all concrete states, where we have "merged" the states s_1 and s_2 to a single state. The abstraction satisfies the *Simulation* condition but not *Upward Termination* because s_1 is a final state in the concrete transition system, but the corresponding abstract state $\{s_1, s_2\}$ is not a final state in the abstract transition system. The abstraction has a closed recurrence set $\{\{s_0\}, \{s_1, s_2\}, \{s_3\}\}$, but the concrete transition system has no recurrence set.

4. Classes of Live Abstractions for Automation

As mentioned earlier, in our automation we focus on program fragments of a special shape: lassos.

Definition (Lasso) A *lasso* is a program fragment that contains a sequence of commands called a *stem* followed by a *simple loop* with guarded updates. The *guard* of a simple loop is a conjunction of atomic conditions. Formally a lasso L is a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ where S is the set of states in the domain, R_{loop} is the transition relation of the loop, and I_{loop} is the set of initial states for the loop. I_{loop} represents the strongest postcondition after execution of the stem. F_{loop} is a set of final states for the loop such that for every final state there is no transition inside the loop.

Abstracting nonlinear commands. We describe the abstraction that our tool uses to abstract nonlinear commands present in the lassos. In our abstraction nonlinear assignment commands are abstracted, but loop guards are kept unchanged.

Towards the purpose of abstracting assignments we first compute a linear location invariant at the end of the loop (using APRON's [20] octagon abstract domain [26] in our implementation). We then replace the nonlinear update command with a nondeterministic choice and add an assume statement with the invariant at the end of the loop. Instead of octagons, here also dedicated disjunctive analyses for nonlinearity (*e.g.* the technique by Alonso *et al.* [1]) can be used to increase precision of the overapproximation. However, as our experiments show, here we can already get quite far using standard octagons.

Consider the nonlinear lasso in Fig. 1(a) and its linear abstraction in Fig. 3 that our tool computes. Here, $\mathsf{i} - 1 \geq 0 \land \mathsf{i} + \mathsf{j} - 3 \geq 0 \land \mathsf{i} - \mathsf{j} + 1 \geq 0 \land \mathsf{i} + \mathsf{k} - 3 \geq 0$ is the invariant computed at location ℓ of the original lasso from Fig. 1(a) by the APRON library using the octagon abstract domain.

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    assume(i − 1 ≥ 0 and i + j − 3 ≥ 0 and
        i − j + 1 ≥ 0 and i + k − 3 ≥ 0); // location ℓ
done
```

Fig. 3. Linear overapproximation of the program in Fig. 1(a) computed by our tool using APRON [20]

```
                             assume(...);
assume(...);                 assume(...);
assume(...);                 v := i × j;
while i × j ≥ 0 do           while v ≥ 0 do
    i := ...                     i := ...
    j := ...                     j := ...
    ......                       ......
done                             v := i × j;
                             done
        (a)                          (b)
```

Fig. 4. Lasso **(a)** with nonlinear guards and equivalent lasso **(b)** with auxiliary variable with linear guards

Mapping nonlinear assignments to nondeterministic assignments is clearly an overapproximation. This abstraction of assignments satisfies the *Simulation* condition of live abstraction because it adds extra abstract transitions only when a concrete transition (the assignment) is already possible. Since we do not alter loop guards, *Upward Termination* holds as well because all the final states of the original lasso are final states in the abstract lasso too. Clearly this abstraction satisfies the conditions of a live abstraction. Formally for a concrete lasso with a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ our tool computes an abstract lasso with a transition system $(S^\alpha, R^\alpha_{loop}, I^\alpha_{loop}, F^\alpha_{loop})$ where $S^\alpha = S, R_{loop} \subseteq R^\alpha_{loop}, I^\alpha_{loop} = I_{loop}, F^\alpha_{loop} = F_{loop}$ and the concretion function is essentially the identity, *i.e.*, $\forall a \in S^\alpha.\ [\![a]\!] = \{a\}$.

Dealing with nonlinear guards. We use a simple trick to get rid of nonlinearity out of guards. Consider Fig. 4. We remove nonlinearity present in the guards by adding an auxiliary variable v. The rest of the analysis proceeds as before.

This approach yields nonlinear commands in the stem of our lassos. The stem commands enter our constraints only existentially (as we will see in Section 5). Thus constraint solvers can deal with such constraints efficiently.

Abstracting heap-based commands. Magill *et al.* [25] propose an overapproximating abstraction from programs operating on the heap to purely arithmetic programs. The abstraction is obtained by instrumenting a memory safety proof for the program. Since in general memory safety only holds under certain preconditions, the user can specify the shape of the

```
while p ≠ null do
    p := p→next;
done
```

```
while k ≥ 1 do
    assume(k > 1);
    l := nondet();
    assume(l ≥ 1 and k = l + 1);
    m := nondet();
    assume(m = l + 1);
    n := nondet();
    assume(n = l + 1);
    k := n;
done
```

(a) **(b)**

Fig. 5. Heap-based program **(a)** with precondition that p points to a nonempty cyclic list and linear overapproximation **(b)** computed by THOR [25]

heap data structures by user-defined predicates in separation logic [28]. We can use Magill's tool THOR [25] to abstract heap-based C programs into linear arithmetic programs operating over the integers. This is exemplified in Fig. 5. In the arithmetic program the variable k tracks the length of the list segment from p to null, and the other variables are temporaries used in the update of k.

Magill's PhD thesis [24, Def. 29] describes the notion of stuttering simulation and proves (in his Thm. 18) that the abstraction satisfies the properties of stuttering simulation. In stuttering simulation for a transition in the concrete system, the corresponding transition in the abstract system may contain a sequence of steps and vice versa. An abstraction satisfying stuttering simulation obeys standard simulation condition and additionally for stuttering simulation to hold, the *Upward Termination* condition is needed. Thus Magill's abstraction satisfies the properties of a live abstraction and thus is safe for our approach of nontermination proving.

We could also abstract linked-list programs via the results connecting lists and counter automata [4]. These results are in fact stronger, a bisimulation rather than a simulation, for lists.

Combining over- and underapproximation. As previously mentioned, closed recurrence sets must in some cases be used in conjunction with underapproximation. Here we can use existing techniques for underapproximation in combination with our own. Note that closed recurrence sets form a complete method when combined with underapproximation, in the sense that every nonterminating program also has an underapproximation with a closed recurrence set.

Underapproximation. We call a transition system (S, R', I', F') an underapproximation of a transition system (S, R, I, F) *iff* $R' \subseteq R$, $I' \subseteq I$, $F \subseteq F'$.

Theorem (Open Recurrence Sets Always Contain Closed Recurrence Sets [8]) *There exists a recurrence set* $\mathcal{G}$ *for* (S, R, I, F) iff *there exist an underapproximation* (S, R', I', F') *of* (S, R, I, F) *and* $\mathcal{G}' \subseteq \mathcal{G}$ *such that* $\mathcal{G}'$ *is a closed recurrence set for* (S, R', I', F').

5. Finding Closed Recurrence Sets

In the previous section we showed how it is possible to prove nontermination of a program by proving the existence of a closed recurrence set for an abstraction of the program. Here we address the problem of how to *find* a closed recurrence set for the abstracted program, *i.e.*, a program over linear integer arithmetic. We will search for a closed recurrence set $\mathcal{G}$ described by a conjunction of linear inequalities $Q\boldsymbol{x} \leq q$.

We adapt the Farkas-based approach used in TNT to find closed recurrence sets rather than recurrence sets. In our application the restriction to deterministic relations from TNT can be lifted. This is particularly important when working with abstractions of programs, which can introduce nondeterminism even when the concrete program is deterministic. It is also essential for treating the heap, because of the nondeterminism inherent in `malloc`.

In this section it will be convenient to phrase our discussion in terms of lassos expressed in linear arithmetic, as such lassos are convenient for automation. In the domain of linear arithmetic, a state s is just a vector $\boldsymbol{x}$ that represents the valuation of program variables. A lasso L in linear arithmetic can be expressed as a transition system $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$. In terms of programs, $I_{loop}(\boldsymbol{x})$ represents the strongest postcondition of a path leading to the loop body, with precondition 'true' from which the program starts, and $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ is the transition relation corresponding to the composition of a sequence of (possibly nondeterministic) assignment statements in the loop body, guarded by a condition. $F_{loop}(\boldsymbol{x})$ represents the set of final states such that no loop transition can take place from any final state. As we are working in linear arithmetic, we can represent the transition relation of the loop by systems of inequalities

$$R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \triangleq G\boldsymbol{x} \leq g \wedge U\boldsymbol{x} + U'\boldsymbol{x'} \leq u$$

where $G\boldsymbol{x} \leq g$ describes the guards and $U\boldsymbol{x} + U'\boldsymbol{x'} \leq u$ the updates. Here G, U and U' are matrices, g and u are vectors. We make the following assumption:

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}. G\boldsymbol{x} \leq g \rightarrow U\boldsymbol{x} + U'\boldsymbol{x'} \leq u. \quad (6)$$

The assumption says that whenever the guards of a lasso can be satisfied we are guaranteed to have a next state given by the updates. This holds in a lasso with a satisfiable transition system when every row in U' contains a non-zero coefficient, which corresponds to an update of the variables.

We are in search of a predicate $\mathcal{G}$ expressed as a system of inequalities using coefficients, *i.e.* $\mathcal{G} \equiv Q\boldsymbol{x} \leq q$, where Q is a matrix and q a vector of existentially quantified variables. The number of rows in Q and q then corresponds to the number of inequalities which we use.

We wish to employ a constraint solver (*e.g.* Z3 [21]) to find the coefficients Q and q. A difficulty in doing so is that these conditions contain mixtures of existential and universal quantifiers: Q and q are existentially quantified at the top-level, and both (4) and (5) use universals. Many constraint solvers struggle to solve problems such as these. The standard approach (*e.g.* in invariant generation [9], rank function synthesis [5] and recurrence set synthesis [18]) is to apply Farkas' lemma to convert the problem into a purely existential one that is easier for existing solvers.

In the remainder of this section we describe a Farkas-based reduction to automate the search for closed recurrence sets. To find a closed recurrence set for $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$ we must find Q and q such that the following conditions are satisfied (here we have substituted $Q\boldsymbol{x} \leq q$ for $\mathcal{G}$ in Conditions (3), (4), and (5)):

$$\exists \boldsymbol{x}.Q\boldsymbol{x} \leq q \wedge I_{loop}(\boldsymbol{x}) \tag{7}$$

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}.Q\boldsymbol{x} \leq q \rightarrow R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \tag{8}$$

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.Q\boldsymbol{x} \leq q \wedge R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \rightarrow Q\boldsymbol{x'} \leq q \tag{9}$$

In order to apply Farkas' lemma we must eliminate the $\forall\exists$ alternation in Condition (8).[2] Assumption (6) lets us remove the existential quantifier in (8),[3] which now becomes:

$$\forall \boldsymbol{x}.Q\boldsymbol{x} \leq q \rightarrow G\boldsymbol{x} \leq g \tag{10}$$

Next, although it is not essential, because of (10) we can drop $G\boldsymbol{x} \leq g$ from $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ in (9), thus giving us a simpler constraint to solve:

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.Q\boldsymbol{x} \leq q \wedge U\boldsymbol{x} + U'\boldsymbol{x'} \leq u \rightarrow Q\boldsymbol{x'} \leq q \tag{11}$$

Conditions (7), (10), and (11) are sufficient constraints for finding a closed recurrence set. Furthermore, (10) and (11) are now in a form which facilitates applications of Farkas' lemma to eliminate the universal quantifiers, and we obtain:

$$\exists \Lambda_1 \geq 0.\Lambda_1 Q = G \wedge \Lambda_1 q \leq g \tag{12}$$

and

$$\exists \Lambda_2 \geq 0.\Lambda_2 \begin{pmatrix} Q \\ U \end{pmatrix} = 0 \wedge \Lambda_2 \begin{pmatrix} 0 \\ U' \end{pmatrix} = Q \wedge \Lambda_2 \begin{pmatrix} q \\ u \end{pmatrix} \leq q \tag{13}$$

The constraints that we finally generate are (7), (12), and (13). These conditions are readily solved by off-the-shelf constraint solving tools. A satisfying assignment for these constraints gives us values of coefficients in Q and q, thus giving us the closed recurrence set.

Note that if the constraints are unsatisfiable, like Gupta *et al.* [18] we can use Q and q with increasingly many rows (and hence inequalities) in $Q\boldsymbol{x} \leq q$. In this way, we can increase the precision of our method further.

6. Implementation and Experiments

In order to assess the practicality of our approach we have developed a prototype implementation called ANANT. Given a program's CFG, ANANT exhaustively searches for candidate lassos.[4] For every lasso the tool applies our method, using Z3 as the constraint solver for the constraints from Section 5 together with abstractions for heap and nonlinear commands described in Section 3. If a lasso under consideration contains a loop variable with a nondeterministic update that also appears in the loop guard, before applying the abstraction the tool first applies an underapproximation strategy. To obtain the desired underapproximation the tool adds an **assume**-statement at the end of the loop body that enforces the loop guard (as done for variable m in Fig. 2(b)).

We make ANANT available for download along with its source code at the following URL:

http://www0.cs.ucl.ac.uk/staff/K.Nimkar/live-abstraction

We compared ANANT experimentally to several other tools. As a benchmark set (also available at the above URL), we have gathered 33 example programs containing nonlinear, nondeterministic and heap-based commands from various sources.

Since nontermination usually indicates a bug, some of our benchmarks implement functions computing factorial, logarithm, *etc.*, with typical programming mistakes that lead to nontermination. The set also includes the nonterminating examples from Berdine *et al.* [3], in particular the bug in a Windows device driver discussed in this paper. While Berdine *et al.* report that their analysis uncovers this bug by absence of a successful termination proof, we can now go a step further and actually *prove* nontermination of such heap programs.

We compared ANANT to the following tools:

- APROVE [15], using the Java bytecode frontend with the nontermination analysis by Brockschmidt *et al.* [7].
- JULIA [33], implementing a reduction to constraint logic programming described by Payet and Spoto [31].

Like Brockschmidt *et al.* [7], we were unable to obtain a working version of the tool INVEL [34]. Note that the other nontermination provers (*e.g.* TNT [18], T2 [8] and CPPINV [23]) are not applicable, as they do not support programs with nonlinear or heap-based commands.

Fig. 6 shows the results of our experiments with ANANT, APROVE, and JULIA. We ran ANANT and APROVE on an Intel i7-2640M CPU clocked at 2.8 GHz under Linux. For JULIA, an unknown cloud-based configuration was used. All tools were run with 600 s timeout. As Fig. 6 shows, ANANT succeeded on 29 of 33 benchmarks, whereas APROVE and JULIA succeeded on only 2 and 4 benchmarks, respectively. This difference is not surprising since overapproximation was thus far not applicable to *dis*proving termination for nonlinear and heap-based programs. In contrast, as our experiments show, we can now disprove termination in many such cases.

It is worth highlighting that *e.g.* on benchmark 9, ANANT took over 4 min to disprove termination, *vs.* JULIA's <7 s. This difference may partly be due to different machine configurations. However, note that a combined prover for termination and nontermination (like APROVE or JULIA) can discard parts of the program proved terminating and only analyze the rest for nontermination. This can lead to a more focused search for a nontermination proof than ANANT's approach of enumerating arbitrary lassos (whose termination might be easy to prove). Thus, ideally, our contributions for disproving termination should be combined with a termination prover.

[2] When Gupta *et al.* [18] search for recurrence sets, they also need to eliminate the $\forall\exists$ alternation in their constraints for automation. They do so by instantiating the existential variable explicitly with the value of the update. The price for this is that the update must be deterministic. We do not have this restriction.

[3] The statements (6) $\wedge$ (8) and (6) $\wedge$ (10) are equivalent.

[4] ANANT uses the same syntax for transition systems as the termination prover T2 [6]. For heap-based programs in C syntax, the lasso extraction is currently conducted manually.

	ANANT		APROVE		JULIA	
Benchmark	Res	Runtime	Res	Runtime	Res	Runtime
1	✓	0.50 s	×	*timeout*	×	7.01 s
2	✓	0.55 s	×	*timeout*	×	7.80 s
2a	✓	0.82 s	×	*timeout*	×	12.01 s
3	✓	0.56 s	×	*timeout*	×	7.74 s
4	✓	125.66 s	×	*timeout*	×	12.85 s
5	✓	0.45 s	×	18.59 s	×	7.24 s
6	✓	0.48 s	×	235.79 s	✓	7.70 s
7	✓	0.59 s	×	23.51 s	✓	11.83 s
8	✓	0.26 s	×	3.15 s	✓	5.08 s
9	✓	243.00 s	×	5.10 s	✓	6.72 s
10	✓	246.83 s	×	27.42 s	×	11.29 s
11	✓	0.63 s	×	*timeout*	×	8.69 s
12	×	2.35 s	×	*timeout*	×	10.67 s
13	×	1.40 s	×	108.61 s	×	8.54 s
14	✓	121.69 s	×	147.54 s	×	7.33 s
15	✓	131.80 s	×	*timeout*	×	8.45 s
16	✓	57.41 s	×	18.81 s	×	7.07 s
17	✓	0.54 s	×	24.18 s	×	7.06 s
18	×	0.66 s	×	28.03 s	×	6.92 s
19	✓	0.44 s	×	*timeout*	×	7.27 s
20	×	0.74 s	×	*timeout*	×	6.95 s
factorial	✓	0.38 s	×	*timeout*	×	7.57 s
log	✓	0.46 s	×	3.17 s	×	8.59 s
log_by_mul	✓	0.63 s	×	*timeout*	×	7.68 s
lasso_ex1	✓	0.45 s	×	*timeout*	×	7.03 s
lasso_ex2	✓	1.21 s	×	72.25 s	×	8.79 s
lasso_ex3	✓	0.48 s	×	*timeout*	×	7.28 s
nCr_combi	✓	0.70 s	×	10.45 s	×	17.26 s
power	✓	0.43 s	×	*timeout*	×	7.03 s
Create	✓	3.47 s	✓	1.75 s	×	4.94 s
Insert	✓	177.69 s	×	16.86 s	×	7.77 s
Traverse	✓	1.23 s	✓	2.12 s	×	50.28 s
WindowsBug	✓	21.69 s	×	14.46 s	×	50.92 s

Fig. 6. Results ("Res") and runtimes of ANANT, APROVE, and JULIA on 29 benchmarks with nonlinear arithmetic and 4 heap-based benchmarks from Berdine *et al.* [3]. Here ✓ denotes that the tool proved nontermination, × means that the tool returned without a definite answer, and *timeout* means that the run was terminated externally after 600 s.

7. CONCLUSION

Overapproximation is the workhorse of program analysis. Unfortunately, overapproximation can invalidate conventional techniques for disproving termination. In this paper we have introduced the notion of a *live abstraction* to show how overapproximation can *help*, not hinder nontermination proving. The idea is to prove the existence of a *closed recurrence set* rather than simply a recurrence set. This modification in strategy allows us to use off-the-shelf overapproximating abstractions, leading to a new set of methods for disproving termination of real programs.

Acknowledgments. We thank the anonymous reviewers for helpful suggestions and Fabian Emmes and Fausto Spoto for help with the experiments.

REFERENCES

[1] Diego Esteban Alonso, Puri Arenas, and Samir Genaim. Handling non-linear operations in the value analysis of COSTA. In *Proc. BYTECODE '11*.

[2] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *Proc. CAV '12*.

[3] Josh Berdine, Byron Cook, Dino Distefano, and Peter O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*.

[4] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Thomas Vojnar. Programs with lists are counter automata. In *Proc. CAV '06*.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*.

[6] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.

[7] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *Proc. FoVeOOS '11*.

[8] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. In *Proc. TACAS '14*.

[9] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV '03*.

[10] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *Proc. PLDI '13*.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.

[12] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *AAECC*, 12(1-2), 2001.

[13] Fabian Emmes, Tim Enger, and Jürgen Giesl. Proving non-looping non-termination automatically. In *Proc. IJCAR '12*.

[14] Samir Genaim, Michael Codish, John P. Gallagher, and Vitaly Lagoon. Combining norms to prove termination. In *Proc. VMCAI '02*.

[15] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*.

[16] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*.

[17] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proc. PLDI '08*.

[18] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*.

[19] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Proc. CAV '06*.

[20] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. CAV '09*.

[21] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. In *Proc. IJCAR '12*.

[22] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proc. CAV '10*.

[23] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using Max-SMT. In *Proc. CAV '14*.

[24] Stephen Magill. *Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2010.

[25] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.

[26] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[27] Greg Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4), 1989.

[28] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. CSL '01*.

[29] Étienne Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theor. Comput. Sci.*, 403(2-3), 2008.

[30] Étienne Payet and Frédéric Mesnard. A non-termination criterion for binary constraint logic programs. *TPLP*, 9(2), 2009.

[31] Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*.

[32] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In *Proc. TACAS '12*.

[33] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *TOPLAS*, 32(3), 2010.

[34] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*.

Faster Temporal Reasoning for Infinite-State Programs

Byron Cook Microsoft Research & University College London
Heidy Khlaaf University College London
Nir Piterman University of Leicester

***Abstract*—In this paper, we describe a new symbolic model checking procedure for CTL verification of infinite-state programs. Our procedure exploits the natural decomposition of the state space given by the control-flow graph in combination with the nesting of temporal operators to optimize reasoning performed during symbolic model checking. An experimental evaluation against competing tools demonstrates that our approach not only gains orders-of-magnitude performance improvement, but also allows for scalability of temporal reasoning for larger programs.**

I. Introduction

Branching-time temporal logics like CTL allow us to reason about safety, termination, and nontermination via the system's interaction with inputs and nondeterminism in a way that linear-time temporal logics like LTL do not. This style of reasoning can be useful in applications ranging from planning, games, security analysis, disproving, and environment synthesis [19], [29]. CTL-based tools also have been used as the basis for higher-performance LTL tools [13].

In this paper we propose a new symbolic CTL model checker for infinite-state programs. We adapt the well-known bottom-up strategy for finite-state CTL model checking [9] to infinite-state programs using precondition synthesis as the enabling technology. We leverage recent techniques for proving safety, termination, and nontermination of programs to synthesize preconditions asserting the satisfaction of CTL sub-formulae of an input property. The key insight to our approach is the exploitation of the natural decomposition of the state space given by the control flow graph. That is, using a counterexample-guided precondition synthesis strategy, we compute program-location-specific preconditions. Our model checker drastically improves performance by reducing the amount of redundant and irrelevant reasoning performed through information sharing extracted from reachability analysis. That is, several preconditions for each program location can be computed simultaneously.

Take for example the fact that the set of states respecting a property such as $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$ *before* a program command is very often the same or nearly the same as the set of states respecting $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$ *after* the command. In comparison to existing tools (*e.g.* [10], [4]) we reduce the amount of reasoning performed as part of the procedure. We can infer whether a command is likely to affect the truth of $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$. So, sequential locality implies that the precondition of a location is easily computed if the preconditions of its successors are known.

This approach gives way to compositional reasoning. For instance, given a program and a desired property, we can, in parallel, synthesize preconditions, desired environments, and plans of individual procedures of a program with the goal of composing the found preconditions into a proof of the whole program. The advantage to this approach is that the program verification tools never have to examine the program as a whole, but instead find a modular proof using compositional reasoning. Recent success in this style of reasoning can be found in areas such as proving correctness of non-blocking algorithms [20], and the analysis of biological models [11].

We also suggest a new method of treating existential path quantification in the infinite-state setting. Existential formulas are handled by considering their universal dual, allowing counterexamples of said duals to serve as a witness asserting the satisfaction of the existential CTL formula.

An experimental evaluation using examples from the benchmark suites of the competing tools (which are drawn from industrial benchmarks) demonstrates orders-of-magnitude performance improvements in many cases. This evaluation is discussed at the conclusion of the paper.

Related work. Model checking has been extensively studied in the context of finite-state systems (*e.g.* [3], [5], [7], [8], [25]) as well as for various types of systems with limitations on their infiniteness (*e.g.* pushdown systems [17], parameterized systems [16], timed systems [2], etc.). In recent years new tools have been developed for proving temporal properties of integer programs, *e.g.* [12], [32], [33], [34], [22], [10], [4]. These tools go beyond, *e.g.* SMV, which is restricted to finite-state programs [6].

In this work we are aiming to prove CTL properties with nested combinations of existential and universal path quantifiers of integer programs. Song & Touili [32] perform a coarse one-time abstraction that takes programs and produces pushdown automata, however the abstraction produced is imprecise and leads to significant information loss. Gurfinkel *et al.* [22] do not reliably support mixtures of nested universal/existential path quantifiers, etc. The two tools closest in their feature set to our setting are from Cook & Koskinen [10] and Beyene *et al.* [4]. Cook & Koskinen implement the Kesten and Pnueli [24] deductive proof system using an incremental reduction to program analysis tools. Beyene *et al.* [4] implement the same idea as Cook & Koskinen using a reduction to Horn-clause reasoning. Neither Cook & Koskinen nor Beyene *et al.* make use of the locality in programs as we do. Effectively,

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

these tools carryout unnecessary computation in their analysis.

In addition, our new approach to the treatment of existential path quantification based on dualization contrasts to that of Cook & Koskinen, which attempts to find a non-trivial restriction on the state-space such that AF can be used to reason about EF, or AG can be used to reason about EG. Our approach also contrasts to the tool of Beyene *et al.* [4], as their tool requires existential quantification over predicates to be supported by the underlying constraint solver, whereas our technique can make use of off-the-shelf verification tools.

Limitations. We do not support programs with heap, nor do we support recursion or concurrency. The heap-based programs we consider during our experimental evaluation have been abstracted using the over-approximation from the technique of Magill *et al.* [26]. Note that this abstraction can lead to unsoundness when we use the existential subset of CTL. Our comparison to existing tools remains fair, as each of the previous tools uses the same abstraction. Effective techniques for proving temporal properties of programs with heap remains an open research question.

As our technique heavily relies on calculating pre-images, it is important that fragments of the underlying program logic are closed under pre-impages, *e.g.* integer linear arithmetic, a fragment of integer arithmetic. Our procedure is not complete as we use a series of incomplete subroutines.

II. Preliminaries

Programs. As is standard [27], we treat programs as control-flow graphs, where edges are annotated by the updates they perform to variables. A program is a triple $P = (\mathcal{L}, E, \mathsf{Vars})$, where $\mathcal{L}$ is a set of locations, E is a set of edges/transitions, and Vars is a set of variables. Each edge $\tau = (\ell, \rho, \ell')$ in E, where $\ell, \ell' \in \mathcal{L}$ and ρ is a condition, specifies possible transitions in the program. A *valuation* associates with the variables in Vars values in Vals. A *trace* or a *path* of a program is either a finite or an infinite sequence of edges allowed by the program. The condition ρ is an assertion in terms of Vars and Vars', a primed copy of Vars, where constants range over Vals. Intuitively, Vars refers to the values of variables before the update and Vars' refers to the values of variables after the update. The set of locations includes the first location ℓ_0 that has no incoming transitions from other program locations. That is, for every $\tau = (\ell, \rho, \ell') \in E$ we have $\ell' \neq \ell_0$. Transitions exiting ℓ_0 have their conditions expressed in terms of Vars'. Locations with incoming transitions from ℓ_0 are *initial locations*. This allows us to encode more complex initial conditions. In figures we usually omit ℓ_0 and add edges with no source to locations having an incoming transition from ℓ_0. The program gives rise to a transition system $T = (S, R)$, where S is the set of program states of the form $S = (\mathcal{L} - \{\ell_0\}) \times (\mathsf{Vars} \to \mathsf{Vals})$ and $R \subseteq S \times S$. That is, a program state is a pair (ℓ, s) where $\ell \neq \ell_0$ and s is a valuation, i.e., a function from program variables to values. The program can transition from (ℓ, s_1) to (ℓ', s_2) if there is a transition $(\ell, \rho, \ell') \in E$ such that $(s_1, s_2) \models \rho$. Here the valuation (s_1, s_2) is a function from $\mathsf{Vars} \cup \mathsf{Vars}'$ to Vals such that for every $v \in \mathsf{Vars}$ we have $(s_1, s_2)(v) = s_1(v)$ and $(s_1, s_2)(v') = s_2(v)$. A state (ℓ, s) is initial if there is a transition (ℓ_0, ρ, ℓ) such that $(s_{-1}, s) \models \rho$, where s_{-1} is some arbitrary state. Notice that in such a case ρ is expressed in terms of Vars' and hence the state s_{-1} does not affect the evaluation of ρ. For example, Figure 1 includes the representation of the program **while** $\mathsf{x} \leq 0$ **do if** * **then** x := x + 1; **fi**; **done**; y := 1; with initial condition $\mathsf{x} = * \wedge \mathsf{y} = 0$ and where $*$ indicates a non-deterministic value.

A finite set of program locations $C \subseteq \mathcal{L}$ is called a *cut-point set* if $\ell_0, \ell_n \in C$, where $n \in \mathbb{N}$ and every cycle in the program's graph contains at least one cut-point.

CTL. We are interested in verifying state-based temporal properties in computational tree logic (CTL) [9]. A CTL formula is of the form

$$\varphi ::= \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AG}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi] \mid \mathsf{EF}\varphi \mid \mathsf{EG}\varphi \mid \mathsf{E}[\varphi \mathsf{U} \varphi]$$

where α is an atomic proposition (*e.g.* $\mathsf{x} < \mathsf{y}$).

To give intuition behind the semantics of CTL, here $P, s \models \mathsf{AF}\varphi$ asserts that in program P and in all possible executions starting from s the property φ will eventually hold in some future state reachable from s, whereas $P, s \models \mathsf{EF}\varphi$ asserts that there is a state in which φ holds and that it can be reached from s. The formula $\mathsf{AG}\varphi$ asserts that φ must hold throughout all possible executions, while $\mathsf{EG}\varphi$ asserts that there exists an execution such that φ would be true throughout. $\mathsf{A}\varphi_1 \mathsf{W} \varphi_2$ asserts that for all executions, φ_1 has to hold until φ_2 holds, signifying that φ_2 does not necessarily have to hold as long as φ_1 holds. Contrarily, $\mathsf{E}\varphi_1 \mathsf{U} \varphi_2$ asserts that there exists an execution in which φ_1 has to hold *at least until* at some position φ_2 holds. AU and EW are represented as syntactic sugar as usual.

For a program P and a CTL property φ, we say that φ holds in P, denoted by $P \models \varphi$ if for every initial state s we have $P, s \models \varphi$.

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 non-empty 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, a set of ranking functions $\{f_1, \ldots, f_n\}$ denotes the disjunctively well-founded relation $\{(s, s') \mid f_1(s') \prec f_1(s) \vee \ldots \vee f_n(s') \prec f_n(s)\}$.

Counterexamples. In our setting new ranking functions can be automatically synthesized by examining counterexamples produced by an underlying safety prover (discussed in more detail in Section IV). Due to the recursive nature of our procedure it is only necessary to handle counterexamples to formulas of nesting depth 1. For example, $\mathsf{A}\varphi$, where φ is a path formula that includes no nesting of additional operators, or $\alpha_1 \vee \alpha_2$, where α_1 and α_2 are assertions. A counterexample for an atomic proposition α is a state in which α does not hold. A counterexample for a conjunction $\varphi_1 \wedge \varphi_2$ is a state

where either φ_1 or φ_2 does not hold. A counter example for disjunction $\varphi_1 \vee \varphi_2$ is a state where both sub-formulas do not hold. A counterexample to an $\mathsf{AG}\varphi$ property is a path to a place where φ does not hold. A counterexample to an $\mathsf{AF}\varphi$ property is a "lasso": a stem path to a particular program location, then a (not necessarily simple) cycle which returns to the same program location, and the property φ does not hold along the stem and the cycle. Finally, a counterexample to $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ is a path to a place where there is a sub-counterexample to φ_1 as well as one to φ_2. A counterexample to $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ can be of the same form as that of $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, as well as one where φ_1 holds while φ_2 does not hold anywhere along the path.

Calculating pre-images. Let $\pi = (\ell_0, \rho_0, \ell'_0), (\ell_1, \rho_1, \ell'_1), \ldots, (\ell_n, \rho_{n-1}, \ell'_n)$ be a path. We compute a pre-image for every possible suffix of π. That is, we denote $pre_{n+1} = S$ and $pre_i = pre((\ell_i, \rho_i, \ell'_i), \ldots, (\ell_n, \rho_n, \ell'_n))$ as the set of states such that $pre_i = \{s \mid \exists s' \in pre_{i+1} \text{ s.t. } ((\ell_i, s), (\ell'_i, s')) \models \rho_i\}$. Generaly speaking, given an assertion α (in terms of Vars) representing pre_{i+1}, and an assertion ρ_i (in terms of Vars and Vars') we must compute an assertion representing pre_i. Let α' denote $\exists$ Vars. $\mathsf{Vars} = \mathsf{Vars}' \wedge \alpha$. We thus consider $\exists \mathsf{Vars}'(\mathsf{Vars} = \mathsf{Vars}' \wedge \exists \mathsf{Vars}. (\mathsf{Vars} = \mathsf{Vars}' \wedge (\alpha' \wedge \rho_i)))$. We use Fourier-Motzkin for quantifier elimination.

III. Intuition and Example

We first informally explain our technique and demonstrate it with an example.

Intuition. The idea of the procedure is to find for each sub-formula φ a precondition $\wp\langle\varphi\rangle$ that ensures its satisfaction. To utilize sequential locality of a counterexample's control-flow graph further on, a precondition $\wp\langle\varphi\rangle$ is thus partitioned to $\wp\langle\ell_i, \varphi\rangle$ for every location ℓ_i in the program. Thus, $\wp\langle\varphi\rangle$ takes the form $\bigwedge_{\ell_i}(\mathsf{pc} = \ell_i \Rightarrow \wp\langle\ell_i, \varphi\rangle)$. Here $\mathsf{pc} = \ell_i$ is used to assert that the state is at location ℓ_i in the program's control-flow graph. We find preconditions by iteratively recursing over the structure of the given CTL formula. That is, we start by finding the precondition of the innermost sub-formula followed by search for the preconditions of the outer sub-formulas dependent on it. We note that the precondition of an atomic proposition is the proposition itself, hence from this point on, we shall treat the precondition of an atomic proposition and the atomic proposition itself synonymously.

Consider a universal CTL formula. Initially, we approximate its precondition as true. We then search for counterexamples from every possible reachable program location. Failures to the proof attempt will result in the strengthening of the precondition through adding the negation of the pre-image of the discovered counterexample. We use the control-flow graph of a counterexample to simultaneously synthesize preconditions of multiple locations. That is, a counterexample that consists of multiple program locations can be utilized to update the precondition of each contained program location. This is done by iterating along the counterexample path, and for each suffix computing a pre-image from a program location onwards.

Each counterexample found further strengthens a precondition, we thus eliminate said counterexample and search for other proof failures for the given CTL property. Eventually, the precondition will imply the correctness of the sub-formula when no further counterexamples are returned.

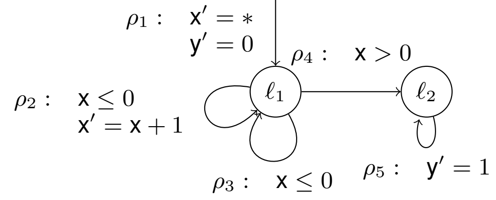


Fig. 1: The control-flow graph of an example program for which we wish to prove the CTL property AGEF $\mathsf{y} = 1$.

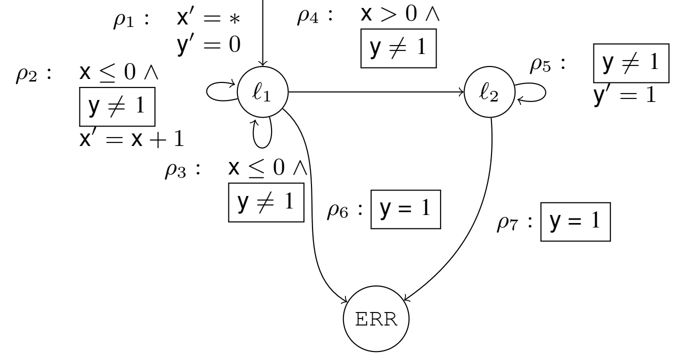


Fig. 2: The transformation of the program from Figure 1 for the property EF $\mathsf{y = 1}$ using its dual AG $\mathsf{y} \neq 1$.

Existential sub-formulas are handled by considering their universal dual. We thus seek a set of counterexamples generated from the property's universal dual to serve as an existential witness. Hence we begin with an initial precondition approximation false. More directly, pre-images of counterexamples to the negation of the sub-formula serve as a witnesses to the satisfaction of our existential formula. Counterexamples are similarly treated in the existential case, we iteratively calculate their pre-images followed by their elimination until no more counterexamples are generated. As before, we utilize a counterexample's control flow graph to simultaneously update preconditions of multiple locations.

Example. Consider the program in Figure 1 and the property $\varphi \equiv \mathsf{AGEF}\ \mathsf{y} = 1$, which states that for all states, it is always possible that eventually $\mathsf{y} = 1$. The approach followed by nearly all tools supporting CTL would be to find, in this instance, a set of states $\wp$ such that $\mathsf{AG}\wp$ holds, and such that $\wp \models \mathsf{EF}\ \mathsf{y} = 1$ holds. In this paper we suggest a strategy based on precondition synthesis.

Consider the sub-formula $\psi \equiv \mathsf{EF}\ \mathsf{y} = 1$. For the proposition $\mathsf{y} = 1$, for every program location ℓ_i we have $\wp\langle\ell_i, \mathsf{y} = 1\rangle \triangleq \mathsf{y} = 1$. We now attempt to prove that $\wp \not\models \mathsf{AG}\ \mathsf{y} \neq 1$ given that AG is EF's universal dual. We start with $\wp\langle\psi\rangle \triangleq \mathsf{false}$ as only failures to proving $\mathsf{AG}\ \mathsf{y} \neq 1$ can necessitate that there exists a witness such that $\mathsf{EF}\ \mathsf{y} = 1$. Failures to the proof attempt will result in refinements to $\wp$ through the iterative calculation of the pre-image of each discovered counterexample. Recall that we are interested in counterexamples starting from all program locations:

$$\wp\langle\psi\rangle \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \wp\langle\ell_1, \psi\rangle) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \wp\langle\ell_2, \varphi\rangle).$$

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

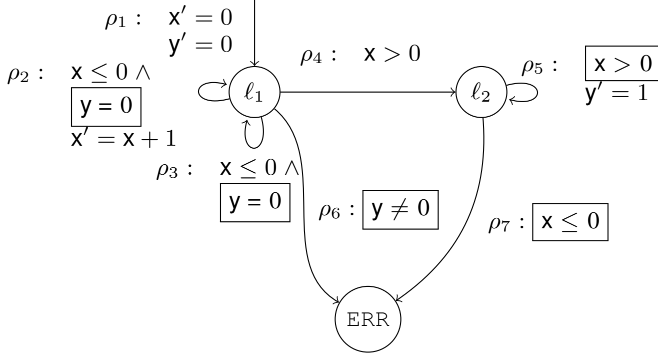


Fig. 3: The transformation of the program from Figure 1 for the sub-property AGEF y = 1 to be utilized in the verification algorithm. The nested property EF y = 1 is substituted with its precondition resulting in a transformation for AG $((\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \vee (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0))$ instead.

We begin with ℓ_1. To check AG $\mathsf{y} \neq 1$ we use a source-to-source transformation that reduces checking of universal CTL properties to safety [10]. The transformation returns the program in Figure 2 (new conditions outlined), on which we use a safety prover to check reachability of ERR. We get counterexample CEX_1: $\langle \ell_0, \rho_1, \ell_1\rangle$, $\langle \ell_1, \rho_3, \ell_1\rangle$, $\langle \ell_1, \rho_2, \ell_1\rangle$, $\langle \ell_1, \rho_4, \ell_2\rangle$, $\langle \ell_2, \rho_5, \ell_2\rangle$, $\langle \ell_2, \rho_7, \mathrm{ERR}\rangle$.

We then calculate the pre-image of CEX_1 for multiple locations along the counterexample. We do so by iterating along the counterexample path, and for every reachable location $\ell \in \mathcal{L}$ in CEX_1, we compute a pre-image utilizing the suffix of CEX_1 from ℓ onwards. Thus we can avoid redundant reasoning by utilizing sequential locality based upon the program's control-flow graph to compute a refinement for ℓ_2 from a counterexample generated for ℓ_1. In this case, we compute $\wp \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0)$

One existential witness may not be sufficient to find all states that satisfy ψ in the respective locations, we thus rule out CEX_1 by adding $\neg\wp\langle \ell_i, \psi\rangle$ to each transition from ℓ_i to the error state. We re-run our safety checker and find that we do not generate anymore counterexamples, thus completing our precondition synthesis for EF $\mathsf{y} = 1$.

Note that the technique used by Cook & Koskinen [10] imposes that they spend time computing both $\wp\langle \ell_1, \psi\rangle$, $\wp\langle \ell_2, \psi\rangle$ separately while the technique used by Beyene *et al.* [4] solves a constraint based on an entire path when it's only necessary to reason about a single state.

We now modify φ by using $\wp\langle\psi\rangle$ and get $\varphi' = \mathsf{AG}\ ((\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0))$. The constructed transformation reducing the property φ' to safety can be seen in Figure 3. Note that in this particular transformation, the outlined instrumented conditions correspond to each of the location preconditions generated for EF $\mathsf{y} = 1$. As φ' is universal, we begin with the initial precondition $\wp\langle\varphi\rangle \triangleq \mathsf{true}$. Failures to the proof attempt will result in strengthening the precondition by adding *negated* pre-images of discovered counterexamples. In this case no counterexamples are returned and we get $\wp\langle\varphi\rangle \triangleq \mathsf{true}$. This proves that AGEF $\mathsf{y} = 1$ holds.

IV. Procedure

In this section we describe the details of our CTL model checking procedure. Figure 4 depicts VERIFY, which wraps the main procedure TEMPORALWP in Figure 5. Other subroutines used in TEMPORALWP are in Figures 6–10.

```
1  let VERIFY (φ,P) : bool =
2
3      (ℒ,E,Vars) = P
4      ℘ = TEMPORALWP(φ,P)
5      return ∀(ℓ₀,ρ,ℓ) ∈ E ∀s . (s,s) ⊨ ρ ⇒ s ⊨ ℘⟨ℓ,φ⟩
```

Fig. 4: Procedure VERIFY, which wraps TEMPORALWP and then checks all initial states.

```
1  let rec TEMPORALWP(ψ,P) : map =
2    ℘ = INITIALIZEMAP (ψ,P)
3    ℳ = ∅
4    κ = [ ]
5    (ℒ,E,Vars) = P
6    if ψ = α  is atomic  then
7      foreach {ℓ | (ℓ,t,ℓ′) ∈ E}
8         ℘⟨ℓ,ψ⟩ = pre(t,α) ; ℘⟨ℓ,¬ψ⟩ = ¬pre(t,α)
9      done
10   else
11     match (ψ) with
12     | ψ′₁∧ψ′₂ | ψ′₁ ∨ ψ′₂ | ψ′₁Uψ′₂ | ψ′₁Wψ′₂ →
13        ℘ = ℘ ∪ TEMPORALWP(ψ′₁,P)∪ TEMPORALWP(ψ′₂,P)
14     | AFψ′₁ | AGψ′₁ | ¬ψ′₁ →
15        ℘ = ℘ ∪ TEMPORALWP(ψ′₁,P)
16     C = FINDCUTPOINTS(P)
17     foreach ℓ ∈ C  do
18        P′ = TRANSFORM(⟨ℓ,ψ⟩,ℳ,P,℘)
19        CEX, ℳ = REFINE(P′,ψ,℘,ℳ)
20        while CEX ≠ ∅ do
21          ℘,P′ = PROPAGATE(CEX,P′,κ,ψ,ℓ,℘)
22          κ = CEX :: κ
23          CEX,ℳ = REFINE(P′,ψ,℘,ℳ)
24        done
25     done
26   ℘
```

Fig. 5: Procedure TEMPORALWP getting a temporal property and a program and returning the map from program locations and sub-formulas to assertions.

We exploit the natural decomposition of the state space given by the control flow graph. That is, using a counterexample-guided precondition synthesis strategy, we compute program-location-specific preconditions. In our approach the table $\wp$ is the key data structure which maps pairs of program locations and sub-formulae to assertions which represent the current candidate *precondition* that would guarantee the sub-formulae at a respective location. That is, $\wp\langle \ell, \varphi\rangle$ should be a sufficient and most general precondition to prove that φ holds at location ℓ. If such is not the case, a counterexample is produced and the procedure attempts to refine $\wp$

```
1  let INITIALIZEMAP (ψ,P) : map =
2
3    ℘ = ∅
4    (ℒ,E,Vars) = P
5    if ψ = Eψ′ then
6      foreach ℓ ∈ ℒ do
7        ℘⟨ℓ,ψ⟩ = false;
8        ℘⟨ℓ,¬ψ⟩ = true
9      done
10   else
11     foreach ℓ ∈ ℒ do
12       ℘⟨ℓ,ψ⟩ = true;
13       ℘⟨ℓ,¬ψ⟩ = false
14     done
15   return ℘
```

Fig. 6: Initializing the map from program locations and sub-formulas to assertions.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

```
1  let Refine(P,ψ,℘,M) : map =
2
3    CEX = Reachable(P,ERR)
4    while P can reach ERR do
5      if CEX contains stem and lasso then
6         if ∃ witness f showing CEX' w.f. then
7            M = M ∪ {f}
8         else
9            return CEX, M
10     else
11         return CEX, M
12     CEX = Reachable((Transform(⟨ℓ,ψ⟩,M,P,℘),ℓ0,ERR)
13   done
```

Fig. 7: Procedure REFINE getting a program, a temporal property, the map from locations and temporal properties to assertions, and a set of ranking functions and returning a counter example reaching location ERR and a (possibly) larger set of ranking functions.

```
1  let Propagate(CEX,P,κ,ψ,n,℘): map =
2    α = true
3    (L,E,Vars) = P
4    foreach (ℓ,ρn,ℓ') ∈ CEX reachable from n do
5      if CEX in κ ∧ ℓ = n then
6         α = Strengthen(pre(ℓ,CEX),CEX)
7      else
8         α = pre(ℓ,CEX)
9      if ψ = Eψ' then
10        ℘⟨ℓ,ψ⟩ = ℘⟨ℓ,ψ⟩ ∨ α
11        ℘⟨ℓ,¬ψ⟩ = ℘⟨ℓ,¬ψ⟩ ∧ ¬α
12     else
13        ℘⟨ℓ,ψ⟩ = ℘⟨ℓ,ψ⟩ ∧ ¬α
14        ℘⟨ℓ,¬ψ⟩ = ℘⟨ℓ,¬ψ⟩ ∨ α
15     if ℓ' = ERR then
16        ρ ∈ E = ρ ∧ ¬℘⟨ℓ,ψ⟩
17   done
18   ℘,P
```

Fig. 8: Procedure PROPAGATE getting a counter example, the program, a list of previous counter examples, the location to which the counter example is applicable, and the map of previously discovered preconditions and returning an updated map and updated program. The map of preconditions is updated by adding the weakest preconditions of the current counter example. The program is updated by eliminating handled counter example from reaching the ERR location again.

given the counterexample path. Each precondition synthesized substitutes its temporal sub-property in the original formula, and we then continue with the next most outer formula. After a short description of TEMPORALWP and a brief description of each of its subroutines, we give an in depth explanation of TEMPORALWP.

TEMPORALWP: performs both a recursive and a refinement-based computation to construct $\wp$. It starts by initializing the map of preconditions using procedure INITIALIZEMAP (Figure 6) and then calling itself recursively for each sub-formula (lines 7–9 and 11–15). TRANSFORM and REFINE are part of the model checking procedure for the current sub-formula while PROPAGATE (Figure 8) updates the map by synthesizing the pre-images given a counterexample. We then reduce the amount of redundant and irrelevant reasoning performed through information sharing extracted from

```
1  let Strengthen(α,CEX) : AP =
2
3    W = {v | v gets updated in CEX}
4    QE(∃W.α)
```

Fig. 9: If divergence is suspected due to infinitely many counterexamples, the sub-procedure strengthens the candidate precondition towards the limit.

```
1  let Transform(⟨k,φ⟩,M,P,℘) : Program =
2
3  (L,E,Vars) = P
4  match(φ) with
5  | ψ∧ψ' →
6    α1 = ℘⟨k,¬ψ⟩ ; α2 = ℘⟨k,¬ψ'⟩
7    E = E ∪ (k, α1 ∨ α2, ERR)
8  | ψ∨ψ' →
9    α1 = ℘⟨k,¬ψ⟩ ; α2 = ℘⟨k,¬ψ'⟩
10   E = E ∪ (k, α1 ∧ α2, ERR)
11 | A[ψWψ'] →
12   foreach (ℓ,ρ,ℓ') ∈ E reachable from k do
13      α1 = ℘⟨ℓ,ψ⟩ ; α2 = ℘⟨ℓ,ψ'⟩
14      ρ = ρ ∧ α1 ∧ ¬α2
15      E = E ∪ (ℓ, ¬α1 ∧ ¬α2, ERR)
16 | E[ψUψ'] →
17   P = Transform(⟨k,A[¬ψ'W(¬ψ ∧ ¬ψ')]⟩,M,P,℘)
18 | AFψ →
19   foreach ('ℓ,ρ,k) ∈ E do
20      ρ = ρ ∧dup=false
21   foreach (ℓ,ρ,ℓ') ∈ E reachable from k do
22      α = ℘⟨ℓ,ψ⟩
23      ρ = (ρ ∧ ¬α) ∨ (ρ ∧ ¬dup∧¬α
24           ∧ dup=true ∧ ('s = ℓ × Vars → Vals))
25      c = dup ∧ ¬α ∧ ¬(∃f ∈ M. f(s) ≺ f('s))
26      E = E ∪ (ℓ, c, ERR)
27 | EGψ →
28   P = Transform(⟨k,AF¬ψ⟩,M,P,℘)
29 | AGψ →
30   foreach (ℓ,ρ,ℓ') ∈ E reachable from k do
31      α = ℘⟨ℓ,ψ⟩
32      ρ = ρ ∧ α
33      E = E ∪ (ℓ, ¬α, ERR)
34 | EFψ →
35   P = Transform(⟨k,AG¬ψ⟩,M,P,℘)
36
37 P
```

Fig. 10: Reduction of model checking of temporal properties to safety and ranking function synthesis.

reachability information. That is, several preconditions for each program location can be computed simultaneously. When TEMPORALWP returns to VERIFY, it is only necessary to check if the precondition of the outermost temporal sub-property is satisfied by the initial states of the program.

TRANSFORM: implements the reduction of model checking to safety checking and well-foundedness, inspired by the procedure from [10]. The TRANSFORM transformation utilizes the map $\wp$, which maps the preconditions synthesized previously for sub-properties and their negations (lines 6,9,13,22, and 31). The program is then transformed according to the CTL sub-property by modifying the program from a given program location $k \in \mathcal{L}$. The reduction is only applied from a location k onwards (see loop invariants in lines 12, 21, and 30), that is, we only wish to verify the sub-property starting from transitions stemming from k. Whenever φ does not hold for a location ℓ, a new reachable transition to an error location ERR is added.

As mentioned, existential path quantifiers are handled by considering their universal dual. For both existential and universal properties, our mapping function is also updated with the precondition for the negation of the property on line 8 in TEMPORALWP and lines 11 and 14 in PROPAGATE. This allows us to conveniently access the negation of the property when encoding existential properties as their universal duals.

REFINE: uses a safety prover (similar to IMPACT [28]) to obtain counterexamples from the transformed system, if a

counterexample exists. A produced counterexample to a liveness property (such as AF) contains a lasso fragment, we then attempt to find an accompanying set of ranking functions $\mathcal{M}$ that will show that the counterexample is not valid. We thus attempt to enlarge the set of ranking functions $\mathcal{M}$ using the well known method of [14]. Otherwise, the absence of a set of ranking function indicates the existence of a recurrent set. Note that proving liveness is undecidable, thus techniques used in [14] are incomplete.

A. TEMPORALWP: *computing* $\wp$

In order to synthesize a precondition for a temporal property ψ, we first recursively accumulate the preconditions generated when considering the sub-properties of ψ at lines 8, 13, and 15. The base case, an atomic proposition α, is computed as is standard, *e.g.,* in [15]. For the sake of clarity, we omit the descriptions of both FINDCUTPOINTS and the use of sequential locality in PROPAGATE till later, as we solely wish to describe the fundamental procedure underlying our precondition synthesis for each temporal sub-property. We will then discuss how these sub-procedures provide the key to making use of the program's control-flow graph to construct multiple preconditions.

Given the omission of FINDCUTPOINTS, let C be the set of all locations in a program P, that is $\mathcal{L}$. We wish to synthesize a precondition for each $\ell \in \mathcal{L}$ such that the precondition asserts the satisfaction of ψ. Hence, we iterate over these locations (line 17) and generate a transformed program corresponding to each location using the subroutine TRANSFORM at line 18.

Recall that TRANSFORM allows us to reduce the checking of temporal properties to a program analysis task from a given program location. Each transformed program is then verified through the subroutine REFINE (line 19). A counterexample-guided precondition refinement loop then begins at line 20, where we iteratively refine a precondition for $\ell \in \mathcal{L}$ until no more counterexamples are found. We now discuss the refinement process for each type of quantifier separately below.

Universal precondition synthesis. For a universal CTL sub-property ψ, a precondition $\wp\langle \ell, \psi \rangle$ for a program location ℓ is initialized to true (Figure 6 line 12). If REFINE returns a counterexample, we refine $\wp\langle \ell, \psi \rangle$ by taking the negation of pre-image of the returned counterexample at location ℓ in PROPAGATE on line 21. Given our temporary omission of sequential locality in PROPAGATE, consider the loop in Figure 9 on line 3 to only iterate over one element, that is the current ℓ. As we are handling a universal sub-property its precondition is then strengthened to become $\wp\langle \ell, \psi \rangle = \wp\langle \ell, \psi \rangle \wedge \neg pre(\ell, \mathsf{CEX})$ (line 13 in PROPAGATE).

We then rule out the aforementioned counterexample by adding the assumption $\neg pre(\ell, \mathsf{CEX})$ to each ingoing transition to the error location on the counterexample path, as shown on lines 15 and 16 in PROPAGATE. We then continue to unfold the loop in TEMPORALWP whenever a new counterexample is discovered while iteratively refining $\wp\langle \ell, \varphi \rangle$, resulting in:

$$\wp\langle \ell, \varphi \rangle = \bigwedge_{n \in \mathbb{N}} \neg pre(\mathsf{CEX_n})$$

Existential precondition synthesis. For an existential CTL property, a precondition must entail an existential witness satisfying the sub-property ψ at program location ℓ. We thus verify the universal dual of the existential property (as instrumented by our encoding) and seek a set of counterexamples generated from the property's universal dual to serve as an existential witnesses.

A precondition $\wp\langle \ell, \varphi \rangle$ for a program state is initially false (line 7 in Figure 6). If a counterexample is returned, $\wp\langle \ell, \varphi \rangle$ is refined through the disjunction of the pre-image of the counterexample returned, that is $\wp\langle \ell, \psi \rangle = \wp\langle \ell, \psi \rangle \vee pre(\ell, \mathsf{CEX})$ (line 10 in Figure 8).

We rule out the aforementioned counterexample by adding the assumption $\neg pre(\ell, \mathsf{CEX})$, and continue to unfold the loop with each newly discovered counterexample while iteratively refining $\wp\langle \ell, \psi \rangle$. Note that finding one witness is not sufficient to satisfy an existential property, as $\wp\langle \ell, \psi \rangle$ must characterize *all* the states satisfying the sub-property ψ at a location. Thus,

$$\wp\langle \ell, \psi \rangle = \bigvee_{n \in \mathbb{N}} pre(\mathsf{CEX_n})$$

Upon the return of our precondition method to its caller, $\wp$ will contain the precondition for the most outer temporal property of the original CTL property φ.

In our procedure, divergence can occur due to the possibility of generating infinitely many counterexamples. In practice this is rare, but not unheard of. We thus implement the following heuristic introduced by [10]:

- If we suspect we are looking at a sequence of repeated counterexamples that will result in divergence, we call the procedure STRENGTHEN (Figure 9, line 5 in PROPAGATE). The sub-procedure strengthens the candidate precondition towards the limit.
- STRENGTHEN takes the calculated pre-image α, then proceeds to quantify out all variables that are updated proceeding the program location ℓ by applying quantifier elimination (QE).
- This heuristic can lead to unsoundness, as STRENGTHEN may over-approximate the set of states, causing $\wp$ to be potentially unsound for temporal properties involving existential path quantifiers. To check that the guessed candidate precondition is in fact a real precondition, *e.g.* that $\wp \Rightarrow \mathsf{EG}\ \wp'$, we can use the approach from Beyene *et al.* [4] to double check the small lemma.
- If the check succeeds we continue, if the check fails we raise an exception.

Reducing redundant and irrelevant reasoning. Given that our approach synthesizes counterexample guided preconditions over program locations, we utilize sequential locality to simultaneously calculate preconditions for the set of locations that are arranged and can be accessed from a CEX starting from a given location ℓ. Our propagation sub-procedure PROPAGATE (Figure 8) is called from TEMPORALWP at line 21. We iterate along the counterexample path, and for every reachable location $\ell \in \mathcal{L}$, we compute a pre-image utilizing a suffix of CEX from ℓ onwards. In more informal terms, every program location along the path can utilize the same counterexample to show that the property does or does not hold. Practically, the computation of a pre-image is performed by going backwards

over the counter example.

PROPAGATE alone does not eliminate redundant or irrelevant reasoning, as we would still iterate over locations whose preconditions have already been computed for. We thus calculate a cut-point set C such that $C \subseteq \mathcal{L}$ and every cycle in the program's graph contains at least one cut-point (line 16 in TEMPORALWP). That is, we only wish to synthesize a precondition over each program location $\ell \subseteq C$. We choose to verify the set of cut-points [18] instead of all program locations, as cut-points provide locality across program locations given the nature of cycles. We will thus be able to propagate a cut-point precondition to all locations reachable from a cycle of a generated counterexample. Other program analysis inspired techniques may be used for the selection of initial locations to be verified. A cycle independent analysis can be run for those locations unreachable from program cut-points.

We now state the correctness of our procedure.

Proposition 4.1: If the algorithm in Figure 5 terminates, for every sub-formula ψ of φ, every location $\ell \in \mathcal{L}$, and every reachable state s we have: $s \models \wp\langle \ell, \psi \rangle$ implies $P, (\ell, s) \models \psi$ and $s \models \neg\wp\langle \ell, \psi \rangle$ implies $P, (\ell, s) \models \neg\psi$.

Proof Sketch: We prove the proposition by induction on the structure of the formula. It is clear for an atomic proposition and for Boolean operators. Consider a universal path formula. As the counter examples obtained from the underlying program analysis tool are real counter examples, it follows that their pre-images do not satisfy the formula. We then get additional counter examples, which are all sound. The termination of the loop searching for counter examples implies that the disjunction of all pre-images is sound and complete. Existential path formulas are dual.

Corollary 4.2: For every symbolic program P we have $P \models \varphi$ iff for every $(\ell_0, \rho, \ell) \in E$ we have $\rho \Rightarrow \wp\langle \ell, \varphi \rangle$.

V. EVALUATION

In this section we discuss the results of our experiments with an implementation of the procedure from Figure 4. Our implementation is built as an extension to the open source project T2 [1], which uses a safety prover similar to IMPACT [28] alongside previously published techniques for discovering ranking functions, etc [31], [21] to prove both liveness and safety properties. The source code for our tool can be found at `http://heidyk.com/T2source/`.

We have compared our tool to that of Cook & Koskinen [10] and Beyene *et al.* [4]. The benchmarks used are the same as those used in [10] and [4]. These benchmarks were originally created by Cook & Koskinen using the examples drawn from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the SoftUpdates patch system [23]. The benchmarks can be found at `http://www.cims.nyu.edu/~ejk/ctl/`. For each program and CTL property φ, we verify both φ and $\neg\varphi$. The various tools were executed on an Intel x64-based 2.8 GHz single-core processor.

Program commands. We now discuss the format in which we interpret a program's commands. A deterministic assignment statement of the form $v' = \mathsf{exp}$ where $v' \in \mathsf{Vars}'$ and exp is an expression over program variables is translated to the condition $v' = \mathsf{exp} \ \wedge \ \forall x \in \mathsf{Vars}\backslash\{v\}.\ x = x'$. A nondeterministic assignment $v' = *$ is translated to $\forall x \in \mathsf{Vars}\backslash\{v\}.\ x = x'$. A conditional statements exp is encoded to $\mathsf{exp} \wedge \forall x \in \mathsf{Vars}.\ x = x'$.

In Figure 11 we display the comparison of our results. For each program and its set of CTL properties, we display the number of lines of code (LOC), and report the time it took to verify a CTL property (Time column) in seconds. A ✓ in the "Result" column indicates that the tool was able to verify the property. Likewise, an $\times$ indicates that the tool failed to prove the property. A timeout or memory exception is indicated by T/O. A timeout is triggered if verification of an experiment exceeds 3000 seconds. The symbol "–" in the Time and Result column indicates that the experiment was not run.

Overall, our tool demonstrates a significant increase in performance and scalability. Contrary to existing tools, our tool produces no timeouts and programs are often verified in under a second or less. The aforementioned tools often take minutes (the former more-so than the latter). Furthermore, the previous tools produce T/Os in cases where the temporal formula is complex, the size of the program is large, or both. Although a few of our results are on par with Beyene *et al.*, one can speculate from our evaluations that said tool is not well equipped to handle larger programs. Contrarily, our tool demonstrates the potential for scalability. On average, we show orders-of-magnitude performance improvement over existing tools, particularly when dealing with larger programs.

In a few cases our tool produces results that differ with one of the previous tools, due to bugs in the previous tools. As an example, in the S/W Updates case we are unable to repeat the result of Cook & Koskinen on $c > 5 \ \wedge \ \mathsf{AG}(r \leq 5)$ and $c > 5 \ \wedge \ \mathsf{EG}(r \leq 5)$. Our result agrees with that of Beyene *et al.*. Finally, OS frag. 2 requires fairness, and it is unclear how [10] and [4] verified said program, as all these tools lack support for fairness. Cook & Koskinen acknowledge their bug.

VI. CONCLUDING REMARKS

In this paper we have described a procedure for CTL model checking that takes advantage of the structure of control-flow graphs available in programs. Our procedure works recursively on the structure of the property and computes (location-based) preconditions for the satisfaction of each sub-formula. The idea is to use a decomposition based on program-location (thus facilitating the use of program analysis techniques), but to maintain the current state of the intermediate lemmas in a way their results can be used to quickly facilitate the computation of results for nearby program locations. As is evident from the outcome of our experimental evaluation, our method leads to dramatic performance improvement over competing tools that support CTL verification for infinite-state programs. Additionally, we wish to further experiment with the scalability that our methodology can perhaps provide.

REFERENCES

[1] "T2 source code," http://research.microsoft.com/t2.
[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, 126(2), 1994.

			φ						$\neg\varphi$					
			Fig. 4		Beyene [4]		Cook [10]		Fig. 4		Beyene [4]		Cook [10]	
Program	**LOC**	**Property** (φ)	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**
OS frag. 6	1050	$\mathsf{AG}(b = 1 \Rightarrow \mathsf{AF}(u = 0))$	67.3	✓	T/O	–	T/O	–	82.9	×	T/O	–	T/O	–
OS frag. 6	1050	$\mathsf{EG}(b = 1 \Rightarrow \mathsf{EF}(u = 0))$	36.2	✓	T/O	–	T/O	–	38.8	×	T/O	–	–	–
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	5.9	✓	43.4	✓	38.9	✓	6.2	×	40.4	×	18.0	×
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	6.8	✓	35.45	✓	90.0	✓	3.4	×	36.57	×	107.3	×
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	4.7	✓	T/O	–	T/O	–	3.1	×	T/O	–	T/O	–
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	2.3	✓	2.52	✓	1680.7	✓	6.0	×	2.52	×	1930.0	×
OS frag. 4	370	$\mathsf{AF}(io = 1) \vee \mathsf{AF}(ret = 1)$	18.5	✓	270.6	✓	34.3	✓	13.9	×	58.06	×	18.8	×
OS frag. 4	370	$\mathsf{EG}(io \neq 1) \wedge \mathsf{EG}(ret \neq 1)$	13.5	✓	T/O	–	7.6	✓	14.2	×	T/O	–	61.3	×
OS frag. 4	370	$\mathsf{EF}(io = 1) \wedge \mathsf{EF}(ret = 1)$	14.7	✓	T/O	–	1261.0	✓	4.8	×	T/O	–	T/O	–
OS frag. 4	370	$\mathsf{AG}(io \neq 1) \vee \mathsf{AG}(ret \neq 1)$	8.0	✓	0.1	✓	–	–	3.7	×	1.3	×	–	–
PgSQL arch	90	$\mathsf{AG}(\mathsf{AF}(w = 1))$	2.0	✓	0.7	✓	T/O	–	1.3	×	1.4	×	38.1	×
PgSQL arch	90	$\mathsf{AG}(\mathsf{EF}(w = 1))$	2.0	✓	0.7	✓	T/O	–	0.0	×	0.2	×	42.7	×
PgSQL arch	90	$\mathsf{EF}(\mathsf{AG}(w \neq 1))$	2.0	✓	0.1	✓	T/O	–	2.4	×	0.7	×	30.2	×
PgSQL arch	90	$\mathsf{EF}(\mathsf{EG}(w \neq 1))$	0.1	✓	0.1	✓	35.2	✓	0.1	×	0.5	×	45.3	×
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{AF}(u = 1))$	0.8	×	1.4	✓	2.1	✓	0.2	✓	0.7	×	1.8	×
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{EF}(u = 1))$	2.0	×	1.3	✓	3.7	✓	0.2	×	0.4	×	1.5	×
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{AG}(u \neq 1))$	0.8	✓	0.1	✓	5.6	✓	0.2	×	0.7	×	8.7	×
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{EG}(u \neq 1))$	1.0	✓	0.1	✓	1.2	✓	1.2	×	1.8	×	6.5	×
OS frag. 5	58	$\mathsf{AG}(\mathsf{AF}(w \geq 1))$	1.0	✓	0.6	✓	569.7	✓	0.2	×	0.4	×	65.1	×
OS frag. 5	58	$\mathsf{AG}(\mathsf{EF}(w \geq 1))$	1.0	✓	0.7	✓	T/O	–	0.0	×	0.1	×	T/O	–
OS frag. 5	58	$\mathsf{EF}(\mathsf{AG}(w < 1))$	0.1	✓	0.5	✓	255.8	✓	0.1	×	0.2	×	85.5	×
OS frag. 5	58	$\mathsf{EF}(\mathsf{EG}(w < 1))$	0.1	✓	0.1	✓	351.1	✓	0.0	×	0.2	×	1471.7	×
S/W Updates	36	$c > 5 \Rightarrow \mathsf{AF}(r > 5)$	0.1	×	5.27	✓	70.2	✓	1.1	✓	0.8	×	32.4	×
S/W Updates	36	$c > 5 \Rightarrow \mathsf{EF}(r > 5)$	0.1	✓	0.2	×	18.5	✓	0.8	×	0.1	×	1.3	×
S/W Updates	36	$c > 5 \wedge \mathsf{AG}(r \leq 5)$	0.1	×	0.1	×	0.3	✓	1.1	✓	0.1	×	0.5	×
S/W Updates	36	$c > 5 \wedge \mathsf{EG}(r \leq 5)$	0.4	×	0.1	×	4.5	✓	0.7	✓	0.1	×	0.4	×
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	1.0	✓	0.3	✓	4.6	✓	1.4	×	0.7	×	9.1	×
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	0.1	✓	0.3	✓	9.5	✓	0.1	×	0.3	×	1.5	×
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	0.1	✓	0.1	✓	105.7	✓	0.1	×	0.4	×	18.1	×
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	0.1	✓	0.1	✓	3.5	✓	0.7	×	0.3	×	12.5	×

Fig. 11: The results of applying our CTL model checking procedure on benchmarks from [10], [4]. For each program we verify a set of properties (φ) and their negations ($\neg\varphi$) and compare our results with [10], [4]. A timeout (T/O) is triggered if verification of a benchmark exceeds 3000 seconds.

[3] O. Bernholtz, M. Y. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking (extended abstract)," in *CAV'94*. Springer, 1994.

[4] T. A. Beyene, C. Popeea, and A. Rybalchenko, "Solving existentially quantified horn clauses," in *CAV'13*. Springer, 2013.

[5] J. Burch, E. Clarke *et al.*, "Symbolic model checking: 10^{20} states and beyond," *Information and computation*, 98(2), 1992.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV'02*. Springer, 2002.

[7] E. Clarke, S. Jha, Y. Lu, and H. Veith, "Tree-like counterexamples in model checking," in *LICS*, 2002.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *TOPLAS*, 1986.

[9] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop on Logic of Programs*, Springer, 1981.

[10] B. Cook and E. Koskinen, "Reasoning about nondeterminism in programs," in *PLDI'13*. ACM, 2013.

[11] B. Cook, J. Fisher, E. Krepska, and N. Piterman, "Proving stabilization of biological systems," in *VMCAI'11*. Springer, 2011.

[12] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi, "Proving that programs eventually do something good," in *POPL'07*, 2007.

[13] B. Cook and E. Koskinen, "Making prophecies with decision predicates," in *POPL'11*. ACM, 2011.

[14] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI'06*. ACM, 2006.

[15] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1(1), 1959.

[16] E. A. Emerson and K. S. Namjoshi, "Automatic verification of parameterized synchronous systems (extended abstract)," in *CAV'96*. Springer, 1996.

[17] J. Esparza, A. Kucera, and S. Schwoon, "Model checking ltl with regular valuations for pushdown systems," *Information and Computation*, 186, 2003.

[18] R. W. Floyd, "Assigning meaning to programs," in *Mathematical Aspects of Computer Science*, ser. Proc. of Symposia in Applied Mathematics. American Mathematical Society, 1967.

[19] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning*, Springer, 2000.

[20] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that non-blocking algorithms don't block," in *POPL'09*. ACM, 2009.

[21] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," *SIGPLAN Not.*, 43, 2008.

[22] A. Gurfinkel, O. Wei, and M. Chechik, "Yasm: A software model-checker for verification and refutation," in *CAV'06*. Springer, 2006.

[23] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *VSTTE'12*, 2012.

[24] Y. Kesten and A. Pnueli, "A compositional approach to ctl* verification," *Theor. Comput. Sci.*, 331(2-3), 2005.

[25] O. Kupferman, M. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," *Journal of the ACM*, 47(2), 2000.

[26] S. Magill, J. Berdine, E. M. Clarke, and B. Cook, "Arithmetic strengthening for shape analysis," in *SAS'07*. Springer, 2007.

[27] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety*. Springer Verlag, 1995, vol. 2.

[28] K. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.

[29] H. Peng, Y. Mokhtari, and S. Tahar, "Environment synthesis for compositional model checking," in *Computer Design: VLSI in Computers and Processors*, 2002.

[30] A. Podelski and A. Rybalchenko, "Transition invariants," in *LICS'04*. IEEE, 2004.

[31] ——, "Transition invariants," in *LICS*, 2004.

[32] F. Song and T. Touili, "Pushdown model checking for malware detection," in *TACAS'12*. ACM, 2012.

[33] I. Walukiewicz, "Pushdown processes: Games and model checking," in *CAV'96*. Springer, 1996.

[34] ——, "Model checking ctl properties of pushdown systems," in *FSTTCS'00*. Springer, 2000.

Template-based circuit understanding

Adrià Gascón*, Pramod Subramanyan†, Bruno Dutertre*, Ashish Tiwari* Dejan Jovanović*, Sharad Malik†
*SRI International
adria.gascon@sri.com, bruno@csl.sri.com, tiwari@csl.sri.com, dejan.jovanovic@sri.com

†Princeton University
psubrama@princeton.edu, sharad@princeton.edu

***Abstract*—When verifying or reverse-engineering digital circuits, one often wants to identify and understand small components in a larger system. A possible approach is to show that the sub-circuit under investigation is functionally equivalent to a reference implementation. In many cases, this task is difficult as one may not have full information about the mapping between input and output of the two circuits, or because the equivalence depends on settings of control inputs.**

We propose a template-based approach that automates this process. It extracts a functional description for a low-level combinational circuit by showing it to be equivalent to a reference implementation, while synthesizing an appropriate mapping of input and output signals and setting of control signals. The method relies on solving an exists/forall problem using an SMT solver, and on a pruning technique based on signature computation.

I. Introduction

Digital circuits are designed and implemented in a top-down fashion, typically using computer-aided design (CAD) tools that provide several levels of abstraction. Hence, a variety of components must be understood separately to derive the high-level functionality of the whole system. However, after the original high-level description is mapped to a low-level digital circuit—i.e., a flattened netlist—most of the modularity that made the original description understandable is lost. For this reason, an unavoidable task in reverse-engineering of industrial size digital circuits is to extract subcircuits of the original design to verify them independently. This task is referred as the functional block identification step in [1]. Techniques that tackle this problem include structural, functional, and mixed approaches such as

1) FSM extraction [2]
2) Functional aggregation and matching [3]
3) Word identification and propagation [4]
4) Identification of repeated structures [5]

After identification of a component $\mathcal{C}$ by these methods, an important step is understanding $\mathcal{C}$'s functionality. Ideally, we would like a systematic way of obtaining a reasonable approximation of the high-level description of $\mathcal{C}$ in some Hardware Description Language (HDL). A possible approach is to try matching the function computed by $\mathcal{C}$ against a library of predefined components. However, this option is typically too strict in practice. A source of difficulty is that the mapping between the inputs of $\mathcal{C}$ and the component to be matched is usually unknown. *Permutation-Independent Equivalence Checking* (PIEC) addresses this problem [6], [7], [8]. It has been applied in the context of technique 2) above. Once the wires in a flattened netlist have been grouped into *unordered words*, and a combinational subcircuit $\mathcal{C}$ operating on those words has been extracted, $\mathcal{C}$ is checked for equivalence with known library components *modulo a permutation* of such words that *is determined* by the matching algorithm.

The research presented in this paper has been partially supported by the National Science Foundation under grant CCF-1423296.

Even with an equivalence checking algorithm that synthesizes a suitable input/output permutation, another practical difficulty may be that suitable library components are not available. Because of optimization steps applied in the flattening process, and the specifics of the design, $\mathcal{C}$ does not necessarily have a standard functionality. For example, our benchmark includes a subcircuit automatically obtained from a real design by unfolding an FSM found using a technique in the first category above. The circuit has 170 wires and 120 components from a circuit synthesis library, and it has 30 inputs out of which six are control signals. The circuit implements a 22-bit up counter modulo $(2^{20} + 2^{21})$ with synchronous reset and hold. This design can be described in fewer than ten lines of VHDL, but it is not reasonable to assume that we have a predefined reference circuit that exactly matches its functionality, even modulo a permutation of inputs and outputs. This motivates the need for more flexible functional matching algorithms that enables reverse engineering without prior low-level knowledge of the circuit under investigation.

Ideally, we would like to synthesize a suitable permutation of the inputs and the corresponding VHDL code for $\mathcal{C}$. Unfortunately that is not possible in practice. Instead, we solve a more constrained version of this synthesis problem: the combinational circuit $\mathcal{C}$ is checked for equivalence against a template spanning a finite, but possibly huge, family of high-level circuit descriptions. More specifically, our goal is to describe the functionality of a *combinational* circuit $\mathcal{C}$ using word-level operations such as concatenation, extraction, shifting, and rotation, as well as arithmetic functions on words such as addition, subtraction, multiplication, modulo, and the usual arithmetic comparison functions for signed and unsigned integers.

Our solution is inspired by recent progress in the area of program synthesis. Synthesizing a program from an abstract specification is not achievable in practice, but *template-based* synthesis is much more practical [9]. In this approach, the designer provides a *template* that captures the shape of the intended solution(s) together with the specification. A synthesis algorithm fills in the details. This general idea has

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

been successfully applied to several domains. For example, imperative programs can be obtained from a given sketch, as long as their intended behavior is also provided [10]; efficient bitvector manipulations can be synthesized from naïve implementations [11]; agent behavior in distributed algorithms can be synthesized from a description of a global goal [12]; circuits can be repaired given a specification of their intended behavior [13]; deobfuscated code can be obtained using similar ideas [14]. Although all these applications rely on template-based synthesis, different synthesis algorithms are used in different domains.

In our setting, the functional specification is the circuit $\mathcal{C}$ itself, and our goal is to generate a high-level description of its functionality by instantiating a user-provided template that operates at the word level. The template is a convenient way of integrating user knowledge to reduce the search space. Our approach automatically synthesizes both an input/output permutation and a set of Boolean conditions on some inputs, under which the circuit $\mathcal{C}$ is equivalent to a high-level description. We call this problem *Permutation-Independent Conditional Equivalence Checking* (PICEC).

Our approach to solving PICEC relies on (i) a set of syntactic transformations similar to the ones used in [15], (ii) an efficient implementation of validity checking for Boolean formulas over the theory of bitvectors with two levels of quantification, i.e., $\exists\forall$ QF_BV formulas, and (iii) the use of distinguishing signatures to handle the search for suitable input and output permutations.

We evaluated our techniques on a set of reverse-engineering benchmarks that were generated by synthesizing a variety of circuits described in high-level (behavioral) Verilog using the Synopsys Design Compiler (DC). All our benchmarks and circuits, both as high-level Verilog and as flattened netlist, are available at [16]. Our results indicate that our functional matching approach can be very effective in practice for any task that requires getting a precise understanding of the high-level functionality of a digital system.

In Section II, we introduce some notation and precisely state the Permutation Independent Conditional Equivalence Checking problem. In Section III, we briefly present our approach for solving the synthesis problem, including the preprocessing techniques. In Section IV, we review previous work on the use of output and input distinguishing signatures for solving PIEC and show how we used it in our context. In Sections V and VI, we present our experimental results and future lines of research.

II. Template-based circuit understanding

As mentioned in the previous section, our goal is to extract a high-level understanding of the behavior of a given combinational circuit $\mathcal{C}$. More specifically, we would like to raise the level of abstraction of the description of the functionality of $\mathcal{C}$ from bits and standard logical gates to a variety of word-level manipulation operations and arithmetic functions. Motivated by this goal, we first formulate a generic *Permutation Independent Conditional Equivalence Checking* (PICEC, pronounced "pieces") problem, then present a refined PICEC problem. Finally, we show how it can be solved using an exists-forall solver.

Let I and O be disjoint sets of variables ranging over some domain D. Intuitively, I and O correspond to the inputs and outputs of our circuit $\mathcal{C}$. In all our experiments, D is the Boolean domain, but the PICEC problem can be defined for arbitrary domains. Given a set V of variable ranging over D, by $\mathtt{Words_k}(V)$ we denote the set of words over V of length k. We simply refer to $\mathtt{Words}(V)$ when k is clear from the context or irrelevant.

Since our goal is to raise the level of abstraction of the description of the functionality of $\mathcal{C}$ from bits to words, a key challenge is *finding the right words* from the sets I and O. To do so, our procedure must consider all possible functions that produce a word of a certain size from I and O, so-called extraction functions. An *extraction function* is a function that maps a set V of variables to a word in $\mathtt{Words}_k(V)$, for some positive constant k.

We are now ready to define our problem precisely. As commented in Section I, our goal is to provide a *flexible* procedure for checking whether a circuit exhibits a certain behavior, that is, checking whether a circuit may compute a certain function under conditions *to be determined* and for some selection of its inputs and outputs that is also *to be determined*. The *Generic* PICEC Problem captures this idea. The goal of the following definition is to provide the reader with a high-level intuition of the goal of our formalization. As comented above, this general definition is then refined to the formulation of the problem being addressed in this work, which is presented in Definition 3 below.

Definition 1 (Generic PICEC): Given a quantifier-free formula $\mathcal{C}(I,O)$ (over free variables I and O), and a function $\phi : \mathtt{Words}(D) \times \mathtt{Words}(D) \mapsto \mathtt{Words}(D)$, the *PICEC problem* seeks to find
(a) a partition $I_C \cup I_D$ of I into control variables I_C and data variables I_D,
(b) a satisfiable formula $\psi(I_C)$ with free variables in I_C,
(c) extraction functions $\mathtt{ex}_1, \mathtt{ex}_2$ on I_D, and
(d) an extraction function $\mathtt{ex}_3$ on O,
such that the sentence

$$\begin{aligned}\forall I, O \ : \ & \mathcal{C}(I,O) \Rightarrow \\ & (\psi(I_C) \Rightarrow (\mathtt{ex}_3(O) = \phi(\mathtt{ex}_1(I_D), \mathtt{ex}_2(I_D))))\end{aligned}$$

is valid in the theory of the underlying domain $\mathtt{Words}(D)$.

Intuitively, a solution of the generic PICEC problem shows that, under the condition $\psi(I_C)$, the circuit $\mathcal{C}$ behaves like the function ϕ on some suitably identified input and output words. Solving the generic PICEC problem amounts to synthesizing the parts (a)–(d) in Definition 1.

Example 1: Consider a circuit $\mathcal{C}(I,O)$ with set of binary inputs $I = \{i_1, i_2, i_3, i_4, c\}$ and a single output o. Assume that $\mathcal{C}$ implements the following function

$$f(i_1, i_2, i_3, i_4, c) = \begin{cases} i_1 i_2 > i_3 i_4, & \text{if } c = 0 \\ i_1 i_1 \geq i_3 i_4, & \text{otherwise.} \end{cases}$$

and consider the function $\phi(w_1, w_2) = (w_1 \geq w_2)$. An *interesting solution* of this PICES instance consists of: (a) the partition $I = \{c\} \cup \{i_1, i_2, i_3, i_4\}$, (b) the Boolean formula $\psi(\{c\}) = c$, (c) extraction functions $\mathtt{ex}_1(I \setminus \{c\}) =$

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

$i_1 i_1, \texttt{ex}_2(I \setminus \{c\}) = i_3 i_4$, and (d) the extraction function $\texttt{ex}_3(\{o\}) = o$.

We have defined ϕ as a binary function to keep the presentation simple, but the definition generalizes to functions of any arity. Furthermore, in practice, we do not have just one function ϕ that we are "searching for" in a circuit $\mathcal{C}$, but a whole set $\phi_1, \ldots, \phi_m$ of functions. In this case, we want to share the partition synthesized in Part (a) across all the m functions, but synthesize different Parts (b)–(d) for the m different functions. This extension corresponds to the *Generic m-PICEC problem* defined as follows:

Definition 2 (Generic m-PICEC): Given a quantifier-free formula $\mathcal{C}(I, O)$ (over free variables I and O), and given m functions $\phi_1, \ldots, \phi_m$ each with signature $\texttt{Words}(D) \times \texttt{Words}(D) \mapsto \texttt{Words}(D)$, the *generic m-PICEC problem* seeks to find
(a) a partition $I_C \cup I_D$ of I into control variables I_C and data variables I_D,
(b) m satisfiable formulas $\psi_i(I_C)$ with free variables in I_C,
(c) $2m$ extraction functions $\texttt{ex}_{i,1}, \texttt{ex}_{i,2}$ on I_D, and
(d) m extraction functions $\texttt{ex}_{i,3}$ on O,
(where $i \in \{1, \ldots, m\}$ in Items (b)–(d)) such that the sentence

$$\forall I, O \ : \ \mathcal{C}(I, O) \Rightarrow \Big(\bigwedge_i \psi_i(I_C) \Rightarrow (\texttt{ex}_{i,3}(O) = \phi_i(\texttt{ex}_{i,1}(I_D), \texttt{ex}_{i,2}(I_D)))\Big)$$

is valid in the theory of the underlying domain D.

Note that a solution to the m-PICEC problem does not necessarily specify a total mapping between inputs and outputs values of $\mathcal{C}$, but only a mapping under the condition $\bigvee_i \psi_i(I_C)$, and hence the first C in PICEC. This flexibility is very helpful in a reverse-engineering process to *incrementally* understand the high-level functionality of the circuit.

The generic m-PICEC problem raises two issues. First, its synthesis search space (that is, the state space of the synthesis parameters in Parts (a)–(d) above) is huge. More importantly, it does not provide a way of integrating user-provided knowledge to reduce the synthesis search space. In particular, the user might have some knowledge about which variables form *unordered words*. The user may also wish to put constraints on the different extraction functions used for different choices of i (saying that some of them have to be the same extraction function). This is typically the case in practice, for example, when trying to understand an ALU-like circuit.

The user's knowledge of the circuit is captured in a *template*. A template T for a circuit $\mathcal{C}(I, O)$ is an 8-tuple

$$\langle O_T, \{S_1, \ldots, S_n, I_C\}, p, \{\phi_1, \ldots, \phi_m\}, \mathit{arg_1}, \mathit{arg_2}, \mathit{perm_1}, \mathit{perm_2} \rangle$$

where $O_T \subseteq O$ is a subset of output variables, $I = (I_C \cup \bigcup_{i=1}^{n}(S_i))$ is a partition of the input variables, $p \geq 1$ is a natural number, the ϕ_i's are binary functions over words as before, and $\mathit{arg_1}, \mathit{arg_2} : \overline{m} \mapsto \overline{n}$ and $\mathit{perm_1}, \mathit{perm_2} : \overline{m} \mapsto \overline{p}$ are mappings. Here by $\overline{m}$ we denote the set $\{1, \ldots, m\}$.

Intuitively, O_T represents the subset of outputs of T explained in the template, the partition of I captures knowledge on how input words and control inputs are grouped. The problem is to correctly order the wires within those words by synthesizing p input permutations $\sigma_1, \ldots, \sigma_p$. The ϕ_i's are functions that the circuit is expected to implement *under some conditions on the inputs in I_C*. The template specifies that the input to ϕ_i are the two sets of $S_{\mathit{arg_1}(i)}$ and $S_{\mathit{arg_2}(i)}$ and that these sets of wires must be ordered according to permutations $\sigma_{\mathit{perm_1}(i)}$ and $\sigma_{\mathit{perm_2}(i)}$, respectively.

We are now ready to define the m-PICEC problem.

Definition 3 (m-PICEC problem): Given a quantifier-free formula $\mathcal{C}(I, O)$ (over free variables I and O), and given a template T as defined above, the *m-PICEC problem* seeks to find
(a) $p + 1$ permutations $\theta, \sigma_1, \ldots, \sigma_p$ and
(b) m satisfiable formulas $\psi_i(I_C)$ with free variables in I_C,
such that the sentence

$$\forall I, O : \mathcal{C}(I, O) \Rightarrow \bigwedge_i (\psi_i(I_C) \Rightarrow Eq_i) \qquad (1)$$

is valid in the theory of the underlying domain D, where Eq_i stands for

$$(\theta(O_T) = \phi_i(\sigma_{\mathit{perm_1}(i)}(S_{\mathit{arg_1}(i)}), \sigma_{\mathit{perm_2}(i)}(S_{\mathit{arg_2}(i)})))$$

Since encoding the "satisfiable" condition in Part (b) is tricky, we assume that the formula $\psi_i(I_C)$ denotes *an assignment* to the variables in I_C. Then, it immediately follows that the m-PICEC problem reduces to checking validity of the following exists-forall *synthesis constraint*:

$$\exists \psi_1, \ldots, \psi_m, \sigma_1, \ldots, \sigma_p, \theta : \forall I, O : \Phi \qquad (2)$$

where Φ is the matrix (quantifier-free part) of Formula (1).

Example 2: In practice, we specify the templates using an extension of the Yices language [17], as illustrated in Figure 1. In this example, we wish to determine whether a circuit $\mathcal{C}$ behaves as an adder under some condition and as a comparator under another condition. The corresponding formal template is given by

$$O_T := \texttt{outputs}$$
$$\{S_1 := \texttt{inputsA}, S_2 := \texttt{inputsB}, I_C := \texttt{control}\}$$
$$p := 2$$
$$\{\phi_1 := \texttt{bv-add}, \phi_2 := \texttt{bv-slt-int}\}$$

with $\mathit{arg_1}$, $\mathit{arg_2}$, $\mathit{perm_1}$, and $\mathit{perm_2}$ defined by $\mathit{arg_1}(i) = 1, \mathit{arg_2}(i) = 2, \mathit{perm_1}(i) = 1, \mathit{perm_2}(i) = 2$ for $i = 1, 2$. The function `bv-slt-int` is a signed less-than operator that returns 1 or 0. Let $\prec$ denote the less-than relation on signed integers encoded in two's complement representation. The synthesis constraint is satisfiable if there exist two permutations p and q, and bitvector constants $v1$ and $v2$, such that, for all possible values of `inputsA` and `inputsB`, (1) whenever $\texttt{control} = v1$, then $\mathcal{C}$ outputs $p(\texttt{inputsA}) + q(\texttt{inputsB})$ and (2) whenever $\texttt{control} = v2$, then $\mathcal{C}$ outputs 1 if $p(\texttt{inputsA}) \prec q(\texttt{inputsB})$ and 0 otherwise.

Since we are dealing either with combinational circuits or unfolding of sequential circuits, the relation $\mathcal{C}(I, O)$ can be represented as a Boolean formula. Then Formula 2 belongs to the logic of fixed-sized bit vectors with two levels of quantification $\exists$ and $\forall$. In the following section, we describe our solver, whose implementation is based on the Yices [18] SMT-solver. Our solver applies some general preprocessing

```
(and
    (=>
        (value v1 control)
            (=
               outputs
                   (bv-add
                       (permute p inputsA)
                       (permute q inputsB)
    )   )   )

    (=>
        (value v2 control)
            (= outputs
                  (ite
                      (bv-slt
                          (permute p inputsA)
                          (permute q inputsB)
                      )
                      (mk-bv 32 1)
                      (mk-bv 32 0)
)   )   )   )     )
```

Fig. 1. Example of a user-defined template

techniques also used in [15]. In particular equality resolution is very effective in our setting due to the restricted form of our templates.

III. Solving the ∃∀ problem

The synthesis problem reduces to solving Boolean formulas over the theory of bit-vectors with two levels of quantification, commonly called the ∃∀ QF_BV fragment. Formulas in this fragment have the general form

$$(\exists \vec{x})\,(E(\vec{x}) \wedge (\forall \vec{y}) A(\vec{x}, \vec{y}))$$

Such formulas can be reduced to quantified Boolean formulas and delegated to a general QBF solver (e.g., [19]). Instead, we opt for reasoning at the higher level of bit-vectors and relying on a counterexample-refinement loop, similar to the approach used in 2QBF solvers (e.g., [20], [21]).

This loop is sketched in Figure 2. Given an ∃∀ formula, as above, the procedure is a game between two (quantifier-free) bit-vector solvers. The first solver generates candidate solutions for the existential variables $\vec{x} \mapsto \vec{a}$ by solving E. If there are no solutions to E, then the ∃∀ formula is unsatisfiable. Otherwise, the second solver checks whether the candidate solution $\vec{a}$ is correct, by trying to refute A modulo the assignment $\vec{x} \mapsto \vec{a}$. If the latter formula can not be refuted, then $\vec{a}$ is a solution to the ∃∀ problem. Otherwise, the second solver produces a refutation counterexample $\vec{b}$. This counterexample $\vec{b}$ eliminates $\vec{a}$ from the set of candidates for the existential variables. But $\vec{b}$ can eliminate more candidates than $\vec{a}$: all good candidates must satisfy $A[\vec{y}/\vec{b}]$. This new assertion (on the variables $\vec{x}$) is then added to the first solver's context and the loop proceeds. It is easy to see that this procedure terminates as the variables $\vec{x}$ have a finite domain. It is worth noting that the more general procedure for deciding quantified bit-vectors and uninterpreted functions in the Z3 SMT solver [15] reduces to our procedure when used on the ∃∀ QF_BV fragment.

A. Formula Simplification

The ∃∀ procedure is complete but may be very slow to terminate. High-level preprocessing and simplifications of the ∃∀ formula are essential to make it practical.

loop
  $\langle sat_x, \vec{x} \mapsto \vec{a}\rangle \leftarrow$ SMT-SOLVE(E)
  if not sat_x **then**
    return unsat
  $\langle sat_y, \vec{y} \mapsto \vec{b}\rangle \leftarrow$ SMT-SOLVE$(\neg A[\vec{x}/\vec{a}])$
  if not sat_y **then**
    return $\langle$**sat**$, \vec{x} \mapsto \vec{a}\rangle$
  $E \leftarrow E \wedge A[\vec{y}/\vec{b}]$

Fig. 2. Main loop for solving $(\exists \vec{x})\,(E(\vec{x}) \wedge (\forall \vec{y})\; A(\vec{x}, \vec{y}))$.

For reducing the scope of quantifier we distribute quantifiers over compatible Boolean operators (this is known as *miniscoping*):

$$\begin{array}{lcl}(\exists \vec{x})A \vee B & \Leftrightarrow & (\exists \vec{x})A \;\vee\; (\exists \vec{x})B \\ (\forall \vec{x})A \wedge B & \Leftrightarrow & (\forall \vec{x})A \;\wedge\; (\forall \vec{x})B\end{array}$$

The first simplification decomposes an ∃∀ problem into smaller subproblems, while the second simplification reduces candidate checking to several smaller checks.

It is common for our ∃∀ problems to contain subformulas of the following form:

$$(\exists \vec{x})(E(\vec{x}) \wedge (\forall \vec{y})(\bigwedge_{i \in S} (y_i = x_{j_i}) \Rightarrow B(\vec{y})))$$

where S is a subset of indexes in $1..|\vec{y}|$ and $j_i \in 1..|\vec{x}|$, for every $i \in S$. A naïve application of our procedure does not work well on such problems. To illustrate a worst case scenario, let us assume that B is unsatisfiable. In such a case, each iteration of our procedure will pick a fresh candidate assignment $\vec{x} \mapsto \vec{a}$, then refute the universal subformula with a counterexample $\vec{y} \mapsto \vec{b}$. Since we must have that $b_i = a_{j_i}$, for every $i \in S$, and B evaluates to false under $\vec{b}$, the counterexample instantiation yields the weakest possible explanation $\bigvee_{i \in S}(x_{j_i} \neq a_{j_i})$, which (essentially) eliminates only the current candidate for $\vec{x}$.[1]

To preserve the connections that equalities introduce over quantifiers, we perform *equality resolution*. We detect equalities of the form $(y_i = t_i)$ in the antecedents of universal subformulas, then solve out the variables y_i. In our previous examples, this simplifies the problem to the equivalent formula

$$(\exists \vec{x})(E(\vec{x}) \wedge (\forall \vec{y})B')$$

where B' is the result of solving out the variables y_i from B. On this simplified formula, the procedure now has a chance to eliminate more than one candidate at each iteration.

In addition to these formula simplifications, we reduce the solver's search space by relying on *distinguished signatures*, a technique originally proposed for solving PIEC.

IV. Distinguishing signatures

Functional equivalence checking of circuits is a central problem in logic synthesis and verification. Roughly speaking, it consists of determining whether two given circuits $\mathcal{C}_1(I_1, O_1)$, $\mathcal{C}_2(I_2, O_2)$ implement the same function. Without loss of generality, we can assume that $n = |I_1| = |I_2|$ and $m = |O_1| = |O_2|$.

[1]It may eliminate more than one candidate if S is a strict subset of $1..|\vec{y}|$.

In our setting, $\mathcal{C}_1$ and $\mathcal{C}_2$ are combinational circuits represented as multi-output Boolean functions $f = (f^1, \dots, f^m)$ and $g = (g^1, \dots, g^m)$, respectively. The combinational equivalence checking problem consists in deciding whether the sentence $\forall 1 \le i \le m : \forall x_1, \dots, x_n : f^{\theta(i)}(x_{\sigma(1)}, \dots, x_{\sigma(n)}) = g^i(x_1, \dots, x_n)$ is valid, where σ and θ are input and output correspondence between $\mathcal{C}_1$ and $\mathcal{C}_2$; σ is the input permtation and θ is the output permutation.

In the PIEC problem, the mappings σ and θ are not known and we must synthesize them. We can then formulate the problem as checking validity of the formula

$$\exists\theta,\sigma : \forall 1 \le i \le m, x_1, \dots, x_n : \quad f^{\theta(i)}(x_{\sigma(1)}, \dots, x_{\sigma(n)}) = g^i(x_1, \dots, x_n) \tag{3}$$

This problem has been widely studied [22], [23], [8]. To reduce the size of the search space for θ, a common approach is to assign an abstract *signature* to every output of $\mathcal{C}_1$ and $\mathcal{C}_2$, with two key properties. First, functions with different signatures cannot be equivalent. Second, the signature of a function f_i (or g_i) is invariant under any permutation of the input variables $x_1, \dots, x_n$. If g^i and f^j have different signatures, we can reduce the search to output permutations that satisfy $\theta(i) \neq j$. This method extends to the input signals: one can also assign signatures to every input x_i to eliminate a priori some invalid input permutations σ.

More concretely, let $\mathcal{B}_n$ be the set of all single-output Boolean functions with n input variables, and let $\mathcal{D}$ be an ordered set. An input signature is a function $s_{in} : \{x_1, \dots, x_n\} \times (\mathcal{B}_n)^m \to \mathcal{D}$, such that the equality

$$s_{in}(x_i, f^1(x_1, \dots, x_n), \dots, f^m(x_1, \dots, x_n)) = s_{in}(x_{\sigma(i)}, f^{\theta(1)}(x_{\sigma(1)}, \dots, x_{\sigma(n)}), \dots, f^{\theta(m)}(x_{\sigma(1)}, \dots, x_{\sigma(n)}))$$

holds for every input permutation σ and output permutation θ. Similarly, an output signature s_{out} is a function $s_{out} : \mathcal{B}_n \to \mathcal{D}$ such that $s_{out}(f(x, \dots, x_n)) = s_{out}(f(x_{\sigma(1)}, \dots, x_{\sigma(n)}))$ holds for any input permutation σ.

Consider Formula (3) above and assume that, for some inputs x_i and x_j, we have $s_{in}(x_i, f^1, \dots, f^n) \neq s_{in}(x_j, g^1, \dots, g^n)$. Then, since equal signatures are a necessary condition for i to be mapped to j by the input permutation σ, it follows that $\sigma(i) \neq j$. The case of output permutations and an output signature s_{out} is analogous. We can collect all disequality constraints derived from input signatures in a formula $C_{in}(\sigma)$ and all disequalities derived from output signatures in $C_{out}(\theta)$. Then, Formula 3 is equivalent to

$$\exists\theta,\sigma : C_{in}(\sigma) \wedge C_{out}(\theta) \wedge \forall 1 \le i \le m, x_1, \dots, x_n : f^{\theta(i)}(x_{\sigma(1)}, \dots, x_{\sigma(n)}) = g^i(x_1, \dots, x_n) \tag{4}$$

We apply a similar idea to our synthesis constraint (Formula (2)) from Section II.

A variety of signatures have been presented in the literature, many of them derived from a Reduced Ordered Binary Decision Binary Diagrams (ROBDDs) representation of Boolean functions. For a detailed presentation of a variety of signatures, their applications, and limitations, the reader is referred to [24], [25]. In this paper we focus on two signatures that do not rely on ROBDDs and are thus more scalable.

A. The in_dep and out_dep Signatures

Given a Boolean formula $f(x_1, \dots, x_n)$ and a variable x_i, we say that f essentially depends on x_i, denoted by $f \preceq x_i$, if there exists an Boolean tuple $(\alpha_1, \dots, \alpha_n)$ such that $f(\alpha_1, \dots, \alpha_i, \dots, \alpha_n) \neq f(\alpha_1, \dots, \bar{\alpha_i}, \dots, \alpha_n)$. Consider a circuit defined by m functions $f_1, \dots, f_m$; we define the *input dependence set of* x and the *output dependence set of* f_i as follows:

$$\begin{aligned} \mathit{in_dep_set}(x, f^1, \dots, f^n) &= \{f^j : f^j \preceq x\} \\ \mathit{out_dep_set}(f_i) &= \{x : f_i \preceq x\} \end{aligned}$$

Then, we define the two following signatures:

(a) $\mathit{in_dep}(x, f_1, \dots, f_n) = |\mathit{in_dep_set}(x, f_1, \dots, f_n)|$

(b) $\mathit{out_dep}(f) = |\mathit{out_dep_set}(f)|$

We must adapt these signatures to take templates into account. We want to produce a formula $C(\theta) \wedge C(\sigma_1) \wedge \dots \wedge C(\sigma_n)$ that, as in the case of Formula 4, can be added to our synthesis constraint while preserving validity.

Recall that we defined a template as a tuple

$$\langle O_T = \{o_1, \dots, o_l\}, \{S_1, \dots, S_n, C\}, p, \{\phi_1, \dots, \phi_m\}, \mathit{arg_1}, \mathit{arg_2}, \mathit{perm_1}, \mathit{perm_2}\rangle.$$

The inputs and outputs of functions $\phi_1, \dots, \phi_m$ are all bit vectors. We can then interpret each ϕ_i as a multi-output Boolean function, and we denote by ϕ_i^k the k-th bit of ϕ_i's output. We define our template-version of $\mathit{in_dep}$ and $\mathit{out_dep}$, which we denote by $\mathit{in_dep}_T$ and $\mathit{out_dep}_T$, for every input x_i and output o^k as follows

(c) $\mathit{in_dep}_T(x_i, T) = |\bigcup_{j=1}^m \mathit{in_dep_set}(x_i, \phi_j(S_{\mathit{arg_1}(j)}, S_{\mathit{arg_2}(j)}))|$

(d) $\mathit{out_dep}_T(o^k) = |\bigcup_{j=1}^m \mathit{out_dep_set}(\phi_j^k((S_{\mathit{arg_1}(j)}, S_{\mathit{arg_2}(j)})))|$

To see how to take advantage of this definition of signatures, consider a combinational circuit $\mathcal{C}$ and its representation as single-output Boolean functions $f_{\mathcal{C}}^1, \dots, f_{\mathcal{C}}^m$ with input variables $x_1, \dots, x_n$. Consider a template T for $\mathcal{C}$. The key observation is that fixing the value of variables in C cannot cause $\mathit{in_dep}(x, f_{\mathcal{C}}^1, \dots, f_{\mathcal{C}}^n)$ or $\mathit{out_dep}(f_{\mathcal{C}}^i)$ to increase. Let j be any index in $\{1, \dots, m\}$, and let x and y be input variables in $S_{\mathit{arg_1}(j)}$. Then we have that

$$(\mathit{in_dep}(x_{k_1}, f_{\mathcal{C}}^1, \dots, f_{\mathcal{C}}^n) > \mathit{in_dep}_T(x_{k_2}, T)) \Rightarrow \sigma_{\mathit{perm_1}(j)}(x) \neq y$$

An analogous implication holds for $x, y \in S_{\mathit{arg_2}(j)}$. Similarly, for the output permutation θ, if $f_{\mathcal{C}}^i$ corresponds to a variable in $o_i \in O_T$ then, for any $k \in \{1, \dots, l\}$, we have

$$(\mathit{out_dep}(f_{\mathcal{C}}^i) > \mathit{out_dep}_T(o^k)) \Rightarrow \theta(o_i) \neq o_k$$

The definition of essential dependence at the beginning of this section directly gives us a procedure to precompute input and output signatures of $\mathcal{C}$ and T by means of a quadratic number of calls to an SMT solver. Other signatures based on model counting are known to be very effective but they require circuits to be represented using ROBDDs. Our signature computation scales better than these BDD-based

approaches but we did not put emphasis on efficient signature computation in our investigations. Other symbolic approaches might be more effective.

In summary, we have an effective approach to produce a conjunction of constraints $C_{out}(\theta) \wedge C_{in}(\sigma_1) \wedge \ldots \wedge C_{in}(\sigma_n)$ that eliminates irrelevant permutations, and such that the formula

$$\exists \psi_1, \ldots, \psi_m, \sigma_1, \ldots, \sigma_n, \theta : \\ \forall I, O : C_{out}(\theta) \wedge C_{in}(\sigma_1) \wedge \ldots \wedge C_{in}(\sigma_n) \wedge \Phi$$

is equivalent to our synthesis constraint from section II.

In our implementation, we encode a permutation σ using a quadratic number of Boolean variables $\sigma_{i,j}$ such that $\sigma(i) = j \Leftrightarrow \sigma_{i,j}$. With this encoding, the formulas $C_{out}(\theta)$ and $C_{in}(\sigma)$ are simply conjunctions of literals.

Let us remark that, since the values of the signatures can be computed *independently* in the template and the circuit, we do not report on computation time needed for signature computation in our examples. Nevertheless, using a naive implementation based on calling an SMT solver, we can compute the signatures for all our examples in the order of minutes.

V. Experimental evaluation

We have evaluated our techniques on a set of reverse engineering benchmarks. These are flattened Verilog netlists that contain components such as ALUs, multipliers, shifters and counters. The benchmars were derived from various sources including the ISCAS'85 benchmarks, an ALU from an academic processor implementation, and synthetic examples. The flattened netlists were generated by synthesizing high-level (behavioral) Verilog using the Synopsys Design Compiler (DC). All the circuits, both in high-level Verilog and flattened netlists form are available at [16].

Our benchmarks exemplify the situation where a reverse engineer tries to understand the high-level functionality of a flattened design with limited information about the operation that it may perform and no detailed knowledge of its input/output buses. In less restrictive cases, the reverse engineer is given the grouping of the inputs into *unordered* words, and in some cases no information at all is known. The output of Synopsys DC is an optimized flattened Verilog netlist and our goal is to identify and extract the high-level modules contained within this flat netlist using template-based matching. Our toolchain reads these flattened Verilog netlists and the templates against which they are to be matched. It then encodes the matching problem as satisfiability queries to be solved by a backend solver. Currently, we can generate queries for the SMT solvers Yices and Z3, and QBF instances in the Q-Dimacs format.

Our template library contains modules such as adders, subtracters, shifters, multipliers, and counters of varying bitwidths. Each netlist was matched against a subset of these templates. We ensured that each netlist was matched against an approximately equal number of satisfying (matching) and unsatisfying (not matching) instances. The total number of instances is 40, of which an equal number are satisfiable and unsatisfiable. We believe these instances are a challenging yet realistic set of benchmarks relevant to the reverse engineering/logic deobfuscation problem. We have made the QBF, Yices and SMT2 instances generated by these matching problems available at [16]. The solver binaries used in our experiments are also available at this location.

We evaluated the performance of the following solvers: Yices [18] and Z3 [26], the QBF solvers RAReQS [27], DepQBF [28] and sKizzo [29], and a variant of the algorithm in Figure 2 (and also Algorithm 1 in [21]) that is somewhat similar to the algorithm presented in [13] that operates on a Boolean circuit representation. We refer as Cir-CEGAR to this variant in the rest of this paper. Since the encoding of our instances is written using the Yices language, we converted our instances to (1) the QDIMACS format used by the QBF solvers using the Yices standard bitblasting procedure, (2) SMTLIB-2 format using a simple syntactic transformation (basically a renaming of bitvector operations), and (3) QDIMACS format with a distinguished special literal equivalent to the validity of the whole formula, as required by Cir-CEGAR. Transformations (1) and (3) were performed after the simplification steps presented in Section III. To assess the effectiveness of Cir-CEGAR, we also produced benchmarks from the original formula, without applying the preprocessing steps. Empirical results are presented at the end of this section.

We modified Yices to incorporate the $\exists\forall$ solver algorithm from Section III. We refer to this modified version as Yices_EF in the results. Cir-CEGAR was implemented using Minisat v2.2 as the underlying SAT solver. When testing the QBF solvers, we first simplified the QBF-instances using Bloqqer [30]. The solvers we used include Z3 v4.3.2, for Linux x64 nightly build downloaded on 2014-05-14, RAReQS v1.1, DepQBF v3.0 and sKizzo v0.8.2. We executed the solvers on a cluster with Intel Xeon E31230 and E5645 processors with a one-hour timeout.

The QBF solvers did not work well on our benchmarks. RAReQS solved only three instances, while DepQBF and sKizzo did not solve any. Therefore, we omit results from these three solvers in the rest of this section.

A. Results

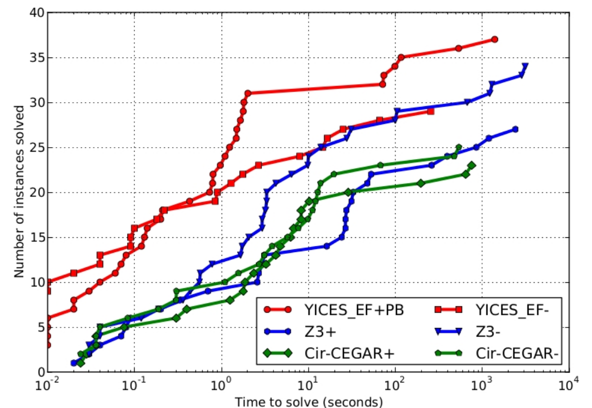


Fig. 3. Comparison of solver performance: Yices, Z3, Cir-CEGAR

Figure 3 shows the number of instances solved by a solver (y-axis) given a particular time limit (x-axis). The encoding of

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

permutations has a significant impact on solver performance. Our default encoding is explained in the previous section. A permutation σ is defined by a quadratic number of Boolean variables $\sigma_{i,j}$ and constraints such that $\sigma(i) = j \Leftrightarrow \sigma_{i,j}$. This *positive encoding* is denoted by the suffix '+' in the plot. We also experimented with a *negative encoding* (denoted by the suffix '-' in the graph). In the negative encoding, the polarity of the Boolean variables is reversed, that is, we have $\sigma(i) = j \Leftrightarrow \bar{\sigma}_{i,j}$ for each i and j in σ's domain.

The choice of encoding is significant as it interacts with the decision heuristics employed by the SMT solvers. By default, Yices uses negative branching with phase caching [31]. With this heuristic, each time Yices makes a decision on the value of a Boolean variable $\sigma_{i,j}$, it gives preference to the value $false$. This leads to poor performance on benchmarks that use the positive encoding, as setting $\sigma_{i,j}$ to $false$ triggers no unit propagations. After noticing this issue, we changed Yices's branching heuristic to use "positive-branching" (i.e., prefer $true$ over $false$). This is denoted by the suffix PB in the graph. With this setting, Yices solves 37 instances within the time limit. It performs worse with the negative encoding (and the default branching heuristics), solving only 29 instances. In its default configuration, Z3 has better results with the negative encoding. It solves 33 instances with this encoding but only 27 instances with the positive encoding. Cir-CEGAR is not as sensitive to the encoding as Yices and Z3.

Figure 4 shows the benefit of signatures. On the x-axis of each graph we show the time to solve the instance without signatures, while the y-axis is the time to solve the instance with signatures. Most points on these graphs are below the diagonal, showing that adding signatures is a gain in most cases. Many instances cannot be solved within our 3600 s timeout without signatures, but can be solved when signatures are added. The few outliers are instances in which the solver "gets lucky" even without signatures, which happens mostly on satisfiable instances.

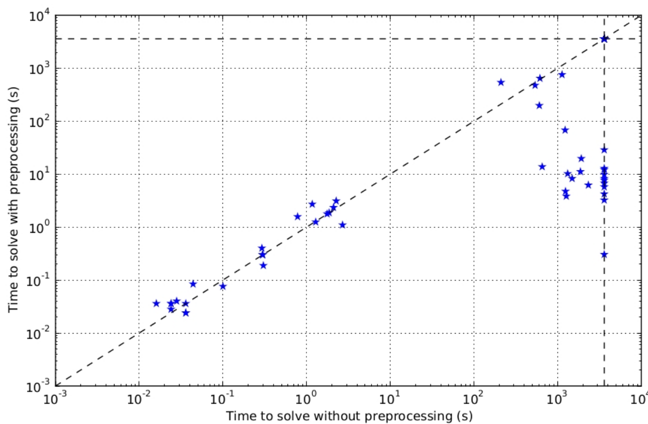


Fig. 5. Improvement in solver performance due to preprocessing. Results are for Cir-CEGAR.

Figure 5 shows the impact of formula simplification presented in Section III on Cir-CEGAR. The x-axis shows the number of seconds taken by the solver when preprocessing is *not* performed on the QBF instances while the y-axis shows the time taken by the solver when preprocessing *is* performed. As before, instances which failed to finish are represented with a value of 3600 seconds. We see that a number of such instances are present on the vertical line with $x = 3600s$. These are instances solved with preprocessing but not when preprocessing was omitted. The behavior is quite interesting. Either preprocessing has little effect on solver performance (the points close to the diagonal) or it has a huge effect (the points where $x > 10^3$ and $y < 10^2$).

VI. Conclusion and Further Work

We have presented the Permutation Independent Conditional Equivalence Checking problem (PICEC) as a method for synthesizing high-level functional descriptions of combinational circuits. PICEC extends permutation independence equivalence checking by considering control signals and conditional matching. We solve the problem using a template-based approach. A template can be seen as describing a (usually very large) family of possible high-level descriptions. Our procedure automatically instantiates the template to match the circuit under investigation. Templates encode partial knowledge about the circuit provided by the user.

PICEC can be reduced to solving formulas in the logic of fixed-sized bit vectors with two levels of quantification $\exists$ and $\forall$— that is, $\exists\forall QF_BV$. We have implemented a solver for this class of problems using the Yices SMT solver. We have shown that distinguishing signatures are effective to prune the solver search space and lead to significant performance improvement.

We have evaluated this approach on a set of realistic reverse-engineering benchmarks, using different solvers and permutation encodings. Our benchmarks are available to the community in four formats: Yices language, SMT2, QDIMACS, and the QDIMACS format with a special top literal used in in Cir-CEGAR.

An interesting line of further research is in exploring more complex signatures, and efficient algorithms to compute their values. We also plan to investigate whether our pruning approach based on signatures can be included as part of the interaction between the two solvers in the algorithm of Section III.

References

[1] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *HOST*. IEEE, 2012, pp. 83–88.

[2] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *ISCAS*. IEEE, 2010, pp. 2610–2613.

[3] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *DATE*, E. Macii, Ed. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 1277–1280.

[4] W. Li, A. Gascón, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *HOST*. IEEE, 2013, pp. 67–74.

[5] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[6] J. Mohnke, P. Molitor, and S. Malik, "Establishing latch correspondence for sequential circuits using distinguishing signatures," *Integration*, vol. 27, no. 1, pp. 33–46, 1999.

[7] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification," in *ICCD*. IEEE Computer Society, 1992, pp. 452–458.

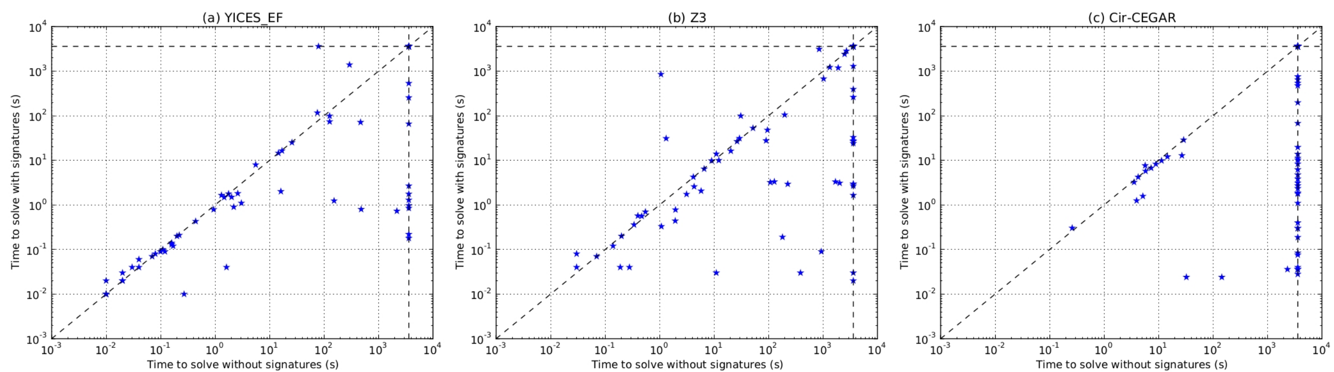


Fig. 4. Improvement in solver performance with signatures.

[8] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 137–148, 1997.

[9] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–17.

[10] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[11] M. W. Hall and D. A. Padua, Eds., *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 2011.

[12] A. Gascón and A. Tiwari, "A synthesized Algorithm for Interactive Consistency," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, J. M. Badger and K. Y. Rozier, Eds., vol. 8430. Springer, 2014, pp. 270–284.

[13] M. Fujita, S. Jo, S. Ono, and T. Matsumoto, "Partial synthesis through sampling with and without specification," in *ICCAD*, J. Henkel, Ed. IEEE/ACM, 2013, pp. 787–794.

[14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE (1)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 215–224.

[15] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "Efficiently solving quantified bit-vector formulas," *Formal Methods in System Design*, vol. 42, no. 1, pp. 3–23, 2013.

[16] "Online repository of benchmarks and experimental results," https://bitbucket.org/spramod/fmcad14-experiments, 2014.

[17] B. Dutertre, "Yices 2 Manual," Computer Science Laboratory, SRI International, Tech. Rep., 2014, available at http://yices.csl.sri.com.

[18] ——, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.

[19] A. Biere, "Resolve and expand," in *Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 59–70.

[20] D. P. Ranjan, D. Tang, and S. Malik, "A Comparative Study of 2QBF Algorithms," in *SAT*, 2004.

[21] M. Janota and J. P. M. Silva, "Abstraction-Based Algorithm for 2QBF," in *SAT*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 230–244.

[22] *Proceedings 1991 IEEE International Conference on Computer Design: VLSI in Computer & Processors, ICCD '92, Cambridge, MA, USA, October 11-14, 1992*. IEEE Computer Society, 1992.

[23] J. Mohnke and S. Malik, "Permutation and phase independent boolean comparison," *Integration*, vol. 16, no. 2, pp. 109–129, 1993.

[24] J. Mohnke, P. Molitor, and S. Malik, "Application of BDDs in Boolean matching techniques for formal logic combinational verification," *STTT*, vol. 3, no. 2, pp. 207–216, 2001.

[25] ——, "Limits of using signatures for Permutation Independent Boolean Comparison," *Formal Methods in System Design*, vol. 21, no. 2, pp. 167–191, 2002.

[26] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[27] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, "Solving QBF with Counterexample Guided Refinement," in *SAT*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 114–128.

[28] F. Lonsing and A. Biere, "DepQBF: A Dependency-aware QBF Solver," *JSAT*, vol. 7, no. 2-3, pp. 71–76, 2010.

[29] M. Benedetti, "sKizzo: A Suite to Evaluate and Certify QBFs," in *CADE*, ser. Lecture Notes in Computer Science, R. Nieuwenhuis, Ed., vol. 3632. Springer, 2005, pp. 369–376.

[30] A. Biere, F. Lonsing, and M. Seidl, "Blocked Clause Elimination for QBF," in *CADE*, ser. Lecture Notes in Computer Science, N. Bjørner and V. Sofronie-Stokkermans, Eds., vol. 6803. Springer, 2011, pp. 101–115.

[31] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing–SAT 2007*. Springer, 2007, pp. 294–299.

Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls

Shilpi Goel Warren A. Hunt, Jr. Matt Kaufmann Soumava Ghosh
Department of Computer Science
The University of Texas at Austin

***Abstract*—We present an approach to modeling and verifying machine-code programs that exhibit non-determinism. Specifically, we add support for system calls to our formal, executable model of the user-level x86 instruction-set architecture (ISA). The resulting model, implemented in the ACL2 theorem-proving system, allows both formal analysis and efficient simulation of x86 machine-code programs; the *logical mode* characterizes an external environment to support reasoning about programs that interact with an operating system, and the *execution mode* directly queries the underlying operating system to support simulation. The execution mode of our x86 model is validated against both its logical mode and the real machine, providing test-based assurance that our model faithfully represents the semantics of an actual x86 processor. Our framework is the first that enables mechanical proofs of functional correctness of user-level x86 machine-code programs that make system calls. We demonstrate the capabilities of our model with the mechanical verification of a machine-code program, produced by the GCC compiler, that computes the number of characters, lines, and words in an input stream. Such reasoning is facilitated by our libraries of ACL2 lemmas that allow automated proofs of a program's memory-related properties.**

I. Introduction

To enable the formal verification of x86 machine-code programs, we are developing a tool suite based on our formal, executable model of the x86 instruction-set architecture (ISA). The x86 ISA has been modeled in the ACL2 programming language; we have formalized the semantics of most user-level instructions with an interpreter that can execute x86 machine-code programs. We have extended our x86 model with a formalization of an x86 system call instruction, namely, `syscall`. The execution of system calls is not provided directly by the x86 ISA; it is provided by a contemporary operating system, like Linux, FreeBSD, Windows, or MacOS, to a user process. Our extension to the x86 ISA model includes the semantics of various system calls, thereby allowing us to prove properties of user-level x86 machine-code programs that rely on an operating system for system call service.

As is the case for all other instructions specified, our extended model enables not only the formal analysis of system calls, but also supports their simulation. In fact, our extended model provides the capability to simulate and verify non-deterministic computations in general, including system calls and x86 instructions like `rdrand`. We achieve this by way of two modes in our model: the *logical mode* that supports reasoning and the *execution mode* that allows simulation.

Our evolving x86 ISA model includes specifications for 64-bit segmentation, paging, supervisor calls/returns, system registers, and many other system-level features. This model is intended to mimic the ISA-level behavior of an x86 processor; it does not currently include a specification of specialized hardware, such as the APIC and RTC.

One might wonder why we choose to formally analyze machine-code programs. In situations where source programs are unavailable, such as executables downloaded from the Web or many software distributions, we have no alternative but to analyze machine-code programs. Compilers may produce incorrect machine-code from higher-level programs, so it is important to verify the actual code that is executed on a processor. Also, programmers often optimize their high-level programs by embedding assembly code in them; the verification of such programs is impossible without the ability to analyze machine code. It quickly becomes intractable to build and maintain tools targeting various aspects of software verification; our approach provides a single, unified model that can serve multiple purposes.

Our contributions are in three areas: one, a highly-validated formal, executable model of the x86 ISA extended with system calls; two, a framework that, for the first time, provides the capability both to formally analyze and to efficiently simulate user-level x86 machine-code programs that exhibit non-determinism; and three, ACL2 libraries of lemmas that facilitate automated machine-code proofs. We present a case study to demonstrate the capabilities of our tool suite: the proof of correctness of a machine-code, word-counting program much like Linux *wc*. This case study suggests the viability of interactive theorem-proving for complex interpreter-based models with non-determinism, as in the case of our x86 model extended with system calls. All the specification and verification of programs in our tool suite is done using the ACL2 logic and its associated mechanical theorem-proving system; we know of no comparably rigorous environment for the analysis of x86 machine-code programs.

We emphasize the difference between our inference-based approach and flow-based static analysis approaches. Though flow-based approaches are being successfully used to detect vulnerabilities like buffer overflows, they can not guarantee that a given program meets its specifications. Indeed, the lack of requirement of specifications as input is considered to be the biggest strength of these approaches, thereby making them accessible to the average programmer. Our approach falls under "heavyweight" verification; given a program's specifications, our focus is on building automated tools to verify whether the program behaves as intended. Note that it is possible to *prove* the absence of vulnerabilities in our approach.

In Section II, we describe our x86 ISA model and its validation process. We discuss the extension of our model with system calls in Section III. We introduce our example program in Section IV, and present its proof of functional correctness in

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Section V. We conclude with discussions of related and future work in Sections VI and VII.

II. x86 ISA Model

Our x86 ISA model [1] implements an interpreter-style operational semantics [2]. Our x86 model's state contains registers like the general-purpose registers (`rax`, `rbx`, etc.), segment registers, flags register, model-specific registers, control registers, instruction pointer `rip`, and memory. Each machine instruction is specified by a *semantic function* that takes an x86 state and returns an appropriately modified next state. A *step* function fetches, decodes, and then executes an instruction by calling the appropriate instruction semantic function. Finally, a *run* function takes the number of instructions, `n`, to be executed and an initial x86 state, and returns a resulting x86 state; the run function either takes `n` steps or stops if an irrecoverable error is encountered, whichever comes first.

Our current modeling focus is on the 64-bit mode of Intel's IA-32e architecture (x86-64). We have a specification of all addressing modes, 121 user-level instructions (223 opcodes), IA-32e paging, and `FS`/`GS`-based segmentation. Our x86 ISA model is around 40,000 lines of code, which includes proofs about the specification, but does not include our tools for binary analysis. The model can execute most user-level integer programs emitted by the GCC/LLVM compiler — notable exceptions are media and floating-point instructions, which we plan to model in the near future.

Our model can be used in either a supervisor-level or programmer-level mode of operation. The supervisor-level mode includes support for IA-32e paging. In this mode, our memory model characterizes a 2^{52}-byte physical address space, which is the largest address space provided by modern x86 implementations. This mode can be used to simulate and verify system software. The programmer-level mode of our model attempts to provide the same environment to a programmer for reasoning as is provided by an OS for programming; it allows the verification of an application program while assuming that services like paging and I/O operations are provided reliably by the operating system. In this mode, our memory model supports the 64-bit linear addresses specified for IA-32e machines.

The simulation speed of our model in programmer-level mode is $\sim$3.3 million instructions/second and in supervisor-level mode, with a two-level page table configuration, is $\sim$920,000 instructions/second on a machine with a 3.50GHz Intel Xeon E31280 CPU. Achieving high simulation speeds facilitates the use of our formal processor model as an instruction-set simulator, which enables its validation against the real machine, as we discuss below. It is a challenge to support efficiency for both reasoning and simulation; specification functions written to maximize simulation efficiency, like those for our memory model specification [3], can be hard to reason about and those written to enable simpler reasoning can run slowly. We use abstraction techniques [4] to attain both reasoning and simulation efficiency. For the rest of this paper, we focus on the programmer-level mode of our x86 model.

ISA Model Validation: How can we trust that our model faithfully represents the x86 ISA? A benefit of using ACL2 to develop our x86 model is that its efficient executability enables validation of the model against the real machine using co-simulation. We compile high-level programs using GCC/LLVM and compare each run of a resulting x86 program on the real machine to the corresponding run on our x86 model. Our model is capable of running unmodified x86 machine-code programs because we do not simplify the semantics of x86 instructions. For example, we have successfully simulated a contemporary SAT solver on our x86 model[1]. When given an instance of the SAT'09 Competition Application benchmark (cmu-bmc-barrel6.cnf), 9,142,833,444 machine instructions are executed at run-time for the solver to run to completion. On all these instructions, our model produced exactly the same effects on the memory and registers as those produced by the real machine.

Our model validation framework uses GDB and Intel's dynamic instrumentation library, Pin [5], to extract the machine state while running programs on the processor. In the execution mode of the x86 model, the framework uses our own dynamic instrumentation library, written entirely in ACL2, to extract our model's state so that it can be compared to the real machine state at a desired level of granularity, be it on a per-instruction or a per-breakpoint basis. This framework is largely automated — it spawns off the GDB/Pin process on the real machine, uses the information captured by GDB/Pin to initialize our x86 model appropriately, runs the model in its execution mode on concrete data, and produces a report containing the differences observed, if any, between the real machine state and the model's state. This automated and easy-to-use framework makes it convenient to run many co-simulations, thereby facilitating fast and thorough model validation.

We have invested several person years of work in our x86 model. We use the Intel manuals [6] as specification documents; ambiguities are resolved by running tests on the real processor and by consulting with processor architects. Our model is a formal specification for the x86 ISA, and it can also serve as the target specification for RTL design verification.

III. System Call Model

User-level programs, either directly or through higher-level interfaces provided in libraries, often make system calls to the underlying operating system to request services such as file I/O and memory management. Though the x86-64 architecture provides other instructions to invoke and return from system calls, we focus on `syscall` and `sysret`; these instructions are the most common and efficient interface between the kernel and a user application. The `syscall` instruction is used by user-level code to call system-level procedures at the highest privilege level by loading the `rip` with the appropriate address from a model-specific register. The companion instruction, `sysret`, returns control from the system procedures to user-level code at the application privilege level. These instructions allow fast privilege-level transitions during the system call invocation and return process by keeping all the information required for the transition in general-purpose and model-specific registers, thereby avoiding the overhead of table references in memory.

[1]This SAT solver was developed by Marijn J. H. Heule; its performance is comparable to those of state-of-the-art solvers.

From the perspective of a user-level program, system calls are non-deterministic — different runs can yield different results on the same machine. Since our x86 model serves both as an executable instruction-set simulator and a formal specification that is used to do proofs about machine code, we need to be able to do the following:

1) Efficiently simulate runs of a program with system calls on concrete data, and
2) Formally reason about such a program given symbolic data.

Ideally, to accomplish both these tasks, modeling enough features of the x86 would allow an operating system to be loaded on the model to service system calls. Consequently, we could both simulate and reason about system calls due to the executable and formal nature of our ACL2-based model. However, loading a modern OS on a processor model is non-trivial; the added complexity of the low-level interaction of the OS with the processor would not only make reasoning about user-level programs harder, but also slow down the simulation speed of concrete program runs.

Instead, for simulation of system calls, we set up the *execution mode* of our x86 model to interact directly with the underlying OS. ACL2 provides a mechanism [7] for allowing arbitrary Common Lisp code to be defined in *raw Lisp*, outside ACL2. The system call service is provided by raw Lisp functions to obtain "real" results from the OS [8]. Simulation of all instructions other than `syscall` happens within ACL2 (and hence, Lisp). Note that since we are abstracting away the system-level procedures that are invoked by the OS when a system call is made, we do not need to make a similar arrangement for the `sysret` instruction.

These raw Lisp functions should not be used for reasoning since they are *impure*: they are not axiomatized logically, and indeed, are not even functions in the logical sense since repeating the same call can yield different results. It is critical for our framework to prohibit proofs of theorems that state that some system call returns a specific value. If that were the case, then due to the non-determinism inherent in system calls, we might be able to prove that the same system call returns some other value in a different ACL2 session. Or perhaps worse yet, we could prove an instance of $x \neq x$ by instantiating x with a term that invokes the system call. Another disturbing scenario would be when such theorems contradict results observed in a program run simulation.

Thus, for reasoning about machine-code programs we use the *logical mode* of our model, which incorporates into the state an *environment* field to represent the part of the external world that affects or is affected by system calls. To reason about a system call's effects, we simply consult that `env` field. A well-formed `env` field contains sub-fields that describe a subset of the file system and an *oracle* that provides information that, though a part of the real environment, cannot be inferred from our model of the file system. An example of such information is the file descriptor of a file to be opened; an OS assigns the file descriptor depending on the number of files already opened for a particular process at the time the open system call is made.

The contents of `env` can be *abstract*. For example, to verify a program like `grep` that searches for occurrences of a pattern in an input file, the pattern can be specified as an arbitrary string and the file can be specified as an arbitrary file in the file system (or not, if we wish to reason about the case when the file does not exist). This ability to reason about arbitrary elements in the environment is precisely what makes reasoning about non-determinism possible. Of course, it is also possible to reason about specific elements in the environment, e.g., `grep` with a specific pattern on a specific file, by simply initializing the `env` field with these elements.

Consider two runs of our model with the same initial x86 state, where one is in execution mode with real environment `ENV` and the other is in logical mode with environment field `env`. We say that `env` *corresponds* to environment `ENV` if the execution of system calls produces the same results in the logical mode as in the execution mode.

The execution mode does not unduly impact the logical mode, since the raw Lisp functions do not influence the reasoning process. Conversely, the `env` field does not interfere with the impure functions in the execution mode. However, the logical and execution modes are far from completely independent, as noted by the following three properties.

(L) For reasoning, all the functions in the logical mode of the x86 model are pure.

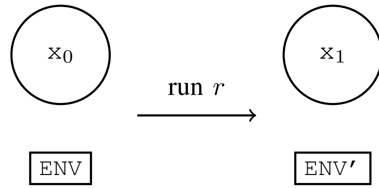
(E) The execution mode allows the use of raw Lisp functions that directly interact with the underlying OS to provide system call service. Note that the logical mode and execution mode are identical for all instructions except `syscall` — all other instructions have the same definitions in both these modes.

(C) The following connection exists between the logical mode and the execution mode. Let x_0 be an x86 state. Suppose in the execution mode, the evaluation of (run x_0) returns x_1 and updates the real environment from `ENV` to `ENV'`. Then, the following is true for the logical mode: if `env` corresponds to `ENV`, and x_0' refers to x_0 augmented with `env`, then the evaluation of (run x_0') in the logical mode produces x_1 augmented with `env'`, for some `env'` corresponding to `ENV'`.

See Figure 1 for an illustration of a program run in both the execution and logical modes. We discuss property (C) in some detail later in this section. Due to property (C), we know that evaluation results produced by raw Lisp functions will not be contradicted by theorems proved about system calls; in fact, each program run in the execution mode produces a theorem under a hypothesis about the well-formedness of the environment in the logical mode. Thus, observations made while performing simulation in the execution mode hold in the logical mode as well. Our method facilitates the maintenance of an integrated software base for the logical and execution modes of the x86 model.

Our framework makes reasoning about non-deterministic computations in programs tractable. As we will see in Section V, the proof of correctness of a program is not complicated by the presence of system calls. We have used this approach to model and implement the following system calls: `read`, `write`, `open`, `close`, `lseek`, `dup`, `link`, and `unlink`. We support the simulation of these system calls on both

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.



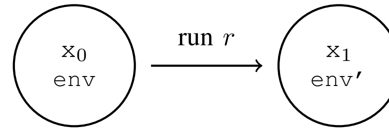


Figure 1. Illustration of a run in the execution mode (left) and in the logical mode (right). A run, r, from an initial state $\mathtt{x}_0$ in the execution mode gives a final state $\mathtt{x}_1$, and the environment ENV on the real machine transitions to ENV′. In the logical mode, env corresponds to ENV, and r produces the same final state, $\mathtt{x}_1$, augmented with env′, which corresponds to ENV′.

Linux and Darwin systems. Our approach can be used to handle various sources of non-determinism, other than just system calls, that arise in user-level programs. One such example is the rdrand instruction, which is used to provide cryptographically secure random numbers to applications.

System Call Model Validation: For all instructions but syscall, comparing the real machine state to the model state extracted in the execution mode validates the logical mode as well since these modes are identical. However, for the syscall instruction, the execution and logical modes of the x86 model consist of different functions, and are thus distinct. Consequently, two validation tasks need to be performed for the syscall instruction:

1. Validate the execution mode of the x86 model against the real system, i.e., processor plus system call service provided by the operating system, and
2. Validate the execution mode against the logical mode of the x86 model.

We accomplish the first validation task using our model validation framework, as discussed in Section II. Since the raw Lisp functions supporting the execution mode of the syscall instruction interact with the underlying OS and hence, pass on the results of the real machine to our framework, the only functions of the execution mode that need to be validated are those that marshal the input arguments and return values of these raw Lisp functions, and those that capture the effects of a return from the system call. The latter accounts for the effects of the sysret instruction as well; for example, sysret always clears the RF and VM flags, and the programmer's view of the processor after a return from a system call should also depict these flags as cleared.

The second validation task is critical to ensure the property (C) stated earlier. The logical mode for syscall can be thought of as the specification for its execution mode. The specification functions supporting the logical mode are written in accordance with the man pages of the system calls and their more detailed descriptions found elsewhere [9]. We validate the execution mode against the logical mode by performing extensive code reviews, and by comparing program runs in the execution mode to corresponding runs in the logical mode. We illustrate this process by a short example. Consider the following five x86 instructions. This snippet of an assembly program makes a read system call to obtain one byte from a file with descriptor equal to 0, usually the standard input. The arguments needed by the read system call are loaded into appropriate registers, as dictated by the x86-64 Application Binary Interface [10]. The rax register contains the Linux read system call number, the rdi register contains a file descriptor, the rsi register contains the address of the memory buffer where the read bytes will be written, and the rdx register contains the number of bytes to be read.

```
mov $0x0,%rax         /* Syscall number   */
xor %rdi,%rdi         /* File descriptor  */
mov -0x20(%rbp),%rsi  /* Buffer address   */
mov $0x1,%rdx         /* Number of bytes  */
syscall
```

In the execution mode, we initialize our x86 model to reflect the state of the real machine when rip points to the address of the first mov instruction. We set up the model to make five steps, i.e., run this snippet. Then, the raw Lisp function for the read system call collects the user's input.

In the logical mode, we initialize the environment field env so that it corresponds to the real environment. As such, the contents of the standard input in the env field should contain the user input that was collected in the corresponding run in the execution mode. After setting up the rest of the fields of the x86 state to be exactly the same as those of the initial state in the execution mode, we run the model to simulate these five instructions. A comparison of the final state obtained in the logical mode and execution mode allows validation of these modes against each other.

IV. Program: Simple Word Count

We analyze the machine code corresponding to a simple word count program taken from "The C Programming Language" by Kernighan and Ritchie [11]. This C program is a bare-bones version of the wc program found on Linux systems. We use this program as a case study to assess the capability of our model to simulate and reason efficiently about programs that make system calls. GCC compilation generated 50 machine instructions (166 bytes) — 17 instructions for the gc procedure, including the syscall instruction, and 36 instructions for the main sub-routine.

The program reads a character from the standard input until the end of input (which is denoted by the character #), each time incrementing the character counter nc. If the character is a newline, then the newline counter nl is also incremented. The word counter nw is incremented at the beginning of every word, i.e., when state transitions from OUT to IN.

```
#define IN   1    /* inside a word  */
#define OUT  0    /* outside a word */
```

```
#define EOF '#'   /* EOF character  */
#include <stdio.h>
int gc(void) {
  char buf[1];
  int n;
  __asm__ volatile
    (
     "mov $0x0, %%rax\n\t"
     "xor %%rdi, %%rdi\n\t"
     "mov %1, %%rsi\n\t"
     "mov $0x1, %%rdx\n\t"
     "syscall"
     : "=a"(n)
     : "g"(buf)
     : "%rdi", "%rsi", "%rdx");
  return (unsigned char) buf[0];
}
/* count lines, words, characters in input */
int main () {
  int c, nl, nw, nc, state;
  state = OUT;
  nl = nw = nc = 0;
  while ((c = gc()) != EOF) {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }
  return 0;
}
```

The original program from Kernighan and Ritchie's book used the C standard library (*glibc*) function `getchar` instead of our function `gc`. The machine code corresponding to `getchar` used SIMD (AVX) instructions in some places to speed up execution. Since we do not yet support SIMD instructions in our model, we chose to write `gc` as our own version of `getchar`. The function `gc` can be thought of as an inefficient, unbuffered `getchar`. Every call of `gc` attempts to read one byte from the standard input and stores it in a memory buffer. An alternative to defining `gc` could be to use a portable and lightweight standard library like *newlib* [12] instead of *glibc*.

Before reasoning about the entire program in the logical mode of our model, we ran simulations in the execution mode. The program behaved as expected on our model, thereby providing confidence that our model faithfully emulates a real x86 system for the instructions of this program.

V. Functional Correctness of Simple Word Count Machine-Code Program

In this section, we discuss the verification of the machine-code program produced by running the GCC compiler on our example program. This machine-code program is structurally quite similar to its C source; in particular, it has a loop that begins with a call to the `gc` sub-routine, which makes a system call. The program variables `nc`, `nl`, `nw`, and `state` are allocated on the stack in consecutive memory locations.

We apply a traditional theorem-proving approach to program verification, since our previous automatic approach using bit-blasting [13] is limited in its handling of loops and large programs. We formally analyze this program using the Boyer-Moore clock function method [14], [15]. We briefly describe this method here. Given a clock function *clock* that specifies the number of steps needed for a program to run to completion, the following theorem states the total correctness of a program: if *x* is an x86 state satisfying specified pre-conditions, then the final state *run(clock(x),x)* satisfies the specified post-conditions. It is the user's responsibility to write these clock functions; there is ongoing research to automate this task in ACL2 [16], comparable to previous work for HOL4 [17], [18].

$$\forall x : \mathit{pre\text{-}conditions}(x) \implies \mathit{final\text{-}state}(\mathit{run}(\mathit{clock}(x), x)) \wedge \mathit{post\text{-}conditions}(x, \mathit{run}(\mathit{clock}(x), x))$$

How can we state functional correctness for our program? We choose to write a trio of simple ACL2 specification functions that compute the character, line, and word counts of a string, respectively. Our post-condition asserts that the values returned by these three specification functions on standard input are found in the expected memory locations of the final x86 state, which is obtained by running the program on our x86 model in its logical mode.

We now outline the proof of functional correctness of the simple word count machine-code program. The program structure can be used as a guide to decompose the proof into two sub-tasks — one, the verification of the initial part of the program when all counters are initialized to 0, and two, the verification of the loop, which begins with a call to the `gc` function. We use the theorems stating correctness of these program components to obtain the final correctness theorem.

We begin by stating the assumptions made about the `env` field in the x86 state, in order to reason about the system call that performs a read in the `gc` function.

1) The file descriptor corresponding to the standard input is `0`. Note that we make this assumption only because the program itself makes this assumption.
2) The contents of the standard input should be terminated by the end-of-file character (# for this program), and thus, be non-empty. We make this assumption because the program does not terminate unless this end-of-file character is encountered.

The read system call has the following interface:

```
ssize_t read (int fd, void *buf, size_t count);
```

This system call tries to read `count` bytes from the file pointed to by the file descriptor `fd` into the memory buffer beginning at `buf` [19]. The read made in the `gc` function of the word-count program has `fd` referencing the standard input, `buf` pointing to a stack address, and `count` equal to one. The specification of the read system call in the logical mode of our x86 model only tells us that one byte is read from the standard input, modeled by the `env` field, and written to the memory buffer unless some error is encountered. We can not deduce the value of this byte. This permits us to reason about our program for all possible bytes that can be returned by one call of `gc`. Various errors, like `buf` pointing to an illegal memory address, are also accounted for by our system call specification.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Let us first focus on the loop. The loop pre-conditions *loop-pre* are as follows.

1) The x86 state is well-formed.
2) The environment assumptions hold for this x86 state.
3) The program is loaded in the memory at its expected location.
4) The instruction pointer, `rip`, points to the first instruction of the loop.
5) The stack pointer, `rsp`, is within a specified range. This guarantees that the stack does not over-write the code during the program's execution.

Pseudo-code for the ACL2 function `loopClk` is shown below. This function is the loop's clock function, which computes the number of steps needed for the loop to complete. The argument `state` of `loopClk` corresponds to the `state` variable of the simple word count program, `offset` corresponds to the position of the next character to be read from standard input, and `strBytes` corresponds to the contents of standard input in bytes. Constants like `cEOF`, `cNL`, `cSpace`, `cTab`, `cOut`, and `cIn` denote the number of instructions that are executed during run-time in one loop iteration, according to which branch of the loop is taken. Thus, the function `loopClk` keeps recurring till `EOF` is encountered; for each recursive call, it adds the number of instructions to be executed at run-time based on the character read.

```
loopClk(state,offset,strBytes):

if !(envAssumptions(offset,strBytes)) then
 // No instructions are run when environment
 // assumptions fail.
 0
else {
 // gcSpec is gc's specification function.
 char = gcSpec(offset,strBytes)
 if (char == EOF) then
   cEOF
 else {
   case (char) {
     newline   :  state = OUT
                  loopSteps = cNL
     space     :  state = OUT
                  loopSteps = cSpace
     tab       :  state = OUT
                  loopSteps = cTab
     otherwise :  if (state == OUT) then
                    state = IN
                    loopSteps = cOut
                   else
                    loopSteps = cIn }
   return(loopSteps +
          loopClk(state,(1+ offset),strBytes))
 } }
```

Given these pre-conditions and loop clock function `loopClk`, the loop correctness theorem is as follows, where we write *l* to abbreviate the application of `loopClk` to the appropriate values stored in the x86 state, *x*.

Theorem 1: $\forall x : \textit{loop-pre}(x) \implies \textit{halted}(\textit{run}(l, x)) \wedge \textit{post}(x, \textit{run}(l, x))$

where *x* is an x86 state that satisfies the loop pre-conditions *loop-pre*, *post* relates the trio of our specification functions to the values in the expected memory locations of the counters in the halted state *run(l,x)*, and *l* specifies the number of steps the entire loop takes to reach the final state, i.e., *l* is a value computed by `loopClk`.

Proof: This theorem can be proved by strong induction on the value *l* of `loopClk`. If *l* is 0, then `envAssumptions` is false; thus *loop-pre(x)* does not hold, which proves the base case. Otherwise the proof splits into cases according to the character read. Let us address the case that this character is a newline, as the other cases are analogous. By the inductive hypothesis, we may assume the following, which is obtained from the theorem by replacing *x* with *run(*`cNL`*, x)* and *l* with $(l - \texttt{cNL})$, and noting that $(l - \texttt{cNL})$ is the the application of `loopClk` to the appropriate values stored in the x86 state, *run(*`cNL`*, x)*.

$$\textit{loop-pre}(\textit{run}(\texttt{cNL}, x)) \implies \textit{halted}(\textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x))) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x))) \quad (1)$$

The following fact is easy to prove by definition of the *run* function.

$$\textit{run}(l, x) = \textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x)) \quad (2)$$

By substituting 2 into 1, we obtain:

$$\textit{loop-pre}(\textit{run}(\texttt{cNL}, x)) \implies \textit{halted}(\textit{run}(l, x)) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}(l, x)) \quad (3)$$

The proof of Theorem 1 follows from the induction hypothesis if we prove the following lemmas:

L1: $\forall x : \textit{loop-pre}(x) \implies \textit{loop-pre}(\textit{run}(\texttt{cNL}, x))$
L2: $\forall x : \textit{loop-pre}(x) \implies \textit{post}(x, \textit{run}(\texttt{cNL}, x))$
L3: $\forall x : \textit{post}(x, \textit{run}(\texttt{cNL}, x)) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}(l, x)) \implies \textit{post}(x, \textit{run}(l, x))$

The proof of L1 is conceptually simple, and we lead ACL2 to simplify expressions representing the values of components of the state *run(*`cNL`*,x)* that are relevant to *loop-pre*, such as its `rip`. The proof of L2 proceeds in the same manner. Given our description of *post* as relating stack values for the counters with our specification functions, the proof of L3 follows from the transitivity of *post*. ■

We then prove the following theorem about the initial part of the program, i.e., the part preceding the loop. Here, *pre* has a form similar to *loop-pre* but with obvious differences, for example: the `rip` points to the first instruction of the program instead of to the first instruction of the loop, and the constant `i` is the number of instructions required to take the program to the beginning of the loop.

Theorem 2: $\forall x : \textit{pre}(x) \implies \textit{loop-pre}(\textit{run}(i, x))$

The proof is similar to the proof of L1. We finally prove total correctness for this program by using Theorems 1 and 2. Here, *c* is the value computed by the clock function for the entire program on initial state *x*, that is, $c = i + l$.

Theorem 3: $\forall x : \textit{pre}(x) \implies \textit{halted}(\textit{run}(c, x)) \wedge \textit{post}(x, \textit{run}(c, x))$

Though the proof of correctness of the word-count program is straightforward, it is worth emphasizing that it was done on a large interpreter-based model of the x86, where the semantic functions of instructions are, on an average, ~200 lines of ACL2. This proof makes heavy use of compositional reasoning and would have been harder to do had we not developed our own libraries to automate reasoning about reads and writes made to the x86 state. We proved many lemmas about registers, flags, etc. Here we briefly discuss one such library that facilitates reasoning about memory accesses and updates.

Reasoning about memory usage is challenging, simply because memory is so large. Moreover, code and data share the memory, which requires establishing that each write to the stack or heap during the program's execution does not overwrite program and data. Verifying position-independent code entails reasoning about disjointness of memory regions that are specified by symbolic or computed memory addresses. As such, a lot of tedious low-level arithmetic reasoning about inequalities and equalities involving these symbolic addresses is required. Our library lifts this problem to reasoning about membership of addresses in lists instead, and list-based reasoning is done largely automatically. An example is the automated proof of the disjointness of the program and the stack done in our word-count case study. Another proof that was discharged automatically was that the word-count program does not modify unintended regions of memory, i.e., the only writes that occur during the program's execution are to the stack, and the rest of the memory is the same as it was before the execution. This is an important theorem because it rules out one kind of potential "evilness" of our word-count program. Other kinds of memory guarantees, like ruling out stack smashing and buffer overflows, can also be established using our library.

In order to facilitate re-use, our proof libraries are designed to be as general as possible. As we verify progressively more complicated programs, we discover new lemmas and extend these libraries. Below is some empirical evidence that illustrates how our libraries can reduce manual effort:

Lines of ACL2 needed to verify the word count program:

- Without the libraries: ∼20K
- With the libraries: ∼8K

∼8K lines of ACL2 might still seem excessive. However, at least half of these lines were generated by ACL2 in response to commands to simplify specific symbolic expressions. The simplified expressions are large because there are many updates to different components of the x86 state to symbolically run even a small program.

VI. Related Work

Machine-code verification has long been an area of active research. As such, many formal models of contemporary processor ISAs have been developed to enable reasoning about machine-code [20]–[23]. Our strategy of modeling an external environment to account for non-determinism in programs is similar to Moore's work [24] in ACL2 to model non-determinism in a pedagogical multiprocessor system. There has been considerable research on the verification of system calls from a micro-kernel point of view [25], [26]. In this paper, we concern ourselves with reasoning about user-level x86 programs that interact with a contemporary operating system. Here, we mention some recent work with goals similar to ours.

Morrisett et al., while working on software fault isolation [27], developed an x86 ISA specification in the Coq [28] proof assistant that can also be used for machine-code verification. Morrisett's x86 specification is not directly executable; an executable OCaml simulator has to be extracted from the x86 specifications in Coq. The resulting simulator has an execution rate of ∼50 instructions/second; it simulates ∼10 million instructions in 60 hours on an 2.6 GHz, 8 core Intel Xeon machine. This work is concerned with restricting certain kinds of computations that can be performed natively on the host machine in order to avoid information leaks to a web browser. It is not designed to handle the verification of general user-level programs that employ system calls. Feng et al. [29] use the Coq proof assistant [28] to prove the functional correctness of machine-code on a formal model of a processor that can handle asynchronous events like signals and interrupts. However, this processor model is a simplified version of the x86, and does not handle 64-bit x86 machine-code programs. Dowse et al. [30] used the Sparkle proof assistant [31] to verify programs that perform I/O. This verification effort is targeted at higher-level programs, specifically lazy functional programs. Malecha et al. [32] also verified high-level Coq programs that perform I/O. Reps et al. [33] have developed a sophisticated system called TSL, that can create retargetable tools for different types of machine code analyses, especially data-flow analyses. We do not know of a TSL-created tool that can prove whether a given machine code program meets its specification.

VII. Conclusion and Future Work

We mechanically verify user-level x86 machine-code programs with our ACL2-based ISA model extended with a specification of system calls. Our effort is the first mechanical verification of a user-level x86 machine-code program that includes the use of system calls.

Our extended model has two modes: (a) a logical mode that formally axiomatizes an external environment to enable reasoning about programs that include instruction-based non-determinism and that make system calls, and (b) an execution mode that supports program simulation by interacting with the underlying OS to produce results just as if executing a user-level machine-code program natively on an x86 processor with contemporary OS support. We regularly validate the accuracy of our x86 model using co-simulation, having already done so for many billions of instructions.

Our approach avoids any special treatment for system calls when proving the functional correctness of a user program. More generally, our framework makes formal analysis of non-determinism in programs tractable. This effort has led to the development of ACL2 libraries that automate machine-code verification, in particular for reasoning about memory reads and writes. Automating such tedious reasoning activities considerably speeds up the proof development process.

Our case study of the verification of the word count program provides compelling evidence that there is much more potential for automating x86 machine-code proofs in our framework. The proof for this program was tedious; similar kinds of theorems were needed to reason about different parts

of the program. However, these proofs were already largely automated due to the support provided by our libraries. We continue to develop tools to support automation in order to make machine-code verification in our framework accessible to those unfamiliar with formal verification of programs on interpreter-based models.

We should note that our model of the file system does not account for concurrent updates by external processes. Our verification work assumes that the input being processed will not be changed during the execution of our program; thus, our specification states the behavior of our programs in the absence of such concurrent updates. Exploring program correctness in view of possible interference by other programs would require, at the very least, a more subtle model of the environment being provided for our verification effort.

We believe that our specification of the x86 ISA, coupled with the ACL2 system, can facilitate regular verification of x86 machine-code programs. To realize this goal, we would begin by verifying various programs in standard libraries; then, we would verify programs that make use of these standard libraries. Such compositional methods can provide a scalable way to prove the functional correctness of machine-code programs.

Acknowledgment

We thank Marijn J. H. Heule for his invaluable feedback on the paper. This work is supported by DARPA under contract number N66001-10-2-4087.

References

[1] Matt Kaufmann and Warren A. Hunt, Jr., "Towards a formal model of the x86 ISA," Department of Computer Science, University of Texas at Austin, Tech. Rep. TR-12-07, May 2012.

[2] J S. Moore, "Mechanized operational semantics," Lectures in the Marktoberdorf Summer School (August 5-16, 2008)., Online; accessed: January 2014. http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html.

[3] Warren A. Hunt, Jr. and Matt Kaufmann, "A formal model of a large memory that supports efficient execution," in *Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012, Cambrige, UK, October 22-25)*.

[4] Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann, "Abstract stobjs and their application to ISA modeling," in *Proceedings ACL2 2013, EPTCS 114, 2013, pp. 54-69*.

[5] Intel, "Pin: A Dynamic Binary Instrumentation Tool," http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[6] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manuals." Order Number: 325462-048US. (September 2013). http://download.intel.com/products/processor/manual/325462.pdf., online; accessed: January 2014.

[7] Matt Kaufmann, J S. Moore, Sandip Ray, and E. Reeber, "Integrating external deduction tools with ACL2," *Journal of Applied Logic*, vol. 7, no. 1, pp. 3–25, Mar. 2009.

[8] CCL, "CCL Manual: Foreign Function Interface," http://ccl.clozure.com/manual/chapter13.html., clozure Common Lisp Manual. Online; accessed: January 2014.

[9] Michael Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.

[10] Michael Matz, Jan Hubiĉka, Andreas Jaeger, and Mark Mitchell, "System V Application Binary Interface: AMD64 Architecture Processor Supplement," http://www.x86-64.org/documentation/abi.pdf, online; accessed: January 2014.

[11] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd ed. Prentice-Hall, 1988.

[12] Newlib, "Newlib C Library," https://sourceware.org/newlib/., online; accessed: January 2014.

[13] Shilpi Goel and Warren A. Hunt, Jr., "Automated code proofs on a formal model of the X86," in *Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, Ernie Cohen and Andrey Rybalchenko, Ed., vol. 8164. Springer Berlin Heidelberg, 2014, pp. 222–241. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54108-7_12

[14] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young, "Special Issue on System Verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 409–530, 1989.

[15] Sandip Ray, Warren A. Hunt, Jr., John Matthews, and J S. Moore, "A mechanical analysis of program verification strategies," *Journal of Automated Reasoning*, vol. 40, no. 4, pp. 245–269, May 2008.

[16] J S. Moore, "Code Walker Tool," (presented as a Rump Session Talk at the ACL2 Workshop, 2013, Laramie, Wyoming).

[17] Magnus O. Myreen, *Formal Verification of Machine-code Programs*. British Computer Society., 2008.

[18] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind, "Decompilation into logic - improved," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*, 2012, pp. 78–81.

[19] Linux, "read(2) - Linux manual page," Retrieved from: http://man7.org/linux/man-pages/man2/read.2.html., online; accessed: January 2014.

[20] Warren A. Hunt, Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, 1989.

[21] J S. Moore, *Piton: A Mechanically Verified Assembly-level Language*. Kluwer Academic Publishers, 1996.

[22] Anthony Fox, "Directions in ISA specification," *Interactive Theorem Proving (ITP)*, pp. 338–344, 2012.

[23] Ulan Degenbaev, "Formal Specification of the x86 Instruction Set Architecture," 2012.

[24] J S. Moore, "A mechanically checked proof of a multiprocessor result via a uniprocessor view," *Formal Methods in System Design*, vol. 14, no. 2, pp. 213–228, 1999.

[25] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer, "Formal verification of a microkernel used in dependable software systems," in *Computer Safety, Reliability, and Security*. Springer, 2009, pp. 187–200.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and Others, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[27] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan, "Rocksalt: Better, faster, stronger SFI for the x86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111

[28] Coq, "Coq proof assistant," http://coq.inria.fr/.

[29] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong, "Certifying low-level programs with hardware interrupts and preemptive threads," *Journal of Automated Reasoning*, vol. 42, no. 2, pp. 301–347, 2009.

[30] Malcolm Dowse, Andrew Butterfield, Marko van Eekelen, and Maarten de Mol, "Towards machine-verified proofs for I/O," *Technical Report 0408 in the Proceedings of Implementation and Application of Functional Languages, 16th International Workshop, IFL'04, Lübeck, Germany.*, pp. 469–480., September 8-10, 2004.

[31] Maarten De Mol, Marko Van Eekelen, and Rinus Plasmeijer, "The mathematical foundation of the proof assistant Sparkle," 2007, Technical Report ICIS-R07025, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands.

[32] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky, "Trace-based verification of imperative programs with I/O," *Journal of Symbolic Computation,*, vol. 46, no. 2, pp. 95–118., 2011.

[33] J. Lim and T. Reps, "Tsl: A system for generating abstract interpreters and its application to machine-code analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 1, p. 4, 2013.

DRUPing for Interpolants

Arie Gurfinkel
Carnegie Mellon Software Engineering Institute
http://arieg.bitbucket.org

Yakir Vizel
Electrical Engineering Department, Princeton University
Computer Science Department, The Technion
http://www.cs.technion.ac.il/~yvizel

***Abstract*—We present a method for interpolation based on DRUP proofs. Interpolants are widely used in model checking, synthesis and other applications. Most interpolation algorithms rely on a resolution proof produced by a SAT-solver for unsatisfaible formulas. The proof is traversed and translated into an interpolant by replacing resolution steps with AND and OR gates. This process is efficient (once there is a proof) and generates interpolants that are linear in the size of the proof. In this paper, we address three known weakness of this approach: (i) performance degradation experienced by the SAT-solver and the extra memory requirements needed when logging a resolution proof; (ii) the proof generated by the solver is not necessarily the "best" proof for interpolantion, and (iii) combining proof logging with pre-processing is complicated. We show that these issues can be remedied by using DRUP proofs. First, we show how to produce an interpolant from a DRUP proof, even when pre-processing is enabled. Second, we give a novel interpolation algorithm that produces interpolants partially in CNF. Third, we show how DRUP proof can be restructured on-the-fly to yield better interpolants. We implemented our DRUP-based interpolation framework in MiniSAT, and evaluated its affect using AVY — a SAT-based model checking algorithm.**

I. INTRODUCTION

SAT-based Model-Checking, i.e., reducing Model Checking to one or several instances of Boolean satisfiability (SAT), has emerged as the most effective approach for scaling model checking to industrial designs. Bounded Model Checking (BMC) [1] is reduced to a single satisfiabilty problem that checks for existence of a counterexample of a given length. Safety verification (or Unbounded Model Checking) is reduced to an iterative process by repeatedly: (a) solving BMC problems with increasing bound, (b) constructing a proof π of bounded safety, and (c) attempting to generalize π to an inductive invariant. The bounded safety proof π is extracted from the resolution refutation proof of unsatisfiability of a BMC instance by the process of *Craig interpolation*. Thus, safety verification requires that a SAT-solver can produce interpolants in addition to deciding satisfiability.

Formally, given an UNSAT formula $G \equiv A \wedge B$ partitioned into A and B, a Craig interpolant is a formula I such that A implies I, I is inconsistent with B, and I is defined over the variables common to A and B. In model checking, G is a BMC instance, A is some prefix that contains the initial condition, and B a suffix that contains the bad states [2]. Thus, the interpolant I is an over-approximation of the set of states reachable by the prefix A that does not contain any bad states. It is convenient to generalize Craig interpolants to a sequence. In this case, $G \equiv G_1 \wedge \cdots \wedge G_N$ is partitioned into N parts, and an interpolant is a sequence $I_1, \ldots, I_{N-1}$ such that I_i is a Craig interpolant between $G_1 \wedge \cdots \wedge G_i$ and $G_{i+1} \wedge \cdots \wedge G_N$. That is, I_i over-approximates the set of states reachable after i steps. The sequence corresponds to an inductive invariant if for some i, I_i implies $\bigvee_{1 \leq j < i} I_j$. Most SAT-based model checking algorithms (e.g., [2]–[7]) are based in some way on sequence interpolants, although, they vary widely in interpolant computation and in many additional details.

Interpolants can be extracted directly from a resolution proof of unsatisfiability. There are several such *proof-based* procedures that convert a resolution refutation into a circuit by replacing resolution steps by AND and OR gates [2], [8], [9]. They are simple to implement and produce interpolants that are linear in the size of the proof. Their variations (for strength [9], [10], structure [11]–[13], and size [13]) and model checking specific properties (e.g., [9], [14]) are widely studied. However, they require a SAT-solver to log the resolution proof. While this is not technically difficult [15], it significantly increases the memory usage of the solver [16]. Furthermore, it appears that combining proof-logging and common pre-processing is difficult. Most solvers (e.g., [17]) treat proof-logging and pre-processing as mutually exclusive.

Alternatively, interpolants can be constructed by partitioning the clauses of $G \equiv A \wedge B$ into two groups and restricting the SAT-solver to work with either A or B clauses, but not with both at the same time. The solver is allowed to communicate implicants of B to A, and consequences of A to B. The interpolant is the set of all communicated A-consequences. This *proof-less* approach was pioneered by IC3 [5] (together with many other improvements), has been applied for interpolation by Chockler et al. [18], and has been further refined by Bayless et al. [19] by allowing additional communication between partitions. Such algorithms compute interpolants in CNF (which is often desired) and do not need proof-logging. However, partitioning the clauses and restricting the solver significantly degrades performance. This is less of an issue when these techniques are a part of a tightly integrated verification loop, as in IC3. Finally, partitioning negatively affects pre-processing.

Goldberg and Novikov [20] suggest a low-overhead proof logging technique by showing that the sequence of all learnt clauses, in the order learnt by a CDCL SAT-solver – *the clausal proof* – is both easy to log and sufficient to reconstruct the complete resolution proof. Recently, Heule et al. [16] introduced a *trimmed* variant, called *DRUP-proofs*, that additionally account for the clauses deleted by the solver. They show that DRUP-proofs can be expanded (or validated) into a resolution proof efficiently, and suggest them for solver certification, UNSAT core extraction, and interpolation.

In this paper, we propose a novel interpolation algorithm

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001563.

based on DRUP-proofs. We are motivated by the fact that logging DRUP-proofs is easy even in the presence of pre-processing. The naïve approach is to expand a DRUP-proof into a resolution proof (e.g., [15]) and apply existing interpolation techniques. While this is reasonable, we take a different, more flexible, approach.

Our contributions are as follows. We present a framework for computing sequence interpolants from DRUP-proofs implemented on top of MiniSAT. The approach consists of two phases. The first traverses the DRUP-proof backward, trimming it, and identifying the core. Unlike [16], our traversal is geared towards interpolation and not proof minimization. The second traverses the trimmed proof forward constructing an interpolant on-the-fly. During this phase, local transformations are applied to the proof to guide it to a better interpolant. It is important to note that our framework is focused on interpolation and not solver certification. Hence, it is made efficient through reuse of many of the solver's data-structures and procedures, and through reuse of the final state of the *trail* when the final conflict is reached. Note that while it seems theoretically trivial to expand a DRUP-proof into a resolution proof, such expansion may take as much time as solving the original SAT instance [16]. Thus, our careful implementation and reuse of the solver's final state is beneficial.

Furthermore, we present a novel interpolation algorithm that computes an interpolant as a pair of formulas $p \wedge g$ such that g is in CNF. In some cases this results in a pure CNF interpolant. To our knowledge, this is unique. Finally, our local proof restructuring, mentioned above, aims at maximizing the CNF component of the interpolant. This restructuring procedure is possible partially due to the flexibility our framework enables when constructing an interpolant.

We evaluated our framework in the context of model checking using AVY [7], a SAT-based model checking algorithm that heavily relies on sequence interpolants. We show the effect our DRUP-based interpolation framework has on AVY's performance when compared to a proof-logging SAT-solver. In addition, we evaluate our different heuristics and show their effect on the computed interpolants. Our experiments show that DRUP-based interpolation is efficient and improves the underlying model checking algorithm. In addition, our new interpolation technique, together with our local proof restructuring result in a significant number of clauses in the CNF component of the computed interpolants.

II. PRELIMINARIES

Given a set U of Boolean variables, a *literal* ℓ is a variable $u \in U$ or its negation $\neg u$, a *clause* is a disjunction of literals, and a formula in *Conjunctive Normal Form*, or a CNF for short, is a conjunction of clauses. It is convenient to treat a clause as a set of literals, and a CNF as a set of clauses. We write $\Box$ to denote the empty clause, $Var(\alpha)$ for variables of a clause α, and $Var(G)$ for variables of a set of clauses G.

The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are clauses and v and $\neg v$ are literals, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. Application of the resolution rule is denoted by $\alpha_1, \alpha_2 \vdash^{v}_{\text{RES}} \alpha_3$, and v is called the *pivot* variable. We omit v when it is clear from the context or irrelevant.

A *resolution derivation* of a clause α from a CNF formula G is a sequence $\pi = (\alpha_1, \alpha_2, \ldots, \alpha_n \equiv \alpha)$, where each clause α_k is either an *initial* clause of G or is *derived* by applying the resolution rule to clauses α_i, α_j with $i, j < k$. A resolution derivation of the empty clause $\Box$ from G is called a *refutation* or a *proof*, and shows that G is unsatisfiable.

A resolution derivation $(\alpha_1, \ldots, \alpha_k)$ is *trivial* [21] if all variables resolved upon are distinct, and each α_i, for $i \geq 3$, is either an initial clause or is derived by resolving α_{i-1} with an initial clause. It is convenient to capture a trivial resolution derivation by a rule. A *chain resolution rule*, written $\alpha_1, \ldots, \alpha_k \vdash^{\vec{x}}_{\text{TVR}} \alpha$, states that α can be derived from $\alpha_1, \ldots, \alpha_k$ by trivial resolution derivation. We call $\alpha_1, \ldots, \alpha_k$ the *chain* and α_1 – the *anchor*, and variables $\vec{x} = (x_1, \ldots, x_{k-1})$ the *chain pivots*. Without loss of generality, we assume that the chain and chain pivots are resolved in the order given. That is, first α_1 is resolved with α_2 on x_1, then the resolvent is resolved with α_3 on x_2, etc. A *chain derivation* is a sequence $\pi \equiv (\alpha_1, \ldots, \alpha_n)$ where each α_k is either an initial clause or is derived by chain resolution from preceding clauses. A *derivation witness* of a chain derivation π is a total function D from clauses of π to sub-sequences of π such that

$$D(\alpha) = [\,] \Rightarrow \alpha \text{ is initial} \quad D(\alpha) \neq [\,] \Rightarrow D(\alpha) \vdash_{\text{TVR}} \alpha \quad (1)$$

Note that a derivation witness is not unique. As usual, a derivation of an empty clause is called a *proof*. Chain proofs capture concisely the proofs produced by CDCL SAT-solvers by logging learned clauses only. For example, the TraceCheck proof format [22] is based on chain derivation.

A *Craig interpolant* [23] of a pair of inconsistent formulas A and B is a formula I such that

$$A \Rightarrow I \qquad I \Rightarrow \neg B \qquad Var(I) \subseteq Var(A) \cap Var(B) \qquad (2)$$

where $Var(A)$ is the set of all variables of A. It is well known that an interpolant can be computed in polynomial time from a resolution proof of unsatisfiability of $A \wedge B$ [2], [8].

For interpolation, it is convenient to partition clauses of a CNF as belonging to A or B. More generally, an *N-colored CNF* is a pair (G, κ) of a CNF formula G and a coloring function $\kappa : G \to [1, .., N]$ that assigns to every clause $\alpha \in G$ a color between 1 and N. We omit the coloring function κ when it is clear from the context or irrelevant and write G for (G, κ). For a colored CNF (G, κ), we write $G_i = \kappa^{-1}(i)$ for the set of all clauses colored i. The coloring extends naturally to variables. For each $v \in Var(G)$, we define its minimum and maximum color as follows:

$$\kappa_{\downarrow}(v) = \min\{i \mid \exists \alpha \in G_i \cdot v \in \alpha\} \qquad (3)$$

$$\kappa_{\uparrow}(v) = \max\{i \mid \exists \alpha \in G_i \cdot v \in \alpha\} \qquad (4)$$

A variable v is called *local (to partition i)* if $\kappa_{\downarrow}(v) = \kappa_{\uparrow}(v) = i$, and *shared* otherwise. A clause α is *shared* if for all $v \in Var(\alpha)$, v is shared and $\kappa(\alpha) < \kappa_{\uparrow}(v)$. A colored CNF G is *striped* if for all $v \in Var(G)$, $\kappa_{\uparrow}(v) - \kappa_{\downarrow}(v) \leq 1$. That is, every variable is either local, or shared between partitions with adjacent colors. Note that every non-striped CNF can be made striped by adding fresh variables and equality constraints. In the rest of the paper, for simplicity, we assume that all colored CNFs are striped. Given a chain refutation π of a colored

CNF (G, κ) and a derivation witness D of π, we define the maximum color for the clauses of π inductively as follows:

$$\kappa_{\uparrow}(\alpha) = \begin{cases} \kappa(\alpha) & \text{if } \alpha \in G \\ \max\{\kappa_{\uparrow}(\beta) \mid \beta \in D(\alpha)\} & \text{otherwise} \end{cases} \quad (5)$$

Minimum color $\kappa_{\downarrow}(\alpha)$ is defined similarly.

A *sequence (or path) interpolant* for an N-colored unsatisfiable striped CNF (G, κ) is a sequence of formulas $(\top \equiv I_0, \ldots, I_N \equiv \bot)$ such that for all $1 \leq i \leq N$:

$$I_{i-1} \wedge G_i \Rightarrow I_i \quad \forall v \in \mathit{Var}(I_i) \cdot \kappa_{\downarrow}(v) = i \wedge \kappa_{\uparrow}(v) = i + 1$$

We assume that the reader is familiar with the basic CDCL SAT algorithm, as presented in [24]. We assume that the solver maintains all currently implied and decided (i.e., assigned) literals in a queue, called the *trail*, in the order they are assigned. We assume that the solver provides the following API:

- `UnitPropagation` exhaustively applies unit propagation (UP) rule by resolving all unit clauses;
- `ConflictAnalysis` analyzes the most recent conflict and learns a new clause;
- `IsOnTrail` checks whether a clause is in antecedent of a literal on the trail;
- `Enqueue` enqueues one or more literals on the trail;
- `IsDeleted`, `Delete`, `Revive` checks whether a clause is deleted, deletes a clause, and adds a previously deleted clause, respectively;
- `SaveTrail`,`RestoreTrail` save and restore the state of the trail.

III. Trimming Clausal Proofs

Clausal proofs were introduced by Goldberg and Novikov [20] who showed that the sequence of all the learned clauses, in the order they are learned by a CDCL solver, forms a chain derivation. They show that the chain derivation can be validated using UP facilities of the solver. The correctness is based on the following lemma that shows the connection between UP and trivial resolution.

Lemma 1 ([21]) *Given a CNF G and a clause c, c is deducible from G by unit propagation iff c is deducible from G by trivial resolution. That is, $F \vdash_{\text{UP}} c$ iff $F \vdash_{\text{TVR}} c$.*

Two algorithms are suggested in [20], one for backward and one for forward validation. The forward validation replays the proof forward, checking that each clause is subsumed (using UP) by prior clauses. Dually, backward validation walks the proof backwards, removing clauses, and checking that each removed clause is subsumed by the remaining ones.

Recently, backward validation has been improved by Heule et al. [16] who noticed that (a) CDCL solvers aggressively delete unnecessary clauses, and (b) keeping track of clause deletion significantly reduces the number of clauses used by UP during validation. They define a *DRUP-proof* as a sequence $\pi \equiv ((\alpha_0, d_0), \ldots, (\alpha_n, d_n) \equiv (\Box, \bot))$, where each d_k is a Boolean flag indicating whether the clause is deleted, and α_k is either an initial clause or is derived by chain resolution from the set of k-active clauses $\{\alpha_j \mid j < k \wedge d_j = \bot \wedge (\forall j < i < k \cdot \alpha_i \neq \alpha_j)\}$. Validation of DRUP-proofs is efficient because validation of a clause α_k depends only on the k-active clauses.

Forward validation walks the proof from the leaves to the empty clause. Thus, it is well suited for interpolation. However, clausal proofs produced by a CDCL solver contain many useless clauses making forward validation inefficient. Heule et al. [16] suggest that in this case, backward validation should be used to trim a clausal proof by removing all clauses that do not contribute to the derivation of the empty clause.

In this section, we present an efficient trimming procedure, called `Trim` and shown in Alg. 1, based on backward validation. Unlike Heule et al., our goal is not to certify a solver, but to trim the proof. Thus, we trust the solver and reuse its intermediate state (namely, the final state of the trail and deletion status of clauses) and routines (namely, unit propagation and conflict analysis). This makes our procedure efficient and easy to implement.

The input to `Trim` is a CDCL solver S in a conflicting state, and a corresponding DRUP-proof π_o. The output is a chain derivation π such that all clauses of π participate in a derivation of the empty clause. In the terminology of Heule et al., all clauses of π are *core*. The algorithm maintains a set C of core clauses. It walks the input DRUP-proof π_o backwards. Deleted clauses are revived (line 3). If the current clause α_i is on the trail, `UndoTrailCore` is used to pop the literals of the trail up to and including the literal whose antecedent is α_i. In the process, antecedents of any core literal on the trail are marked core as well. Next, α_i is removed from the solver, and, if it is not initial, validated using UP. For that, the negation of the literals is put on a trail and `UnitPropagate` is used to derive the conflict. Note that this always succeeds since we assume that the solver S and the proof π_o are valid. Finally, `ConflictAnalysisCore` is used to analyze the conflict, and, in the process, marks all clauses in the implication graph of the conflict as core. When the main loop terminates, π is a chain proof in reversed order.

We use `Trim` to trim a DRUP-proof before interpolation using forward validation. In the rest of the paper, we assume that all chain proofs are trimmed. The interpolation procedure is described in Section IV. `Trim` provides two degrees of freedom. First, different UP strategies result in different proofs. For example, Heule et al. prefer core clauses during UP to minimize the total size of the trimmed proof. Second, `ConflictAnalysisCore` can introduce additional clauses corresponding to different cuts of the implication graph. We propose strategies that result in better interpolants in Section V.

IV. Interpolation Algorithm

In this section, we present our interpolation algorithm.

Let (G, κ) be an N-colored striped CNF formula. Throughout this section, we assume, for simplicity, that $N = 3$. However, our results easily extend to an arbitrary number of colors. We denote shared variables of partition j by $V_j = Var(G_j) \cap Var(G_{j+1})$. For a clause $\alpha \in G$, we write $\alpha|_{[k,l]}$ for a clause obtained from α by removing all variables v with color less than k ($\kappa_{\uparrow}(v) < k$) or greater than l ($\kappa_{\uparrow}(v) > l$). We write $\alpha|_{\leq l}$ for $\alpha|_{[1,l]}$ and $\alpha|_{\geq k}$ for $\alpha|_{[k,N]}$. Recall that a clause α is *shared* w.r.t. j if $\mathit{Var}(\alpha) \subseteq V_j$ and $\kappa(\alpha) = j$.

Algorithm 1: $\texttt{Trim}(S, \pi_o)$

Input: A SAT-solver instance S with $\Box$ on the trail and the corresponding DRUP-proof $\pi_o = ((\alpha_0, d_0), \ldots, (\alpha_n, \bot) \equiv (\Box, \bot))$
Output: A chain derivation $(\beta_0, \ldots, \beta_m \equiv \Box)$

```
1  π = [ ] ; C = {α_n}
2  for i = n to 0 do
3      if S.IsDeleted(α_i) then S.Revive(α_i)
4      else
5          if S.IsOnTrail(α_i) then
6              S.UndoTrailCore(α_i, C)
7          S.Delete(α_i)
8          if α_i ∈ C then
9              if α_i is not initial then
10                 S.SaveTrail()
11                 S.Enqueue(¬α_i)
12                 c = S.UnitPropagation()
13                 S.ConflictAnalysisCore(c, C)
14                 S.RestoreTrail()
15             π.Append(α_i)
16 Reverse(π)
```

Our procedure, called ChainItp, is shown in Alg. 2. The inputs are a N-colored CNF (G, κ) and a (trimmed) chain derivation π. The output is a sequence interpolant $I_0, \ldots, I_N$. ChainItp walks π forward from α_0 to α_n and computes partial interpolants for each partition (or color) separately. For partition i and a clause α_j, a partial interpolant is a conjunction of a pair of formulas $p_i(\alpha_j) \wedge g_i$. g_i contains the CNF part of the interpolant, and $p_i(\alpha_j)$ contains the rest. The final interpolant is obtained as a partial interpolant of the empty clause $\alpha_n \equiv \Box$.

For a fixed color k, we partition the clauses of π into two groups: *leaf* and *non-leaf*. A clause is a leaf (for color k) if it is either initial, or derived only using clauses with color less than or equal to k. Otherwise, it is non-leaf. The leaf and non-leaf clauses are interpolated using helper functions Leaf and Tvr, respectively. Before going into detail, let us introduce the following notion:

Definition 1 Let (G, κ) be an N-colored striped CNF formula, π a chain refutation of G, D a derivation witness for π, and k a natural number $1 \leq k \leq N$. A shared leaf $\alpha \in \pi$ is *shared-derivable* w.r.t. k and D if for all $\beta \in D(\alpha)$, $\kappa_{\downarrow}(\beta) = k$ or β is shared-derivable w.r.t. $k - 1$ and D.

Clearly, for initial shared clauses, this definition holds trivially. Intuitively, α is shared-derivable w.r.t. k if it is derived using only clauses from G_k and shared-derivable clauses w.r.t. $k - 1$. Let us assume that our stripped CNF formula is $G_1 \wedge G_2 \wedge G_3$. All shared clauses w.r.t. G_1 are also shared-derivable. A shared clause w.r.t. G_2 is shared-derivable w.r.t. 2 iff it is derived using clauses from G_2 and clauses that are shared-derivable w.r.t. G_1. Note that we maintain a derivation witness D as part of the definition due to the fact that a chain derivation represent a space of possible resolution steps that may lead to a derived clause. Thus, in order for our recursive definition to apply, we must make sure a specific derivation witness is used.

Lemma 2 *Let (G, κ) be an N-colored striped CNF formula. Given a chain derivation π, let D be a derivation witness of π. Let $(g_0 = \top, g_1, \ldots, g_N)$ be a sequence such that g_i is a CNF containing all shared-drivable clauses w.r.t. a color i and D, then $g_{i-1} \wedge G_i \Rightarrow g_i$ for $1 \leq i \leq N$.*

The proof is immediate from the definition of shared-derivable clauses.

We now go into more detail about the mechanics of Leaf and Tvr. The function Leaf is applied to initial clauses (line 4) and to derived leaf clauses (line 15). The input is a clause α, a color j and a derivation witness D. The output is a pair (p, g) such that $p \wedge g$ is a partial interpolant of α for color j, and g is in CNF. It works according to the following rules:

- if α is shared-derivable w.r.t. j and D: $p = \top$ and $g = \alpha$.
- otherwise, if $\kappa(\alpha) \leq j$ then $p = \alpha|_{\geq j+1}$ and $g = \top$
- otherwise, $p = g = \top$

The function Tvr is applied to derived clauses. The input is a clause α, a corresponding chain derivation $\vec{\beta} \vdash^{\vec{x}}_{\text{TVR}} \alpha$, and a color j. The chain $\vec{\beta} = (\beta_0, \ldots, \beta_b)$ is obtained by UP and conflict analysis (lines 8-10) as described in Section III. The output is a formula q_b, where

$$q_l = \begin{cases} p_j(\beta_0) & \text{if } l = 0 \\ q_{l-1} \bowtie^j_{x_l} p_j(\beta_l) & \text{ow} \end{cases} \qquad \bowtie^j_x = \begin{cases} \wedge & \text{if } \kappa_{\uparrow}(x) \leq j \\ \vee & \text{ow} \end{cases}$$

That is, Tvr walks up the chain $\vec{\beta}$, and, at each resolution step, either conjoins or disjoins the partial interpolants of the chain clauses. Tvr is effectively a direct extension of the interpolation rules of [2] from resolution to chain resolution.

It is important to note that the derivation witness D is not stored explicitly in our implementation of the algorithm, and it is used implicitly by Leaf. We only mention it in Algorithm 2 for clarity.

Our interpolation algorithm is somewhat unorthodox since it treats some of the derived clauses as leaves. Furthermore, it keeps a CNF part of the interpolant separately (using g_j). We show that none-the-less, it still produces a valid sequence interpolant.

Definition 2 Given an unsatisfiable N-colored striped CNF (G, κ) and a chain derivation π. A sequence of partial interpolants $(\top, p_1, \ldots p_{N-1}, \bot)$ and a set of CNF formulas $\{g_j\}_1^{N-1}$ are *valid* iff for every $1 \leq k \leq N$, and for every $\alpha \in \pi$, $(p_k(\alpha) \wedge g_k \wedge G_{k+1}) \Rightarrow (p_{k+1}(\alpha) \vee \alpha|_{\geq k+1}) \wedge g_{k+1}$.

Note that a valid partial interpolant sequence results in a valid sequence interpolant. We show that the partial interpolants of ChainItp satisfy validity requirement of Def 2.

Theorem 1 *Given an N-colored striped CNF (G, κ) and a chain derivation π, the sequence of partial interpolants $(\top, p_1, \ldots p_{N-1}, \bot)$ and the set of CNF formulas $\{g_j\}_1^{N-1}$ computed by* ChainItp *are valid.*

Proof: For simplicity, we show the proof for the case $N = 3$. The proof for the general case is similar. Furthermore, we rely on the fact that without our special leaf handling, ChainItp is a straightforward extension of McMillan's procedure [2] to chain resolution. We use $q_j(\alpha)$ to denote the partial interpolant of [2].

Algorithm 2: `ChainItp`

Input: A SAT-solver instance S, colored CNF (G, κ), $\kappa : G \to [1..N]$, and a chain derivation $\pi = (\alpha_0, \ldots, \alpha_n \equiv \bot)$
Output: An interpolation sequence $(\top \equiv I_0, I_1, \ldots, I_N \equiv \bot)$

1 **for** $i = 0$ **to** n **do**
2   **if** $\alpha_i \in G$ **then**
3   **for** $j = 1$ **to** $N - 1$ **do**
4       $(p_j(\alpha_i), g) \leftarrow \texttt{Leaf}(\alpha_i, j)$
5       $g_j \leftarrow g_j \wedge g$
6   **else**
7   S.`UnitPropagate()`, S.`SaveTrail()`
8   S.`Enqueue`$(\neg\alpha_i)$
9     $\beta_0 = S$.`UnitPropagate()`
10     $\vec{\beta} = S$.`ConflictAnalysisTvr`(β_0, α_i)
      /* $\vec{\beta} = (\beta_0, \ldots, \beta_b)$ is a subsequence of π s.t. $\vec{\beta} \vdash^{\vec{x}}_{\texttt{TVR}} \alpha_i$ */
11     $D(\alpha_i) \leftarrow \vec{\beta}$
12     $\kappa(\alpha_i) \leftarrow \max\{\kappa(c) \mid c \in \vec{\beta}\}$
13   **for** $j = 1$ **to** $N - 1$ **do**
14   **if** $\kappa(\alpha_i) \leq j$ **then**
15         $(p_j(\alpha_i), g) \leftarrow \texttt{Leaf}(\alpha_i, j)$
16         $g_j \leftarrow g_j \wedge g$
17   **else**
18         $p_j(\alpha_i) \leftarrow \texttt{Tvr}(\vec{\beta}, \alpha_i, j)$
19   S.`RestoreTrail()`
20   S.`Revive`(α_i)
21 $I_0 \leftarrow \top, I_N \leftarrow \bot$
22 **for** $j = 1$ **to** $N - 1$ **do** $I_j \leftarrow p_j(\alpha_n) \wedge g_j$

The proof is by induction on the graph induced by π and D. The base case follows from [14] since for an initial clause α $p_j(\alpha) \wedge g_j = q_j(\alpha)$. For the inductive step, we only consider the case of a single resolution step. Let c_1 and c_2 be two clauses that resolve on v to get c. W.l.o.g., assume $v \in c_1$ and $\neg v \in c_2$. By inductive hypothesis:

$$p_1(c_1) \wedge g_1 \wedge G_2 \Rightarrow (p_2(c_1) \vee c_1|_{\geq 2}) \wedge g_2 \tag{6}$$
$$p_1(c_2) \wedge g_1 \wedge G_2 \Rightarrow (p_2(c_2) \vee c_2|_{\geq 2}) \wedge g_2 \tag{7}$$

Since we rely on [2], [14], we only need to prove the correctness for our modifications, namely treating derived clauses as leaves. Thus, there are only two cases: (1) c is derived using only clauses from G_1, or (2) c is derived using only clauses from G_1 and G_2. Case (1) is immediate by Lemma 2. For case (2), w.l.o.g., assume that c_1, c_2, and c are not leaves w.r.t. 1, but are leaves w.r.t. 2. In this case, we can substitute p_2 with a partial interpolant for the leaf. The induction hypothesis becomes:

$$p_1(c_1) \wedge g_1 \wedge G_2 \Rightarrow (c_1|_{\geq 2}) \wedge g_2 \tag{8}$$
$$p_1(c_2) \wedge g_1 \wedge G_2 \Rightarrow (c_2|_{\geq 2}) \wedge g_2 \tag{9}$$

By the definition of p_1 we know that if $v \in Var(G_3)$ then $p_1(c) = p_1(c_1) \wedge p_1(c_2)$, otherwise $p_1(c) = p_1(c_1) \vee p_1(c_2)$. We take care of the following two cases. Case 1, c is shared-derivable. We need to show that $(p_1(c) \wedge g_1) \wedge G_2 \Rightarrow (\top \vee c) \wedge g_2$. Since c is shared-derivable $c \in g_2$ and g_1 is unchanged. By Lemma 2, $p_1(c) \wedge g_1 \wedge G_2 \Rightarrow g_2$ holds.

Case 2, c is not shared-derivable. We need to show that $(p_1(c) \wedge g_1) \wedge G_2 \Rightarrow (c|_{\geq 3} \vee c|_{\geq 2}) \wedge g_2$. Since c is not shared-derivable both g_1 and g_2 are unchanged. Assume that $v \notin Var(G_3)$, then $p_1(c) = p_1(c_1) \vee p_1(c_2)$. Assume, to the contrary, that $p_1(c) \wedge g_1 \wedge G_2 \Rightarrow c|_{\geq 2} \wedge g_2$ does not hold. Then, there is an assignment s.t. $(p_1(c) \wedge g_1) \wedge G_2$ evaluates to $\top$ while $c|_{\geq 2} \wedge g_2$ evaluates to $\bot$. From Lemma 2, we know that g_2 evaluates to $\top$, therefore, $c|_{\geq 2}$ is $\bot$. W.l.o.g. assume that under this assignment $p_1(c_1)$ evaluates to $\top$. By the induction hypothesis $c_1|_{\geq 2} \wedge g_2$ evaluates to $\top$ as well. Due to our assumption that $c|_{\geq 2}$ evaluates to $\bot$, v must evaluate to $\top$. but, since $v \in c_1|_{\geq 2}$, it must aslo be part of $c|_{\geq 2}$. Thus, indicating that $c|_{\geq 2}$ evaluates to $\top$, in contradiction to our assumption. The other cases are proved similarly. ■

V. COLORS, PROOFS, AND CNF

In this section, we discuss how to combine our framework with a light-weight proof restructuring. The goal of restructuring is to increase the number of shared derived leaves in the proof to increase the CNF component of the interpolant. We first introduce the concept of colorable chain refutations and show that they lead to a simple CNF interpolation procedure. However, an ordinary chain refutation is exponentially stronger than colorable one. Hence, restricting to colorable refutations is not practical. Instead, we propose a polynomial algorithm to restructure a refutation on-the-fly to increase its colaribility.

Let (G, κ) be a striped N-colored CNF, π a chain refutation of G, and D a derivation witness for π. The witness D is called *colored* if for every derived clause $\alpha \in \pi$, the corresponding derivation sequence $D(\alpha) = (\beta_0, \ldots, \beta_n)$ satisfies the following condition: for all $0 \leq i \leq n$, $\kappa_\uparrow(\beta_i) = \kappa_\downarrow(\beta_i) = \kappa_\uparrow(\alpha)$ or $\kappa_\uparrow(\beta_i) < \kappa_\uparrow(\alpha)$ and β_i is shared. A chain refutation π is *colorable* if there exists a colored refutation witness for it.

Colorable refutations induce a simple interpolation procedure. Let $\pi = (\alpha_0, \ldots, \alpha_n)$ be a colorable chain refutation with N colors. Then, the sequence $\vec{I} \equiv (I_0 \equiv \top, \ldots, I_N \equiv \bot)$ defined as follows:

$$I_i = \{\alpha \in \pi \mid \kappa_\uparrow(\alpha) = i \wedge \alpha \text{ is shared}\} \tag{10}$$

is a sequence-interpolant. Furthermore, $\vec{I}$ is in CNF and is linear in the size of the chain refutation π. This is not a coincidence. Colorable chain refutations and CNF interpolants are closely related.

Theorem 2 *For every colorable chain refutation π of N-colored CNF G there exists a sequence interpolant $\vec{I}$ such that $\sum_{i=0}^{N} |I_i| < |\pi|$. For every CNF sequence interpolant $\vec{I}$ of G, there is a corresponding colorable refutation containing the clauses of $\vec{I}$.*

Proof: For simplicity, we only show the case when $N = 2$, where there is only one non-trivial interpolant: I_1. First, we show (10) defines a sequence interpolant. By definition, I_1 is the set of all shared clauses of π colored 1. By definition of coloring, each 1-colored clause is implied by G_1, hence, $G_1 \Rightarrow I_1$. By colorability of π, there is a refutation of $I_1 \wedge G_2$.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Second, for each clause $\alpha \in I_1$, let π_α be a chain refutation of $G_1 \wedge \neg\alpha$, and π_2 be chain refutation of $I_1 \wedge G_2$. The refutation π is obtained by concatenating those refutations. ∎

Ordinary chain refutations are exponentially stronger than colorable ones. For example, let k be a natural number and consider the 2-colored CNF $G^k = G_1^k \wedge G_2^k$, where

$$G_1^k = (\bigvee_{i=1}^{k} x_i) \wedge \bigwedge_{i=1}^{k} (x_i \Rightarrow a_i) \wedge (x_i \Rightarrow b_i) \quad G_2 = \bigwedge_{i=1}^{k} (\neg a_i \vee \neg b_i)$$

The CNF interpolant $I_1^k = \text{CNF}(\bigvee_{i=1}^{k}(a_i \wedge b_i))$ is exponential in k. Therefore, a colorable refutation of G is exponential in k as well. Thus, transforming a chain refutation into a colorable one is worst-case exponential. Note that proof-less interpolation techniques such as [18], [19] correspond to colorable chain refutations, and hence, in the worst case are exponentially more expensive than CDCL.

Given a proof π, if `ChainItp` (Alg. 2) returns the interpolant in CNF, then π is colorable. The converse is not true because `ChainItp` picks an arbitrary witness D. Thus, it might not find a colorable witness, even if one exists. We propose two strategies to improve `ChainItp`.

First, we propose to apply UP on line 9 of `ChainItp` *ordered* by the color of the clauses. In the forward-order, UP is first applied to 1-colored clauses, than two 1- and 2-colored clauses, etc. Conversely, the backward-order starts with N-colored clauses. Both strategies increase the number of clauses that are derived within a partition boundary.

Second, we propose a new algorithm to restructure the chain derivation produced by `ConflcitAnalysisTvr` on line 10 of `ChainItp`. The new algorithm, called `ConflictAnalysisClr`, is shown in Alg. 3. It takes a SAT-solver in a conflicting state and a conflict clause, and produces a sequence of chain derivation Π and a new learned clause α. The interpolation step of `ChainItp` (lines 12–19) is then applied to each chain derivation in Π. The main step of the algorithm is done by the supporting procedure, called `Colorize`, shown in Alg. 4.

The algorithms make the following assumptions about the SAT-solver. All clauses are sorted relative to the current assignment so that $\top$-valued literals precede all $\bot$-valued literals. All implied literals are stored in the trail in the implication order. **nil** indicates undefined values (literals and clauses). `Value`(q) is the value of literal q in the current assignment. `Reason`(q) is the unique clause that implies the literal q or $\neg q$. `Reason`(q) = **nil** if q is not implied by any other clause. `SetReason`(q, c) sets clause c as the reason for q and $\neg q$.

Intuitively, `Colorize` walks the chain derivation from the anchor β_0, and applies only resolutions that are in the same partition as β_0. Clauses from earlier partitions are recursively colorized by attempting to turn them into shared-derived clauses. Clauses from later partitions are ignored. `ConflictAnalysisClr` applies `Colorize` starting from the partition of the anchor, and then as many time as necessary to remove all UP-implied literals from the learned clause. In the worst case, the set Π is linear in the number of clauses in the original chain derivation found by `ConflictAnalysisTvr`.

Algorithm 3: `ConflictAnalysisClr`

Input: A SAT-solver S and a conflict clause *confl*.
Output: A learned clause α and a chain proof Π.

```
1  k ← κ(confl)
2  forever do
3  |  α ← Colorize(S, confl, k)
4  |  let T = {q ∈ α | S.Reason(q) = nil}
5  |  if T = ∅ then break
6  |  k ← min{κ(q) | q ∈ T}
```

Algorithm 4: `Colorize`

Input: A SAT-solver S, a conflict *confl* and a color k
Output: A learned clause α and a chain proof Π

```
1  p ← nil, α = [ ], β = [ ], W = ∅
2  if S.Value(confl[0]) = ⊤ then
3  |  p ← confl[0], α.Append(p)
4  forever do
5  |  if κ(confl) < k then
6  |  |  confl ← Colorize(S, confl, κ(confl))
7  |  |  S.SetReason(confl[0], confl)
8  |  β.Append(confl)
9  |  foreach q ∈ confl do
10 |  |  if q = p ∨ q ∈ W ∨ q ∈ α then continue
11 |  |  r ← S.Reason(q)
12 |  |  if r ≠ nil ∧ κ(r) ≤ k then  W ← W ∪ {¬q}
13 |  |  else α.Append(q)
14 |  if W = ∅ then break
15 |  p ← q ∈ W s.t. q has the largest trail index
16 |  W ← W \ {p}, confl ← S.Reason(p)
17 if β ≠ [ ] then Π.Append((β ⊢_TVR α))
```

VI. Experiments

We have implemented our DRUP-based interpolation framework on top of MiniSAT 2.2. It is available at part of AVY model checker at http://arieg.bitbucket.org/avy. For evaluation, we used two sets of experiments. First, we compared the sizes of the sequence interpolants and the time it takes to extract them for Bounded Model Checking (BMC) problems. Second, we evaluated the framework within our interpolation-based model checker AVY [7]. In both cases, we use benchmarks from HWCCC'13[1]. For baseline, we compare against proof-based interpolation in ABC [25]. Note that we have extended the ABC implementation to sequences in a straight-forward way. However, the comparison with ABC has to be taken with a grain of salt since ABC uses a customized version of an older version of MiniSAT, rewritten in C with some new features back-ported. None-the-less, ABC implementation is the state-of-the-art used by many other hardware model checkers, and we found it to perform well (compared to MiniSAT 2.2).

Fig. 1 shows the sizes of interpolants for BMC problems of depth 20. All problems were given a 180 seconds timeout. In majority of cases, the DRUP-based approaches produce smaller interpolants, measured as number of AIG nodes. Note that for our interpolation algorithm, we conjoin the CNF into the AIG. Clearly, without conjoining this part the interpolants

[1]Benchmarks are available from http://fmv.jku.at/hwmcc13.

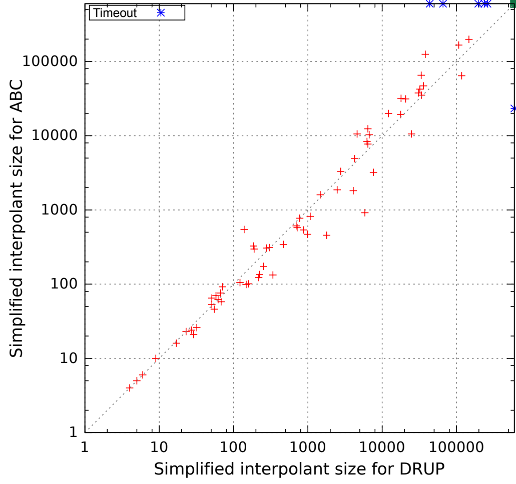


Fig. 1: Comparing sizes (AND gates) of interpolants

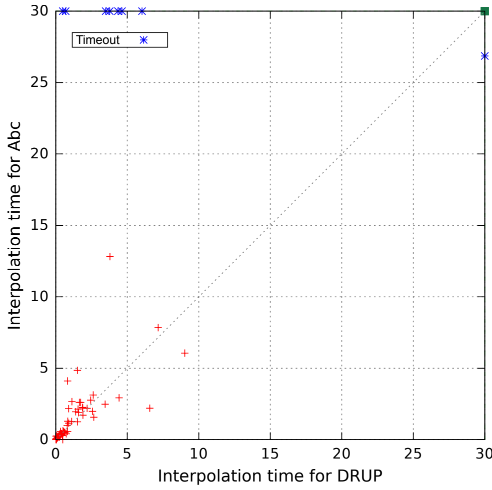


Fig. 2: Comparing time (seconds) to extract interpolants

are even smaller. Careful inspection of the results shows that only two cases were insolvable by DRUP-based methods. On the other hand, 9 cases were not solved by the traditional proof logging solver. This gives us an indication about the strength of a DRUP-based solver, which was apparent in all of our experiments: when the SAT problem becomes hard, the DRUP-based approach outperforms a traditional proof logging solver. Fig. 2 shows the extraction times for the same BMC problems. Note that the extraction time is comparable to the resolution proof based method, but consumes less memory (since the resolution proof is not logged)[2].

Table I analyzes the performance of our model checker AVY when using different interpolation algorithms. This is an important analysis as it shows the affect on the runtime of the model checker and on the depth at which convergence is achieved. First, note that all DRUP-based approaches outperforms our ABC baseline w.r.t. number of solved instances. In addition to that, AVY performs better when using DRUP on the majority of cases. One can note that in most cases, there is a correlation between depth of convergence and performance where lower depth of convergence indicates better runtime.

[2]More comparison charts are at http://arieg.bitbucket.org/avy/drup/plots.

TABLE I: Running time for AVY using different interpolation algorithms. ABC is for ABC's MiniSAT, DRUP is for MiniSAT with DRUP and ordered UP; +Clr adds our colorizing algorithm; and +Pre adds MiniSAT's Pre-processor. 't' stands for time, 'd' for depth of the solution, '–' for time-out or other failure.

Name	**ABC**		**DRUP**		**DRUP+Clr**		**DRUP+Pre**		**DRUP+Pre+Clr**	
	t (s)	d	t (s)	d	t (s)	d	t (s)	d	t (s)	d
6s102	203	23	91	24	340	37	175	33	399	37
6s121	–	–	418	50	248	34	–	–	–	–
6s130	136	8	144	9	165	9	192	9	216	9
6s144	–	–	533	25	583	24	668	26	560	22
6s189	622	21	382	21	572	26	396	21	552	23
6s206rb025	66	5	13	3	13	3	15	3	15	3
6s207rb16	46	8	96	8	96	8	127	8	110	8
6s209b1	181	24	106	24	115	24	142	24	157	24
6s215rb0	7	7	4	7	4	7	4	7	4	7
6s216rb0	21	13	12	13	11	13	13	13	13	13
6s218b1246	588	9	283	9	272	9	293	9	281	9
6s271rb045	371	11	256	10	262	10	–	–	–	–
6s273b37	162	20	216	20	217	20	277	20	284	20
6s275rb253	4	6	7	6	7	6	8	6	10	6
6s276rb318	19	10	11	10	11	10	15	10	16	10
6s277rb342	18	10	15	13	10	10	13	10	22	13
6s282b15	103	18	99	25	106	19	184	17	209	17
6s288r	–	–	376	24	338	22	468	23	444	22
6s289rb00529	77	7	38	7	37	7	38	7	37	7
6s291rb18	517	78	341	74	283	73	–	–	717	73
6s305rb069	270	18	139	18	128	18	133	18	136	18
6s306rb03	219	17	58	13	56	13	58	13	59	13
6s307rb06	127	13	86	13	90	13	102	13	109	13
6s311rb1	69	2	19	2	18	2	20	2	20	2
6s326rb02	34	11	14	11	15	11	17	11	17	11
6s327rb10	25	9	11	9	11	9	12	9	12	9
6s330rb11	10	3	5	3	4	3	5	3	5	3
6s335rb60	2	4	1	4	1	4	1	4	1	4
6s343b31	–	–	–	–	–	–	332	15	503	15
6s349rb12	185	13	143	15	142	15	158	15	169	15
6s364rb03158	519	2	198	2	198	2	191	2	188	2
6s372rb31	358	29	322	30	162	21	295	29	276	26
6s374b029	467	9	264	9	258	9	264	9	256	9
6s380b129	226	20	109	20	109	20	131	20	122	20
6s384rb194	–	–	–	–	–	–	786	22	868	30
6s385rb444	441	13	237	12	257	13	218	12	203	12
6s386rb07	–	–	871	13	868	13	855	13	828	13
6s388b07	0	0	0	0	0	0	0	0	0	0
6s389b11	6	4	3	4	3	4	3	4	3	4
6s38	341	14	301	15	296	13	624	19	347	14
6s403rb0609	17	5	11	5	12	5	14	5	14	5
6s404rb4	55	4	45	4	65	5	69	4	78	4
6s405rb611	85	6	53	6	54	6	58	6	65	6
6s406rb111	735	16	521	16	612	16	544	16	662	17
6s407rb296	417	12	354	12	360	12	378	12	405	12
6s408rb191	264	8	452	8	420	8	340	8	371	8
6s410rb043	193	9	150	9	156	9	275	10	273	10
6s9	166	10	194	9	219	9	309	9	221	9
SOLVED	42		46		46		45		46	

Also note that this experiment confirms the results of the above figures which show that interpolation time is comparable with a proof-logging SAT solver and that sizes are in favor of DRUP.

Another important analysis is the effect *Colorizing* has on AVY's performance. Clearly, using colorize results in different interpolants. We can see from the results that there are cases where this results in better convergence depths and thus better performance. Note that using this feature is more demanding than simply extracting an interpolant since it restructures local chain derivations. Even though, when the convergence depth is similar the performance degradation due to the extra computation is small. It is important to note that colorizing results in many shared-derivable leaves, which means that the CNF component of the interpolant is meaningful. Currently, we did not make any special use of the CNF component and we leave this option for future research and exploration.

Finally, in Table II, we show the number of shred-derivable leaves, i.e. number of clauses in the CNF component of the interpolant computed by our method. Recall that DRUP is used

TABLE II: Number of shared-derivable leaves using our interpolation algorithm when solving BMC problems using bound 20. Algorithm names are as in Table I.

Name	DRUP	DRUP+Clr	DRUP+Pre	DRUP+Pre+Clr
6s102	0	73	0	257
6s119	0	976	0	0
6s122	0	223	0	216
6s152	0	449	0	217
6s188	0	651	0	521
6s196	0	648	0	642
6s276rb318	0	230	0	122
6s27	0	507	0	572
6s282b15	0	1684	0	270
6s291rb18	0	420	24	177
6s292rb024	0	1043	0	577
6s302rb09	0	1257	0	369
6s309b046	0	641	0	669
6s310r	0	1334	1	810
6s351rb02	0	6956	0	6910
6s384rb194	0	1144	0	408
6s44	0	1701	0	1188
6s50	0	617	0	166
6s7	0	1372	1	860
6s8	0	1123	1	615

with ordered UP while DRUP+Clr is used with ordered UP and colorize. Here too, we use fixed bound BMC problems. It is clear that our colorizing algorithm is very effective in finding a large number of clauses. While we present only a selected subset, this trend holds in all our experiments.

Note that while the underlying model checking algorithm AVY did not make a special use of the CNF component, we believe that specialized usage of the CNF component will result in better performance [5], [7], [12].

VII. Related Work

To our knowledge, this paper is the first to present and *evaluate* a DRUP-based interpolation framework. Moreover, we introduce a novel algorithm that computes a *sequence interpolant* partially in CNF. Finally, our restructuring algorithm is not based on pivot reordering as in previous works, but tries to keep resolution steps within a given partition (colorizing). We have already discussed proof-based and proof-less interpolation methods in Sec. I, and clausal proofs in Sec. III. Thus, in this section, we only focus on proof restructuring for CNF.

Many works deal with generating better interpolants, either using new interpolation algorithms or by proof restructuring. Our work is a synergy of these two approaches. [11] and [13] suggest local transformation rules that are based on pivots reordering to get CNF interpolants. Rollini et al. [13] also suggest a compression of a resolution proof as a pre-processing step. Unlike our work, they rely on explicit resolution proofs. Furthermore, our restructuring does not rely on pivot reordering and supports sequence interpolation natively.

Our interpolation algorithm identifies the CNF component of an interpolant even if the interpolant itself is not in CNF. Vizel at al. [12] introduce an interpolation procedure that also produces (near) interpolants in CNF. However, unlike [12], our framework does not rely on explicit resolution proofs and produced complete interpolants. We leave extending [12] to DRUP-proofs for future work.

VIII. Conclusion

In this paper, we introduce a DRUP-based interpolation framework. We show how DRUP-proofs can be trimmed and restructured for interpolation. We develop a novel interpolation algorithm that computes interpolants partially in CNF. Furthermore, we show how DRUP-proofs can be locally restructured to maximize the size of the CNF component without exponentially increasing the proof. Based on previous works [5], [7], [12], we believe that getting a CNF component for an interpolant is beneficial for the underlying model checking algorithm. Our framework is implemented in MiniSAT and is publicly available. Our experiments show that the framework is very effective in the context of both bounded and unbounded model checking applications.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[2] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV*, 2003, pp. 1–13.

[3] ——, "Lazy abstraction with interpolants," in *CAV*, 2006, pp. 123–136.

[4] Y. Vizel and O. Grumberg, "Interpolation-sequence based model checking," in *FMCAD*, 2009, pp. 1–8.

[5] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[6] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[7] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *CAV*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 260–276.

[8] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. Symb. Log.*, vol. 62, no. 3, 1997.

[9] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *VMCAI*, 2010, pp. 129–145.

[10] G. Weissenbacher, "Interpolant strength revisited," in *SAT*, 2012.

[11] R. Jhala and K. L. McMillan, "Interpolant-Based Transition Relation Approximation," in *CAV*, 2005, pp. 39–51.

[12] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in cnf," in *CAV*, 2013, pp. 330–346.

[13] S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich, "Resolution proof transformation for compression and interpolation," *CoRR*, vol. abs/1307.2028, 2013.

[14] A. Gurfinkel, S. F. Rollini, and N. Sharygina, "Interpolation properties and sat-based model checking," in *ATVA*, 2013, pp. 255–271.

[15] A. Van Gelder, "Producing and verifying extremely large propositional refutations - have your cake and eat it too," *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.

[16] M. Heule, W. A. Hunt, Jr, and N. Wetzler, "Trimming while checking clausal proofs," in *FMCAD*, 2013, pp. 181–188.

[17] S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification," in *LPAR*, 2013, pp. 683–693.

[18] H. Chockler, A. Ivrii, and A. Matsliah, "Computing interpolants without proofs," in *Haifa Verification Conference*, 2012, pp. 72–85.

[19] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu, "Efficient modular SAT solving for IC3," in *FMCAD*, 2013, pp. 149–156.

[20] E. I. Goldberg and Y. Novikov, "Verification of Proofs of Unsatisfiability for CNF Formulas," in *DATE*, 2003, pp. 10 886–10 891.

[21] P. Beame, H. A. Kautz, and A. Sabharwal, "Towards understanding and harnessing the potential of clause learning," *J. Artif. Intell. Res. (JAIR)*, vol. 22, pp. 319–351, 2004.

[22] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[23] W. Craig, "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[24] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*, 2009, pp. 131–153.

[25] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.

Efficient Extraction of Skolem Functions from QRAT Proofs

Marijn J.H. Heule
The University of Texas at Austin
marijn@cs.utexas.edu

Martina Seidl and Armin Biere
Johannes Kepler University, Linz
{martina.seidl, biere}@jku.at

***Abstract*—Many synthesis problems can be solved by formulating them as a quantified Boolean formula (QBF). For such problems, a mere true/false answer is often not enough. Instead, expressing the answer in terms of Skolem functions reflecting the quantifier dependencies of the variables is required. Several approaches have been presented to extract such functions from term-resolution proofs. However, not all solvers and preprocessors are able to produce term-resolution proofs, especially when universal expansion is involved. In previous work, we developed the QRAT proof system consisting of three simple rules which allowed us to overcome this issue and to equip modern expansion-based tools like the preprocessor bloqqer with proof tracing. In this paper, we show how to extract Skolem functions from QRAT proofs. We present a general extraction tool and compare its performance to similar resolution-based tools. We show that the Skolem functions extracted from QRAT proofs are smaller than those produced by alternative approaches making our method in particular useful for synthesis applications.**

I. Introduction

Synthesis problems, which aim at the automatic derivation of an implementation from a given specification, typically ask whether for all possible inputs by the environment there exists a strategy for the system such that certain properties like safety hold. If the answer is positive, then this strategy provides the means to synthesize the required implementation of the system which by construction obeys the given specification. Therefore, a tool solving such a synthesis problem should not only provide a yes/no answer, but also a strategy of the specification in the case of realizability.

A natural way to encode synthesis problems is offered by quantified Boolean formulas (QBFs) [1], [2], which extend propositional logic by quantifiers over propositional variables [3]. The QBF formalism provides a convenient framework for modeling finite two-player games [4] which is reflected by the popular game-based view on the semantics of QBFs. Here, the evaluation of a QBF is described as a game between the existential player who owns the existential variables and the universal player who owns the universal variables of the formula. The existential player wants to satisfy the formula, while the universal player wants to falsify the formula. The moves are assignments to the variables, where the order of the variables in the quantifier prefix has to be respected. If the formula is satisfiable, then there exists a strategy for the existential player to always win the game and respectively, there is a strategy for the universal player to win the game if the formula is unsatisfiable.

By using existential and universal quantification, QBFs allow for exponentially more succinct encodings than propositional logic, with the consequence that the satisfiability problem of QBF is PSPACE-complete [3]. Therefore, the field of application of QBF ranges from efficient encodings of verification problems like model checking tasks to planning (see [2] for a survey on QBF applications).

All these applications have in common that they require models if the formula encoding the application problem is satisfiable. Whereas in SAT a model is given by a variable assignment, in QBF the situation is more complicated [5]. A model is an assignment tree giving a winning strategy to the existential player. In practice, a more compact representation than given by an assignment tree is required. To this end, the concept of Skolem functions, which for example are used in first-order logic to eliminate existential quantifiers, is transferred to the context of QBF [5]. A Skolem function for an existential x variable is a Boolean function over the universal variables preceding x in the quantifier prefix.

Today, it is known how Skolem function extraction can be realized in the context of DPLL-based QBF solving [6], which is the solving paradigm realized by most state-of-the-art solvers. Basically, the possibility of search-based approaches is exploited to generate term-resolution proofs from which winning strategies for the existential player can be generated [7], [8]. Unfortunately, this approach does not apply to expansion-based techniques [9], [10], which have been shown to be extremely powerful if realized in preprocessors. Until recently, it was not possible to generate any proofs when preprocessing is applied, because it is an open question if expansion can be simulated by resolution [11]. However, when preprocessing is restricted or even omitted, the solving performance drastically decreases. To overcome this issue, we presented the QRAT proof system [12] which is able to capture all state-of-the-art preprocessing techniques by a few simple rules. We could further show that emitting QRAT proofs causes only small overhead and that the validation of QRAT proofs is computationally cheap. Hence we were able to provide a tool to certify the result of a preprocessor efficiently.

In this paper, we go one step further and show how to extract Skolem functions from QRAT proofs of satisfiability. With this work we solve one important issue hindering the practical application of QBF. Moreover, the size of the Skolem functions that we extract is smaller than in other approaches.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

In the following, we first recapitulate QBF basics in Section II and review literature in Section III. Then we introduce the QRAT proof system in Section IV which is the basis for the Skolem function extraction approach presented in Section V. Implementation details are discussed in Section VI, followed by an experimental evaluation in Section VII. Finally, we conclude this paper with an outlook to future work.

II. Preliminaries

The language of QBF extends the language of propositional logic by existential and universal quantifiers over the propositional variables. As usual, we assume a QBF to be in *prenex conjunctive normal form* (PCNF).[1] A QBF in PCNF has the structure $\Pi.\psi$ where the prefix Π has the form $Q_1X_1Q_2X_2\dots Q_nX_n$ with disjoint variable sets X_i and $Q_i \in \{\forall, \exists\}$. The formula ψ is a propositional formula in conjunctive normal form, i.e., a conjunction of clauses. A clause is a disjunction of literals and a literal is either a variable (positive literal) or a negated variable (negative literal). The variable of a literal is denoted by $\mathsf{var}(l)$ where $\mathsf{var}(l) = x$ if $l = x$ or $l = \bar{x}$. The negation of a literal l is denoted by $\bar{l}$. The quantifier $\mathsf{Q}(\Pi, l)$ of a literal l is Q_i if $\mathsf{var}(l) \in X_i$. Let $\mathsf{Q}(\Pi, l) = Q_i$ and $\mathsf{Q}(\Pi, k) = Q_j$, then $l \leq_\Pi k$ if $i \leq j$. We sometimes write formulas in CNF as sets of clauses and clauses as sets of literals. We consider only closed QBFs, so ψ contains only variables which occur in the prefix. The variables occurring in the prefix of ϕ are given by $\mathsf{vars}(\phi)$. The subformula ψ_l consisting of all clauses of matrix ψ with literal l is defined by $\psi_l = \{C \mid l \in C, C \in \psi\}$. By $\top$ and $\bot$ we denote the truth constants true and false. QBFs are interpreted as follows: a QBF $\forall x\Pi.\psi$ is false iff $\Pi.\psi[x/\top]$ or $\Pi.\psi[x/\bot]$ is false where $\Pi.\psi[x/t]$ is the QBF obtained by replacing all occurrences of variable x by t. Respectively, a QBF $\exists x\Pi.\psi$ is false iff both $\Pi.\psi[x/\top]$ and $\Pi.\psi[x/\bot]$ are false. If the matrix ψ of a QBF ϕ contains the empty clause after eliminating the truth constants according to standard rules, then ϕ is false. Accordingly, if the matrix ψ of QBF ϕ is empty, then ϕ is true. Two QBFs ϕ_1 and ϕ_2 are *satisfiability equivalent* (written as $\phi_1 \sim \phi_2$) iff they have the same truth value. Two QBFs ϕ_1 and ϕ_2 are *logically equivalent* (written as $\phi_1 \approx \phi_2$) if they have the same set of (counter) models.

Whereas in propositional logic a model of a formula is given by a satisfying variable assignment, for a QBF a model has to reflect the variable dependencies between existential and universal variables. Hence, QBF models are either expressed in form of subtrees of assignment trees or as Skolem functions.

Definition 1. Let x be an existential variable of $\phi = \Pi.\psi$ and let $y_1, \dots, y_n$ be all universals of ϕ with $y_i \leq_\Pi x$. Then a propositional formula $f_x(y_1, \dots, y_n)$ is a *Skolem function* for x, and called *valid* iff $\phi[x/f_x] \sim \phi$. A set of Skolem functions $\mathcal{F}$ which contains exactly one Skolem function for every existential x is called a *Skolem set*. It is called *valid* iff it only contains valid Skolem functions.

[1] Note that any QBF of arbitrary structure can be efficiently transformed to a satisfiability equivalent formula in PCNF.

Obviously, for any satisfiable QBF ϕ, a valid Skolem set $\mathcal{F}$ gives a strategy for the existential player to satisfy the formula. In the remainder of this paper, given a QBF $\Pi.\psi$ containing an existential variable x, the function $f_x(U)$ denotes a Skolem function for x with the set of universal variables U that are outer to x in Π, as parameters.

To check that a Skolem set $\mathcal{F}$ is valid, it is necessary to substitute in ϕ all existential variables by their corresponding Skolem functions in $\mathcal{F}$ and check that the resulting propositional formula is valid. This can be done by a SAT solver. In practice, it also needs to be checked that a given Skolem function for x does not contain universal variables y_i with $y_i >_\Pi x$. This syntactic criterion can easily be checked. Thus while the satisfiability checking problem of QBF is PSPACE complete, checking validity of a Skolem set is in co-NP [5].

We conclude the preliminary section by introducing the concept of *asymmetric literal addition*.

Definition 2 (Asymmetric Literal Addition)**.** Given a QBF $\Pi.\psi$ and a clause C. The clause $\mathsf{ALA}(\psi, C)$ is the unique clause obtained by repeatedly applying the extension rule

$$C := C \cup \{\bar{l}\} \text{ if } \exists l_1, \dots, l_k \in C \text{ and } (l_1 \vee \dots \vee l_k \vee l) \in \psi$$

called *asymmetric literal addition* to C until fixpoint.

Asymmetric literal addition is indifferent with respect to the quantification type of the involved literals. Originally it was introduced for propositional logic in order to uniformly characterize preprocessing and inprocessing techniques [13]. It turned out that asymmetric literal addition is the basis for several powerful redundancy criteria which allow to safely add and delete clauses in propositional logic. As ALA is model preserving, for any QBF $\phi = \Pi.\psi \wedge \{C\}$ holds that $\phi \approx \phi[C/C']$ where $C' = \mathsf{ALA}(\psi \setminus \{C\}, C)$ [12].

III. Related Work

The importance of Skolem function generation for true QBFs has been acknowledged to be a vital problem. Yet for a long time, only solvers internally working with Skolemization like Skizzo [14] and squolem [15] as well as the BDD-based solver ebddres [16] were able to produce Skolem functions. All three solvers are not maintained any more and to best of our knowledge no recent solver is built based on internal Skolemization. Instead, two solving paradigms have shown to be successful over the last years: most solvers implement a variant of the *search-based* DPLL algorithm [6] with clause and cube learning which is closely related to the techniques found in state-of-the-art SAT solvers. Alternatively, *expansion-based* systems [17], [9], [10] are developed which use variable elimination and universal expansion for simplifying a formula. The latter techniques have been shown to be extremely powerful when used as preprocessing steps where they are not applied until completion but where they just transform the formula such that it becomes easier to solve for search-based solvers.

For solvers and tools which are able to produce term-resolution proofs, the approaches presented by Balabanov

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

TABLE I
THE QRAT PROOF SYSTEM

	Rule	Preconditions	Postconditions
(N1)	$\Pi.\psi \xrightarrow{\mathsf{ATE}(C)} \Pi.\psi\backslash\{C\}$	C is an asymmetric tautology	
(N2)	$\Pi.\psi \xrightarrow{\mathsf{ATA}(C)} \Pi'.\psi \cup \{C\}$	C is an asymmetric tautology	$\Pi' = \Pi\exists X$ with $X = \{x \mid x \in \mathsf{vars}(C), x \notin \mathsf{vars}(\Pi)\}$
(E1)	$\Pi.\psi \xrightarrow{\mathsf{QRATE}(C,l)} \Pi.\psi\backslash\{C\}$	$C \in \psi$, $\mathsf{Q}(\Pi, l) = \exists$ C has QRAT on l w.r.t. ψ	
(E2)	$\Pi.\psi \xrightarrow{\mathsf{QRATA}(C,l)} \Pi'.\psi \cup \{C\}$	$C \notin \psi$, $\mathsf{Q}(\Pi, l) = \exists$ C has QRAT on l w.r.t. ψ	$\Pi' = \Pi\exists X$ with $X = \{x \mid x \in \mathsf{vars}(C), x \notin \mathsf{vars}(\Pi)\}$
(U1)	$\Pi.\psi \cup \{C\} \xrightarrow{\mathsf{QRATU}(C,l)} \Pi.\psi \cup \{C\backslash\{l\}\}$	$l \in C$, $\mathsf{Q}(\Pi, l) = \forall$, $\bar{l} \notin C$, C has QRAT on l w.r.t. ψ	
(U2)	$\Pi.\psi \cup \{C\} \xrightarrow{\mathsf{EUR}(C,l)} \Pi.\psi \cup \{C\backslash\{l\}\}$	$l \in C$, $\mathsf{Q}(\Pi, l) = \forall$, $\bar{l} \notin C$, C has EUR on l w.r.t. ψ	

and Jiang [7] and presented by Goultiaeva et al. [8] can be applied to extract strategies from the proofs. With these works it became possible to generate certificates for search-based solvers.

For expansion-based solvers and tools, however, the situation is different. As soon as universal expansion is involved in the solving process, it remains an open question if and how it can be translated to resolution. Therefore, it is not possible to produce resolution proofs for expansion-based systems [11], what was especially problematic if a formula is only solvable by the application of universal expansion. In previous work [18], we showed how to produce partial certificates for the variables of the outermost quantifier block, but here only single variable assignments are involved. Janota et al. [19] proposed to use only techniques that can be translated into resolution in order to bring certification and Skolem function extraction to state-of-the-art preprocessing. However, then the preprocessor looses a lot of its power.

To avoid any restriction of the applicable techniques when certification is required, we introduced the QRAT proof system [12] which is able to capture universal expansion as well as all state-of-the-art preprocessing techniques by three simple rules which can be checked easily. The obvious question is how to extract Skolem functions from such proofs which we answer in this paper.

IV. QRAT: QUANTIFIED RESOLUTION ASYMMETRIC TAUTOLOGIES

The QRAT proof system which we introduced in [12] is the first proof system for QBFs which captures all preprocessing techniques as well as expansion-based solving. The rules of the proof system are shown in Table II. The basic idea is to use syntactic redundancy criteria to add, remove, or modify clauses until the truth value of the formula is known. Soundness of the rules is shown in [12], completeness follows from the fact that the QRAT proof system simulates resolution. Please note that for the sake of readability, we work with a definition consisting of six rules in this paper instead of the more compact three rule variant of our QRAT proof format [12], where (N1)+(E1), (N2)+(E2), as well as (U1)+(U2) form the three rules. By splitting up the original three rules in six rules, we do not gain any additional expressiveness, but it allows us a more focused view on the problem of extracting Skolem functions. As we will see only those rules are relevant which delete clauses in a satisfiability equivalence preserving manner. This applies to (E1) only. Further, this allows us to present the rules irrelevant for the Skolem function extraction in an intuitive by abstracting from the concrete technical details. For the complete formal definition of the proof system, we kindly refer to [12].

The rules (N1) and (N2) eliminate and respectively add asymmetric tautologies (AT). A clause C is an asymmetric tautology w.r.t. a QBF $\Pi.\psi$ iff $\mathsf{ALA}(\psi\backslash\{C\}, C)$ is a tautology. As the addition of asymmetric literals is model preserving, it holds that $\Pi.\psi \approx \Pi.\psi \cup C$ iff $\mathsf{ALA}(\psi\{C\}, C)$ is a tautology. Hence, the application of (N1) and (N2) is model preserving [12].

Rule (E1) and (E2) apply the QRAT redundancy criterion which is defined below.

For some intuition about QRAT consider the following scenario. Let $\Pi.\psi$ be a QBF, C a clause, and l a literal in C. We are interested in the situation that for every assignment satisfying ψ and falsifying C, it holds that all clauses $D_i \in \psi$ with $\bar{l} \in D_i$ are satisfied on literal $k \in D_i$ with $k \neq \bar{l}, k \leq_\Pi l$. As all clauses D_i with $\bar{l} \in D_i$ are satisfied by at least two literals if C is falsified by a variable assignment σ, the assignment can be modified by flipping the value of l such that C becomes satisfied while all D_i stay satisfied. Hence, the addition of C to ϕ preserves satisfiability and the deletion

of C to ϕ preserves unsatisfiability. The reasoning for QRAT is similar and uses the following three definitions.

Definition 3 (Outer Clause)**.** Let C be a clause occurring in QBF $\Pi.\psi$. The *outer clause* of C on literal $l \in C$, denoted by $\mathcal{O}(\Pi, C, l)$, is given by the clause $\{k \mid k \in C, k \leq_\Pi l, k \neq l\}$.

Definition 4 (Outer Resolvent)**.** Let C be a clause with $l \in C$ and D a clause occurring in QBF $\Pi.\psi$ with $\bar{l} \in D$. The *outer resolvent* of C with D on literal l w.r.t. Π, denoted by $\mathcal{R}(\Pi, C, D, l)$, is given by the clause $O \cup (C \setminus \{l\})$ if $Q(\Pi, l) = \forall$ and by $O \cup C$ if $Q(\Pi, l) = \exists$ assuming $O = \mathcal{O}(\Pi, D, \bar{l})$.

Definition 5 (Quantified Resolution Asymmetric Tautology (QRAT))**.** Given a QBF $\Pi.\psi$ and a clause C. Then C has QRAT on literal $l \in C$ with respect to $\Pi.\psi$ iff it holds for all $D \in \psi_{\bar{l}}$ that $\mathsf{ALA}(\psi \setminus \{C\}, R)$ is a tautology for the outer resolvent $R = \mathcal{R}(\Pi, C, D, l)$.

The rules (U1) and (U2) eliminate universal variable occurrences for which redundancy criteria ensure that on those the universal player will never be forced to use them to satisfy the clauses. In particular, if a clause C has QRAT on universal literal l w.r.t. to a QBF $\phi = \Pi.\psi$ with $C \in \psi$, then it can be shown that removing l from C preserves satisfiability. This rule subsumes universal pure literal elimination (i.e., literals occurring in one polarity), which is an indispensable rule for state-of-the-art QBF solvers. Finally, the rule (U2) allows the elimination of a universal literal by the means of extended universal reduction [12]. Universal reduction is part of the resolution calculus for QBFs. It removes a literal l from a non-tautological clause C iff C does not contain any existential literal occurring to the right of l in the prefix. From the game view this means that whenever the universal player has to assign l, he can immediately falsify the clause, because there is no existential literal left allowing the existential player to satisfy the clause. The idea behind extended universal reduction goes in a similar direction, but here existential literals to the right of l in the prefix are allowed if they have certain properties. As these properties are irrelevant for the remainder of this paper, we refer the reader to [12] for the details.

Example 1. Consider the true QBF $\Pi.\psi = \forall a\, \exists b, c.(a \vee b) \wedge (\bar{a} \vee c) \wedge (b \vee \bar{c})$. Clause $(a \vee c)$ has QRAT on c w.r.t. $\Pi.\psi$: the only clause that contains literal $\bar{c}$ is $(b \vee \bar{c})$, which produces the outer resolvent $(a \vee b \vee c)$. Since $\mathsf{ALA}(\psi \setminus \{(a \vee c)\}, (a \vee b \vee c)) = (a \vee \bar{a} \vee b \vee \bar{b} \vee c \vee \bar{c})$ is a tautology, QRATA can add $(a \vee c)$ to ψ. Now, consider a new existential variable d in the innermost quantifier block. The clause $(\bar{b} \vee c \vee d)$ has QRAT on c (and d) w.r.t. ψ. Adding $(\bar{b} \vee c \vee d)$ to ψ results in the true QBF $\forall a\, \exists b, c, d.(a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (b \vee \bar{c}) \wedge (\bar{b} \vee c \vee d)$.

Definition 6 (Outer Formula)**.** Let l be a literal occurring in QBF $\Pi.\psi$. The *outer formula* of l, denoted by $\mathcal{OF}(\Pi, \psi, l)$, is $\{\mathcal{O}(\Pi, D, \bar{l}) \mid D \in \psi, \bar{l} \in D\}$.

If a clause C has QRAT on $l \in C$ w.r.t. a QBF formula $\Pi.\psi$, we know that: 1) if the outer formula $\mathcal{OF}(\Pi, \psi, l)$ is falsified by an assignment then C is satisfied by that assignment; and 2) the outer formula $\mathcal{OF}(\Pi, \psi, l)$ is satisfied by an assignment, then l can be assigned to true, thereby satisfying C. We use this property of QRAT clauses to construct Skolem functions. Given a QBF formula $\Pi.\psi$, a valid Skolem set $\mathcal{F}$ for $\Pi.\psi$ and a clause C that has QRAT on $l \in C$ w.r.t. $\Pi.\psi$. We can now make a valid Skolem set $\mathcal{F}'$ for $\Pi.\psi \wedge \{C\}$ by updating the Skolem function $f_{\mathsf{var}(l)}(U)$ as follows. First, create a new variable y and make $f_y(U) := f_{\mathsf{var}(l)}(U)$. Second, replace $f_{\mathsf{var}(l)}(U)$ by the function stating that if $\mathcal{OF}(\Pi, \psi, l)$ evaluates to true then return polarity of l else return $f_y(U)$.

V. From Proof Validation to Skolem Functions

In earlier work [12], we showed how to validate QRAT proofs. In this section, we describe how to extend that method to obtain Skolem functions after a satisfaction proof, i.e., a proof for a true QBF, has been validated. We start with a brief discussion on how to validate satisfaction proofs. Afterwards, we explain how to integrate the extraction of Skolem functions into the checking. We end this section with a running example of the algorithm.

A. *Validating* QRAT *Proofs*

QRAT proofs are sequences of clause additions and deletions. They are build using three kind of lines: addition (N2+E2), deletion (N1+E1), and universal elimination (U1+U2). In the QRAT proof format, addition lines have no prefix and are unconstrained in the sense that one can add any clause at any point in the proof. Clause deletion lines have prefix "`d`", while universal elimination lines have prefix "`u`". Both are restricted in the following way: the clause after a "`d`" or "`u`" prefix must be either present in the original formula or as a clause added earlier in the proof. For satisfaction proofs, a universal elimination line can be replaced by a clause addition (N2) and clause deletion line (N1). We assume that all universal elimination lines have been replaced and ignore their existence for the remaining part of the paper.

Fig. 1 shows the basic algorithm to validate QRAT proofs. Let us ignore line 1, 2, 9, and 13 for the moment, because they are only required to produce Skolem functions. We loop over the clauses in the proof in the order a QBF solver or preprocessor added or removed clauses (line 3). The first unexamined clause is obtained from the proof together with its flag and pivot (line 4). The flag can be either `add` or `delete` (in the proof format no prefix or a "`d`" prefix, respectively). If the flag is `add`, no checking is required because this is a strengthening step. The new clause is simply added to ψ (line 11). Else, the clause will be removed (line 6). This elimination step needs to be validated. We check if the clause is logically implied by ψ by computing whether the clause is an asymmetric tautology (line 7). If that is not the case, the clause needs to have QRAT w.r.t. ψ (line 8), otherwise the proof is invalid (line 10). This procedure continues until all clauses in the proof have been processed. At this point, ψ should be empty, showing that the original formula is

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

satisfiability equivalent to the empty formula. If ψ is empty, the proof is valid (line 14), otherwise it is invalid (line 12).

```
     validateQRAT (QBF formula Π.ψ, QRAT proof P)
v1   let V = vars(P)
v2   initSkolem (Π, V)
v3   while P ≠ ∅ do
v4     ⟨flag, l, C⟩ := P.dequeue()
v5     if flag = delete then
v6       ψ := ψ \ {C}
v7       if ALA(ψ, C) is a tautology then continue
v8       else if C has QRAT on l ∈ C w.r.t. Π.ψ then
v9         addSkolem (Π.ψ, C, l)
v10      else return 'INVALID PROOF'
v11    else ψ := ψ ∪ {C}
v12  if ψ ≠ ∅ then return 'INVALID PROOF'
v13  finishSkolem (V)
v14  return 'VALID PROOF'
```

Fig. 1. Procedure to check QRAT proofs and output Skolem functions.

B. Extracting Skolem Functions

The basic QRAT validation algorithm can easily be enhanced to produce Skolem functions. In Fig. 1 this is shown by the lines 1 and 2 (initialization), line 9 (producing Skolem functions), and line 13 (termination). Notice that although the QRAT proof system uses six rules, recall Table II, the Skolem functions only depend on one of them, i.e., quantified resolution asymmetric tautology elimination (E1). The reason why only (E1) has be to taken into account is that the other rules either strengthen ψ or preserve logical equivalence.

Fig. 2 shows the procedures used to extract the Skolem functions. An important part of the extraction algorithm is the global array last which contains for each variable in the QRAT proof a pointer to the last variable on which its Skolem function depends. Initially, see *initSkolem*, $\mathsf{last}[x] := x$ for existential variables and $\mathsf{last}[x] := 0$ for universal variables, since there are no Skolem functions for universal variables. After the QRAT proof has been validated, for all variables x pointed to in the last array, a Skolem function $f_x(U)$ is added that is always true ($\top$) (see *finishSkolem*).

The real work is done in the *addSkolem* procedure. This algorithm is inspired on the solution reconstruction procedure for RAT proofs, the variant of QRAT for propositional (SAT) formulas [20]. The algorithm works as follows: pick an arbitrary assignment. Loop over all clauses in the RAT proof in reverse order. If a clause is falsified by the current assignment, flip the truth value of the pivot. In order to make this algorithm produce Skolem functions out of QRAT proofs, some changes have to be made. Most importantly, we need to respect the quantifier prefix, because Skolem functions may not depend on variables that are more inner w.r.t. the quantifier prefix.

The *addSkolem* procedure uses two sub-procedures of which the pseudo-code is not shown: $\mathsf{pol}(l)$ and $\mathsf{eval}(F)$. The procedure $\mathsf{pol}(l)$ simply returns the polarity of literal l, i.e., $\top$ if l is a positive literal and $\bot$ if l is a negative literal. The procedure $\mathsf{eval}(F)$ returns the clause set F under the current Skolem functions. It replaces each positive literal x with $f_{\mathsf{last}[x]}(U)$ and each negative literal $\bar{x}$ by $\neg f_{\mathsf{last}[x]}(U)$. For example, the expression $\mathsf{eval}((a \vee \bar{b}) \wedge (c))$ is replaced by $(f_{\mathsf{last}[a]}(U) \vee \neg f_{\mathsf{last}[b]}(U)) \wedge f_{\mathsf{last}[c]}(U)$.

At the end of Section IV, we discussed how to update Skolem functions when adding a QRAT clause C. This update step requires to evaluate the outer formula of the pivot. The outer formula can be large which in turn would make the Skolem functions large. Therefore, we first check whether we can avoid computing the outer formula. This can be done when C', a copy of C with all inner literals to the pivot removed, has QRAT as well. In that case, we only need to check whether the outer clause of C w.r.t. the pivot is falsified.

Now we have all elements to explain the *addSkolem* procedure. A new existential variable y is created (line 2). Afterwards, we compute C', a copy of C with all inner literals to the pivot are removed (line 3). If C' has QRAT on l w.r.t. $\Pi.\psi$ (line 4) then the Skolem function for the pivot becomes as shown in the pseudo-code on line 5. Otherwise, we need to compute the outer formula of the pivot and use it for the Skolem function (line 7). The procedure terminates by updating the last array using y (line 8).

```
     initSkolem (prefix Π, set of variables V)
iS1  foreach x ∈ V do
iS2    if Q(Π.x) = ∀ then last[x] := 0 else last[x] := x

     addSkolem (QBF formula Π.ψ, clause C, literal l)
aS1  let x be last[var(l)]
aS2  let y be a new existential variable
aS3  let C' := {k ∈ C | k ≤_Π l}
aS4  if C' has QRAT on l w.r.t. Π.ψ then
aS5    f_x(U) := if(eval(O(Π, C, l)) then f_y(U) else pol(l)
aS6  else
aS7    f_x(U) := if(eval(OF(Π, C, l))) then pol(l) else f_y(U)
aS8  last[var(l)] := y

     finishSkolem (set of variables V)
fS1  foreach x ∈ V do
fS2    if last[x] ≠ 0 then f_last[x](U) := ⊤
```

Fig. 2. Procedures to init, add, and finish Skolem functions.

C. Running Example

The true QBF below is used to illustrate how the extraction of Skolem functions from a QRAT proofs works:

$$\Pi.\psi := \exists a, b \forall x, \exists c.(a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (\bar{a} \vee x \vee c) \wedge (\bar{x} \vee \bar{c})$$

Fig. 3 shows how this formula looks in the QDIMACS format, which is used by most QBF solvers and preprocessers (left) and a QRAT proof for that formula (right). Note that in QRAT proofs, the pivot of clause C is the first literal appearing in the clause deletion line corresponding to C. The initialization

true QBF formula in DIMACS

```
p cnf 4 4
e 1 2 0
a 4 0
e 3 0
   1  2 0
  -1 -2 0
-1  3  4 0
  -3 -4 0
```

satisfaction QRAT proof

```
d     -2 -1 0
    2  3  4 0
d -1  3  4 0
d     1  2 0
d   2  3  4 0
d    -3 -4 0
```

Fig. 3. A true QBF (left) with a satisfaction proof (right). The formula and proof are spaced to improve readability. Proofs consist of two kind of lines: addition (no prefix) and deletion ("d " prefix). The formula and proof represent our running example in the DIMACS and QRAT format, respectively. Variables in both formats are numbers. The following mapping is used a corresponds to 1, b to 2, c to 3, and x to 4. Negative literals are shown as negative numbers.

of Skolem functions will assign the last array as follows: $\mathsf{last}[a] := a$, $\mathsf{last}[b] := b$, $\mathsf{last}[c] := c$, and $\mathsf{last}[x] := 0$.

The proof consists of the following steps. First, $C := (\bar{a} \vee \bar{b})$ is removed from ψ using $\bar{b}$ as pivot. Since C has no inner literals, $C' := C$ and consequently C' has QRAT on $\bar{b}$ w.r.t. the new ψ (from which C has been removed). We introduce a new existential variable b_1. Now the Skolem function $f_b(U)$ will be – with $\mathsf{eval}(\bar{a})$ replaced by $\neg f_a(U)$, and $\mathsf{pol}(\bar{b})$ by $\bot$:

$$f_b(U) := \quad \text{if}(\neg f_a(U))\,\text{then}\, f_{b_1}(U)\,\text{else}\,\bot$$

The second step in the proof is adding clause $(b \vee x \vee c)$ to ψ. Since this involves clause addition, no Skolem function is added. The third step is the most tricky one. Clause $C := (\bar{a} \vee x \vee c)$ is now removed using pivot $\bar{a}$ and checked for redundancy. Both x and c are inner to $\bar{a}$, so $C' := (\bar{a})$. C' does not have QRAT on $\bar{a}$ w.r.t. the new ψ (which now contains $(b \vee x \vee c)$, but no longer has $(\bar{a} \vee x \vee c)$). In this case the outer formula $F := (b)$. Although F is small, it can be almost as large as ψ in the worst case. Now, the Skolem function $f_a(U)$ will be – with $\mathsf{eval}(F)$ replaced by $f_{b_1}(U)$ and $\mathsf{pol}(a)$ by $\bot$:

$$f_a(U) := \quad \text{if}(f_{b_1}(U))\,\text{then}\,\bot\,\text{else}\, f_{a_1}(U)$$

The fourth step concerns the removal of $(a \vee b)$ using pivot a. This step is practically the same as the first step. Now $\mathsf{last}[a] = a_1$, so we compute for Skolem function $f_{a_1}(U)$:

$$f_{a_1}(U) := \quad \text{if}(f_{b_1}(U))\,\text{then}\, f_{a_2}(U)\,\text{else}\,\top$$

In the fifth step, $C := (b \vee x \vee c)$, which was added in step two, is removed. Again, the literals x and c are inner to b. This results in $C' := (b)$. In contrast to step three, this C' has QRAT on b w.r.t. the current ψ because ψ no longer contains any clause with literal $\bar{b}$.

$$f_{b_1}(U) := \quad \text{if}(\bot)\,\text{then}\, f_{b_2}(U)\,\text{else}\,\top$$

Finally, the last clause $(\bar{x} \vee \bar{c})$ is removed with pivot $\bar{c}$. The Skolem function $f_c(U)$ will be:

$$f_c(U) := \quad \text{if}(\neg x)\,\text{then}\, f_{c_1}(U)\,\text{else}\,\bot$$

After the QRAT proof has been validated, we call *finishSkolem* which does the following assignments: $f_{a_2}(U) := f_{b_2}(U) := f_{c_1}(U) := \top$. Using these Skolem functions, we can set the earlier Skolem functions to $f_a(U) := \bot$, $f_b(U) := \top$, and $f_c(U) := \neg x$.

VI. Implementation, Optimization and Validation

We enhanced our QRAT checking tool, called QRAT-trim, with Skolem function extraction capabilities.[2] The Skolem functions can be emitted as a propositional formula in DIMACS format or as an and-inverter-graph in AIGER format. This section describes some details about our implementation, optimizations and validation of Skolem function extraction.

A. Reducing the Size of the Outer Formula

The outer formula computed in line 7 of the *addSkolem* procedure (Fig. 2) is typically much larger than necessary and consequently makes Skolem functions larger than necessary. In order to produce smaller Skolem functions, we implemented the following optimization (using the notation in *addSkolem*): For all $D \in \psi$ with $\bar{l} \in D$, we compute the outer resolvent $R = \mathcal{O}(\Pi, D, \bar{l}) \cup C'$. We check whether $\mathsf{ALA}(\varphi, R)$ is a tautology and store all $\mathcal{O}(\Pi, D, \bar{l})$ for which the corresponding R is *not* a tautology. The alternative outer formula becomes the conjunction of these $\mathcal{O}(\Pi, D, \bar{l})$ together with the negation of $C' \setminus \{l\}$.

B. Value of Final Skolem Functions

We presented *finishSkolem* such that it assigns all Skolem functions $f_{\mathsf{last}[x]}(U) := \top$. However, for some variables, $f_{\mathsf{last}[x]}(U) := \bot$ is much more effective. We observed that the best truth value for final Skolem functions $f_{\mathsf{last}[x]}$ is based on the polarity of a literal that was a pivot for a QRAT check. For some variables x (typically a few hundred for each benchmark), there are QRAT checks with literal x as a pivot, but no QRAT checks with literal $\bar{x}$ as pivot (or the other way around). By assigning $f_{\mathsf{last}[x]}(U) := \top$ (or $f_{\mathsf{last}[x]}(U) := \bot$, respectively), and apply simplification, we obtain the Skolem functions $f_x(U) := \top$ (or $f_x(U) := \bot$, respectively).

C. Validation of Skolem Functions

Validating a set of Skolem functions consists of two checks. If both checks succeeds, the set of Skolem functions is valid. Let $\mathcal{F}$ be a set of Skolem functions for a QBF $\Pi.\psi$. The first check consists of substituting the existential variables in ψ by all the skolem functions in $\mathcal{F}$. The resulting formula is negated and checked by a SAT solver to be unsatisfiable.

The second check uses the AIG representation of the Skolem functions and Π to check that no input gate g_i (universal variable) influences the truth value of output gate g_o (existential variable) with $g_i >_\Pi g_o$. So no universal variable influences the truth of an inner-more existential variable.

Apart from implementing a tool that extracts Skolem functions from a QRAT proof, we also implemented a tool that

[2]The QRAT-trim version with Skolem function extraction is available on http://www.cs.utexas.edu/~marijn/skolem/.

checks whether the Skolem functions are correct. A tool, called CertCheck [21], has the same functionality but uses a more strict check for the second part, i.e, whether the truth of no existential variable x depends on the truth value of any variable (also existential) inner to x. This check is too restrictive to validate our Skolem functions.

Example 2. Consider the formula $\exists a \forall b \exists c.(a \vee b \vee c) \wedge (\bar{a} \vee \bar{c})$. A possible QRAT proof for this formula removes first $(a \vee b \vee c)$ with pivot c and afterwards $(\bar{a} \vee \bar{c})$ with pivot $\bar{a}$. The latter is allowed because after removing $(a \vee b \vee c)$ the prefix collapses to $\exists a, c$. Our procedure for extracting Skolem functions can result in $f_a(U) = \neg f_c(U)$ and $f_c(U) = \top$ (depending on which optimizations are used). Although the Skolem function for a depends on the Skolem function for c which is inner to a, the Skolem functions are correct because the Skolem function of a does not depend on a universal variable inner to a.

Our validation tool called cheskol uses the less restrictive dependency check and emits the result of substitution and negation as a formula in DIMACS format (after Tseitin encoding), the typical input format for SAT solvers.

TABLE II
STATISTICS OF EXTRACTING SKOLEM FUNCTIONS FROM QRAT PROOFS PRODUCED BY BLOQQER ON QBF EVAL 12 BENCHMARKS.

formula	sol-t	ext-t	tr-s	ch-t	qbc-s
c3_BMC_p1_k2	1.08	2.10	1777	0.04	71
counter_8	0.39	0.38	266	0.07	37
itc-b13-fixpoint-8	16.32	75.53	124969	337.80	14756
k_branch_n-16	6.12	137.40	13127	7.14	1296
k_branch_n-7	3.71	25.33	25449	337.41	7467
k_d4_n-10	1.08	3.23	5688	31.45	1988
k_d4_n-11	1.22	3.97	6401	42.20	2244
k_d4_n-14	1.58	7.36	9379	25.05	3158
k_d4_n-15	1.75	8.78	10536	86.55	3459
k_d4_n-20	2.53	18.94	16637	94.76	4885
k_d4_n-21	2.75	22.08	18442	140.58	5296
k_dum_n-10	0.17	0.42	262	0.06	62
k_dum_n-11	0.23	0.49	335	0.07	74
k_dum_n-12	0.26	0.47	342	0.07	63
k_dum_n-16	0.26	0.60	432	0.09	95
k_dum_n-20	0.33	0.85	657	0.18	158
k_dum_n-21	0.44	0.93	761	0.32	204
lights3_021_1_022	0.18	0.46	253	0.09	54
lights3_021_1_033	0.23	0.41	388	0.07	47
lights3_035_1_059	0.39	0.70	639	0.34	111
rankfunc0_unsigned_64	1.70	7.55	3244	36.69	4985
rankfunc16_unsigned_16	0.33	1.42	783	0.83	342
rankfunc24_signed_32	0.56	1.82	1049	10.37	902
rankfunc27_unsigned_32	0.44	0.80	1446	1.90	569
rankfunc52_signed_64	1.80	9.23	3652	363.51	5058
s3330_d2_s	10.60	19.23	122569	3.59	798
stmt137_903_911	0.39	0.54	1037	0.27	112
stmt1_629_630	0.72	1.20	1671	0.74	187
stmt17_99_98	1.22	2.69	3388	2.19	373
stmt27_584_603	0.42	0.63	975	0.27	132
stmt27_946_955	0.39	0.53	977	0.32	113
stmt41_118_131	0.36	0.52	774	0.68	166

sol-t/ext-t/ch-t: solving/extraction/checking time (sec)
tr-s/qbc-s: size of QRAT file/qbc file (kilobyte)

VII. EXPERIMENTAL EVALUATION

At the moment, our preprocessor bloqqer is the only tool able to produce QRAT proofs. As it is not a complete solver, we consider the true formulas of the benchmark suite from QBF Eval 12 which can be solved by bloqqer. We showed in earlier work [12] that bloqqer with and without QRAT proof logging solves the same instances. In the following, we evaluate how our Skolem function extraction algorithm performs on the formulas for which QRAT proofs are available. Table II shows the results of our Skolem function extraction tool, i.e., a modified version of QRAT-trim. We converted all our proofs to the QBC format in order to make a comparison with other tools more clear. Notice that the extracted Skolem functions are typically smaller than the used QRAT proofs. We validated our Skolem functions using cheskol which checks the dependencies and computes whether the Skolem functions imply the formula using the SAT solver lingeling [22].

Janota et al. [19] restricted bloqqer such that it is able to produce resolution (RES) proofs. However, several preprocessing techniques are not supported by that approach. As a consequence, their modified version of bloqqer solves less formulas (only 22 out of 32). If a formula cannot be solved by bloqqer they use the solver depQBF to compute a resolution proof of the simplified formula. They merge the certificate (resolution proof) obtained from depQBF with the partial certificate obtained from their bloqqer. The results of resolution-based approaches, our approach and some older tools [15], [16], [14] that are shown in Table III. The only tool that has comparable performance compared to our bloqqer+QRAT approach is bloqqer+RES+depQBF — although it cannot solve four of the harder benchmarks in the test suite.

A. Comparing the Size of Skolem Functions

If all preprocessing techniques are turned on, the average size of the Skolem functions produced by bloqqer+QRAT is larger than those from bloqqer+RES+depQBF. Recall that bloqqer+RES+depQBF does not support several preprocessing techniques. Consequently, unsupported techniques such as covered clause elimination (QCCE) [23] are turned off. Although bloqqer+QRAT supports QCCE, using it has a negative impact on the size of Skolem functions. On harder benchmarks, the Skolem functions are about four times larger due to this technique. Hence, turning QCCE off reduces the size of Skolem functions significantly — at the cost of solving one formula less (s3330_d2_s).

For a fair comparison between the size of Skolem functions produced by both approaches, we turn off QCCE for bloqqer+QRAT as done by bloqqer+RES+depQBF. The AIG files of the bloqqer+QRAT approach are converted into QBC certificates to have the same file format. The scatter plot shown in Fig. 4 illustrates that the Skolem functions extracted by bloqqer+QRAT are smaller than those produced by bloqqer+RES+depQBF, especially for the harder benchmarks.

TABLE III
RESULTS OF SKOLEM FUNCTION PRODUCING TOOLS ON QBF EVAL 12

solver	sol-#	sol-t	ch-#	ch-t	cer-s
bloqqer+QRAT	32	1	32	47	1851
bloqqer+RES	22	1	22	1	861
bloqqer+RES+depQBF	28	113	27	13	1040
depQBF	2	843	2	1	224
ebdd	15	491	7	118	409479
squolem	16	465	16	2	382
sKizzo	23	275	23	1	108750

sol-#: # solved formulas, sol-t: avg. solving time (s),
ch-#: checked certificates, ch-t: avg. checking time (s)
cer-s: avg. certificate size (kilobyte)

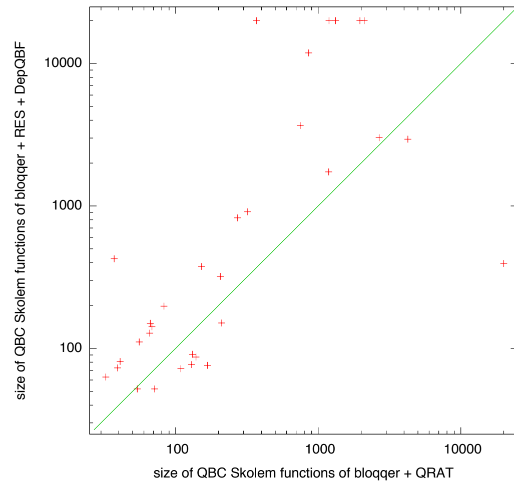


Fig. 4. Comparison between the size (in kilobyte) of QBC Skolem functions produced by the bloqqer+QRAT and bloqqer+RES+depQBF approaches on QBF Eval 2012 benchmarks. Above the line means that bloqqer+QRAT produces smaller files. Unsolved formulas are shown as a 20,000 kb file.

VIII. CONCLUSION AND FUTURE WORK

The QRAT proof system is the first framework that allows verification of all preprocessing techniques for QBFs. We developed an algorithm that extracts Skolem functions of QRAT proofs. Hence, the techniques presented in this paper allow us to obtain Skolem functions for all QBF preprocessing techniques. Moreover, these Skolem functions are smaller than those produced by alternative approaches – a very favorable property for many synthesis applications.

We expect that the produced Skolem functions can be further reduced in size: we applied the circuit simplification tool ABC [24] on the AIGs representing the set of Skolem functions and noticed a significant reduction. However, the simplified Skolem functions are not necessarily valid, as ABC is not aware of the dependency restrictions. In future, we want to consider Skolem function reduction by circuit simplification while taking into account the dependencies.

ACKNOWLEDGEMENTS

This work was supported by the Austrian Science Fund (FWF) through the national research network RiSE (S11408-N23), Vienna Science and Technology Fund (WWTF) under grant ICT10-018, DARPA contract number N66001-10-2-4087 and National Science Foundation grant no. CCF-1153558.

REFERENCES

[1] R. Bloem, R. Könighofer, and M. Seidl, "SAT-Based Synthesis Methods for Safety Specs," in *VMCAI*, ser. LNCS, vol. 8318. Springer, 2014.
[2] M. Benedetti and H. Mangassarian, "QBF-Based Formal Verification: Experience and Perspectives," *JSAT*, vol. 5, no. 1-4, pp. 133–191, 2008.
[3] H. Kleine Büning and U. Bubeck, "Theory of Quantified Boolean Formulas," in *Handbook of Satisfiability*, 2009.
[4] C. Ansótegui, C. P. Gomes, and B. Selman, "The achilles' heel of qbf," in *AAAI*. AAAI Press / The MIT Press, 2005, pp. 275–281.
[5] H. Kleine Büning, K. Subramani, and X. Zhao, "Boolean functions as models for quantified boolean formulas," *J. Aut. Reas.*, vol. 39, no. 1, 2007.
[6] E. Giunchiglia, P. Marin, and M. Narizzano, "Reasoning with quantified boolean formulas," in *Handbook of Satisfiability*. IOS Press, 2009, vol. 185, pp. 761–780.
[7] V. Balabanov and J.-H. R. Jiang, "Resolution Proofs and Skolem Functions in QBF Evaluation and Applications," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 149–164.
[8] A. Goultiaeva, A. Van Gelder, and F. Bacchus, "A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas," in *IJCAI*. IJCAI/AAAI, 2011, pp. 546–553.
[9] A. Biere, "Resolve and expand," in *SAT (Selected Papers)*, ser. LNCS, vol. 3542. Springer, 2004, pp. 59–70.
[10] A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for QBF," in *CADE 2011*, ser. LNCS, vol. 6803. Springer, 2011.
[11] M. Janota and J. Marques-Silva, "On Propositional QBF Expansions and Q-Resolution," in *SAT 2013*, ser. LNCS, vol. 7962. Springer, 2013, pp. 67–82.
[12] M. J. H. Heule, M. Seidl, and A. Biere, "A Unified Proof System for QBF Preprocessing," in *IJCAR 2014*, ser. LNCS, vol. 8562. Springer, 2014, pp. 91–106.
[13] M. J. H. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for CNF formulas," in *LPAR-17*, ser. LNCS, vol. 6397. Springer, 2010, pp. 357–371.
[14] M. Benedetti, "Skizzo: A suite to evaluate and certify QBFs," in *CADE-20*, ser. LNCS, vol. 3632. Springer, 2005, pp. 369–376.
[15] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. Wintersteiger, "A first step towards a unified proof checker for QBF," in *SAT 2007*, ser. LNCS. Springer, 2007, vol. 4501, pp. 201–214.
[16] T. Jussila, C. Sinz, and A. Biere, "Extended resolution proofs for symbolic sat solving with quantification," in *SAT*, ser. LNCS, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 54–60.
[17] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, "Solving QBF with Counterexample Guided Refinement," in *SAT 2012*, ser. LNCS, vol. 7317. Springer, 2012.
[18] R. Könighofer and M. Seidl, "Partial witnesses from preprocessed quantified boolean formulas," in *DATE*. IEEE, 2014, pp. 1–6.
[19] M. Janota, R. Grigore, and J. Marques-Silva, "On QBF Proofs and Preprocessing," in *LPAR*, ser. LNCS, vol. 8312. Springer, 2013.
[20] M. Järvisalo, M. J. H. Heule, and A. Biere, "Inprocessing rules," in *IJCAR*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.
[21] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere, "Resolution-Based Certificate Extraction for QBF," in *SAT 2012*, ser. LNCS, vol. 7317, 2012, pp. 430–435.
[22] A. Biere, "Lingeling, Plingeling and Treengeling entering the SAT competition 2013," in *Proceedings of SAT Competition 2013*, 2013.
[23] M. J. H. Heule, M. Järvisalo, and A. Biere, "Covered clause elimination," in *LPAR-17-short*, ser. EPiC Series, A. Voronkov, G. Sutcliffe, M. Baaz, and C. Fermüller, Eds., vol. 13. EasyChair, 2013, pp. 41–46.
[24] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/.

Small Inductive Safe Invariants

Alexander Ivrii
IBM Research
alexi@il.ibm.com

Arie Gurfinkel
Software Engineering Institute
http://arieg.bitbucket.org

Anton Belov
Synopsys
http://anton.belov-mcdowell.com

***Abstract*—Computing minimal (or even just small) certificates is a central problem in automated reasoning and, in particular, in automated formal verification. For example, Minimal Unsatisfiable Subsets (MUSes) have a wide range of applications in verification ranging from abstraction and generalization to vacuity detection and more. In this paper, we study the problem of computing minimal certificates for safety properties. In this setting, a certificate is a set of clauses Inv such that each clause contains initial states, and their conjunction is safe (no bad states) and inductive. A certificate is minimal, if no subset of Inv is safe and inductive. We propose a two-tiered approach for computing a Minimal Safe Inductive Subset (MSIS) of Inv. The first tier is two efficient approximation algorithms that under- and over-approximate MSIS, respectively. The second tier is an optimized reduction from MSIS to a sequence of computations of Maximal Inductive Subsets (MIS). We evaluate our approach on the HWMCC benchmarks and certificates produced by our variant of IC3. We show that our approach is several orders of magnitude more effective than the naïve reduction of MSIS to MIS.**

I. INTRODUCTION

Computing minimal (or even just small) certificates is a central problem in automated reasoning, and, in particular, in Model Checking. For reachability, the certificates take the form of counterexamples. It is widely believed that small counterexamples are the key to success of Model Checking in practice, as they increase user comprehension and provide better fault localization. In SAT-based Bounded Model Checking (BMC), the certificates for bounded safety (i.e., absence of counterexamples bounded by a given fixed length) correspond to unsatisfiable subsets. Minimal Unsatisfiable Subsets (MUSes) have a wide range of applicability. For example, they are a key ingredient in Proof-Based Abstraction [1], and have also been used to improve user's comprehension of verification results through vacuity [2]. For Unbounded Model Checking (or unreachability) the certificates are represented by *safe inductive invariants*. A recent trend, borrowing from the breakthroughs in Incremental Inductive Verification (such as IMC [3], IC3 [4], and PDR [5]), is to represent such invariants by a set of simple lemmas. In this paper, we study the problem of efficiently minimizing the set of such lemmas, and especially constructing a *minimal safe inductive subset* of a given safe inductive invariant. We focus on the algorithmic aspects of the problem and on empirical evaluation, and leave exploring the numerous potential applications for future work.

Throughout the paper, we assume that all formulas are in CNF and that a safe inductive invariant is represented by a set of clauses Inv such that each clause contains the initial states, and their conjunction is invariant under the transition relation and does not contain any bad states. The set Inv is minimal, called *Minimal Safe Inductive Invariant (MSIS)*, if, in addition to being safe and inductive, no subset of Inv is safe and inductive.

In this paper, we make the following contributions. First, in Section III, we show that computing an MSIS is reducible to a sequence of computations of Maximal Inductive Subset (MIS). While this yields a simple-to-implement algorithm, we show that it is not efficient. Second, we propose a two-tiered algorithm. The first tier, described in Section IV, consists of two approximation algorithms. The first algorithm under-approximates an MSIS by identifying the necessary clauses that are shared between all MSISs. The second, uses a sequence of MUS computations to over-approximate an MSIS. While these algorithms do not guarantee minimality, they can be used as an effective pre-processing step. The second tier, described in Section V, consists of two alternative optimized reductions from MSIS to MIS. The key idea is to combine the basic MSIS to MIS reduction with some of the pre-processing techniques to reduce the number of redundant SAT calls in each MIS computation. Third, we evaluate all of the algorithms on the benchmarks from the Hardware Model Checking Competition. We show that our ultimate algorithm that combines pre-processing and optimizations is several order of magnitude faster than the naïve approach. Furthermore, we show that the technique is extremely effective at reducing the size of the certificate, compared to the certificate produced by our custom variant of IC3.

To our knowledge, the problem of computing MSIS is not widely studied in SAT-based Model Checking (as opposed to computing minimal counterexamples or minimal unsatisfiable subsets). The only alternative solution is proposed by Bradley et al. [6] in the context of `FAIR` algorithm, which is similar to our base algorithm in Section III. However, we show that it does not scale in our context. On the other hand, we believe that efficient algorithms for computing MSIS are just as important as efficient algorithms for computing minimal unsatisfiable subsets, and they are necessary for extending many of the applications (in particular vacuity and abstraction) from BMC to Unbounded Model Checking. We believe that our work lays the foundation for numerous applications of small safety certificates in SAT-based Model Checking.

II. PRELIMINARIES

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. It is often convenient to treat a clause

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001309.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Input: $P = (Init, Tr, Bad)$, CNF $\mathcal{L}$
Output: $Inv \subseteq \mathcal{L}$ the MIS of $\mathcal{L}$ relative to P

```
1 Inv ← L
2 forever do
3 |  let R = {d ∈ Inv | (Inv ∧ Tr) ⇏ d'}
4 |  if R ≠ ∅ then
5 |  |  Inv ← Inv \ R
```

Fig. 1. A generic MIS algorithm.

Input: $(Init, Tr, Bad)$, safe inductive invariant Inv_o
Output: A minimal safe inductive subset $Inv \subseteq Inv_o$

```
1 Inv ← Inv_o ; W ← Inv_o
2 while W ≠ ∅ do
3 |  c ← a clause from W ; W ← W \ {c}
4 |  X ← MIS((Init, Tr, Bad), Inv \ {c})
5 |  if (X ⇒ ¬Bad) then Inv ← X
```

Fig. 2. A naïve MSIS algorithm for Minimal Safe Inductive Subset.

as a set of literals, and a CNF as a set of clauses. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

A variable assignment is a map $\sigma : \mathcal{V} \to \{\top, \bot\}$ that assigns $\top$ or $\bot$ to every variable in $\mathcal{V}$. A clause c is satisfied by an assignment σ if $\sigma(\ell) = \top$ for a literal $\ell \in c$. A CNF formula F is satisfied by σ if σ satisfies every clause in F. A CNF formula is SAT if there exists an assignment that satisfies it and is UNSAT otherwise.

A SAT-solver is a complete decision procedure for propositional formulas in CNF. We assume that the reader is familiar with the basic interface of an incremental solver. We use the following API: (a) `Sat_Add`(φ) adds clauses corresponding to the formula φ to the solver; (b) `Sat_Checkpoint`$()$ saves the current state of the solver; (c) `Sat_Rollback`$()$ restores the solver to the previously saved state.

Let F be an UNSAT CNF formula. A *minimal unsatisfiable subset (MUS)* of F is a subset of clauses $U \subseteq F$ such that U is UNSAT, and for every clause $c \in U$, $U \setminus \{c\}$ is SAT. There are many efficient algorithms for computing an MUS [7]–[9]. In the paper, we write `Sat_Mus`(F) for a call to an unspecified MUS algorithm. We assume that the MUS is always computed relative to the clauses already added to the solver using `Sat_Add`.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (Init, Tr, Bad)$, where $Init(\mathcal{V})$ and $Bad(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $Tr(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that $Init$ and Tr are in CNF, and that Bad is a single literal.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$Init(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(\vec{v}_i, \vec{v}_{i+1}) \right) \wedge Bad(\vec{v}_n) \qquad (1)$$

P is UNSAT (or SAFE) iff there exists a formula $Inv(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$Init(\vec{v}) \Rightarrow Inv(\vec{v}) \qquad Inv(\vec{v}) \wedge Tr(\vec{v}, \vec{v}') \Rightarrow Inv(\vec{v}') \qquad (2)$$

$$Inv(\vec{v}) \Rightarrow \neg Bad(\vec{v}) \qquad (3)$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*. Without loss of generality, we assume that $\neg Bad \in Inv$.

Throughout the paper, we fix a problem $P = (Init, Tr, Bad)$. Let $\mathcal{L}$ be a formula in CNF. A *maximal inductive subset (MIS)* of $\mathcal{L}$ relative to P is the largest subset $Inv \subseteq \mathcal{L}$ that satisfies (2). There are several algorithms for computing MIS [10]–[12]. A generic MIS algorithm is shown in Fig. 1, in which we first set Inv to $\mathcal{L}$, and then we repeatedly remove those clauses $R \subseteq Inv$ that fail to be inductive relative to Inv. We write `MIS`$(\mathcal{L})$ for a call to an MIS algorithm.

III. Minimal Safe Inductive Subset

Fix a safety verification problem $P = (Init, Tr, Bad)$, and let Inv be a safe inductive invariant of P in CNF. A subset of clauses $S \subseteq Inv$ is called a *safe inductive* subset of Inv relative to P if S is inductive and safe. S is *minimal* if any subset of S is either not safe or not inductive. In this section, we give a basic algorithm to compute a minimal safe inductive subset (MSIS) of a safe inductive invariant in CNF.

The algorithm is shown in Fig. 2. It works by a repeated application of the MIS algorithm. The input is a safety problem and a safe inductive invariant Inv_o. The algorithm keeps a work-set $\mathcal{W}$ of yet unprocessed elements of Inv_o. In each iteration of the loop, a clause $c \in \mathcal{W}$ from the work-set (line 3) is removed, and an MIS algorithm is used to compute the maximal inductive subset X of $Inv \setminus \{c\}$ (line 4). If X is also safe, then X represents a smaller safe inductive invariant of Inv (not containing c and possibly some additional clauses), and so Inv is replaced by X (line 5). Otherwise, c must belong to an MSIS of Inv_o. The algorithm terminates when there are no more unprocessed clauses, at which point we claim that Inv is an MSIS of Inv_o. The fact that Inv remains safe follows from the fact that the initial invariant Inv_o was safe and that each update of Inv maintains that. For the sake of contradiction, suppose that Inv is not minimal, i.e. that there is a minimal safe inductive invariant $Inv_m \subsetneq Inv$. Take any clause $c \in Inv \setminus Inv_m$. Consider the iteration of the loop corresponding to the removal of c from $\mathcal{W}$ and let Inv_1 represent Inv on that iteration. Since c was not removed from Inv_1, the maximal inductive subset of $Inv_1 \setminus \{c\}$ is not safe. On the other hand, Inv_m is a safe inductive subset of $Inv \setminus \{c\}$ and hence of $Inv_1 \setminus \{c\}$, leading to a desired contradiction.

While this naïve algorithm is simple, it is not efficient. In our experience, the calls to MIS are the bottleneck. Furthermore, the algorithm makes a lot of redundant calls because it does not take into account the dependency between clauses. Often, the inductive clauses occur in a group such that removing any one of the clauses makes the MIS of the result unsafe. In the rest of the paper, we propose two significant improvements. First, in Section IV, we give efficient algorithms to under- and over-approximate MSIS. While these algorithms

Input: $(Init, Tr, Bad), Inv, \mathcal{N}_o \subseteq Inv$ s.t. $\neg Bad \in \mathcal{N}_o$
Output: safe necessary set $\mathcal{N}$ s.t. $\mathcal{N}_o \subseteq \mathcal{N} \subseteq Inv$

```
1  φ ← (⋀_{c∈N_o} c) ∧ (⋀_{c∈Inv\N_o} a_c ⇔ c) ∧
2      (∑_{c∈Inv\N_o} ā_c ≤ 1) ∧ Tr
3  Sat_Add(φ)
4  N ← N_o ; W ← N_o
5  while W ≠ ∅ do
6  |  d ← a clause from W ; W ← W \ {d}
7  |  while φ ∧ ¬d′ is SAT (with model M) do
8  |  |  let c ∈ Inv \ N be a clause s.t. M ⊨ (a_c = 0)
9  |  |  Sat_Add(a_c)
10 |  |  N ← N ∪ {c} ; W ← W ∪ {c}
```

Fig. 3. NEC algorithm.

do not necessarily compute a minimal set, they are used as effective pre-processing steps. Second, in Section V, we give a more efficient variant of MSIS that attempts to minimize the amount of wasted work in each iteration and show how to combine it with the over- and under-approximating algorithms. Our results in Section VI show that this achieves orders of magnitude improvements in performance.

IV. Approximating SIS

In this section, we present two algorithms to approximate a MSIS of a given inductive invariant Inv. The first algorithm, called NEC, under-approximates an MSIS by identifying a set of clauses that must be included in any safe inductive subset. The second algorithm, called FEAS, over-approximates an MSIS by removing clauses that do not belong to some SIS. Throughout the section, we fix a verification problem $P = (Init, Tr, Bad)$ and let Inv be a safe inductive invariant of P.

A. Necessary Under-Approximation

A clause $c \in Inv$ is called *safe necessary* (or *necessary* for short) if c is included in *every* MSIS of Inv. While computing all necessary clauses is expensive, they can be approximated by the set NEC defined as the smallest subset of Inv that satisfies the following recursive definition:

$$\neg Bad \in NEC$$
$$\forall c \in Inv, d \in NEC \cdot (Inv \setminus \{c\} \wedge d \wedge Tr \not\Rightarrow d') \Rightarrow c \in NEC$$

That is, NEC contains the $\neg Bad$ clause, and all clauses that are necessary to ensure that other clauses in NEC remain inductive. It is easy to show by induction that if a clause $c \in NEC$ then c is safe necessary. However, NEC does not contain all necessary clauses. For example, consider the problem $P_1 = (Init_1, Tr_1, Bad_1)$ and Inv_1, where

$$Init_1 = Inv_1 \equiv x \wedge y \wedge z \quad (4)$$
$$Tr_1 \equiv x' = y \wedge y' = x \wedge z' = x \vee y \quad (5)$$
$$Bad_1 \equiv \neg z \quad (6)$$

Inv_1 is a MSIS of itself. Thus, all of its clauses are necessary. However, $NEC_1 = \{z\}$ because

$$x \wedge z \wedge Tr_1 \Rightarrow z' \qquad y \wedge z \wedge Tr_1 \Rightarrow z'$$

Input: $(Init, Tr, Bad), Inv_o, \mathcal{N} \subseteq Inv_o$ s.t. $\neg Bad \in \mathcal{N}$
Output: A safe inductive set Inv s.t. $\mathcal{N} \subseteq Inv \subseteq Inv_o$

```
1  Inv ← N, W ← N
2  while W ≠ ∅ do
3  |  Sat_Checkpoint()
4  |  φ ← Inv ∧ Tr ∧ (⋁_{c∈W} ¬c′)
5  |  Sat_Add(φ)
6  |  W ← Sat_Mus(Inv_o \ Inv)
7  |  Inv ← Inv ∪ W
8  |  Sat_Rollback()
```

Fig. 4. FEAS algorithm.

The set NEC can be computed efficiently using an incremental SAT solver, as shown in the algorithm in Fig. 3. The algorithm takes as input a verification problem, an inductive invariant Inv, and a starting subset $\mathcal{N}_o$ of Inv of safe necessary clauses. We require that $\neg Bad$ is in $\mathcal{N}_o$, but it is possible for $\mathcal{N}_o$ to include additional clauses as well (the value of this will become clear in Section V-C). The output of the algorithm is a possibly enlarged safe necessary subset $\mathcal{N}$ of Inv.

The algorithm starts by creating a Boolean formula φ (line 1) consisting of the following components:

- The clauses $c \in \mathcal{N}_o$.
- For each clause $c \in Inv \setminus \mathcal{N}_o$, we introduce a new variable a_c and clauses for $c \Leftrightarrow a_c$ (so that c is satisfied if and only if a_c evaluates to 1).
- Clauses for the at-most-one constraint over the negations of the variables a_c. In practice, we implement such constraints using a sequential counter construction [13].
- The transition relation clauses Tr.

It maintains a work-set of yet unprocessed elements of $\mathcal{N}$ in $\mathcal{W}$. In each iteration of the outermost loop (line 5), a clause $d \in \mathcal{N}$ is selected and tested using the SAT query shown on line 7. Note that $\neg d'$ is passed via assumptions interface. Suppose that this query is satisfiable. We claim that in the satisfying assignment exactly one of the variables a_c is assigned to 0. Indeed, since $Inv \wedge Tr \Rightarrow d'$, not all a_c can be 0. On the other hand, assigning more than one a_c to 0 is prohibited by the at-most-one constraint. Letting $c \in Inv \setminus \mathcal{N}$ be the corresponding clause, we obtain that $Inv \setminus \{c\} \wedge d \wedge Tr \not\Rightarrow d'$, and c can be added to NEC. This is accomplished on lines 8–10 by marking c as necessary and permanently setting a_c to 1. The algorithm terminates when all of the necessary clauses have been processed. The algorithm makes at most $2|\mathcal{N}|$ SAT queries: one satisfiable query for each new clause in $\mathcal{N}$ and one unsatisfiable query for each clause in $\mathcal{N}$.

B. Feasible Over-Approximation

Given two subsets $C, D \subseteq Inv$, D is *inductively supported* (*supported* for short) by C iff C is a set such that $C \wedge D \wedge Tr \Rightarrow D'$. That is, D is inductive relative to C. If D is supported by C, then C inductively supports D. Given a safe inductive invariant Inv_o, it is possible construct a SIS Inv of Inv_o by first adding $\neg Bad$ to Inv, and then, repeatedly, adding to Inv supporting clauses of Inv until fix-point. Note that the fix-point always exists. In the worst case, $Inv = Inv_o$.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

An optimized implementation of this idea is shown in Fig. 4. In addition to Inv, we maintain a work-set $\mathcal{W} \subseteq Inv$ of clauses which are not yet supported. On line 1, we initialize both Inv and $\mathcal{W}$ to $\mathcal{N}$ (which includes $\neg Bad$ and possibly some additional clauses as well). Let us consider one iteration of the loop (lines 3-8). Our goal is to support $\mathcal{W}$ by including in Inv as few additional clauses as possible and we achieve it by a reduction to Sat_Mus. Let $\varphi = Inv \wedge Tr \wedge \neg(\bigvee_{c \in \mathcal{W}} \neg c')$. Since $\mathcal{W}$ can be supported by including *all* of the clauses in $Inv_0 \setminus Inv$, the formula $(Inv_0 \setminus Inv) \wedge \varphi$ is unsatisfiable. Thus, the set of clauses required to support $\mathcal{W}$ can be computed as Sat_Mus$(Inv_0 \setminus Inv)$ (with respect to φ). After this set is found, we include it in Inv (line 7) and by induction this set represents exactly the set of clauses of Inv not known to be supported. If empty, then Inv is already a SIS, and the algorithm terminates. The algorithm makes at most $|F|$ queries to Sat_Mus (and much fewer in practice).

We remark that even though we always choose a minimal set of clauses to be added to Inv, the overall algorithm does not necessary produce a MSIS. We illustrate this using the following example. Consider the problem $P_2 = (Init_2, Tr_2, Bad_2)$ and Inv_2, where

$$Init_2 = Inv_2 \equiv x \wedge y \wedge z \quad (7)$$

$$Tr_2 \equiv x' = y \wedge y' = y \wedge z' = x \vee y \quad (8)$$

$$Bad \equiv \neg z \quad (9)$$

It is easy to see that $\{z\}$ is not inductive, but can be supported by either x or y. Suppose that x is chosen and is included to F. Since $\{x\}$ itself is not supported, the next iteration will include y as well, ending up with $F = Inv_2$. However, the MSIS of Inv_2 is $\{y, z\}$.

We conclude this section with several observations on the interaction between NEC and FEAS algorithms. First, we have found that running FEAS after NEC produces tighter over-approximations and takes less time on average then running FEAS alone. This can be explained, as illustrated by the example above, by the fact that FEAS heavily depends on the order in which clauses are added to Inv. On the other hand, NEC marks the necessary clauses that must be eventually included in any SIS. Thus, FEAS makes better choices when started with those clauses upfront and is faster on average. Second, for a similar reason, we have found that the effort spent on finding a *minimal* set $\mathcal{W}$ to be incrementally added to Inv also pays off – both in terms of the quality of the final over-approximation and the time spent by the algorithm. Finally, Bradley et al. [6] suggest to over-approximate a SIS by computing a "global" unsatisfable core of $Inv_o \wedge Tr \wedge \neg Inv_o'$ by minimizing the set of clauses of Inv_o required for unsatisfiability. We have not found this approach useful, even with the MUS version of the computation. In fact, on our benchmarks it seems that there are large sets of clauses which can be removed from Inv_o but which are required to support themselves. In such a case, the global approach keeps these clauses in the over-approximation, while the iterative approach has a good chance for removing them.

V. Minimal Inductive Safe Invariant

In this section, we present two algorithms for finding a minimal inductive safe subset of a given safe inductive

Input: $(Init, Tr, Bad), Inv_o, \mathcal{N}_o \subseteq Inv_o$ s.t. $\neg Bad \in \mathcal{N}_o$
Output: An MSIS $Inv \subseteq Inv_o$

```
1  Inv ← N_o ; W ← Inv_o \ N_o
2  while W ≠ ∅ do
3    c ← a clause from W
4    W ← W \ {c} ; U ← W
5    forever do
6      let R = {d ∈ Inv ∪ U | (Inv ∧ U ∧ Tr) ⇏ d'}
7      if R = ∅ then
8        W ← U ; break
9      else if R ∩ Inv ≠ ∅ then
10       Inv ← Inv ∪ {c} ; break
11     else U ← U \ R
```

Fig. 5. An optimized SIS algorithm (OptMSIS).

invariant Inv. Our first algorithm (Section V-A) is a simple yet powerful optimization of the basic algorithm from Fig. 2 which identifies the necessary clauses as soon as possible. Our second algorithm (Section V-B) additionally exploits the support dependency between different clauses in a MSIS and avoids performing redundant computations as much as possible.

A. Optimized MSIS algorithm

The OptMSIS algorithm is shown in Fig. 5. As before, the input is a verification problem, an initial safe inductive invariant Inv_o, and a subset $\mathcal{N}_o$ of safe necessary clauses of Inv_o. The output is a minimal safe inductive invariant Inv. We maintain two sets of clauses Inv and $\mathcal{W}$ such that the while-loop satisfies the following invariants: (1) $Inv \cup \mathcal{W}$ is a safe inductive invariant, and (2) Inv is safe necessary for $Inv \cup \mathcal{W}$. Initially, $Inv = \mathcal{N}_o$ and $\mathcal{W} = Inv_o \setminus Inv$. Intuitively, the algorithm proceeds by selecting a clause $c \in \mathcal{W}$ and either deducing that c is safe necessary (adding it to Inv) or finding a safe inductive subset of $Inv \wedge \mathcal{W}$ that does not contain c (shrinking $\mathcal{W}$ accordingly). The algorithm terminates when $\mathcal{W} = \emptyset$ at which point Inv is indeed a minimal safe inductive invariant.

In more details, on each iteration of the while-loop we select a clause $c \in \mathcal{W}$, remove it from $\mathcal{W}$, and denote the resulting set by $\mathcal{U}$ (lines 3-4). Next, on each iteration of the inner loop, we compute the set of clauses $R \subseteq Inv \cup \mathcal{U}$ that are no longer supported. On one hand, if $R = \emptyset$, then $Inv \cup \mathcal{U}$ remains a SIS, which means that we have succeeded in removing c (and possibly some other clauses) from $\mathcal{W}$, in which case we update $\mathcal{W}$ and proceed with the next unprocessed clause (lines 7-8). On the other hand, if $R \cap Inv \neq \emptyset$, then one of the necessary clauses in Inv becomes unsupported, in which case we conclude that c must be included in any MSIS of $Inv \cup \mathcal{W}$, mark c as necessary, and proceed with the next unprocessed clause (lines 9-10). Finally, if all of the necessary clauses in Inv remain supported but $R \neq \emptyset$, then the clauses in R cannot be part of any SIS of $Inv \cup \mathcal{U}$, and so we remove these clauses from $\mathcal{U}$ and make another iteration of the inner loop (line 11).

In our implementation, we compute the clauses in R incrementally, making a separate SAT query for each clause $d \in Inv \cup \mathcal{U}$. This computation is aborted as soon as a clause

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Input: $(\mathit{Init}, \mathit{Tr}, \mathit{Bad}), \mathit{Inv}_o, \mathcal{N}_o \subseteq \mathit{Inv}_o$ s.t. $\neg \mathit{Bad} \in \mathcal{N}_o$
Output: An MSIS $\mathit{Inv} \subseteq \mathit{Inv}_o$

```
1  Inv ← N_o ; W ← Inv_o \ N_o
2  while W ≠ ∅ do
3    c ← a clause from W
4    S ← ⟨{c}⟩
5    forever do
6      Suppose that S = ⟨C_1, ..., C_n⟩
7      let R = {d ∈ Inv ∪ (W \ C_n) |
8            (Inv ∧ (W \ C_n) ∧ Tr) ⇏ d'}
9      if R = ∅ then
10       W ← W \ C_n
11       S ← ⟨C_1, ..., C_{n-1}⟩
12       if S is empty then break
13     else if R ∩ Inv ≠ ∅ then
14       Inv ← Inv ∪ C_1 ∪ ··· ∪ C_n
15       W ← W \ (C_1 ∪ ··· ∪ C_n)
16       break
17     else if R ∩ C_i ≠ ∅ for some i ≤ n then
18       S ← ⟨C_1, ..., C_{i-1}, C_i ∪ ··· ∪ C_n⟩
19     else
20       d ← a clause from R
21       S ← ⟨C_1, ..., C_n, {d}⟩
```

Fig. 6. Binary Implication Graph MSIS algorithm (BigMSIS).

from Inv is added to R. Furthermore, the clauses in Inv are checked before the remaining clauses in $\mathcal{U}$. In other words, the OptMSIS corresponds to the naïve MSIS algorithm in which (1) the safe necessary clauses are marked as soon as they are discovered, (2) computing an MIS is aborted as soon as one of the necessary clauses becomes unsupported, and (3) necessary clauses are checked first. In practice, this significantly reduces the number of SAT queries done by the algorithm.

B. B.I.G. MSIS algorithm

Our ultimate algorithm for finding MSIS exploits the dependency of including some clauses to a SIS based on the inclusion of other clauses. We say that a clause c is *necessary* for a clause d if c is included in *every* MSIS of Inv that contains d. In particular, if $\mathit{Inv} \setminus \{c\} \wedge \mathit{Tr} \not\Rightarrow d'$, then c is necessary for d. From the definition, the necessary relation is transitive: if c is necessary for d and d is necessary for e, then c is necessary for e as well.

Consider a directed graph G on the clauses of Inv so that there is an edge from a clause $c \in \mathit{Inv}$ to $d \in \mathit{Inv}$ if and only if c is necessary for d. Them, for every strongly connected component C of G and every MSIS Inv of Inv_o either all of the clauses of C are included in Inv, or none of the clauses of C are included in Inv.

The BigMSIS algorithm shown in Fig. 6 makes use of these observations by incrementally learning and exploiting the underlying graph structure. It has the same input and output as the OptMSIS algorithm, and similarly keeps two sets Inv and $\mathcal{W} \subseteq \mathit{Inv}_o$ such that $\mathit{Inv} \cup \mathcal{W}$ is safe and inductive and Inv is safe necessary for $\mathit{Inv} \cup \mathcal{W}$. In addition, we use a vector of sets $\langle C_1, \ldots, C_n \rangle$, with the following properties: **(1)** The sets C_i are pairwise disjoint and contained in $\mathcal{W}$; **(2)** For any $i \leq j$, any clause $c \in C_i$, and any clause $d \in C_j$ the clause c is necessary for d. In particular, for every i all of the clauses in C_i belong to the same connected component of the graph.

Input: $(\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, safe inductive invariant Inv_0
Output: minimal safe inductive invariant Inv s.t. $\neg \mathit{Bad} \in \mathit{Inv} \subseteq \mathit{Inv}_0$

```
1  Inv ← Inv_0; N ← {¬Bad}
2  N   ← NEC((Init, Tr, Bad), Inv, N)
3  Inv ← FEAS((Init, Tr, Bad), Inv, N)
4  N   ← NEC((Init, Tr, Bad), Inv, N)
5  Inv ← MSIS((Init, Tr, Bad), Inv, N)
```

Fig. 7. Combined MSIS algorithm.

In the outermost while-loop of the algorithm we pick the next unprocessed clause $c \in \mathcal{W}$ and initialize S to consist of a single component $\{c\}$. We always focus on the last component C_n of S. On each iteration of the inner loop we compute the set R of clauses that become unsupported if C_n is removed from $\mathit{Inv} \cup \mathcal{W}$. Let us analyze the possible outcomes of this query in detail.

- (lines 9-12) The set $\mathit{Inv} \cup (\mathcal{W} \setminus C_n)$ is inductive. Since $\neg \mathit{Bad} \in \mathit{Inv}$, it is also safe. In this case, we tighten Inv by removing all of the clauses $c \in C_n$ (and focus on C_{n-1}, or proceed with the next unprocessed clause if $n = 1$).
- (lines 13-16) A safe necessary clause in Inv is no longer supported. It follows that *every* clause in $C_1 \cup \cdots \cup C_n$ is safe necessary as well. In this case we update the set Inv by including all of the clauses in S (and proceed with the next unprocessed clause).
- (lines 17-18) A clause $d \in C_i$ is no longer supported. In this case all of the clauses in $C_i \cup \cdots \cup C_n$ belong to the same connected component, and we replace the sets $C_i, \ldots, C_n$ in S by a single set $C_i \cup \cdots \cup C_n$ (and focus on this new set).
- (lines 19-21) A clause $d \in \mathcal{W}$ is no longer supported. Moreover, d is not one of the clauses in S. In this case, we add a new component $C_{n+1} = \{d\}$ to S (and focus on C_{n+1}).

As before, in our implementation R is computed incrementally. Moreover, we have found it beneficial to abort the computation as soon as the first unsupported clause $d \in \mathit{Inv} \cup (\mathcal{W} \setminus C_n)$ is found, and executing the corresponding branch (13-16, 17-18 or 19-21) right away. In this respect, BigMSIS is highly customizable: we can prioritize checking first the known necessary clauses (Inv), or the clauses already visited (S), or the clauses not yet explored.

Even though the high-level descriptions of OptMSIS and BigMSIS are rather similar, there is an important theoretical difference between the two algorithms: in the worst case, OptMSIS executes its inner-loop *a quadratic number of times*, while BigMSIS executes its inner-loop only *a linear number of times*. We illustrate this using the following example. Consider the problem $P_3 = (\mathit{Init}_3, \mathit{Tr}_3, \mathit{Bad}_3)$ and an invariant

Inv_3, where

$$Init_3 = Inv_3 \equiv x_1 \wedge \cdots \wedge x_n \quad (10)$$

$$Tr_3 \equiv x'_1 = x_n \wedge x'_2 = x_1 \wedge \cdots \wedge x'_n = x_{n-1} \quad (11)$$

$$Bad_3 \equiv \neg x_n \quad (12)$$

Further, suppose that initially $\mathcal{N}_o = \{x_n\} \equiv \{\neg Bad\}$. Suppose that OptMSIS picks the clause x_1 for removal. Then, after one iteration of the inner loop, the clause x_2 will be removed, after another iteration – the clause x_3, and so on, in total requiring n iterations to detect that $\neg Bad$ is removed. However, this only allows to deduce that x_1 is safe necessary (and can be included in *Inv*) giving no information about the other clauses. So OptMSIS would then proceed to remove x_2, leading to yet another $n-1$ iterations of the inner loop, and so on, leading to a quadratic number of iterations. The BigMSIS algorithm, on the other hand, requires one iteration of the inner loop to detect that x_1 cannot be removed unless x_2 is removed, another iteration to detect that x_2 cannot be removed unless x_3 is removed, and so on, overall requiring only n iterations to detect that none of $x_1, \ldots, x_n$ can be removed.

C. The combined algorithm

In practice even our best MSIS algorithm is slow due to a large number of required SAT queries. Fortunately, we achieve a significant improvement in runtime by suitably combining the computation of an MSIS with the the under- and over-approximating approaches. The combined algorithm is shown in Fig. 7. The algorithm takes as input a verification problem and an initial safe inductive invariant Inv_0. In the rest of this section, we analyze the suggested approach in detail.

- (Line 1) We mark the $\neg Bad$ clause as necessary and we set *Inv* to Inv_0.
- (Line 2) We run NEC to detect additional clauses that must be included in any MSIS of *Inv*. We emphasize the number of SAT calls performed by NEC is proportional to the number of necessary clauses detected.
- (Line 3) We run FEAS to prune the set of clauses in *Inv*. As discussed in Section IV-B, FEAS uses the necessary clauses found NEC for better overall choices of the algorithm.
- (Line 4) After some of the clauses were removed, we have new opportunities to mark additional clauses as necessary, and indeed we have found it beneficially to do so. The second run of NEC reuses the necessary clauses found in the first run.
- (Line 5) Finally we call an MSIS algorithm.

VI. Experiments

In this section, we present our experimental results. All experiments were performed on a 2.0 GHz Linux-based machine with Intel Xeon E7540 processor and 4 GB of RAM. We consider the unsatisfiable single property benchmarks from the 2011 and 2013 Hardware Model Checking Competitions [14], [15]. To obtain initial invariants, we preprocessed each of the benchmarks using a combinatorial logic optimization, and, if needed, ran (our implementation of) IC3 with 3 hours time limit. Altogether, IC3 successfully completed verification (and produced safe inductive invariants) of 305 benchmark instances. We use these to evaluate the techniques presented in this paper.

We denote by NAIVE the naïve MSIS algorithm from Fig. 2 (Section III) with the additional optimization described by Bradley et al. [6]: the computation of the maximal inductive subset of $Inv \setminus \{c\}$ (see Fig. 1) aborts as soon as the current subset becomes unsafe. We denote by OPT the OptMSIS algorithm from Fig. 5 (Section V-A), and by BIG the BigMSIS algorithm from Fig. 6 (Section V-B). We denote by NAIVE+NFN, OPT+NFN, and BIG+NFN (respectively) the combination of each of these algorithms with preprocessing (computing under-approximations using NEC, and over-approximations using FEAS), as described in Fig. 7 (Section V-C). We have run each of these 6 algorithms on each of the 305 testcases with a time limit of 1 hour.

The cactus plot in Fig. 8 presents a comparison between the algorithms. Note that preprocessing has a huge impact on any of the three MSIS algorithms, both in terms of instances solved and the total time (for example, NAIVE is able to solve 258 problems without preprocessing, and 293 problems with preprocessing). The best algorithm is BIG+NFN (solving 294 problems). The effectiveness of preprocessing is further corroborated by the fact that on average the initial NEC pass identifies about 70% of the final MSIS clauses as necessary; the following FEAS pass on average over-approximates the MSIS by only 4%; finally, after the second NEC pass, over 90% of the final MSIS clauses are marked as necessary. Thus, the final MSIS pass has to deal with only about 10% of the MSIS clauses on average.

We emphasize that when searching for small (and not necessary minimal) inductive invariants, the preprocessing stage alone (or, more precisely, NEC + FEAS) produces an almost optimal invariant in most cases, and as such the final MSIS stage can be skipped. Furthermore, any of the MSIS algorithms can be adapted to run with a resource limit, providing a safe and inductive over-approximation in case this limit is reached (such as the set $\mathcal{W}$ in the naïve MSIS algorithm, and the set $Inv \cup \mathcal{W}$ in the OptMSIS and BigMSIS algorithms).

The scatter plot on the left of Fig. 9 demonstrates that BIG+NFN is 2 to 3 orders of magnitude more effective than NAIVE, and hence represents the overall improvement of our best algorithm over prior work. The scatter plot in the center compares BIG and BIG+NFN, and serves to highlight that preprocessing is a crucial technique for most of the problems. Finally, the scatter plot on the right shows that even when preprocessing is used, BIG is an order of magnitude better than NAIVE. It should be noted the most of the points on the diagonal correspond to the problems solved by preprocessing alone.

It is also interesting to compare the number of clauses in the MSIS with the number of clauses in the original invariant computed by IC3. In this aspect there is very little variance between different algorithms, so we provide the data only for BIG+NFN, see Fig. 10. On average, the reduction is the more pronounced the larger is the initial invariant. In fact, this says something about IC3: even when IC3 learns a lot of invariants (and takes a long time to solve a problem), it does not mean that these invariants are useful for the final proof. Finally, we note that on our benchmarks MSIS on average removes 20%

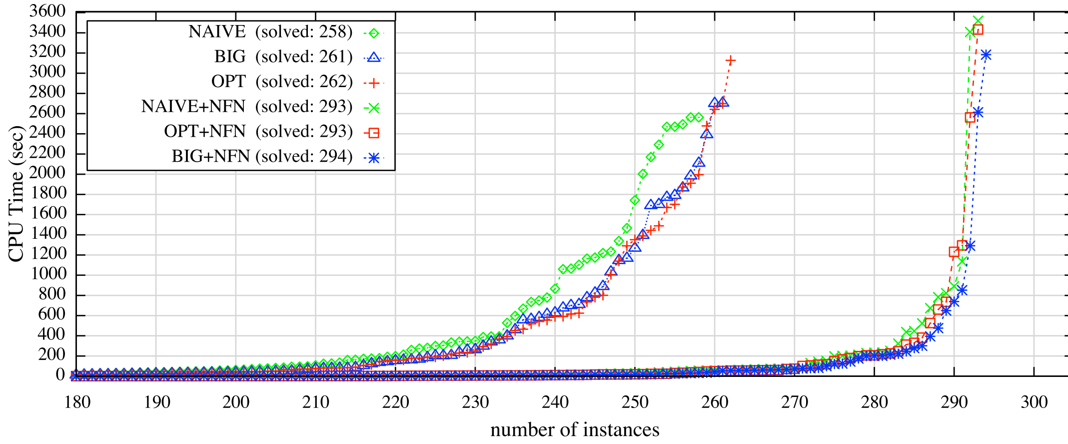


Fig. 8. Comparison between various algorithms, with and without preprocessing.

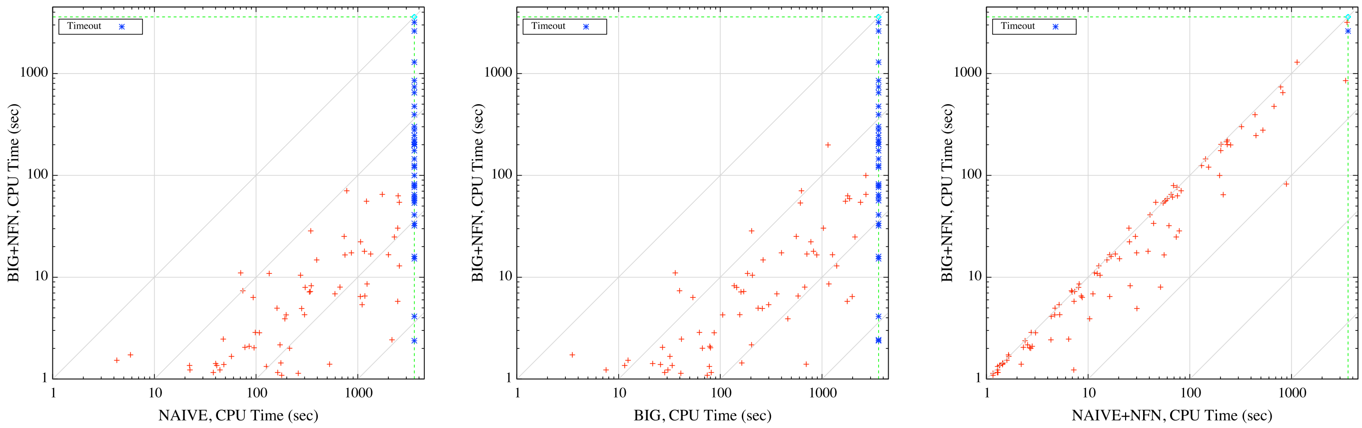


Fig. 9. CPU time comparison between selected algorithms.

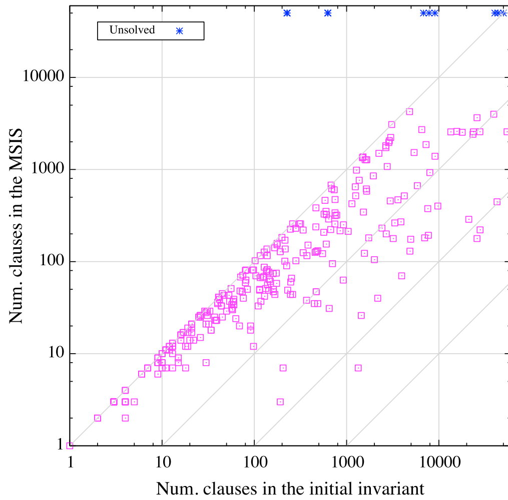


Fig. 10. Number of clauses in the MSIS (computed by BIG+NFN) vs. the initial invariant.

of variables.

VII. Conclusion

In this paper, we advocate for the problem of computing small inductive certificates in the context of unbounded model checking. We believe that this problem is as fundamental as computing minimal unsatisfiable subsets, and that it has just as wide of a variety of applications.

We propose an efficient algorithm for finding minimal safe inductive invariants, which combines: (1) An algorithm to under-approximate an MSIS by identifying necessary clauses that must be included in any safe inductive subset; (2) An algorithm to over-approximate an MSIS by removing clauses that do not belong to some safe inductive subset; (3) Two alternative algorithms to compute an MSIS via an optimized reduction to a series of computations of a maximal inductive subset. We show that on the benchmarks from the Hardware Model Checking Competition, our combined algorithm is several orders of magnitude more efficient than a naïve approach.

References

[1] K. L. McMillan and N. Amla, “Automatic Abstraction without Counterexamples,” in *TACAS*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 2–17.

[2] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, "Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC," in *FMCAD*. IEEE Computer Society, 2007, pp. 3–12.

[3] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[4] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87.

[5] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134.

[6] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 144–153.

[7] A. Belov and J. Marques-Silva, "MUSer2: An Efficient MUS Extractor," *JSAT*, vol. 8, no. 1/2, pp. 123–128, 2012.

[8] J. Marques-Silva, M. Janota, and A. Belov, "Minimal sets over monotone predicates in boolean formulae," in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 592–607.

[9] A. Nadel, V. Ryvchin, and O. Strichman, "Efficient MUS extraction with resolution," in *FMCAD*. IEEE, 2013, pp. 197–200.

[10] A. Gurfinkel, A. Belov, and J. Marques-Silva, "Synthesizing Safe Bit-Precise Invariants," in *TACAS*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 93–108.

[11] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic Abstraction in SMT-Based Unbounded Software Model Checking," in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 846–862.

[12] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 135–143.

[13] C. Sinz, "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints," in *CP*, ser. Lecture Notes in Computer Science, P. van Beek, Ed., vol. 3709. Springer, 2005, pp. 827–831.

[14] "Hardware Model Checking Competition 2011," http://fmv.jku.at/hwmcc11.

[15] "Hardware Model Checking Competition 2013," http://fmv.jku.at/hwmcc13.

[16] P. Bjesse and A. Slobodová, Eds., *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*. FMCAD Inc., 2011.

[17] N. Sharygina and H. Veith, Eds., *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013.

On Interpolants and Variable Assignments

Pavel Jancik, Jan Kofroň
Faculty of Mathematics and Physics,
Charles University, Prague, Czech Republic
Email: name.surname@d3s.mff.cuni.cz

Simone Fulvio Rollini, Natasha Sharygina
Faculty of Informatics,
University of Lugano, Switzerland,
Email: name.surname@usi.ch

***Abstract*—Craig interpolants are widely used in program verification as a means of abstraction. In this paper, we (i) introduce *Partial Variable Assignment Interpolants (PVAIs)* as a generalization of Craig interpolants. A variable assignment focuses computed interpolants by restricting the set of clauses taken into account during interpolation. PVAIs can be for example employed in the context of DAG interpolation, in order to prevent unwanted out-of-scope variables to appear in interpolants. Furthermore, we (ii) present a way to compute PVAIs for propositional logic based on an extension of the Labeled Interpolation Systems, and (iii) analyze the strength of computed interpolants and prove the conditions under which they have the path interpolation property.**

I. Introduction

In software model checking Craig interpolants play an important role. They are typically used to refine an abstraction of a program. Many techniques have been introduced to compute interpolants for various theories such as propositional logic, conjunctive fragments of linear arithmetic, and octagon domain. For propositional logic, McMillan's [9] and Pudlák's [11] interpolation systems are well established; they are generalized by the Labeled Interpolation Systems [6] (LISs), which permit to systematically compute interpolants of different logical strength from the same refutation.

Given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig interpolant is a formula I such that A implies I, I is inconsistent with B and I is defined over the common variables of A and B. In other words, I is an over-approximation of A (which can be used to abstract the behavior of a system, represented by A) disjoint from B (which often represents unacceptable behaviors).

In this paper, we introduce *Partial Variable Assignment Interpolants (PVAIs)* – a generalization of Craig interpolants – which, in addition to the standard subdivision of an unsatisfiable formula (the *interpolation problem*) into A and B, is parametric in a *partial variable assignment (PVA)*. A PVA defines a *sub-problem* on which a PVAI is focused. A sub-problem is obtained from the interpolation problem by removing the clauses (constraints) satisfied by the assignment. Due to the specialization, (1) it is possible to restrict the variables occurring in an interpolant to those relevant to the sub-problem, i.e. those shared between the A and B parts of the sub-problem. Moreover, since the irrelevant constraints (those not occurring in the sub-problem) need not be considered by interpolation, (2) the interpolants for the sub-problem can be of smaller size, compared to Craig interpolants computed from the interpolation problem.

In the motivating example in Sec. II we show how PVAIs apply to program verification. For instance, in the context of abstract reachability graphs (ARG) (and DAG interpolation [2]), an interpolation problem is the encoding of a whole ARG (representing all paths in the ARG), while for a given ARG node i the related sub-problem represents the set of paths that pass through that node. An over-approximation of the states reachable at i via these paths (a *node interpolant*) can be computed by means of a PVAI. Properties of PVAIs guarantee that the interpolant contains only in-scope program variables.

An alternative approach could be to solve each sub-problem separately, which involves calling a SAT/SMT solver for each sub-problem and applying standard Craig interpolation. The method we propose allows one to perform just a single call to a solver for an interpolation problem which encompasses all the sub-problems, thus (i) processing the parts common to multiple sub-problems only once. A single solver call results in a single proof from which all the interpolants for the sub-problems are computed. The presence of a single proof, in turn, enables (ii) generating collections of interpolants which satisfy properties relevant to verification, such as path interpolation [7], [13]. Such collections are hard to obtain if multiple proofs are involved. In the case of PVAIs, a collection may consist of the interpolants associated with different sub-problems.

We also propose the new framework of *Labeled Partial Assignment Interpolation Systems* (LPAISs) – a generalization of LISs, which computes PVAIs for propositional logic. We define the notion of logical strength for LPAISs and show how introducing a partial order over LPAISs allows to systematically compare the strength of the computed interpolants (a feature intuitively relevant to verification since it affects the coarseness of the over-approximations realized by interpolants [12]). We also show how LPAISs can be used to generate collections of interpolants which enjoy the path interpolation (inductive step) property. These results can be applied in the context of ARGs, where the path interpolation property of computed node interpolants (labels) guarantees well-labeledness [10] of the ARG.

This work is partially supported by: ICT COST Action IC0901, the Grant Agency of the Czech Republic project 14-11384S, and Charles University Foundation grant 203-10/253297.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

```
1: int max(int i, int j) {
2:   if (i > j)
3:     return i;
   else
4:     return j;
5: }
   // The main function
6: assert(max(random(), 0) >= 0);
```

Figure 1. Motivating example

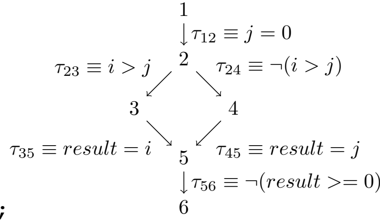


Figure 2. Abstract reachablity graph

$$
\begin{aligned}
\mu_1 &\equiv (n_1 \Rightarrow n_2) && \wedge ((n_1 \wedge n_2) \Rightarrow \tau_{12}) \\
\mu_2 &\equiv (n_2 \Rightarrow (n_3 \vee n_4)) && \wedge ((n_2 \wedge n_3) \Rightarrow \tau_{23}) \wedge \\
& && \wedge ((n_2 \wedge n_4) \Rightarrow \tau_{24}) \\
\mu_3 &\equiv (n_3 \Rightarrow n_5) && \wedge ((n_3 \wedge n_5) \Rightarrow \tau_{35}) \\
\mu_4 &\equiv (n_4 \Rightarrow n_5) && \wedge ((n_4 \wedge n_5) \Rightarrow \tau_{45}) \\
\mu_5 &\equiv (n_5 \Rightarrow n_6) && \wedge ((n_5 \wedge n_6) \Rightarrow \tau_{56})
\end{aligned}
$$

$$
\mathsf{Cond} \equiv n_1 \wedge \mu_1 \wedge \mu_2 \wedge \mu_3 \wedge \mu_4 \wedge \mu_5
$$

Figure 3. The Cond formula

II. Motivation

In the following, we illustrate a possible application of PVAIs, which originally motivated this work; nonetheless, the proposed PVAIs are not limited to this context. As an example, consider the source code on the left-hand side of Fig. 1 and the corresponding ARG in Fig. 2. Node i is associated with location i in the program. Node 1 is the initial node, while node 6 is the node representing an error location. The *edge constraints* τ_{ij} encode the semantics of the corresponding program statements. Note that τ_{12} originates from the call to the `max` function in `main`, on line 6. Further, in node 3, the parameter i is the only in-scope variable; similarly in node 4 the parameter j is the only in-scope variable. A variable is in-scope at a given node, if there is a path through the node where the variable is used before as well as after the node.

In the context of software verification, an important question is whether an error location is actually reachable from the initial location of a program – this is known as the *reachability problem*. The question can be answered by computing, for each node i, the set of states reachable at i via paths in the program ARG [4], [10]. Typically, it is enough to compute an over-approximation of these states, i.e. a *node interpolant*. To this end, the ARG is converted into a Cond formula[1], which represents all execution paths in the ARG. An auxiliary *structure-encoding* Boolean variable n_i is introduced for each node i in the ARG; for each i (except for the error node), a *node formula* μ_i is created, which encodes the labels on the outgoing edges (Fig. 3).

For illustration, we describe the meaning of μ_2. The first conjunct $n_2 \Rightarrow (n_3 \vee n_4)$ expresses that after reaching node 2, a path has to proceed to a successor node (3 or 4). The second conjunct $(n_2 \wedge n_3) \Rightarrow \tau_{23}$ guarantees that if a path goes via the edge $2 \rightarrow 3$, the semantics of the edge is preserved (i.e., the constraint τ_{23} is satisfied). Similarly, the third conjunct enforces the semantics of the edge $2 \rightarrow 4$.

The Cond formula is satisfiable if and only if a feasible path exists that leads from node 1 to node 6 in the ARG. Suppose now that Cond is unsatisfiable; then a node interpolant for each node i can be computed. First the ARG needs to be partitioned into A and B – so that A corresponds to the antecedents of i, B to all the other nodes in the ARG – and then a Craig interpolant I is generated as an over-approximation of the states reachable at i. For instance, in the case of node 3, A would be set to $n_1 \wedge \mu_1 \wedge \mu_2$ and B to $\mu_3 \wedge \mu_4 \wedge \mu_5$. However, employing standard Craig interpolation in this manner to compute a node interpolant I is not sufficient; out-of-scope variables might in fact belong to both A and B, they could therefore appear in I, and should be consequently eliminated. Variable j, for example, could appear in the interpolant for node 3. Even though out-of-scope variables can be eliminated by resorting to quantification, followed by a quantifier-elimination phase, this approach is a well-known bottleneck in verification.

Computing node interpolants using PVAIs effectively solves the problem of out-of-scope program variables. Suppose that a node interpolant is to be computed for a node k; the created PVA assigns False to all structure-encoding variables corresponding to nodes not lying on the paths through k. By setting a variable n_j to False, in fact, the paths via node j are blocked; moreover, the whole node formula μ_j is satisfied and thus μ_j is not a part of the sub-problem for node k. On the other hand, the PVA assigns n_k to True to express that each considered path has to pass through k (the node for which the interpolant is computed). In particular, to compute an interpolant for node 3, we assign n_3 to True and n_4 to False to block the path through node 4; the rest of variables remain unassigned. This assignment satisfies (and thus removes) $n_2 \Rightarrow (n_3 \vee n_4)$, $(n_2 \wedge n_4) \Rightarrow \tau_{24}$ and μ_4 from the sub-problem (see Fig. 4). In the A part, the sub-problem for node 3 contains the edge labels (and consequently the program state variables) related to the path from node 1 to node 3, and in the B part information related to the path from node 3 to node 6. The program state variables shared by the A and B parts of the sub-problem are the in-scope variables, which are exactly those that may appear in PVA interpolants.

III. Preliminaries

A *clause* is a finite disjunction of literals. We use angle brackets $\langle \Theta \rangle$ to denote the clause built over the literals in Θ. Let $\langle \Theta, p \rangle$ and $\langle \Theta', \overline{p} \rangle$ be clauses. Using variable p as the *pivot*, their resolution yields the clause $\langle \Theta, \Theta' \rangle$. In the following, we consider propositional formulas in *conjunctive normal form*,

$$
\begin{aligned}
\pi_3 &\equiv n_3 \wedge \overline{n_4} \\
A_3 &\equiv n_1 \wedge \\
&\quad (n_1 \Rightarrow n_2) \wedge ((n_1 \wedge n_2) \Rightarrow j = 0) \wedge \\
&\quad\quad \wedge ((n_2 \wedge n_3) \Rightarrow i > j) \\
B_3 &\equiv (n_3 \Rightarrow n_5) \wedge ((n_3 \wedge n_5) \Rightarrow result = i) \wedge \\
&\quad (n_5 \Rightarrow n_6) \wedge ((n_5 \wedge n_6) \Rightarrow \neg(result >= 0))
\end{aligned}
$$

Figure 4. The A and B parts of the sub-problem for node 3

[1] Cond has the same meaning as ArgCond in [3].

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

i.e., as conjunctions (or equivalently sets) of clauses. We use $\mathsf{Var}(l)$ to denote the variable of literal l and $\mathsf{Var}(A)$ for the variables occurring in the set of clauses A.

We adopt the definition of resolution proof from [6]: a resolution proof is a tuple (V, E, cl, piv, s), where V is a set of vertices in the proof, E is a set of edges. Each inner vertex v represents resolution of its antecedent vertex-clauses (specified by cl) using the pivot $piv(v)$. A refutation proof derives an empty clause in the sink vertex s.

Since the resolution proofs take the set of clauses as input, the input formula is first converted into a conjunction of clauses. Thus in the following we use the terms formula and set of clauses interchangeably.

A *Craig interpolant* [5] for the pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable is a formula I such that (1) $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$.

An *interpolant sequence* for the unsatisfiable formula $A_1 \wedge A_2 \wedge ... \wedge A_n$ is a tuple of formulas $(I_0, I_1, ...I_n)$, where I_i is an interpolant for $(A_1 \wedge ... \wedge A_i, A_{i+1} \wedge ... \wedge A_n)$. If for all i, $I_i \wedge A_i \Rightarrow I_{i+1}$, then $(I_0, I_1, ...I_n)$ is said to satisfy the *path interpolation* (PI) property. In [7], it was proved that the path interpolation property holds for any LISs, including the well-known McMillan's and Pudlák's systems, whenever the interpolant sequence is computed from the same proof.

Let A be a set of clauses. A *variable assignment* assigns either True $(\top)$ or False $(\bot)$ to each variable in the $\mathsf{Var}(A)$ set. The variable assignment can be seen as a conjunction of literals. A *partial variable assignment* (PVA) π assigns values only to a subset of variables in $\mathsf{Var}(A)$. A PVA π can be used as an assumption w.r.t. A (i.e., $\pi \models A$) to restrict the set of models of A to those compatible with π.

Definition 1 (Clauses under assignment): Let A be a set of clauses and π be a PVA over $\mathsf{Var}(A)$. We define the sets of *satisfied* clauses $A_\pi = \{\langle\Theta\rangle | \langle\Theta\rangle \in A \text{ and } \pi \models \langle\Theta\rangle\}$ and *unsatisfied* clauses $A_{\overline{\pi}} = \{\langle\Theta\rangle | \langle\Theta\rangle \in A \text{ and } \pi \not\models \langle\Theta\rangle\}$.

Satisfied clauses contain at least one literal evaluated to $\top$ under π, while, for unsatisfied clauses, every literal is either unassigned or falsified. The unsatisfied clauses $A_{\overline{\pi}}$ determine the sub-problem. We use $\pi \models l$ to express that a literal l evaluates to $\top$ in a given PVA π.

IV. Partial Variable Assignment Interpolants

In this section, we formally define *partial variable assignment interpolation*, which, in addition to the subdivision of an unsatisfiable formula into A and a B parts, requires specification of a PVA.

Definition 2: Let R be an (A, B)-refutation and π a partial variable assignment over $\mathsf{Var}(A \wedge B)$. A *partial variable assignment interpolant* (PVAI) is a formula I such that:

(D2.1) $\pi \models A \Rightarrow I$
(D2.2) $\pi \models B \wedge I \Rightarrow \bot$
(D2.3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$
(D2.4) $\mathsf{Var}(I) \cap \mathsf{Var}(\pi) = \emptyset$

In the following we use (A, B, π) to denote that a PVAI is computed from an (A, B)-refutation using the partial assignment π.

Since $\pi \models (A \Leftrightarrow A_{\overline{\pi}})$, D2.1 and D2.2 can be equivalently rewritten as $\pi \models A_{\overline{\pi}} \Rightarrow I$ and $\pi \models B_{\overline{\pi}} \wedge I \Rightarrow \bot$; in other words, I is an interpolant for the sub-problem $(A_{\overline{\pi}} \wedge B_{\overline{\pi}})$. Note that even after removing (the satisfied) clauses, the sub-problem remains unsatisfiable (assuming π).

On the other hand, a PVAI cannot be obtained from standard interpolants by application of a partial assignment ($I[\pi]$). The reason is that, in addition to assigned variables (disallowed by D2.4), rule D2.3 excludes from the PVAI also all unassigned (out-of-scope) variables that occur in satisfied clauses only, which can still appear in $I[\pi]$.

Calling a solver multiple times can be quite resource-consuming. An (A, B)-refutation proof is independent of a PVA; this important fact allows to call the solver only once on the overall problem $A \wedge B$, and later to introduce various PVAs (representing relevant sub-problems) for which the PVAI can be efficiently computed.

Although Craig interpolation has many applications in program verification, verification tools often require interpolation sequences with specific properties [7]. The PVAI for all the sub-problems are computed from the same proof, thus they are related to each other. The existence of a single proof permits the application of a standard proving technique in the area of interpolation – structural induction over a refutation proof – to show various properties of PVA interpolant sequences. All the techniques where interpolants for different sub-problems are computed using different proofs (e.g., applying a solver directly on each sub-problem, or incremental solving with assumptions) do not, per se, guarantee any properties of their sequences.

V. Labeled Partial Assignment Interpolation System

To show that PVAIs are not just a theoretical concept, we present the framework of *Labeled Partial Assignment Interpolation Systems*, a generalization of LISs [6], which computes PVAIs for propositional logic, and prove its soundness. Next, in order to prove the path interpolation property, we introduce the concept of logical strength on LPAISs, which allows one to systematically compare the strength of the generated interpolants.

In order to define LPAISs, first we have to extend the definitions of labeling functions and locality from LISs to take variable assignments into account. Note that if no variable is assigned, LPAISs are equivalent to LISs.

A labeling function assigns labels to literals in a refutation; the labeling drives the computation of an interpolant from the proof and determines its strength.

Definition 3 (Labeling function): Let $L = (S, \sqsubseteq, \sqcap, \sqcup)$ be the lattice of Fig. 6, where $S = \{\bot, a, b, ab, d^+\}$ and $\bot$ is the least element, and let $R = (V, E, cl, piv, s)$ be a resolution proof over a set of literals Lit. A function $\mathsf{Lab}_{R,L} : V \times \mathsf{Lit} \rightarrow S$

Leaf v:	$\langle\Theta\rangle, [I]$		
$I =$	$\langle\Theta\rangle[\pi]\|_{b,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in A_{\overline{\pi}}$	Hyp-$A_{\overline{\pi}}$
	$\neg\langle\Theta\rangle[\pi]\|_{a,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in B_{\overline{\pi}}$	Hyp-$B_{\overline{\pi}}$
	$\top$	if $\langle\Theta\rangle \in A_{\pi} \cup B_{\pi}$	Hyp-A_{π}, Hyp-B_{π}

Inner vertex v:	$\dfrac{v_1 : \langle p, \Theta_1\rangle, [I_1] \qquad v_2 : \langle \overline{p}, \Theta_2\rangle, [I_2]}{\langle \Theta_1, \Theta_2\rangle, [I]}$		
$I =$	$I_1 \vee I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$	Res-a
	$I_1 \wedge I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$	Res-b
	$(I_1 \vee p) \wedge (I_2 \vee \overline{p})$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$	Res-ab
	I_2	if $\mathsf{Lab}(v_1, p) = d^+$	Res-d^+
	I_1	if $\mathsf{Lab}(v_2, \overline{p}) = d^+$	Res-d^+

Figure 5. Labeled Partial Assignment Interpolation System

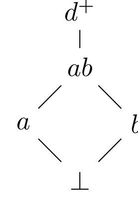


Figure 6. Lattice of labels (according to $\sqsubseteq$)

is called *labeling function* for a refutation R iff $\forall v \in V$ and $\forall l \in \mathsf{Lit}, \mathsf{Lab}_{R,L}$ satisfies the following conditions:

(D3.1) $\mathsf{Lab}_{R,L}(v, l) = \bot$ if and only if $l \notin cl(v)$, and

(D3.2) $\mathsf{Lab}_{R,L}(v, l) = \mathsf{Lab}_{R,L}(v_1, l) \sqcup \mathsf{Lab}_{R,L}(v_2, l)$, where v_1, v_2 are the predecessor vertices.

From condition D3.2 it follows that the labeling function is fully determined once the labels in the leaves have been specified. We omit subscripts R and L if clear from the context.

Naming conventions: Let us assume a pair of sets of clauses (A, B) and a PVA π. The clause sets are split into four groups, the unsatisfied clauses $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$ which specify the sub-problem and are taken into account during interpolation, and the satisfied clauses A_{π} and B_{π}, which are disregarded.

We distinguish among the following kinds of variables, depending on the standard notions of locality and sharedness, as well as on where the variables appear in the four groups of clauses. We say that a variable k is *unassigned* if $k \notin \mathsf{Var}(\pi)$. An unassigned variable k is:

$A_{\overline{\pi}}$*-local*	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$
$B_{\overline{\pi}}$*-local*	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$*-shared*	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$*-clean*	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$

The properties above are independent of the occurrence of k in $\mathsf{Var}(A_{\pi})$ and $\mathsf{Var}(B_{\pi})$. The "clean" variables occur only in the satisfied clauses, thus are out-of-scope and cannot appear in a PVA interpolant.

We say that a variable k is *McMillan-labeled* if, whenever k is $A_{\overline{\pi}}B_{\overline{\pi}}$-shared or $A_{\overline{\pi}}B_{\overline{\pi}}$-clean, k is labeled b (the labels of the remaining variables are not limited to b). If all variables are McMillan-labeled, a LIS reduces to McMillan's interpolation system [6], which yields the strongest interpolant that LISs (and LPAISs) can produce from a given refutation proof.

A variable k is labeled *consistently* if all occurrences of k in a refutation have the same label.

Not all labeling functions can be used to generate interpolants; in LPAIS, interpolants are computed if a locality preserving labeling is used.

Definition 4: A labeling function Lab for an (A, B, π)-refutation R is *locality preserving* iff $\forall v \in V, \forall l \in cl(v)$:

(D4.1) $\mathsf{Lab}(v, l) = d^+ \Leftrightarrow \pi \models l$

(D4.2) $\mathsf{Var}(l)$ is unassigned and $A_{\overline{\pi}}$-local $\Rightarrow \mathsf{Lab}(v, l) = a$

(D4.3) $\mathsf{Var}(l)$ is unassigned and $B_{\overline{\pi}}$-local $\Rightarrow \mathsf{Lab}(v, l) = b$

(D4.4) $\mathsf{Var}(l)$ is unassigned and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean $\Rightarrow$ it is consistently labeled a or b.

Locality constraints provide freedom in labeling $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables; the choice of labels directly affects the strength of the computed interpolants. The label of $A_{\overline{\pi}}B_{\overline{\pi}}$-shared variables can be set freely to a, b, or ab. The same holds for falsified literals; their labels are irrelevant since they are removed by the assignment filter (defined below).

The D4.2 and D4.3 rules are equivalent to the locality requirements of LIS, where A-local and B-local variables must be labeled a and b, respectively. D4.1 concerns the satisfied literals. The label d^+ is used in the interpolation process to identify resolutions with an assigned pivot and parts of the proof which are not relevant to the sub-problem. The D4.4 requirement is specific to PVAI and deals with variables which occur in the satisfied clauses only. The requirement guarantees that such variables do not occur in the interpolant, because ab-resolution cannot be applied. Further, note that for the empty assignment the locality constraints reduce to those of LISs, since D4.1 and D4.4 do not apply to any literal.

Filters: For a clause $\langle\Theta\rangle$, a labeling function Lab, a resolution-proof vertex $v \in V$, and a label c, we define the *match filter* $|$ as $\langle\Theta\rangle|_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c = \mathsf{Lab}(v, l)\}$; it preserves only the literals with the specified label. Similarly, we define the *upward filter* $\upharpoonright$ as $\langle\Theta\rangle\upharpoonright_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c \sqsubseteq \mathsf{Lab}(v, l)\}$; it preserves the literals with labels above c in Fig. 6. The subscripts Lab, v are omitted if clear from the context. Given a partial assignment π and a clause $\langle\Theta\rangle$, we also define the *assignment filter* $\langle\Theta\rangle[\pi] = \{l \in \langle\Theta\rangle \mid \mathsf{Var}(l) \notin \mathsf{Var}(\pi))\}$, which removes all the assigned literals (satisfied and falsified ones).

Moreover, we assume that filters have a higher precedence than negation. E.g., $\neg\langle\Theta\rangle[\pi]\upharpoonright_a$ can be equivalently rewritten as $\neg((\langle\Theta\rangle[\pi])\upharpoonright_a)$.

An interpolation system is a procedure for computing an interpolant from a refutation. It assigns a partial *vertex-*

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

interpolant to each vertex of the refutation, yielding the final interpolant at the sink vertex.

Definition 5: For a locality preserving labeling function Lab and an (A, B, π)-refutation R, Fig. 5 defines the *Labeled Partial Assignment Interpolation System* $\mathsf{LpaItp}(\mathsf{Lab}, R)$.

An LPAIS produces interpolants in the following way: first the vertex-interpolants for leaves of the refutation proof are computed using the rules in the upper part of Fig. 5 (hypothesis rules). Depending on the occurrence of the vertex-clause $\langle\Theta\rangle$ in A or B sets, the corresponding rule describes the transformation of the vertex-clause into a vertex-interpolant. Later, going down through the proof from leaves to the sink, the vertex-interpolants for inner vertices are computed using rules in the lower part of Fig. 5. The labels assigned to the pivots determine how vertex-interpolants of both predecessors are combined. This process ends at the sink vertex where the PVAI is derived. The interpolants are computed in time linear to the size of the proof.

The main difference compared to LISs are the additional d^+ rules. For instance, consider the last rule, where $\mathsf{Lab}(v_2, \overline{p}) = d^+$. In contrast to the standard rules, the partial interpolant is simpler, because it does not contain I_2, omitted due to the variable assignment. Generally, these rules *cut out* the satisfied sub-tree of the proof. Usually, the later in the refutation the assigned variable is resolved, the larger sub-tree is pruned and the smaller the resulting interpolant is.

The differences between LPAISs and LISs are motivated by the way variable assignments work. The new d^+ rules can be seen as a specialization of the ab resolution rule if a PVA π is assumed. A similar relationship holds for the hypothesis rules in the leaves of a refutation. These rules are equivalent to LIS hypothesis rules if applied on a clause under the assumed assignment. The changes we introduce w.r.t. LISs are of two kinds: those in LPAISs rules force specialization of the interpolant on a sub-problem, while the changes in the locality constraints remove unassigned out-of-scope variables from the interpolant.

Theorem 1 (Correctness): $\mathsf{LpaItp}(\mathsf{Lab}, R)$, for an ($A$, B, π)-refutation R and a locality preserving labeling function Lab, generates a partial variable assignment interpolant.

Proof sketch: By structural induction over R we show that, for each vertex v of a resolution proof, the following invariants hold:

$$\pi \models A \wedge \neg\langle\Theta\rangle\restriction_{a,v,\mathsf{Lab}} \Rightarrow I_v$$
$$\pi \models B \wedge \neg\langle\Theta\rangle\restriction_{b,v,\mathsf{Lab}} \Rightarrow \neg I_v$$

I_v is the partial vertex-interpolant and $\langle\Theta\rangle$ is a vertex-clause of v. These invariants yield the PVAI constraints (D2.1, D2.2) at the sink vertex, where $\neg\langle\Theta\rangle = \top$. The full proof can be found in [8]. □

The attentive reader may notice that the locality constraints, as well as the way LPAISs compute interpolants, are *symmetric* for the A_π and B_π sets of satisfied clauses. It reflects the fact that these clauses are not a part of the sub-problem under consideration, thus irrelevant for PVAI interpolants. Given a fixed π, the satisfied clauses can be moved freely between the A and B sets; both computed interpolants and locality of the labeling functions are not affected if satisfied clauses are moved. This fact allows us to articulate the *strength* theorem in an elegant way.

A. Strength

Interpolation systems based on labeling provide some freedom in the choice of labels (e.g., for shared variables); this choice affects the resulting interpolants, in particular their strength. In the following we investigate this relationship in more detail.

$$\begin{array}{c} b \\ | \\ ab = d^+ \\ | \\ a \\ | \\ \bot \end{array}$$

Figure 7. Strength ordering ($\preceq$)

Definition 6 (Strength order): Let $\preceq$ be a pre-order relation defined on the set of labels $S = \{\bot, a, b, ab,\ d^+\}$ as: $b \preceq ab = d^+ \preceq a \preceq \bot$ (see Fig. 7). Let Lab and Lab' be labeling functions for a refutation R. We say Lab *is stronger than* Lab', denoted as $\mathsf{Lab} \preceq \mathsf{Lab}'$, if for all vertices $v \in V$ and for all literals $l \in cl(v)$ it holds that $\mathsf{Lab}(v, l) \preceq \mathsf{Lab}'(v, l)$.

Note that labels ab and d^+ are of the same strength and can be exchanged if the locality requirements permit; b is the strongest label, while a is the weakest one a literal can get.

The following theorem states that the introduced strength order on labeling functions also orders the produced interpolants by logical strength.

Theorem 2 (Interpolant strength): Let Lab be a locality preserving labeling function for an (A, B, π)-refutation R, and Lab' be a locality preserving labeling function for (A, B, π')-R. Let I be a partial variable assignment interpolant for $\mathsf{LpaItp}(\mathsf{Lab}, R)$ and I' be a PVAI for $\mathsf{LpaItp}(\mathsf{Lab}', R)$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi, \pi' \models I \Rightarrow I'$.

Note that when π and π' are *empty* assignments, we obtain exactly the theorem on interpolant strength from [6]. Also note that the theorem permits different variable assignments for the interpolants. Thus it relates the interpolants generated for different sub-problems (e.g., interpolants considering different sets of paths through a given ARG node). Since both π and π' are assumptions of the formula $I \Rightarrow I'$, the theorem applies to cases common to both sub-problems (i.e., to the shared paths). Both interpolants (I and I') have to be computed using the same A and B parts, thus interpolants for different ARG nodes cannot be compared using this theorem; a generalization in this direction is shown in the following sub-section.

In the following proof, we need a new type of filter. Let Lab and Lab' be labeling functions to be compared by strength and v be a vertex of the refutation proof. The new *weakened-labels filter* $\Vert_v^{\mathsf{Lab},\mathsf{Lab}'}$ preserves the literals whose label is weaker in Lab' than in Lab. E.g., the filter preserves a literal l if the strongest labels b ($\mathsf{Lab}(v, l) = b$) is weakened into label a or

ab in $\mathsf{Lab}'(v,l)$, while it filters-out a literal if both functions assign label a to it. The vertex and the labeling functions are omitted if clear from the context.

Proof sketch (Theorem 2): By structural induction over R, we show that for each vertex of the resolution proof the following invariant holds:

$$\pi, \pi' \models I_v \wedge \neg\langle\Theta\rangle|\downarrow_v \Rightarrow I'_v$$

$\langle\Theta\rangle$ is the vertex-clause, I_v and I'_v are the partial vertex-interpolants for the vertex v as generated by our interpolation system using the labeling functions Lab and Lab', respectively. The full proof in [8] shows that the invariant holds for all combinations of rules that can be used to define the vertex-interpolants I_v and I'_v. □

Similarly to LISs, for a fixed variable assignment there is a lattice of LPAISs ordered according to the strength of labeling functions. The top element of the lattice involves the strongest labeling function, which assigns label b to $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables, while the labeling function of the bottom element assigns label a to them. Theorem 2 claims that LPAISs produce interpolants ordered by strength according to the lattice.

B. Path interpolation property

Several verification approaches such as [3], [10], [14] depend on the *path interpolation* property (PI). In [13] the authors show that LISs can be employed to generate path interpolants by providing a sequence of labeling functions that are decreasing in terms of strength. In this subsection we study conditions for labeling functions that have to be satisfied in order to guarantee the PI property of interpolant sequences generated by LPAISs.

First, we show that the PI property holds if the same partial assignment along a sequence is used to compute the interpolants (i.e., considering the same set of paths at different ARG nodes). Later on, we generalize the result to permit different partial assignments for particular interpolants (i.e., relating node interpolants).

Fixed PVA: To show the PI property, it is enough to prove that, for any consecutive interpolants in the sequence, it holds: $I \wedge S \Rightarrow I'$, where I is an interpolant for $(A, S \cup B, \pi)$, I' is an interpolant for $(A \cup S, B, \pi)$, and S is a set of clauses.

For LISs, [13] defines a set of *labeling constraints* on the labeling functions used to compute the interpolants I and I'; if the labeling constraints are satisfied, the interpolants have the PI property. However, we prove the PI property in another way, more suitable for LPAISs. Given a labeling function to compute the interpolant I, we define the strongest labeling function which can be used to compute the successor interpolant I'.

Definition 7: Let Lab be a labeling function for an $(A, S \cup B, \pi)$-refutation R. The *strongest successor labeling* function Lab^S (for the set S) is defined in Fig. 8.

It is easy to see that Lab^S is a valid labeling function and that if Lab is locality preserving, then Lab^S is locality preserving for $(A \cup S, B, \pi)$. Hence, Lab^S can be used to compute an interpolant for $(A \cup S, B, \pi)$.

The first alternative (D7.1) forces label a for all literals which become $(A_{\overline{\pi}} \cup S_{\overline{\pi}})$-local due to the shift of the clauses in S from the B to the A part. Any locality preserving function Lab' has to also assign the label a to these literals. So, it is easy to see that if $\mathsf{Lab} \preceq \mathsf{Lab}'$ then also $\mathsf{Lab}^S \preceq \mathsf{Lab}'$. This expresses the meaning of *strongest*. Moreover, $\mathsf{Lab} \preceq \mathsf{Lab}^S$, because either the labels are equal or the weakest label a is used in the labeling Lab^S.

The following lemma states the PI property for the strongest successor labeling.

Lemma 1: Let Lab be a locality preserving labeling function for an $(A,\ S \cup B,\ \pi)$-refutation R and let $\mathsf{LpaItp}(\mathsf{Lab}, R) = I$. Let Lab^S be the strongest successor labeling for Lab and S, and $\mathsf{LpaItp}(\mathsf{Lab}^S,\ (A \cup S, B, \pi)) = I'$.
Then $\pi \models I \wedge S \Rightarrow I'$.

Proof sketch: By structural induction over R, we show that for each vertex v of the resolution proof the following invariant holds:

$$\pi \models I_v \wedge S \wedge \neg\langle\Theta\rangle|\downarrow_v \Rightarrow I'_v$$

$\langle\Theta\rangle$ is the vertex-clause, I_v and I'_v are the partial vertex-interpolants for the vertex v as generated by our interpolation system using the labeling functions Lab and Lab^S, respectively. The full proof can be found in [8]. □

Lemma 1 guarantees the PI property only if the sequence of the strongest successors labeling functions is used. Below we generalize this result in such a way that the strength of the labeling function can decrease along the sequence; Theorem 3 states the main result for a fixed partial assignment – the path interpolation property.

Theorem 3: Let Lab and Lab' be locality preserving labeling functions for an $(A,\ S \cup B,\ \pi)$-refutation R and $(A \cup S,\ B,\ \pi)$-R, respectively. Let $\mathsf{LpaItp}(\mathsf{Lab}, R) = I$ and $\mathsf{LpaItp}(\mathsf{Lab}', R) = I'$.
If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi \models I \wedge S \Rightarrow I'$.

Proof: Let I^S be the partial variable interpolant for the strongest successor labeling function Lab^S. From Lemma 1 it holds that $\pi \models I \wedge S \Rightarrow I^S$. As shown above $\mathsf{Lab}^S \preceq \mathsf{Lab}'$; so Theorem 2 can be applied and $\pi \models I^S \Rightarrow I'$. □

The result in this case is the same as for LISs. In the following we focus on the case when different PVAs are used, and the situation becomes more challenging.

Different PVAs: The goal to prove when different partial assignments π and π' are used to compute interpolants I and I' (respectively) is:

$$\pi, \pi' \models I \wedge S \Rightarrow I'$$

Looking back at the motivating example, for each node in the ARG a different partial variable assignment is typically used; thus, the generalization done in this section is needed to relate the interpolants of adjacent ARG nodes. Assume node

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

$$\mathsf{Lab}^S(v,l) = \begin{cases} a & \text{if } \mathsf{Var}(l) \in \mathsf{Var}(S_{\overline{\pi}}) \ \wedge\ \mathsf{Var}(l) \notin \mathsf{Var}(B_{\overline{\pi}}) \ \wedge\ \mathsf{Var}(l) \notin \mathsf{Var}(\pi) \quad (D7.1) \\ \mathsf{Lab}(v,l) & \text{otherwise} \quad (D7.2) \end{cases}$$

Figure 8. Strongest successor labeling function

interpolants I_2 for node 2 and I_3 for node 3. The desired property is then $I_2 \wedge \tau_{23} \Rightarrow I_3$ (well-labeledness in the context of ARGs [3], [10]), which follows from the aforementioned goal. In Theorem 4, we work out the conditions the labeling functions (for I_2 and I_3) have to satisfy so that the interpolants have the desired property.

Assignments: Having two different PVAs π and π', the expression (π, π') represents the PVA formed by the union of π and π'. We say that a PVA σ is an *extension* of a PVA π, if $\sigma \Rightarrow \pi$ (viewing the PVAs as conjunctions of literals). In other words, σ can be created from π by assigning additional variables. In case of conflicting π and π' (assigning one $\top$ and the other $\bot$ to a particular variable), the goal above holds trivially and therefore we omit the case from now on.

Definition 8: We say that the variable is *assignable* if it is McMillan-labeled and not $A_{\overline{\pi}}$-local.

Each assignable variable must have label b, therefore, after assigning it, its label becomes weaker. The following theorem states the main result for different PVAs.

Theorem 4: Let Lab be a locality preserving labeling function for an $(A, S \cup B, \pi)$-refutation R and let $I = \mathsf{LpaItp}(\mathsf{Lab}, (A, S \cup B, \pi))$. Let Lab' be a locality preserving labeling function for $(A \cup S, B, \pi')$-R and let $I' = \mathsf{LpaItp}(\mathsf{Lab}', (A \cup S, B, \pi'))$.

Suppose that (i) $A_{\overline{\pi}} \subseteq A_{\overline{\pi'}}$, (ii) $B_{\overline{\pi'}} \subseteq B_{\overline{\pi}}$, (iii) the variables assigned by π' and not by π are assignable in Lab, and (iv) the variables assigned by π and not by π' are not $B_{\overline{\pi'}}$-local.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then it holds $\pi, \pi' \models I \wedge S \Rightarrow I'$.

Intuitively, the constraints (i) and (ii) prevent from comparing interpolants of unrelated sub-problems. The only way to violate the constraint (i) $A_{\overline{\pi}} \subseteq A_{\overline{\pi'}}$ is to assign a new variable by π'. In terms of ARGs, it means that π' blocks some paths in addition to those blocked by π. The interpolant I over-approximates the states reachable in the corresponding node via non-blocked paths in the A part. If the assignment π' blocks some paths related to I' in addition to those blocked by π, then I' may not cover (over-approximate) the states coming from the blocked paths, thus it may be not implied by I. A similar reasoning can be used for (ii).

Proof sketch: The overall idea of the proof is shown in Fig. 9. The proof consists of four simpler steps. In the first step (① → ②) new variables get assigned by π', in the second step (② → ③) the clauses of S are moved. In the third step (③ → ④) the assignment π is removed, in the last step (④ → ⑤) the labeling function is weakened. In the second line of Fig. 9, it is expressed how the interpolation problem is divided into A and B parts and which PVA is used. In all but the second step the division into A and B parts does not change, thus Theorem 2 can be used to relate particular interpolants with each other via implications; in the second step the partial variable assignment does not change, so Theorem 3 is utilized.

To be able to apply this scheme (Theorems 2 and 3), locality preserving labeling functions of decreasing strength are needed. The third line of Fig. 9 specifies a labeling function for each step. The idea of the approach is similar to the one used for fixed variable assignments. In each step, we create the strongest possible labeling function; in particular for the first step (① → ②) we create an *extended-assignment labeling* function ($\mathsf{Lab}^{+}_{\pi \to (\pi,\pi')}$) – the strongest locality-preserving labeling function if new variables get assigned. For the second step (② → ③) we use the strongest successor labeling function as defined in Def. 7. For the third step (③ → ④) we create a *restricted-assignment labeling* function ($\mathsf{Lab}^{-}_{(\pi,\pi') \to \pi'}$) – the strongest locality-preserving labeling function if variables get unassigned. For the sake of space, we skip the definitions of the aforementioned labeling functions and proofs of the required properties; they can be found in [8].

Via the above construction we create the strongest locality-preserving labeling function ($\mathsf{Lab}^{-}_{(\pi,\pi') \to \pi'}$) for $(A \cup S, B, \pi')$ which satisfies $\mathsf{Lab} \preceq \mathsf{Lab}^{-}_{(\pi,\pi') \to \pi'}$. In the last step (④ → ⑤) we decrease the strength into Lab', in the same way as it is done for Lab^S in Theorem 3.

The last line of Fig. 9 shows how the interpolants in each step are related to each other and how the overall claim of this theorem follows from the particular steps. □

C. Application to ARGs

While the locality constraints are simple to satisfy for a single interpolant, the situation becomes more complicated if several interpolants need to be related by the path interpolation property. In such a case, the labels of the literals have to be chosen in an appropriate way. In the following, we briefly discuss how to set labels for ARG nodes (using the same encoding as in our motivating example) to apply Theorem 4 and, thus, to obtain well-labeled node interpolants.

Recall that in ARGs there are two kinds of variables – (1) *structure encoding* (n_i), which can be assigned, and (2) program variables, which are not assigned. The first rule is that the structure encoding variables have to be McMillan-labeled (obtaining the strongest possible labels). This rule and the properties of ARG encoding are enough to satisfy the (i)–(iv) requirements of Theorem 4.

Only the last requirement – $\mathsf{Lab}_i \preceq \mathsf{Lab}_j$ – restricts also the labels for program variables. It is easily satisfied in ARGs by a quite simple general rule: once an $A_{\overline{\pi}}B_{\overline{\pi}}$-shared or an $A_{\overline{\pi}}B_{\overline{\pi}}$-clean literal gets a label weaker than the strongest label b at a node, the same or a weaker label has to be assigned at all its successor nodes, until it becomes $A_{\overline{\pi}}$-local.

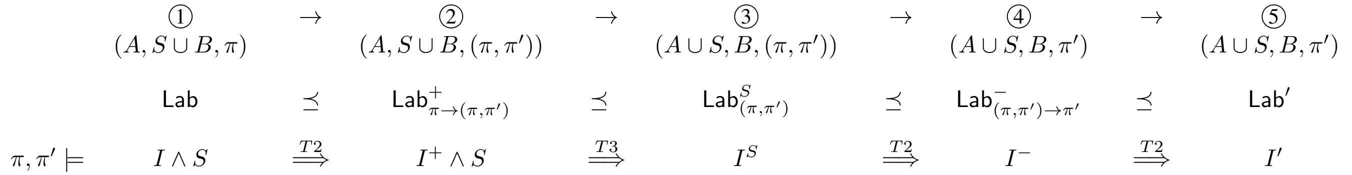


Figure 9. Idea of Theorem 4

Apparently, if for all nodes in an ARG the strongest possible labeling functions are used (i.e., all variables are McMillan-labeled), the aforementioned rules on labeling functions are satisfied, and well-labeled node interpolants are obtained.

A well-known inherent property of node interpolants is that for a path p in ARG the resulting node interpolants do not form path interpolants. A node interpolant summarizes information about all paths via the node. To be able to express this "summary", the variables shared (between A and B) on any path via the node need to be employed; we call these in-scope variables. However these variables are not necessarily AB-shared in the selected path p.

Still, path interpolants for a single path can be computed from the overall problem by means of PVAIs. Using a PVA that blocks all paths except for the one of interest, LPAISs yield path interpolants focused only on that path and over the variables shared on that path.

VI. Related work

To the best of our knowledge, the only strongly related works in this area are [1], [3].

The approach of [3], implemented in the UFO tool, can handle linear integer arithmetic. The main idea of the technique is to linearize a DAG into a single path; after that, standard path interpolants are computed and, if out-out-scope variables are present in the interpolants, quantification is used to remove these variables. So, in general the approach leads to quantified interpolants, while LPAISs yield quantifier-free interpolants.

In [1], the authors present a different solution to the problem of out-of-scope variables. Instead of quantification, the following operations are proposed to remove them: (a) assigning constants to variables in the interpolant ($\top$ or $\bot$ in case of propositional logic) or (b) modifying the structure of the DAG encoding. Comparing to (a), our approach is more general. We naturally handle any provided assignments, thus it is possible to assign additional variables to obtain the same interpolant as suggested by [1]. Moreover, we provide more flexibility, e.g., in the case of $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables one may choose either label b to obtain a stronger interpolant, or label a to get a weaker one. In our work we also show the constraints under which a property relevant to verification – the path interpolation property – holds, which is not guaranteed in [1].

An aspect common to the above approaches is that they are applied as post-processing techniques, after an interpolant has been computed and only if it contains out-of-scope variables. On the contrary, our method is integrated into the computation of the interpolant, and simplifies the proof on the fly according to the corresponding variable assignment, yielding a possibly smaller interpolant.

VII. Conclusion

In this paper, we introduced the new concept of Partial Variable Assignment Interpolants, which, unlike Craig interpolants, permits specialization to sub-problems specified in the form of variable assignments. We showed how PVAIs find application in the context of Abstract Reachability Graphs and DAG interpolation. We also developed the new framework of Labeled Partial Assignment Interpolation Systems, which can be used to compute PVAIs for propositional logic, and showed its properties.

As future work, we plan to extend the framework of LPAISs and to introduce a PVA interpolation system for linear integer arithmetic – a theory particularly relevant to program verification.

Acknowledgment.: Special thanks go to Ondřej Šerý for his valuable contribution.

References

[1] Albarghouthi, A., Gurfinkel, A.: DAG-Interpolation for Software Model Checking (2013), http://cav2013.forsyte.at/files/aws_albarghouthi.pdf
[2] Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: SAS '12. LNCS, vol. 7460, pp. 300–316 (2012)
[3] Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: TACAS '12. LNCS, vol. 7214, pp. 157–172 (2012)
[4] Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In: CAV '12. LNCS, vol. 7358, pp. 672–678 (2012)
[5] Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic pp. 269–285 (1957)
[6] D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI'10. LNCS, vol. 5944, pp. 129–145 (2010)
[7] Gurfinkel, A., Rollini, S.F., Sharygina, N.: Interpolation Properties and SAT-Based Model Checking. In: ATVA '13. LNCS, vol. 8172, pp. 255–271 (2013)
[8] Jančík, P., Kofroň, J.: On Partial Variable Assignment Interpolants. Tech. Rep. 2013/5, Dept. of Distributed and Dependable Systems, Charles University in Prague (2013), http://d3s.mff.cuni.cz/publications/download/D3S-TR-2013-05-PVAI.pdf
[9] McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV '03. LNCS, vol. 2725, pp. 1–13 (2003)
[10] McMillan, K.L.: Lazy Abstraction with Interpolants. In: CAV '06. LNCS, vol. 4144, pp. 123–136 (2006)
[11] Pudlák, P.: Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
[12] Rollini, S., Alt, L., Fedyukovich, G., Hyvärinen, A., Sharygina, N.: PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification. In: LPAR (2013)
[13] Rollini, S.F., Sery, O., Sharygina, N.: Leveraging Interpolant Strength in Model Checking. In: CAV '12. LNCS, vol. 7358, pp. 193–209 (2012)
[14] Vizel, Y., Grumberg, O.: Interpolation-sequence based Model Checking. In: FMCAD '09. pp. 1–8. IEEE (2009)

Post-silicon Timing Diagnosis Made Simple using Formal Technology

Daher Kaiss and Jonathan Kalechstain
Core CAD Technologies, Intel Corporation
Email: {dkaiss, jkalechs}@iil.intel.com

***Abstract*— With the increasing demand for microprocessor core operating frequencies, debugging post silicon synchronization (or speed) failures is a critical time consuming post silicon debug activity. Inability to complete the isolation of all possible speed failures on time, forces companies to go to market with products that run at a lower frequency than their upper frequency limits. This might cause revenue losses or lead to loss of market segment shares. Laser-Assisted Device Alternation (LADA) machines are the main vehicle for debugging post silicon speed failures at Intel. Operating such expensive machines consumes a substantial portion of the overall post silicon debug effort. Moreover, with the increasing complexity of manufacturing processes, these machines need to be renewed from one process generation to the next, which increases the product cost. This paper describes a novel method, based on formal technology, which brings a productivity breakthrough in isolating post-silicon speed failures. We demonstrate that in many cases optical probing using LADA can be fully replaced by our approach.**

I. Introduction

Due to the increasing design size and complexity of modern VLSI design and the decreasing time-to-market, design bugs are more likely to escape the pre-silicon verification and are only found after a chip has been manufactured. Therefore the efficiency of post-silicon debugging is becoming more critical to improve the productivity. With the rising demand for microprocessor core operating frequencies, challenges with on-die synchronization increase accordingly. Such synchronization challenges limit the upper frequency bound of a complex integrated circuit, and thus isolating and fixing performance-limiting circuits continues to consume a significant portion of the post-silicon validation bandwidth [1]. A *speedpath* is a frequency-limiting critical path which affects the performance of a chip [2], [3]. A speedpath that violates timing constraints at the post-silicon stage is called *failing speedpath*[4].

While pre-silicon static timing analysis [5], [6], [7] plays a vital role in facilitating fast and reasonably accurate measurement of circuit timing, post-silicon speed failures appear due to the use of simplified delay models, and due to the limited ability of such static timing analysis tools to consider the effects of logical interactions between signals. Such limitations are the reasons for the *mis-correlation* between the pre-silicon timing models and the post-silicon real behavior.

The process of debugging a speed failure on a multi-billion transistor microprocessor is a challenging, yet well structured process. It starts by applying test vectors to the microprocessor or by running a test program, such as end-user applications or functional tests, on the microprocessor until an error is detected. Such a process is applied on dedicated machines called *testers*. Post-silicon speed failures are normally observed when similar microprocessors produce different results on a tester at different frequencies. Post-silicon debugging is carried out to localize and rectify the root cause of the erroneous behavior. The fix of the failure is normally done by modifying the circuit either by replacing a cell/gate with a faster/slower one, or by performing a simple design retiming operation.

To assist the debugging process, design-for-test (DFT) features such as *scan* [8], [9] are added to the microprocessor. Such features increase the observability of the functional behavior of internal gates in the microprocessor. If the test fails, the values of the DFT scan gates are saved for debug purposes. For historical reasons, and due to cost reduction considerations, Intel didn't adopt the full-scan methodology. Instead, another technique which was developed to increase the debuggability of speed failures is based on on-die clock shrinking [10], [11] which helps narrow the list of failing source candidates. Such a technique is based on driving the microprocessor into clock regions (or domains) where global, regional and local clock distributions are introduced. In this way, post-silicon timing debugging can be improved by *controlling* the clock behavior of each of the clock domains in order to localize the source of the speed failure. The *tester* can be configured to operate the microprocessor at different clock frequencies for each of the above timing domains, and thus bind the source of the timing failure into smaller regions.

However, due to the large number of gates dominated by each clock domain (can reach *thousands* of sequential/storage signals per clock domain), there is still a need to narrow the list of failing source candidates into a smaller group of logic gates in an efficient and reliable manner. Such a technique, which is widely used at Intel today for debugging post-silicon speed failures, is a laser-based analytical technique, referred to as Laser Assisted Device Alteration - LADA [12]. LADA provides the ability to rapidly isolate failing speedpaths and their limiting components, down to the individual logic gate level with high confidence.

The LADA technique uses a laser incident from the device backside, to generate localized photocurrent within the active regions, temporarily altering transistor characteristics. Due to the different effect on PMOS versus NMOS devices, LADA can be used to speed up or slow down devices, so that when applied to devices in critical timing paths, performance

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

limiting circuits can be rapidly isolated.

Despite the successes of LADA in isolating post-silicon speed failures, such a process is still time consuming and labor intensive. In addition to the high cost of the LADA machines (>$1M for each machine), they require operators, sometimes with special expertise, to configure the machine to optically probe the regions of interest. Such a debugging process might take hours up to weeks per speed failure, depending on the complexity of the failure. When dealing with hundreds of speed failures to debug per microprocessor project, the debug process can take months, putting the whole project schedule at risk, or alternatively, forcing project management to make compromises on the marketed microprocessor frequency. A previous attempt to address this issue was presented in [13], but it was based on logic simulation and suffered from accuracy limitations and thus resulted in a large number of false paths. Another SAT-based attempt to address debugging post-silicon failing speedpaths was presented at [14], however the authors use a model which is based on using copies of a gate to represent the value of a gate at different points in time. In the worst case, the size of the model may be exponentially larger than the original circuit.

This paper suggests a novel SAT-based method to dramatically reduce the effort of the LADA based speed debug, saving the cost of the machines, reducing resources for operating them, and reducing the time-to-market (TTM) for launching new products. We will show that our tool can potentially replace the LADA based debug process. The superiority of our approach is in its ability to model speed failures without the need to have a timing model as we use a zero delay model. In addition, we use efficient modeling which avoids the possibility of exponential model size. As it will be shown, such efficient modeling is translated into better performance with the ability to deal with large instances taken from the latest Intel microprocessor designs.

The rest of this paper is organized as follows. In the next section, we present the notion of *functional failing speedpath* and present useful characteristics of it. Section III presents a framework for a precise, yet flexible, representation of the circuit network. Section IV describes the way we isolate failing speedpaths. Section V shows how our algorithm deals with reconverging paths, while section VI describes multiple approaches to dealing with complexity. Experimental results are reported in Section VII. Future work is discussed in section VIII. We conclude in section IX.

II. Characteristics of a failing speedpath

Splitting the design into multiple clock domains enables a flexible way to control the *phase* of the clock dominating a set of sequentials. The basic idea of being able to change the relative phases is to give some paths more (or less) time to complete. By doing this, we can trigger or remove timing problems. One way to reduce the region of interest that contains the source of speed failure is to perform a "trial and error" analysis which changes the relative phase of a given clock. This process is performed in a semi-automatic manner by iterating over the clock domains of the microprocessor. If after changing the phase of the clock of a given domain, the speed failure is still reproduced, we can then conclude that the signals responsible for the failure are not controlled by such a clock domain. We keep this process until we find the clock domain which eliminates the failure(s). This process is completed by finding two domains: the *source* (denoted by *src*) and the *destination* (denoted by *dst*). These are sets of sequential signals which bind combinational logic that contains the signal(s) responsible for the speed failure. See Fig. 1.

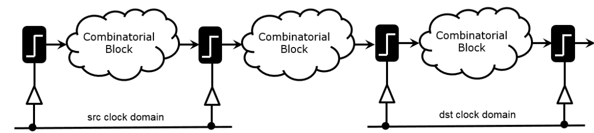


Fig. 1. Clock distribution

Each topological path starting from a sequential signal in the *src* domain, and ending at a sequential signal in the *dst* domain is called a *speed path*. A speedpath containing the errornous signal responsible for the speed failure is called a *failing speedpath*. A signal in a circuit M is said to be *toggling* at phase t in a trace π if the value of the signal at phase t is different from its value at phase $t-1$ in π. The failing speedpath originates normally from a *toggling* sequential in the *src* domain, at some phase, and the new value does not propagate properly. We refer to the first sequential in the path as the *root* signal. Notice that as part of the process of identifying the source and destination domains, the phase of the toggling sequential is detected as well. Another important characteristic of the failing speedpath is that it normally originates from *one* toggling root.

As mentioned earlier, the traditional method to isolate the failing speedpath is based on LADA machines. First, all the possible topological paths between signals in the *src* and the *dst* domains are computed. Then, using LADA machines, laser is used to temporarily alter the operating characteristics of transistors on the devices participating such paths. The device being tested is electrically stimulated and the device output is monitored. This technique is applied to the back side of the semiconductor device, thereby allowing direct access of the laser to the device active diffusion regions. The effect of the laser on the active transistor region is to generate a localized photocurrent. This photocurrent is a temporary effect and only occurs during the time that the laser is stimulating the target region. The creation of this photocurrent alters the transistor operating parameters, which may be observed as a change in the function of the device. The effect of this change in parameters may be to speed up or slow down the operation of the device. This makes LADA a suitable technique for determining critical timing paths within a semiconductor circuit [15].

From the perspective of logic behavior, one can consider the failing speedpath failure as a wrong propagation of a toggling

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

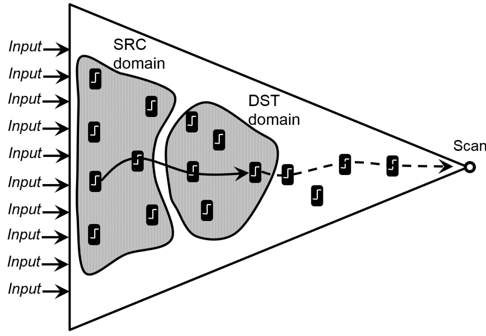


Fig. 2. A failing speedpath from SRC to DST, and its propagation

root sequential in the *src* domain. Instead of propagating a value *v*, the inverse value $\bar{v}$ propagates through its fanout logic and causes some *observable* (or scan signal) to get an inverted value which causes the test to fail. See the illustration in Fig. 2. The line from a sequential in the *src* domain (root) to the *dst* domain is the reported failing speedpath, while the dashed line is the propagation of its impact till the failing scan.

III. LOGIC PRESENTATION OF THE MICROPROCESSOR

A circuit design is modeled at the gate level in terms of combinational signals and sequential signals. Sequential signals can be of two types: (1) a latch, which is a device that transports its input to its output when the clock signal is high (T), and holds the output value when the clock signal is low (F), and (2) a flip flop, which is a device that transports the previous value of the input when the clock signal rises, and holds the output otherwise.

We consider *ternary* modeling of circuit node values. A value could be one of the *binary* values, T or F, or an *undefined* value, $\perp$ (elsewhere also denoted by X). Given a ternary input vector sequence π, and an initial ternary state s, n_t will denote the value of node n in a circuit M at time t after 3-valued simulation of M with π starting in s.

A circuit M can be represented by a collection of *next-state functions* (NSFs) of the sequentials as well as of the outputs, where a NSF is a function of current and next-state values of inputs and sequentials. For example, consider the circuit M which is illustrated in Fig. 3. It consists of five inputs $a, b, c, clk1$ and $clk2$, one latch l, one flop f, and an output (o) which is the output of the circuit. We denote the current state value of a variable "v" using v and the next state value of the same variable using v'. This way, the next state function of the output o is $l' \vee f'$, while the NSF of the active-high latch l is $(clk1' \wedge a' \wedge b') \vee (\neg clk1' \wedge l)$. The NSF of the rising-edge flop f is $(\neg clk2 \wedge clk2' \wedge c) \vee (\neg(\neg clk2 \wedge clk2') \wedge f)$. Available convenient representations for next state functions can be BDDs [16] or boolean expressions (simple graph data structures for representing propositional logic, where nodes of the graph represent binary operation $\wedge, \vee$, with an annotation whether a variable is negated or not, and variables appear as leafs). We adopted boolean expressions in our work since uniqueness

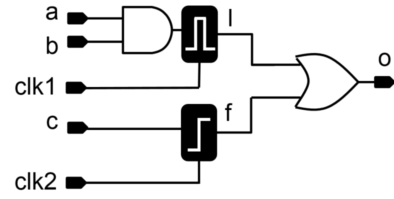


Fig. 3. Example of latch, flop and output functions

of BDDs is not needed. Modeling of sequential logic is done using a *compact* representation of infinite variable sequences. For a signal v, an infinite sequence of *propositional* variables $\{v_0, v_1, v_2, \cdots\}$ represents symbolically its sequential behavior. This allows one to reduce sequential verification problems to propositional satisfiability. The sequence representations can be unrolled to a desired depth k, producing k propositional variables $\{v_0, v_1, \cdots v_{k-1}\}$, which represent all the possible first k values of the signal v. This representation is suitable not only for modeling sequential behavior of inputs, but also for internal combinational signals, sequential signals, and outputs. For example, for a given output o in Figure 3, the sequential behavior is represented by a disjunction of the sequences representing l and f. Similarly, we can define the behavior of any sequential signal by using NSFs.

IV. DETECTING FAILING SPEEDPATHS USING FORMAL TECHNOLOGY

Our goal is to detect the failing speedpath within the hundreds (sometimes thousands) of speedpaths between the sequential signals in the *src* and *dst* domains, and thus bypass the manual and expensive optical probing stage. We provide the following inputs to the tool:

- Logic representation of the circuit in gate level Register Transfer Level (RTL) format (e.g. Verilog)
- A stimuli file containing the microprocessor simulation trace. This file results from simulating the test program on the RTL presenting the circuit starting from a given initial state. Since such trace might consist of thousands of simulation cycles, our tool extracts only a short window of it (see below)
- The name of the *src* and *dst* clock domains
- The phase when a *src* candidate toggled. It will be denoted by t_{src}.
- The scan signal where the failure was observed
- The phase in which the failure at the scan signal was observed. It will be denoted by t_{scan}.

Definition 4.1: Given a circuit M at a given state s, which is a result of 3-valued simulation of M with a ternary input vector sequence π at time t, a signal l is called to be *sensitive* at time t if flipping the value of l at time t in s causes the value of the failing *scan* signals to flip at a given phase t_{scan}.

A failing speedpath is thus a topological path starting from from some *sensitive* sequential in the *src* domain at time t_{src}. Detecting failing speedpaths starts after detecting the *src* and

dst clock domains. The idea behind our solution is to model the *symptom* of the speedpath failure using logic representation. Our method is based on modeling two machines: The *good* and the *bad* machine. The good machine models the functional behavior of the test as it is reproduced on RTL simulation when it is assumed to exhibit the correct functional behavior. The bad machine models the failure at the tester where the speed failure happens. Finding the failing speedpath is performed by running BMC [17], [18] comparing the sequential behavior of the two machines. The bound k for BMC is defined by the user (default value is set as 50) and it is an upper bound to the maximum length of the sequential depth between the phase of the failing scan and any sequential in the *src* domain.

The good machine is built as follows: the machine has one output which is the failing observed/scan signal. For the rest of this paper, we will refer to the failing observed signal as the failing scan signal. The sequential logic that drives the scan signal is modeled based on the real modeling in the circuit. The inputs, the output and the internal signals (both combinational and sequential signals) are constrained by the concrete values taken from each phase of the given trace window (of length 50). If for some reason, a signal does not have a trace in the given simulation traces, it is modeled as X. All signals have dual rail modeling [19], [20], [21], where each signal is modeled by a pair of variables.

Since we are dealing with complete microprocessor simulation, building the logic model of the complete full-chip was normally beyond the capacity limits of our internal logic modeling tools. Normally, the Verilog that models the schematics is very low in its abstraction level, compared to normal RTL which represents the abstract model of the design. In order to overcome such capacity issues, we black-box the irrelevant blocks and keep the ones between the block containing the failing scan, and the block containing the signals in the *src* domain.

The bad machine is built similarly to the good machine, but with the following differences:

- Assuming that the scan signal is failing at phase t_{scan}, and assuming that the trace of the scan signal in the good machine is $[v_0\ v_1 \cdots v_{t_{scan}} \cdots v_{k-1}]$ (where every v_i is a ternary value), we constrain the behavior of the failing scan signal in the bad machine with $[v_0\ v_1 \cdots \neg v_{t_{scan}} \cdots v_{k-1}]$. In other words, if the variable of the failing scan at time t_{scan} is annotated by v and the concrete simulation value of the failing scan at the same time is c, then we add a constraint that v is equivalent to $\neg c$. Notice that it is necessary to have a binary trace value for the failing scan signal at the failing phase t_{scan}. Otherwise, the algorithm aborts.
- The set of the *src* candidate sequentials will be denoted by *S*. For each sequential $s \in$ *S*, we add a new XOR gate with two entries: the first is fed by the sequential s and the second is fed by a new free inputs $s_control$. The logic that was fed originally by the sequential s will be fed now by the new XOR node. Each XOR signal enables modeling the flipping of the value of the sequential s in the sense that if the control signal $s_control$ has a value of T, then the output of the XOR is simply the inverse of the value of s. See Fig. 4 for illustration.

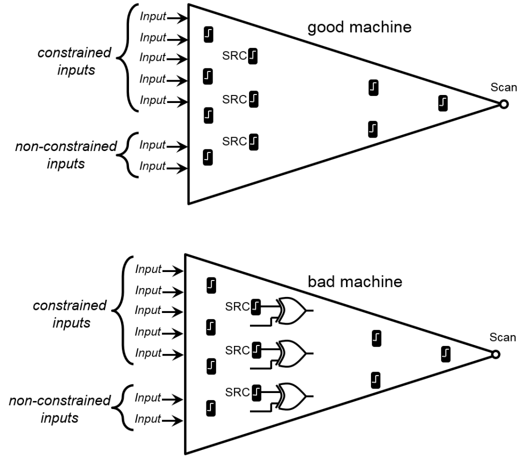


Fig. 4. Good and bad machine modeling

- Assuming that the failing speedpath originates from a sequential which toggles at a given phase t_{src}, we assume that the value of each control input $s_control$ is a free variable at phase t_{src}, and is constrained to F for the rest of the phases:

$$\forall_{s \in \boldsymbol{S}}.\forall_{0 \leq t \leq k \wedge t \neq t_{src}}.(s_control_t = F) \qquad (1)$$

- Since we assume that the failing speedpath originates from only *one* source, we add an extra constraint that only one control variable can be T (at phase t_{src}).

A SAT-based bounded model checker (BMC) is called to find a satisfying assignment to the above constraints. If a satisfying assignment is found, then we extract the root sequential out of the counter examples by finding the control variable which got a value of T. Extracting a complete path is done by backward traversal from the scan signal, by comparing the values of each signal in the good and bad machine.

Definition 4.2: A *failing functional speedpath* is a sequence of pairs of the form $\{(sig_0, ph_0), (sig_1, ph_1), \ldots, (sig_n, ph_n)\}$, where $ph_0 \leq ph_1 \leq \ldots \leq ph_n$, sig_i is a signal name, and ph_i is the *first* phase where the signal sig_i got different values between the good and the bad machines.

Clearly, $t_{src} \leq ph_i \leq t_{scan}$, and the path starts with a pair (s, t_{src}) for some sequential s in the *src* domain, and ends with a pair $(scan, t_{scan})$. Other signals in the speedpath can be either sequential or combinational ones. For the rest for the paper, the failing functional speedpath will be referred to as the failing speedpath. Generating multiple paths is performed via an iterative process where new constraints are added at each iteration to direct the BMC engine not to reproduce the found path for the next time. We annotate the value of a

signal s in the good/bad machine at time t with s_t^{good} and s_t^{bad} respectively. The constraint which is added to prevent a failing functional speedpath P from being generated again is:

$$\neg \bigwedge_{(s,t)\in \boldsymbol{P}} (s_t^{good} \neq s_t^{bad}) \quad (2)$$

Notice that paths which do not go through any sequential in the *dst* domain are excluded from the tool report. The last iteration happens when BMC does not find any new speedpath.

V. Handling reconverging paths

In this section, we describe the main challenge in achieving this kind of symbiosis between timing analysis and formal technology. Consider the simple AND gate illustrated in Fig. 5. From a logic analysis perspective, regardless whether it is

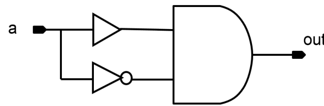


Fig. 5. Differences between timing and logic analysis

logic simulation or formal analysis, the output of the circuit at output *out* is `F`, even if the value of the input a transitions from `F` to `T`. Though, from a timing perspective, a transition from `F` to `T` at the input a might be propagated at different speeds through the buffer and inverter, resulting in two `T`'s at the entries of the AND gate and causing the output *out* to get a value of `T` for a short period of time. We call this phenomenon a *glitch* and it can be one of the reasons for speed failures as the output *out* can be captured with the wrong value. Clearly, the logic representation of the circuit does not capture the speed behavior described in the simple AND illustration, and it definitely limits the tool from being able to isolate real failing speedpaths. Looking into the problem in a more generic view, the problem results when the cone of influence of the scan signal contains internal signals which form a *reconvergence* point of different paths starting from the same root signal. The propagation of the toggling value on the root signals is *masked* by a value of two or more signals feeding the same reconvergence point, which masks further propagation of the value till the failing scan signal.

In this section, we describe a novel technique for dealing with masking values at reconvergence points. The idea is based on performing a naive *topological* analysis on the cone of influence to detect the reconvergence points. For each convergence signal s, with n inputs denoted by I_i where $0 \leq i < n$, we perform the following modeling modifications to the bad machine:

- For each input I_i, we introduce a new MUX gate with two entries. The first entry will capture the signl I_i from the good machine, while the second entry will capture the signals I_i from bad machine. The selector of the MUX gate will be a new free variable $sensitivity_selector_I_i$.

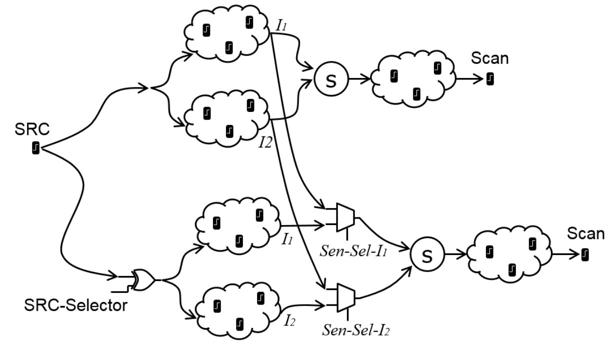


Fig. 6. Handling reconvergence points

- Each signal driven by the signal s will be now driven by the new MUX gate.
- We add a constraint that only one of the sensitivity selector variables $sensitivity_selector_I_i$ ($0 \leq i < n$) can be `T`. To clarify, this constraint is added for each reconvergence signal separately.

The basic assumption behind the above is the fact that a valid failing speedpath contains signals with only *one* driver that has a value in the bad machine which is inverse to the value in the good machine. By adding the above MUX gates, adding an assumption that only one selector can be `T` ensures that only one of the immediate drivers has an inverse value between the good and the bad machines. Fig. 6 illustrates the approach. The upper part of the illustration models the good machine while the lower part models the bad one. The XOR gate models flipping the value of SRC if the selector is `T`. The two MUX gates driving s are responsible for handling the reconverging signals I_1 and I_2, while their selectors guarantee that only one value out of I_1 and I_2 propagates to s in the bad machine. Notice that extracting the failing speedpath is done with a simple modification: for each reconvergence signal, we go backwards at the immediate driver with the active sensitivity selector.

VI. Dealing with complexity

Another challenge that we faced was run time of the algorithm for instances with a large sequential depth between the failing scan signal and the *src* candidates. The cone of influence was computed using a naive breadth-first search (BFS) on the sequential signals up to *src* candidates. In some cases, this computation resulted in cones with thousands of sequential signals which caused the core BMC engine to choke. We have developed an iterative process for computing the cone of influence based on functional detection of sensitivity of sequential signals.

Sensitivity of a sequential signal s at phase t is detected using our algorithm by assuming that s is the *src* candidate and assuming that $t_{src} = t$. The motivation behind the above iterative expansion process is that if a sequential signal is not sensitive during the window $[t_{src}, t_{scan}]$, then there is no need

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

to expand the cone over it, and thus it can be abstracted and considered as an input (constrained by the trace).

Data: scan, t_{src}, t_{scan}
Result: Compute cone of influence
$t_{current}$ = t_{scan};
ExpansionList = {} ;
while $\{t_{current} \neq t_{src}\}$ **do**
  C = ComputeConeOfInfluence(scan, ExpansionList);
  L = ExtractSetOfSequentialsAtBoundary (*C*);
  S = FilterSensitive (*L*, $t_{current}$);
  if *empty S* **then** $t_{current}$ = $t_{current}$ - 1;
  else ExpansionList = ExpansionList ∪ *S* ;
end
return ComputeConeOfInfleuce(ExpansionList) ;

A pseudocode explaining the steps of the algorithm for computing the cone of influence is presented herein: The function ComputeConeOfInfluece accepts as an argument a list of sequential signals and computes the cone of influence starting from the failing *scan* signal going backwards while stopping at the first sequential signals which do not belong to the list of sequential signals included in ExpansionList. The function ExtractSetOfSequentialsAtBoundary computes the sequential signals at the boundary of the cone of influence. The function FilterSensitive finds sensitive sequential signals belonging to the set *L* at phase $t_{current}$. The algorithm keeps expanding at sensitive sequentials at phase $t_{current}$ till no more sensitive sequentials are found, and only then it decreases the phase $t_{current}$. The algorithm stops at phase t_{src}. For illustration, during the first iteration, *scan* will be detected as sensitive at t_{scan} and thus it will be added to ExpansionList.

The algorithm computes a sub-cone of the cone computed by the naive BFS approach, and thus the detection of the failing speedpaths happens in a smaller cone which BMC can handle. The algorithm is sound in the sense that if a failing speedpath exists in the cone generated by the naive BFS approach, then it is guaranteed to be detected at the sub-cone generated by the iterative expansion algorithm. The proof for the soundness of the algorithm is based on showing a contradiction between the existence of such a path, and the fact that the algorithm didn't expand on a sequential in the boundary of the the sub-cone. Illustration of the proof is presented in Fig. 7.

Let M be a circuit with a failing scan signal $scan$ at phase t_{scan} and let **C** be the topological cone of influence of $scan$ computed using the naive BFS, where the stop points are primary inputs of M or sequential signals driving *src* candidates. Let us denote the set of internal sequential signals in **C** by L. Let L_{SRC} be a set of *src* candidates where $L_{SRC} \subseteq L$. Let **C′** be the topological cone of influence of $scan$ computed by expanding on a set of internal sequential signals L', where $L' \subseteq L$. We denote the phase when a sequential was expanded with t^e and the first phase when the sequential was sensitive by t^s.

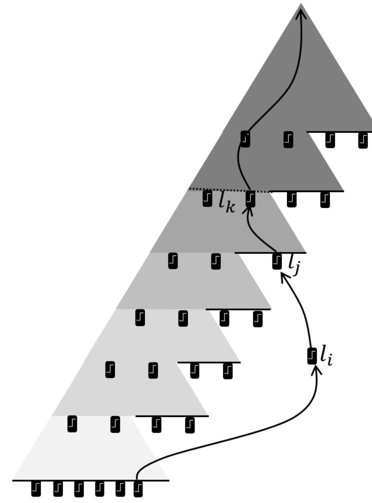


Fig. 7. Iterative expansion soundness

Lemma 6.1: If a sequential signal l_i is sensitive at phase t^s_i, and it was expanded in the iterative expansion process at phase t^e_i, then $t^s_i \leq t^e_i$.

Proof: If l_i is sensitive at phase t^s_i, then there is some failing speedpath P starting at (l_i, t^s_i) and containing the sequentials $\{(l_i, t^s_i), (l_{i+1}, t^s_{i+1}), \ldots, (l_n, t^s_n)\}$ (recall that $l_n = scan$) where every sequential l_i drives l_{i+1} through a combinational cloud, and $t^s_j \leq t^s_{j+1}$ for $i \leq j < n$. Recall also that if (l, t) belongs to a path, then l is sensitive at time t. Let us denote a list of corresponding phases $\{t^e_i, t^e_{i+1}, \ldots, t^e_n\}$ annotating for each sequential l_i the phase t^e_i when it was expanded in the iterative expansion. Recall that t^e_n is the phase when *scan* was expanded and t^s_n and is the phase when the it was sensitive. Both values should be equal to t_{scan}.

We will first show that for each j where $i \leq j < n$, that if $t^s_j \leq t^e_j$ then $t^e_{j+1} < t^s_j$. Let us assume on the contrary that $t^s_j \leq t^e_{j+1}$, since the algorithm detects sensitivity of l_j at the window $[t_{src}, \ldots, t^e_{j+1}]$, and the fact that l_j was found sensitive only at phase t^e_j, means that the algorithm didn't detect sensitivity of l_j at t^s_j which contradicts the fact that l_j is sensitive at t^s_j.

Thus $t^e_{j+1} < t^s_j$ and since $t^s_j < t^s_{j+1}$, we conclude that $t^e_{j+1} < t^s_{j+1}$. Based on that, $t^e_i < t^s_i$ implies $t^e_j < t^s_j$ for each $i \leq j \leq n$, implying that $t^e_n < t^s_n$. This is a contradiction since it means that the expansion phase for the *scan* signal is less than the sensitive phase of the same signal which contradicts the fact that they should be equal.

■

Theorem 6.2: If the algorithm detects a set of failing speedpaths SP for the cone C, then it will detect the same set of failing speedpaths for the sub-cone C'.

Proof: Let us assume on the contrary that there is a failing speedpath $P = \{(l_1, t_1), (l_2, t_2), \ldots, (l_n, t_n)\} \in SP$ which is not detected in C'. Then there exists a sequential

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Test No.	# signals in cone	# of inputs on boundary	# of latches in cone	# of reconverg. signals	# of iterations	path length (in phases)	**# of paths**	Run Time (Sec.)
1	296	26	2	4	5	3	**1**	248
2	509	67	14	11	6	4	**1**	278
5	405	54	3	12	11	8	**1**	214
7	305	19	3	0	6	4	**1**	290
9	248	11	1	0	1	1	**1**	186
13	517	50	14	26	55	44	**1**	227
15	497	83	4	3	7	4	**1**	222
16	1528	212	59	86	14	8	**1**	745
18	27696	3009	635	8569	31	16	**1**	7168
6	3025	617	43	650	15	8	**2**	434
11	2403	345	22	209	12	7	**2**	318
12	1798	258	58	236	33	20	**2**	442
10	855	164	8	27	8	5	**3**	222
17	25895	7279	294	1070	30	16	**3**	6458
21	21864	4618	165	2266	33	18	**3**	3395
8	855	164	8	27	8	5	**4**	242
22	1545	303	46	5	12	6	**5**	5555
14	837	90	39	29	23	12	**6**	619
4	4665	704	106	1149	31	18	**7**	579
19	8789	994	125	2132	26	14	**7**	1713
20	26226	4035	168	2422	27	14	**15**	3285
3	4931	675	167	689	27	14	**40**	780

TABLE I

FAILING SPEEDPATHS FOUND ON NEXT GENERATION INTEL MICROPROCESSOR

signal $(l_i, t_i^s) \in P$ where l_i belongs to L but not L', and there is a combinational path from l_i to a boundary sequential l_j where $(l_j, t_j^s) \in P$ and $l_j \in L'$, and there is a combinational path from l_j to an internal sequential l_k where $(l_k, t_k^s) \in P$ and $l_k \in L'$. Recall that $t_j^s \leq t_k^s$. Let us assume that the l_k was expanded at phase t_k^e. Since l_k is part of the path, then based on lemma 6.1, $t_k^s \leq t_k^e$, and thus $t_j^s \leq t_k^e$. Recall that l_j is driving l_k, and l_k was expanded at phase t_k^e. The fact that the iterative expansion couldn't detect sensitivity for l_j during the window $[t_{src} \dots t_k^e]$, and the fact that t_j^s belongs to that window, contradicts the fact that l_j is sensitive at t_j^s. ■

VII. Results

We are currently at the early deployment stage of our application to the post-silicon speedpath debug lab responsible for the quality of the next generation Intel microprocessor. Most of the speedpaths shown in table I were detected using LADA first, and our tool was run afterwards to demonstrate its ability to detect the same failing speedpaths. In all the testcases shown in the table, we successfully found the same failing speedpath which was detected by LADA. For some cases, it took about two weeks trying detect the failing speedpath using LADA with no success, but after running the tool, the tool was able to isolate the failing speedpath easily. In other cases, our tool was run before LADA and was able to detect the failing speedpath and thus LADA was bypassed totally. Table I presents some information about each failing speedpath, when the cone of influence was produced using the iterative expansion algorithm. Column 2 shows that number of the signals in the cone computed by the iterative expansion algorithm. Column 3 shows the number of variables at the boundary of the cone, while column 4 shows the number of the internal sequential signals in the cone of influence. Column 5 shows the number of the internal reconvergence signals in the cone while column 6 shows the number of the expansion iterations to compute the cone of influence. Columns 7 shows the path length in phases from the path *root* to *scan* while column 8 shows the number of the paths detected by the tool. Run time of the tool is shown in column 8. These results were produced on a 2.6 GHz Intel(R) Xeon(R) CPU processor. The run time demonstrates that isolation of post-silicon failing speedpaths can be completed in less than two hours using our tool compared to the costly, manual LADA based process, which took about a day to debug in average per failing speedpath.

VIII. Future work

We have already started to see first cases where the vision of eliminating the need for optical probing for speed debug becomes a reality. Our next steps are to penetrate this technology to be used across the different microprocessor projects at Intel. Our next challenge is to eliminate the need to generate the RTL simulation trace, which today consumes long hours per test. Our alternative will be to get partial trace from the tester which produces the trace values for the scan signals only. Our algorithm will work the same, but will have to deal with more signals that do not have concrete value, but have X value instead. We hope that the scan signals will have enough coverage of the sequential signals so that they will be able to propagate concrete values coming from the scan

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

signals, forward and backward, and thus eliminating the X values. Current results are encouraging and we are optimistic that we will be able to reduce (and possibly eliminate) the LADA effort for all Intel microprocessor projects.

IX. Summary

We have introduced a new SAT-based algorithm that enables detecting failing speedpaths that are detected at the post-silicon debug stage. The value that this method brings is by reducing (and possibly eliminating) the cost of post-silicon speed debug using optical probing which is done today at Intel using LADA machines. Such a process consumes expensive machines, operators and costly TTM. Our method uses formal technology to model the incorrect behavior of the silicon from a functional perspective. We introduced a novel technique to model glitches by introducing new MUX gates in the reconvergence gates.

References

[1] S. Mitra, S. A. Seshia, N. Nicolici, "Post-silicon Validation Opportunities, Challenges and Recent Advances", in Design Automation Conference (DAC), 2010.

[2] P. Bastani, K. Killpack, L.-C. Wang, and E. Chiprout, "Speedpath prediction based on learning from a small set of examples", in *Design Automation Conf.*, 2008, pp. 217222.

[3] L. Lee, L.-C. Wang, P. Parvathala, and T. M. Mak, "On silicon-based speed path identification", in *VLSI Test Symp.*, 2005, pp. 3541.

[4] L. Xie, A. Davoodi, and K. K. Saluja, "Post-silicon diagnosis of segments of failing speedpaths due to manufacturing variations", in *Design Automation Conf.*, 2010, pp. 274279.

[5] T.M. McWilliams, "Verification of timing constraints on large digital systems", in *DAC*, 1980, pp. 139-147.

[6] G. Martin, J. Berrie, T. Little, D. Mackay, J. McVean, D. Tomsett, L. Weston. "An integrated LSI design aids system", in *Microelectronics Journal*, Vol. 12, Issue 4, 1981, Pages 1822.

[7] R. Hitchcock, G.L. Smith, and D.D. Cheng. "Timing analysis of computer hardware", in *IBM Journal of Research and Development (IBM)*, Vol. 26, Issue 1, 1982, pp. 100105.

[8] R. S. Venkataraman, "A Technique for Fault Diagnosis of Defects in Scan Chains", in *Int. Test Conference Proc.*, 2001, pp. 268-277.

[9] K. Cheng, "Partial Scan Designs Without Using a Separate Scan Clock", in *VLSI Test Symposium. Proc., 13th IEEE*, pp. 277-282.

[10] S. Rusu and S. Tam, "Clock Generation and Distribution for the First IA-64 Microprocessor", in *IEEE Solid State Circuits Conference*, 2000, pp. 176-177.

[11] S. Tam, S. Rusu, U. Nagarji Desai, R. Kim, Ji Zhang, I. Young, "Clock Generation and Distribution for the First IA-64 Microprocessor", in *IEEE Journal of Solid State Circuits*, Vol. 35, 2000, pp. 1545- 1552.

[12] R. Rowlette; T. Eiles, "Critical Timing Analysis in Microprocessors Using Near-IR Laser Assisted Device Alteration (LADA)", in *Proc. IEEE International Test Conf.*, 2003, pp. 264-273.

[13] R. McLaughlin; S. Venkataraman and C. Lim, "Automated Debug of speedpath Failures Using Functional Tests", in *VLSI Test Symposium*, 2009, pp. 91-96.

[14] M. Dehbashi; G. Fey, "Automated Post-Silicon Debugging of Failing Speedpaths", in *Asian Test Symposium*, 2012, pp. 13-18.

[15] C. H. Kong and E. P. Castro. "Application of LADA for Post-Silicon Test Content and Diagnostic Tool Validation", in *Proceedings of the 32nd International Symposium for Testing and Failure Analysis*, pp. 4317, 2006.

[16] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation" in *IEEE Transactions on Computers*, Vol. 35 Issue 8, 1986, pp. 677-691.

[17] A. Biere, A. Cimatti, E. Clarke. "Symbolic model checking without BDDs" in *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 1579, 1999, pp. 193-207.

[18] A. Biere, A. Cimatti, E. Clarke, M. Fujita. Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs" in *DAC*, 1999.

[19] R. E. Bryant, "Boolean Analysis of MOS Circuits", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 6, No. 4, 1987, pp. 634 - 649.

[20] C-J.H. Seger, R.E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories", in *Formal Methods in System Design*, Vol 6, No. 2, 1995, pp. 147-189.

[21] D. Kaiss, M. Skaba, Z. Hanna, Z. Khasidashvili, "Industrial Strength SAT-based Alignability Algorithm for Hardware Equivalence Verification", in *FMCAD*, 2007.

Leveraging Linear and Mixed Integer Programming for SMT

Tim King* Clark Barrett* Cesare Tinelli†

*New York University †The University of Iowa

***Abstract*—SMT solvers combine SAT reasoning with specialized theory solvers either to find a feasible solution to a set of constraints or to prove that no such solution exists. Linear programming (LP) solvers come from the tradition of optimization, and are designed to find feasible solutions that are optimal with respect to some optimization function. Typical LP solvers are designed to solve large systems quickly using floating point arithmetic. Because floating point arithmetic is inexact, rounding errors can lead to incorrect results, making inexact solvers inappropriate for direct use in theorem proving. Previous efforts to leverage such solvers in the context of SMT have concluded that in addition to being potentially unsound, such solvers are too heavyweight to compete in the context of SMT. In this paper, we describe a technique for integrating LP solvers that improves the performance of SMT solvers without compromising correctness. These techniques have been implemented using the SMT solver CVC4 and the LP solver GLPK. Experiments show that this implementation outperforms other state-of-the-art SMT solvers on the QF_LRA SMT-LIB benchmarks and is competitive on the QF_LIA benchmarks.**

I. Introduction

Solvers for Satisfiability Modulo Theories (SMT) combine the ability of fast Boolean satisfiability (SAT) solvers to find solutions for complex propositional formulas with the ability of specialized *theory solvers* to find solutions to systems of constraints with respect to specific first order theories. SMT solvers excel in applications that require reasoning about non-trivial Boolean combinations of specific theory atoms.

Theory solvers for linear real and integer arithmetic are found in nearly every modern SMT solver, and are an essential building block for verification applications built on top of SMT. The best-performing arithmetic theory solvers are based on an algorithm that adapts the well-known simplex method to the SMT setting [1]. Because of their use in verification, SMT solvers typically use exact precision numeric representations internally in order to ensure that their calculations are correct and do not compromise the soundness of the overall system. For many typical SMT problems with significant Boolean structure (such as the majority found in the SMT-LIB benchmark library), this approach is sufficient, as the required theory reasoning is not too complex and the numbers involved in the internal calculations tend to stay relatively small. Moreover, such problems require tens or hundreds of thousands of calls to the theory solver. Thus, the theory solver's ability to incorporate new constraints quickly, to rapidly detect inconsistencies, and to backtrack efficiently, are far more important for overall efficiency than is the speed of the internal numerical calculations. However, there do exist problems for which this is not the case. If the internal simplex solver receives constraints that lead to large and dense linear systems, then using exact precision for the calculations required for the simplex search can overwhelm the solver.

The standard simplex algorithm finds a solution that is "best" according to some criteria. This is made mathematically explicit by adding a linear objective function that is to be maximized. The linear constraints combined with a linear objective are called *Linear Programs* (LPs), and systems that solve them are called *LP solvers*. Simplex-based LP solvers differ from SMT solvers in several important ways, including the following: (i) LP solvers solve only conjunctions of constraints - they cannot handle arbitrary Boolean combinations; (ii) LP solvers focus on both feasibility and optimization rather than just feasibility; (iii) LP solvers (generally) use floating point rather than exact precision arithmetic internally; and (iv) the product of many decades of research, modern LP solvers incorporate highly sophisticated techniques, making them very efficient in practice. The techniques used in LP solvers have been extended to the problem of optimizing constraints where all or some of the variables are required to be integers (Integer Programming (IP) and Mixed Integer Programming (MIP)).

On challenging simplex instances, LP and MIP solvers are considerably more efficient than the techniques used inside of SMT solvers. However, LP and MIP solvers are not optimized for rapid incremental calls, making them inefficient as theory solvers for many SMT applications. Also, their use of floating point means that they will occasionally return incorrect results. In this paper, we show how LP and MIP solvers can be efficiently and soundly incorporated into a modern SMT solver. Our work builds on previous efforts to leverage LP solvers for SMT but is the first to obtain significant improvements in performance by doing so. It is also the first to attempt integrating a MIP solver with SMT.

The rest of the paper is organized as follows. We give an overview of relevant background on SMT and simplex in Section II. Section III discusses our approach for integrating an LP solver in a theory solver for linear real arithmetic, and section IV shows how to extend this strategy to use an MIP solver in a theory solver for linear integer arithmetic. We conclude with section V, which reports and discusses experimental results.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

II. Background

The core SMT problem is to determine whether a first order formula ϕ is satisfiable with respect to a fixed first order theory $\mathcal{T}$ [2]. Most modern SMT solvers rely on a DPLL($\mathcal{T}$) framework which combines a Boolean satisfiability (SAT) solver with decision procedures for various theories. The SAT solver is used to find an assignment of theory literals to truth values that is propositionally consistent with the Boolean skeleton of ϕ. The theory-specific modules used by SMT solvers are called *theory solvers*.

A theory solver for theory $\mathcal{T}$ takes as input a set Φ of theory literals[1] and determines whether Φ is consistent with respect to $\mathcal{T}$. If so, the theory solver responds with $\mathbf{Sat}$ and (optionally) a *model*, an assignment to the free variables in Φ that makes every formula in Φ true. If not, the theory solver responds with $\mathbf{Unsat}$ together with a small (ideally minimal) subset of Φ known to be unsatisfiable in $\mathcal{T}$, called a *conflict* set. A conflict set C is converted into a clause $\bigvee_{l \in C} \neg l$, and sent to the SAT solver. In addition, any $\mathcal{T}$-valid formula can be sent to the SAT solver by the theory solver and such formulas are called *theory lemmas*. Theory lemmas are used by theory solvers to help direct and guide the SAT solver during its search.

The focus of this paper is a novel theory solver for quantifier-free mixed linear integer and real arithmetic. We assume a language that includes the usual arithmetic constants and operators, a vector $\mathcal{V} = \langle x_1 \ldots x_n \rangle$ of variables,[2] and a unary predicate $\texttt{IsInt}$. We assume that atoms are of the form (i) $\sum c_i \cdot x_i \bowtie d$ where c_i and d are rational constants, $\bowtie \in \{<, \leq, =\}$, and $x_i \in \mathcal{V}$, or of the form (ii) $\texttt{IsInt}(x_i)$, where $x_i \in \mathcal{V}$. An *assignment* a maps each $x_i \in \mathcal{V}$ to a value in the set $\mathbb{R}$ of real numbers. An assignment a *satisfies* an atom $\sum c_i \cdot x_i \bowtie d$ whenever $\sum c_i \cdot a(x_i) \bowtie d$ holds and satisfies $\texttt{IsInt}(x_i)$ whenever $a(x_i)$ is an integer. An atom that is satisfied by some assignment is said to be *satisfiable*. We lift the notion of satisfaction to arbitrary Boolean combinations of atoms in the natural way. We write $\phi \models_T \beta$ if every assignment satisfying ϕ also satisfies β. We assume that $\mathcal{V}$ is partitioned into a set $\mathcal{V}_\mathbb{R}$ of real variables and a set $\mathcal{V}_\mathbb{Z}$ of integer variables. The *integer-tightening* of a formula Φ is defined as $\phi \wedge \bigwedge_{z \in \mathcal{V}_\mathbb{Z}} \texttt{IsInt}(z)$, and the *real relaxation* of a formula α is obtained by replacing every application of the $\texttt{IsInt}$ predicate in α by $\texttt{True}$.[3] A formula is *integer-feasible* if its integer-tightening is satisfiable (and *integer-infeasible* otherwise), and an assignment is called *integer-compatible* if it assigns an integer to each integer variable. If a formula's real relaxation is satisfied by some assignment, we say it is *real-feasible* (or just *feasible*).

We first describe the well-known approach for simplex-based theory solvers in SMT, an approach we call *Simplex for DPLL($\mathcal{T}$)* (more details can be found in [1], [3]). The input is a conjunction of atoms of the form $\sum c_i \cdot x_i \leq d$. Weak inequalities are transformed by introducing a fresh real variable s for $\sum c_i \cdot x_i$ and rewriting the constraint as $s = \sum c_i \cdot x_i \wedge s \leq d$. The original x_i variables are called *structural* while the introduced s variables are called *auxiliary*. The orig function maps each auxiliary variable to its definition, $\text{orig}(s) \equiv \sum c_i \cdot x_i$. Strict inequalities $\sum c_i \cdot x_i < d$ are rewritten as $\sum c_i \cdot x_i + \delta \leq d$ where δ is a small constant that can be determined later. To properly reason in the presence of δ, some of the internal constants are represented as special δ-*rationals*, pairs $\langle a, b \rangle$ of rationals interpreted as $a + b \cdot \delta$. Details on this technique can be found in [4].

After applying these transformations, the resulting constraints can always be written as: $T\mathcal{V} = 0 \wedge l \leq \mathcal{V} \leq u$, where T is a matrix, and l and u are vectors of lower and upper bounds on the variables. We use T_i to denote the i-th row of T. We use $l(x)$ and $u(x)$ to denote the lower and upper bound on a specific variable x. If x has no lower (upper) bound, then $l(x) = -\infty$ ($u(x) = +\infty$). The theory solver searches for an assignment $a : \mathcal{V} \mapsto \mathbb{R}$ that satisfies the constraints.

We assume T is an $n \times n$ matrix in *tableau form*: the variables $\mathcal{V}$ are partitioned into the *basic* variables $\mathcal{B}$ and *non-basic* variables $\mathcal{N}$ (to emphasize when a variable x_i is basic, we will write b_i as a synonym for x_i when $x_i \in \mathcal{B}$), and T_i is all zeroes iff $x_i \in \mathcal{N}$. Furthermore, for each column i such that $b_i \in \mathcal{B}$, we have $T_{k,i} = 0$ for all $k \neq i$ and $T_{i,i} = -1$. Thus, each nonzero row T_i of T represents a constraint $b_i = \sum_{x_j \in \mathcal{N}} T_{i,j} \cdot x_j$. Initially, the basic variables are exactly the auxiliary variables.

The simplex solver works by making a series of changes to an initial assignment a and the tableau T until the constraints are satisfied or determined to be unsatisfiable. During this process, $T \cdot a(\mathcal{V}) = 0$ is an invariant. To initially satisfy this invariant, we can set $a(x_i) = 0$ for all i. To maintain the invariant, whenever the assignment to a non-basic variable changes, the assignments to all dependent basic variables are also updated. Changes to the tableau are made via *pivoting*. Pivoting takes a basic variable b_i and a non-basic variable x_j such that $T_{i,j} \neq 0$, and swaps them: after pivoting, x_j becomes basic and b_i becomes non-basic.

Simplex for DPLL($\mathcal{T}$) solvers modify the assignment a and pivot the tableau T until a satisfying assignment is found or a *row conflict* is detected: a basic variable b_i violates one of its bounds but none of the non-basic variables that b_i depends on can be used to fix this without violating their own bounds. For example, suppose $a(b_i) > u(b_i)$ and for all $x_j \in \mathcal{N}$ with positive coefficients in row T_i ($T_{i,j} > 0$), $a(x_j) = l(x_j)$ and for all $x_k \in \mathcal{N}$ with negative coefficients in row T_i ($T_{i,k} < 0$), $a(x_k) = u(x_k)$. Then, $b_i \geq a(b_i)$ is entailed by the row and the constraints on the non-basic variables.[4] Since this contradicts $b_i \leq u(b_i)$, the entire system of constraints is unsatisfiable, and the following conflict set is generated:

$$\bigcup_{T_{i,j} > 0} \{x_j \geq l(x_j)\} \cup \bigcup_{T_{i,k} < 0} \{x_k \leq u(x_k)\} \cup \{b_i \leq u(b_i)\}.$$

[1]We will follow the common practice of overloading Φ to mean $\bigwedge_{\varphi \in \Phi} \varphi$ in contexts where a formula rather than a set is expected.

[2]For convenience, we will also use $\mathcal{V}$ to refer to the set $\{x_1 \ldots x_n\}$.

[3]We assume that $\texttt{IsInt}$ occurs only positively in input formulas.

[4]There is a dual case when $a(b_i) < l(b_i)$.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

The current best implementations of theory solvers for mixed linear integer and real arithmetic use a sound but incomplete procedure that layers integer reasoning on top of a solver for linear real arithmetic. Given a set Φ of atoms, the real solver is first used to solve the real relaxation of Φ. If the solver terminates, the result is either a conflict set or an assignment a (when Φ is real-feasible). In the first case, no additional work is necessary as a conflict set for the real relaxation of Φ is also a conflict set for Φ. In the second case, the assignment a is examined to see whether it is integer-compatible. If not, more work is needed to refine the assignment. The following *branching* technique can be used to ensure that the current assignment is refined in the next invocation of the theory solver: select a variable $x \in \mathcal{V}_{\mathbb{Z}}$ whose assignment is non-integer, and then send the following theory lemma to the SAT solver,

$$\texttt{IsInt}(x) \rightarrow (x \leq \lfloor a(x) \rfloor \vee x \geq \lceil a(x) \rceil) \qquad (1)$$

The SAT solver will assert one of the two new bounds on x before reinvoking the theory solver.

Naive use of this heuristic can trigger an infinite sequence of branches, so more sophisticated methods based on *cutting planes* have been developed [5]. Consider a set Φ of assertions. A cutting plane is a plane through the solution space of the real relaxation of Φ that *cuts off* some of the non-integer-compatible assignments. More precisely, $\sum c_i x_i = d$ is a cutting plane for Φ and $\mathcal{H} \equiv \sum c_i x_i \leq d$ is a *cut* iff the following conditions hold: (i) every assignment satisfying the integer-tightening of Φ also satisfies $\mathcal{H}$; and (ii) at least one assignment satisfying the real relaxation of Φ also satisfies $\neg\mathcal{H}$.[5] The inequality $\mathcal{H}$ can be safely added to Φ without changing any of the (integer-compatible) satisfying assignments. A cut is always entailed by the integer-tightening of Φ and never by the real relaxation of Φ. Cuts can be implemented using theory lemmas, by sending the lemma $\Phi \Rightarrow \mathcal{H}$ to the SAT solver. Previous work has looked at using Gomory and Mixed Gomory cut techniques in SMT solvers [4].

III. Leveraging LP Solvers

The first contribution of this paper is a method for leveraging the strengths of both SMT and LP solvers to construct an efficient and robust theory solver for linear real arithmetic. This idea has been explored before. Early work by Yu and Malik [6] reports results on using an LP solver as a theory solver for SMT, but the issue of potentially incorrect results from the LP solver is not addressed. Faure et al. [7] integrate several LP solvers into the Barcelogic SMT solver [8]. They use an exact solver to lazily check the results from the LP solver to ensure soundness. Finally, in recent work by de Oliveira and Monniaux [9] (a continuation of the work in [10]), extensive experiments are done using an LP solver within OpenSMT [11]. In this work, the LP solver is called first and the results are used to "seed" the search in the exact solver. Thus most of the search is done by the LP solver, while the exact solver still ensures correctness.

In each of these studies, experimental results on SMT-LIB benchmarks show that existing SMT solvers outperform the experimental solvers modified to use LP solvers, even if the LP solver results are not checked for correctness. The main reason for this is that for these benchmarks (and the applications they represent), solving requires many related calls to the theory solver, each of which is relatively simple. The algorithms used in SMT solvers are optimized for this case and thus perform better, even though they use exact arithmetic which in general is much slower than floating point arithmetic. A solution to this problem advocated in [7] is to build a floating-point LP solver optimized for many, simple, related calls.

Here, we present an alternative approach. The idea is to take the two existing algorithms as they are and use each one only in cases when it is likely to do well. We thus use an exact solver optimized for fast incremental checks as the primary theory solver. However, we also instrument this solver so that it can detect when it is starting to have difficulty, and in these cases we have it call the LP solver.

The overall approach is given by the algorithm BALANCED-SOLVE shown in Figure 1. First, an efficient incremental exact solver EXACTSOLVE is called with a heuristic cap on the number of pivots it may perform, k_{EX}. We assume that EXACTSOLVE returns a status c (**Sat**, **Unsat**, or **Unknown**). If the exact solver returns **Sat** or **Unsat**, we are done and return the result. Otherwise, the heuristic cap was exceeded. In this case, the LP solver is called. We must convert the simplex problem described by T, l, and u to an analogous problem for the LP solver. We denote the LP analogs of the exact data by using the $\sim$ annotation. They are constructed (following [9]) as follows. For each auxiliary variable s, the equality $s = \text{orig}(s) \equiv \sum c_i x_i$, is added to $\widetilde{T}$ as $\widetilde{s} = \sum \text{float}(c_i) \cdot \widetilde{x}_i$, where the conversion function float maps a rational to the nearest float. For each variable $\widetilde{x}$, the bounds $\widetilde{l}(x)$ and $\widetilde{u}(x)$ are constructed from the δ-*rationals*, $l(x)$ and $u(x)$ by approximating δ as a small constant ϵ. For example, if $l(x) = \langle c, d \rangle$, then $\widetilde{l}(x)$ becomes $\text{float}(c + \epsilon \cdot d)$.

The LP solver is invoked with its own pivot limit k_{LP}. If the LP solver terminates with **Sat** or **Unsat**, we retrieve the assignment $\widetilde{a}$ as well as the final set of basic variables $\widetilde{\mathcal{B}}$ from the LP solver. The assignment $\widetilde{a}$ is converted into a rational assignment a' by the IMPORTASSIGNMENT routine (given below). The SEEDEXACT procedure takes $\widetilde{\mathcal{B}}$ and a' and tries to verify the result of the LP solver using the exact solver. If this fails (or if the LP solver reaches its heuristic limit), the exact precision solver is run with a final limit k_{FI}. For *final* calls to BALANCEDSOLVE (i.e. the DPLL(T) SAT engine has found a propositionally satisfying assignment), k_{FI} should be $+\infty$. It can be less for non-final calls.

An important contribution of this paper is the procedure shown in Figure 2. This procedure attempts to assign a rational value to each variable that is close to the one given by the LP solver, but biased towards values that are easy to represent, partly because that makes them easier to calculate

[5] Often, an additional requirement is that $\mathcal{H}$ is not satisfied by the current assignment a. We will not require this here.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

```
 1: procedure BalancedSolve
 2:    c ← ExactSolve(k_EX)
 3:    if c is Sat or Unsat then return c
 4:    Construct T̃, l̃, ũ from T, l, u
 5:    ⟨c̃, ã, B̃⟩ ← LPSolve(k_LP, T̃, l̃, ũ)
 6:    if c̃ is Sat or Unsat then
 7:       a′ ← ImportAssignment(ã)
 8:       c ← SeedExact(a′, B̃)
 9:       if c is Sat or Unsat then return c
10:    return ExactSolve(k_FI)
```

Fig. 1: The BalancedSolve procedure.

```
1: procedure ImportAssignment(ã)
2:    for all x ∈ 𝒱 do
3:       r ← DioApprox(ã(x), D)
4:       if |r − a(x)| ≤ ϵ then r ← a(x)
5:       if x ∈ 𝒱_ℤ and |r − ⌊r⌉| ≤ ϵ then r ← ⌊r⌉
6:       if r > u(x) or |r − u(x)| ≤ ϵ then r ← u(x)
7:       else if r < l(x) or |r − l(x)| ≤ ϵ then r ← l(x)
8:       a′(x) ← r
9:    return a′
```

Fig. 2: The ImportAssignment procedure.

with, but also partly because the discarded portion often corresponds exactly to a rounding error. For each variable x in the assignment, ImportAssignment first approximates $\widetilde{a}(x)$ as a rational using a technique based on continued fraction expansion called Diophantine approximation [5]. This technique finds the closest rational value with a denominator less than some fixed constant integer D. Next, we check to see if this value is within ϵ of the last known assignment for x in the exact solver. If so, the last known assignment is used. Next, if $x \in \mathcal{V}_{\mathbb{Z}}$ and the value is within ϵ of an integer z ($\lfloor r \rceil$ denotes the nearest integer to r), then z is used. Finally, ImportAssignment examines the value with respect to $l(x)$ and $u(x)$. If the value violates one of these bounds or is within ϵ of a bound, then the bound is used instead.

The SeedExact routine (Fig. 3) attempts to duplicate the results from the LP solver within the exact solver. First the procedure updates the exact solver assignment by calling Update on each non-basic variable. Next it computes the set, $\mathcal{B}'$, of variables that are non-basic in the exact solver but were marked as basic by the LP solver. We loop until as many variables in $\mathcal{B}'$ as possible have been pivoted to become basic. At the beginning of each iteration, we visit all the rows of T to check for conflicts. ([3] discusses doing this check efficiently.) While checking for conflicts, we can also detect whether any basic variable violates its upper or lower bound. If not, we have a satisfying assignment and stop early. If neither check applies, we search for a pair of variables x_i, x_j such that x_j is in $\mathcal{B}'$ meaning it is non-basic but should be basic, and $T_{i,j} \neq 0$ and $x_i \notin \widetilde{\mathcal{B}}$ meaning that x_i is basic but should be non-basic. If we can find such a pair, we pivot i and j and update the

```
 1: procedure SeedExact(a′, B̃)
 2:    for all x ∈ 𝒩 do
 3:       Update(x, a′(x) − a(x))
 4:    B′ ← 𝒩 ∩ B̃
 5:    while B′ ≠ ∅ do
 6:       if T has a row conflict then return Unsat
 7:       if all variables satisfy their bounds then return Sat
 8:       if ∃ i j. x_j ∈ B′ ∧ x_i ∉ B̃ ∧ T_{i,j} ≠ 0 then
 9:          Pivot(i, j)
10:          Update(i, a′(x_i) − a(x_i))
11:          B′ ← B′ \ {x_j}
12:       else return Unknown
13:    return Unknown
```

Fig. 3: The SeedExact procedure.

```
 1: procedure IntegerSolve
 2:    c ← BalancedSolve()
 3:    if c is Unsat then return c
 4:    Construct T̃, l̃, ũ from T, l, u
 5:    ⟨c̃, ã, B̃, t̃⟩ ← MIPSolve(k_MIP, T̃, l̃, ũ)
 6:    if c̃ is Unsat then c ← Replay(t̃)
 7:    else if c̃ is Sat then
 8:       a′ ← ImportAssignment(ã)
 9:       c ← SeedExact(a′, B̃)
10:       if c is Unknown then c ← ExactSolve(+∞)
11:    if (c is Sat and a is integer-compatible) or
          (c is Unsat) then return c
12:    Generate a branching theory lemma using (1)
13:    return Unknown
```

Fig. 4: The IntegerSolve procedure.

assignment of x_i to $a'(x_i)$. Approximations made by the LP solver or by ImportAssignment mean that SeedExact may fail to detect a satisfying assignment or a conflict in which case it returns **Unknown**. The SeedExact procedure can be seen as achieving a similar effect as ForcedPivot in [10] using rounds of the simplex algorithm in [1].

An alternative to verifying the LP solution would be to use an exact external LP solver (e.g. [12]–[14]). However, the use of an exact external solver (as well as an attempt to implement their rather sophisticated techniques) is beyond the scope of this work. Our goal, rather, is to make a first effort at an efficient integration of *inexact* floating-point solvers within SMT search. Integrating an exact external solver would be an interesting direction for future work.

IV. Using MIP Solvers to Improve Theory Solvers for Mixed Linear Integer and Real Arithmetic

We show how to extend the technique from the previous section to mixed linear integer and real arithmetic. The IntegerSolve algorithm (Fig. 4) illustrates our approach. First, the real relaxation of the problem is solved using the BalancedSolve algorithm described above. If the real

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

PROPAGATE

$$\frac{\tilde{E} \subseteq C_N \cup \tilde{P} \quad \tilde{h} \text{ is an inequality constraint} \quad \tilde{E} \cup I \models_T \tilde{h}}{N_1 := N \cdot \langle \tilde{h}, \tilde{E} \rangle}$$

BRANCH

$$\frac{\tilde{a} \text{ satisfies } \tilde{P} \wedge C_N \quad v \in \mathcal{V}_{\mathbb{Z}} \quad \tilde{a}(v) = \alpha \quad \alpha \notin \mathbb{Z}}{N_1 := N \cdot \langle v \leq \lfloor \alpha \rfloor, \emptyset \rangle \quad \| \quad N_2 := N \cdot \langle v \geq \lceil \alpha \rceil, \emptyset \rangle}$$

Fig. 5: Derivation rules. N is the parent node, N_1 and N_2 its child nodes. The symbol $\cdot$ denotes sequence concatenation.

relaxation is unsatisfiable, then we are done. Otherwise, we construct an MIP instance and call an MIP solver (with a pivot limit k_{MIP}) to search for an integer-compatible solution. When **Unsat** is returned, we also retrieve a *proof tree* $\tilde{t}$, which is a record of the steps taken by the MIP solver, and attempt to verify the tree by *replaying* its proof in the exact solver using the REPLAY procedure described below. Otherwise, if **Sat** is returned, we attempt to verify the assignment as before. If the verification fails, we again call EXACTSOLVE to ensure that we have a solution to the real relaxation before continuing. If we are unable to verify that the problem is **Unsat** or do not find an integer-compatible assignment, we force a branch by generating a theory lemma of the form (1) and return.

We now show how proof trees extracted from the MIP solver can be replayed within the exact solver. For the rest of the section, let M be an MIP instance consisting of an LP problem P of the form $T\mathcal{V} = 0 \wedge l \leq \mathcal{V} \leq u$ with the integer-tightening constraints $I \equiv \bigwedge_{z \in \mathcal{V}_{\mathbb{Z}}} \texttt{IsInt}(z)$. Let $\tilde{P}$ be the approximate version of P obtained by converting all rational constants in P to their corresponding floating point constants.

The process that an MIP solver goes through before concluding that $\tilde{P}$ is integer-infeasible can be described at an abstract level as a search tree. The root node represents the initial problem $\tilde{P}$ and each non-root node is derived from its parent by adding either a cut or a branch to the problem. The leaves of the tree represent real-infeasible problems.

Formally, we define a tree node N as a sequence of pairs $\langle \tilde{h}, \tilde{E} \rangle$, where $\tilde{h}$ is an inequality constraint and $\tilde{E}$ is an *explanation*, a (possibly empty) finite set, each element of which is either some $\tilde{h}'$ where $\langle \tilde{h}', \tilde{E}' \rangle$ appears earlier in N or is a constraint from the initial problem $\tilde{P}$. We denote by C_N the set $\{\tilde{h} \mid \langle \tilde{h}, \tilde{E} \rangle \in N\}$.

The root node of a proof tree is the empty sequence. Each non-root node is the result of applying to its parent node one of the derivation rules in Figure 5. The PROPAGATE rule is used to record when the MIP solver adds a cut. The cut must be entailed by some subset of constraints in the current MIP problem. The cut and its explanation are recorded in the child sequence. The BRANCH rule is used to record when the MIP solver does a case split on an integer variable. This can happen when the MIP solver has a solution $\tilde{a}$ to the real relaxation of the current problem that is not integer-compatible. The MIP solver chooses an integer variable v that has been assigned a real value α and enforces the constraint $v \leq \lfloor \alpha \rfloor \vee v \geq \lceil \alpha \rceil$. The rule has two children, each of which records in its sequence one of the two branch cases (with an empty explanation). A node N is a leaf when the MIP solver concludes that the problem $\tilde{P} \cup C_N$ is (real)-infeasible.

```
1:  procedure REPLAY(H, t)
2:     C_H ← {h | ⟨h, E⟩ ∈ H}
3:     if t is a is a leaf node N then
4:        Construct T, l, u from P ∪ C_H
5:        c ← BALANCEDSOLVE()
6:        if c is not Unsat then return Unknown
7:        Let ψ ⊆ P ∪ C_H be the conflict from BALANCEDSOLVE
8:        return REGRESS(ψ, H)
9:     if the root of t has only one child c then
10:       t' ← subtree of t rooted at c
11:       ⟨h, E⟩ ← IMPORTCONSTRAINT(last(c))
12:       if E ⊆ C_H ∪ P and E ∪ I ⊨_T h then
13:          return REPLAY(H · ⟨h, E⟩, t')
14:       else return REPLAY(H, t')
15:    if the root of t has two children c_1 and c_2 then
16:       for i = 1, 2 do
17:          t_i ← subtree of t rooted at c_i
18:          ⟨h_i, ∅⟩ ← last(c_i)
19:          K_i ← REPLAY(H · ⟨h_i, ∅⟩, t_i)
20:       K ← RESOLVEBRANCH(K_1, K_2)
21:       return REGRESS(K, H)
```

Fig. 6: The REPLAY procedure.

Ideally, a proof tree would allow us to prove that the original problem P is integer-infeasible. However, because of the approximate representation used by the MIP solver, this is not always the case. As a consequence, our theory solver uses the proof tree just as a guide for its own internal attempt to prove that P is integer-infeasible. This process is captured at a high level by the REPLAY function.

The REPLAY function is shown in Figure 6. It takes an initially empty sequence H and a proof tree t, and traverses the tree with the goal of computing a *conflict*, a subset of the constraints in the original LP problem P that are integer-infeasible. As REPLAY traverses the tree, it constructs a sequence H which is analogous to the sequences in the tree nodes, except that it contains only those constraints that the internal exact solver has successfully replayed and so may only be a subset of those in the tree node. (The REPLAY procedure returns **Unknown** if the replay has failed.)

If t is a leaf node, then $\tilde{P} \cup C_N$ should be integer-infeasible. We check the exact analog, $P \cup C_H$. If unsuccessful, we fail, returning **Unknown**; otherwise, we return a conflict. To compute the conflict, we make use of an auxiliary function, REGRESS, which is not shown. REGRESS takes a conflict K and a sequence H of constraint-explanation pairs and recursively replaces any constraint in K by explanation [assuming the explanation is non-empty]. The net effect is to ensure a conflict which does not contain derived cuts.

If the root of t has a single child, this child must have been derived using the PROPAGATE rule. The last element of

the sequence in the child node represents the new cut and its explanation. We convert the cut and its explanation to their exact analogs and then verify that we can derive the cut h from the exact constraints in E. These steps are explained in more detail below. If the cut can be verified, it and its explanation are included in the parameter H passed to the next recursive call to REPLAY. If not, the recursive call is made without h in the hopes that it is not needed to derive a conflict.

The final case is when the root of t has two children, indicating that the BRANCH rule was applied. Because branch constraints only use integers, importing them cannot fail. We are always able to represent them exactly. Thus, we simply call REPLAY recursively on each of the two sub-trees, passing one of the branch conditions to each sub-tree. The RESOLVE-BRANCH procedure constructs a conflict from the two returned conflicts K_1 and K_2. The procedure returns either: (i) the result of resolving K_1 and K_2 to remove the branch literals, (ii) K_i if it does not involve the branch, or (iii) **Unknown** otherwise. (The failure case requires at least one branch to be unknown.) As before, we use REGRESS to ultimately construct a conflict with constraints in P (we require REGRESS to return **Unknown** if K is **Unknown**).

Lines 12 and 13 of REPLAY require converting $\langle \widetilde{h}, \widetilde{E} \rangle$ to an exact analog, $\langle h, E \rangle$, and then verifying that h can be derived from E. We have implemented support for both Mixed-Gomory cuts and a variant of aggregated Mixed Integer Rounding cuts [15]. We will only explain here how reconstruction works for a special case of *Gomory cutting planes*.

The MIP solver can add a Gomory cutting plane $\widetilde{h}$ when the following conditions hold: (i) there is a row in $\widetilde{T}$, $b_i = \sum \widetilde{T_{i,j}} \cdot x_j$; (ii) all of the non-basic variables on the row are assigned to either their upper or lower bound; (iii) a subset of the variables on the row, that must include the basic variable b_i, are integer variables; and (iv) the assignment of b_i is non-integer. The premises (i)-(iv) make up the explanation $\widetilde{E}$.[6] For simplicity of presentation, we additionally assume all of the variables are integer and all the coefficients $\widetilde{T_{i,j}}$ are positive and assigned to their upper bounds. The assignment to b_i is then determined by the upper bounds of the non-basic variables, $\widetilde{a}(b_i) = \sum \widetilde{T_{i,j}} \cdot \widetilde{u}(x_j)$. The cut $\widetilde{h}$ for these constraints is then

$$\sum \frac{\widetilde{T_{i,j}}}{\widetilde{a}(b_i) - \lfloor \widetilde{a}(b_i) \rfloor} \left(\widetilde{u}(x_j) - x_j \right) \geq 1.$$

Given $\langle \widetilde{h}, \widetilde{E} \rangle$, we can attempt to derive a trusted cut and explanation $\langle h, E \rangle$ as follows. To reconstruct the cut, for every bound $x_j \leq \widetilde{u}(x_j) \in \widetilde{E}$, there must be a corresponding bound $x_j \leq u(x_j)$ in the exact system. (Note: $x_j \leq u(x_j)$ can be in either P or C_H.) Next we attempt to reconstruct the row $b_i = \sum \widetilde{T_{i,j}} x_j$ in exact precision as a row vector α. The coefficient for the basic variable in α is -1 ($\alpha_i = -1$). Nonbasic variables' coefficients are estimated from the approximate variables, α_j = DIOAPPROX$(\widetilde{T_{i,j}}, D)$. If after approximation, the sign of α_j does not match the sign of $\widetilde{T_{i,j}}$, this cut cannot be reproduced. The equalities $T\mathcal{V} = 0$ entail $\sum \alpha_k x_k = 0$ iff α is in the row span of T. This entailment can be checked by replacing auxiliary variables with their original definitions,

$$\alpha_i \cdot x_i + \sum_{x_j \text{is structural}} \alpha_j \cdot x_j + \sum_{x_k \text{is auxiliary}} \alpha_k \cdot \text{orig}(x_k),$$

and rejecting this cut if any of the coefficients do not cancel to 0.[7] The row α and the bounds $u(x_j)$ are used to generate $b = \sum \alpha_j \cdot u_j$, which can be thought of as a potential assignment to b_i. The cut cannot be reproduced if $b \in \mathbb{Z}$. If the value of b is non-integer, the Gomory cut h

$$h : \sum \frac{\alpha_j}{b - \lfloor b \rfloor} (u(x_j) - x_j) \geq 1$$

has been reproduced in exact precision. The explanation for h, E, includes the upper bounds $x_j \leq u(x_j)$, the integer constraints, and the equations $x_k = \text{orig}(x_k)$.

V. EXPERIMENTS AND DISCUSSION

All of the algorithms in this paper have been implemented in the CVC4 SMT solver [16].[8] In this section, we report the results of experiments using these implementations.

The implementation contains additional heuristics and several tunable parameters. While the authors have not done a formal tuning of any of these parameters, we include these values for completeness. There are two different simplex implementations in CVC4, one that follows the well-known simplex adapted for SMT described in [1], [4], and one based on sum-of-infeasibilities as described in [3]. The experiments were run using the latter method for the EXACTSOLVE procedure with a pivot cap of $k_{EX} = 200$ in Fig. 1 (with $k_{FI} = 200$ for non-final calls). Values of other parameters used in our experiments are $D = 2^{26}$; $\epsilon = 10^{-9}$; $k_{LP} = 10000$; and $k_{\text{MIP}} = 200000$. For both the LP and MIP solvers, we use the floating-point simplex solver in GLPK version 4.52 [17], instrumented to communicate the additional information needed by CVC4 in order to verify assignments, conflicts, and proof trees.[9] To avoid branching loops in GLPK, GLPK is halted if it branches 100 times on any one variable. To keep the size of the numeric constants manageable, we reject any cut containing a coefficient $\frac{n}{d}$ where $\log_2(|n|) + \log_2(|d|) > 512$. Further, we have a heuristic that dynamically disables the GLPK solver if it claims the problem is real-feasible and then integer-infeasible without generating any branches or cuts, a strange situation that happens with the `convert` benchmarks (see discussion below for details). GLPK is also dynamically disabled if CVC4's bignum package throws an exception while trying to import a floating point number. CVC4 has a heuristic that automatically detects and reencodes benchmarks in the `QF_LRA` family miplib (which are derived from benchmarks in [18]) in something closer to their original form.[10]

[6]See [4] for a Gomory cutting plane rule without additional assumptions.

[7]α can also be generated by Gaussian elimination from $\bigwedge x_k = \text{orig}(x_k)$.

[8]Experiments were run using a branch of CVC4 available at github.com/timothy-king/CVC4/CVC4 (commit 2550b6d).

[9]Source for this modified version of GLPK is available at github.com/timothy-king/glpk-cut-log (commit a35b8e).

[10]A comparison of other solvers on the miplib problems after this reencoding gave similar results to those reported in Table I.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

set	# inst.	# sel.	CVC4+MIP solved	CVC4+MIP time (s)	CVC4 solved	CVC4 time (s)	yices2 solved	yices2 time (s)	mathsat5 solved	mathsat5 time (s)	Z3 solved	Z3 time (s)	altergo solved	altergo time (s)	cutsat solved	cutsat time (s)	scip solved	scip time (s)	glpk solved	glpk time (s)
Selecting all benchmarks in the family																				
QF_LRA	652	652	645	6966	636	8557	632	5350	622	10913	615	5696	-	-	-	-	-	-	-	-
non-conj. QF_LIA	4579	4579	4489	86854	4472	86375	4375	30656	4543	55417	4474	75171	3956	262031	-	-	-	-	-	-
conj. QF_LIA	1303	1303	1249	11130	1068	31054	1111	55691	1154	33260	1039	19015	1232	2055	1018	35330	1255	7164	1173	8895
total	6534	6534	6383	104950	6176	125986	6118	91697	6319	99590	6128	99882	-	-	-	-	-	-	-	-
Selecting QF_LRA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
miplib	42	37	30	1530	21	3037	23	2730	17	5682	18	2435	-	-	-	-	-	-	-	-
DTP-Scheduling	91	4	4	4	4	4	4	0	4	2	4	1	-	-	-	-	-	-	-	-
latendresse	18	18	18	767	18	836	12	85	10	99	0	0	-	-	-	-	-	-	-	-
total	-	59	52	2301	43	3877	39	2815	31	5783	22	2436	-	-	-	-	-	-	-	-
Selecting non-conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
convert	319	282	208	9646	193	9343	188	4337	274	1876	282	118	166	272	-	-	-	-	-	-
bofill-scheduling	652	460	460	5401	458	4490	460	748	460	1519	460	2060	67	55	-	-	-	-	-	-
CIRC	51	11	11	0	11	0	11	0	11	0	11	0	11	0	-	-	-	-	-	-
calypto	37	37	37	3	37	3	37	0	37	6	36	5	35	24	-	-	-	-	-	-
nec-smt	2780	207	207	17276	207	18045	199	777	207	17925	201	7209	184	23724	-	-	-	-	-	-
wisa	5	1	1	0	1	0	1	0	1	1	1	0	1	0	-	-	-	-	-	-
total	-	998	924	32326	907	31881	896	5862	990	21327	991	9392	464	24075	-	-	-	-	-	-
Selecting conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
dillig	233	189	189	49	157	9823	175	8557	188	7185	166	1269	189	5	166	5840	189	42	189	3
miplib2003	16	8	4	307	4	1283	4	507	5	354	5	1089	0	0	6	146	7	17	6	295
prime-cone	37	37	37	2	37	2	37	2	37	1	37	2	37	1	37	4	37	1	37	0
slacks	233	188	166	61	93	2003	107	15672	119	4741	90	1994	188	84	96	6324	161	2361	101	11
CAV_2009	591	424	424	69	346	10035	376	26351	421	10236	354	2759	423	323	377	17015	424	105	424	6
cut_lemmas	93	74	62	9581	64	6865	72	1662	45	9472	38	5858	74	267	15	1887	72	1757	71	760
total	-	920	882	10069	701	30011	771	52751	815	31989	690	12971	911	680	697	31216	890	4283	828	1075

TABLE I: Experimental results on QF_LRA and QF_LIA benchmarks.

The experiments were conducted on the StarExec platform [19] with a CPU time limit of 1500 seconds and a memory limit of 8GB. The first segment of Table I compares our implementation with other SMT solvers over the full sets of QF_LRA and QF_LIA benchmarks from the SMT-LIB library (the "2013-03-07" version on StarExec), extended with the latendresse QF_LRA benchmarks from [3]. The QF_LIA benchmarks are divided into the *conjunctive* subset and the non-conjunctive subset. The conjunctive subset consists of all families, all of whose benchmarks are a simple conjunction of constraints.[11] The primary experimental comparison is between a configuration of CVC4 running just its internal solvers ("CVC") against a configuration with the techniques of this paper enabled ("CVC4+MIP"). We additionally compare with similar state-of-the-art SMT solvers: mathsat5 (smtcomp12 version) [20], z3 (v4.3.1) [21], and yices2 (v2.2.0) [22]. We include a comparison against the version of AltErgo [23] used in [24] on just the QF_LIA benchmarks. For the conjunctive subset, we also give results for several solvers that support only conjunctive benchmarks: cutsat (CADE11) [25], SCIP (scip-3.0.0-ex+spx) [13], [26], and glpk (4.52) [17]. This version of SCIP handles MIP problems in exact precision.

The remaining segments of Table I give more detailed results for QF_LRA benchmarks, non-conjunctive QF_LIA benchmarks, and conjunctive QF_LIA benchmarks respectively. In each segment, we report only the results on benchmarks for which CVC4+MIP invokes GLPK at least once. (For each family, the second column of numbers indicates how many benchmarks in the family are included in the results. See cs.nyu.edu/~taking/fmcad14_selections for a list of selected benchmarks.)

[11]The conjunctive families are dillig, miplib2003, prime-cone, slacks, CAV_2009, cut_lemmas, pidgeons, and pb2010. We translated these into the SMT-LIBv1.0 and MPS formats: cs.nyu.edu/~taking/conjunctive_integers.tbz.

set	# sel.	MIPSOLVE calls	Sat attempts	Sat successes	Unsat attempts	Unsat successes
QF_LIA	1393	3873	2559	1058	652	425
convert	208	2130	1356	1	178	3
bofill-scheduling	254	254	245	245	0	0
CIRC	11	85	6	5	79	77
calypto	37	375	77	23	293	278
wisa	1	1	1	1	0	0
dillig	189	228	225	185	3	2
miplib2003	4	10	3	3	5	4
prime-cone	37	37	19	19	18	18
slacks	166	195	168	162	3	3
CAV_2009	424	469	459	414	8	7
cut_lemmas	62	89	0	0	65	33

TABLE II: Success rate of reproducing results of MIPSOLVE

To better understand how successful the verification and

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

replaying algorithms for integers described in Section IV are, we analyzed all of the `QF_LIA` instances which were solved by CVC4+MIP and for which MIPSOLVE was invoked at least once, and collected the following statistics: the number of times MIPSOLVE was called, the number of attempts and successes at verifying **Sat** results from MIPSOLVE, and the number of attempts and successes at replaying **Unsat** results from MIPSOLVE. The results are shown in Table II.

On `QF_LRA` benchmarks, CVC4+MIP solves all of the problem instances that the already competitive CVC4 does plus 9 additional problems (solving more than any other solver), all from the challenging miplib family. After preprocessing, these benchmarks are represented internally as mixed linear real and integer problems, so the INTEGERSOLVE procedure is invoked. CVC4+MIP is the only solver to solve the `opt1217--{27,37,57}.smt2` benchmarks, and it does so in about 1s each. These and a handful of other miplib problems are real-infeasible and are solved very quickly by BALANCEDSOLVE. INTEGERSOLVE is able to verify that several other miplib benchmarks are **Sat**. It was not able to successfully solve the most difficult problems which are real-feasible but integer-infeasible.

CVC4+MIP is also quite competitive on the `QF_LIA` problem instances. Particularly dramatic is the improvement of CVC4+MIP over CVC4 on the (related) families dillig, slacks, and CAV_2009_benchmarks. These benchmarks are small, randomly generated, conjunctive problems that are mostly satisfiable [25], [27]. It appears from Table II that CVC4+MIP does well on these families due to a high proportion of successes when IMPORTASSIGNMENT and SEEDEXACT are used to verify **Sat** instances. Excluding the convert family, GLPK returned **Sat** 1203 times, and in 1057 cases, we were able to verify this with the exact solver. Given the challenges of implementing branching and cutting within SMT solvers, this suggests that the technique of soundly verifying results from an external solver offers a new powerful tool in designing `QF_LIA` solvers. The empirical results on the REPLAY procedure, while not as dramatic, are also promising. Excluding the convert benchmarks, REPLAY was successful on 425 out of 652 invocations and did particularly well on (relatively) easy benchmarks e.g. calypto and prime-cone.

CVC4+MIP is competitive with the dedicated conjunctive solvers we included. Of course, its performance is limited by that of GLPK (Interestingly, CVC4+MIP outperforms GLPK on these benchmarks.) Though most of the improvement of CVC4+MIP over CVC4 is on conjunctive benchmarks, the authors suspect this to be an artifact of the benchmarks.

The convert family is interesting in that almost every proof reported by GLPK on these benchmarks fails to replay. The benchmarks contain integer equalities between variables with coefficients of massively different scales. To ensure numerical stability, GLPK increases each bound by some amount ϵ, where ϵ is proportional to the size of the bound. Because of the dramatic differences of scale in the coefficients in the convert family, GLPK increases some bounds by a large amount and others by a small amount. As a result, GLPK frequently makes incorrect conclusions (both feasible and infeasible) about subproblems from this family. These benchmarks thus present a challenge for the techniques given in section IV and are a good subject for future research.

Acknowledgments: We would like to thank Morgan Deters for his help running experiments and Bruno Dutertre for providing us with a custom version of yices2.

REFERENCES

[1] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, 2006, pp. 81–94.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*. IOS, 2009, ch. 26.

[3] T. King, C. Barrett, and B. Dutertre, "Simplex with sum of infeasibilities for SMT," in *FMCAD*, 2013, pp. 189–196.

[4] B. Dutertre and L. de Moura, "Integrating Simplex with DPLL(T)," SRI International, Tech. Rep. SRI-CSL-06-01, 2006.

[5] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York, NY, USA: Wiley-Interscience, 1988.

[6] Y. Yu and S. Malik, "Lemma Learning in SMT on Linear Constraints," in *SAT*, 2006, pp. 142–155.

[7] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers," in *SAT*, 2008, pp. 77–90.

[8] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodrguez-Carbonell, and A. Rubio, "The Barcelogic SMT solver," in *CAV*, 2008, pp. 294–298.

[9] D.C.B. de Oliveira and D.Monniaux, "Experiments on the feasibility of using a floating-point simplex in an SMT solver," in *PAAR*.CEUR, 2012.

[10] D. Monniaux, "On using floating-point computations to help an exact linear arithmetic decision procedure," in *CAV*, 2009, pp. 570–583.

[11] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *TACAS*, 2011, pp. 150–153.

[12] D. L. Applegate, W. Cook, S. Dash, and D. G. Espinoza, "Exact solutions to linear programming problems," *Operations Research Letters*, vol. 35, no. 6, pp. 693 – 699, 2007.

[13] W. Cook, T. Koch, D. E. Steffy, and K. Wolter, "A hybrid branch-and-bound approach for exact rational mixed-integer programming," *Math. Program. Comput.*, vol. 5, no. 3, pp. 305–344, 2013.

[14] A. Neumaier and O. Shcherbina, "Safe bounds in linear and mixed-integer linear programming," *Mathematical Programming*, vol. 99, no. 2, pp. 283–296, 2004.

[15] H. Marchand and L. A. Wolsey, "Aggregation and mixed integer rounding to solve mips," *Operations Research*, vol. 49, no. 3, pp. 363–371, 2001.

[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011, pp. 171–177.

[17] A. Makhorin, "GNU Linear Programming Kit, Version 4.52," jun 2012. [Online]. Available: http://www.gnu.org/software/glpk/glpk.html

[18] T. Achterberg, T. Koch, and A. Martin, "Miplib 2003," *Operations Research Letters*, vol. 34, no. 4, pp. 361 – 372, 2006.

[19] A. Stump, G. Sutcliffe, and C. Tinelli, "StarExec: a Cross-Community Infrastructure for Logic Solving," in *IJCAR*, 2014, pp. 367–373.

[20] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS*, 2013.

[21] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[22] B. Dutertre, "Yices 2.2," in *CAV*, 2014, pp. 737–744.

[23] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout, "The Alt-Ergo Automated Theorem Prover." [Online]. Available: http://alt-ergo.lri.fr

[24] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, A. Mahboubi, A. Mebsout, and G. Melquiond, "A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic," in *IJCAR*, 2012, pp. 67–81.

[25] D. Jovanovic and L. M. de Moura, "Cutting to the Chase Solving Linear Integer Arithmetic," in *CADE*, 2011, pp. 338–353.

[26] T. Achterberg, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[27] I. Dillig, T. Dillig, and A. Aiken, "Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers," in *CAV*, 2009, pp. 233–247.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

A Program Transformation for Faster Goal-Directed Search

Akash Lal and Shaz Qadeer
Microsoft Research
Email: {akashl, qadeer}@microsoft.com

Abstract—
A goal-directed search attempts to reveal only relevant information needed to establish reachability (or unreachability) of the goal from the initial state of the program. The further apart the goal is from the initial state, the harder it can get to establish what is relevant. This paper addresses this concern in the context of programs with assertions that may be nested deeply inside its call graph—thus, far away interprocedurally from `main`. We present a source-to-source transformation on programs that lifts all assertions in the input program to the entry procedure of the output program, thus, revealing more information about the assertions close to the entry of the program. The transformation is easy to implement and applies to sequential as well as concurrent programs. We empirically validate using multiple goal-directed verifiers that applying this transformation before invoking the verifier results in significant speedups, sometimes up to an order of magnitude.

I. Introduction

Automated program verification attempts to establish reachability (or unreachability) of a goal from the initial state of the program. The goal is usually expressed as the violation of an assert statement in the program. Modern automated program verifiers are typically goal-directed, i.e., they attempt to use program information parsimoniously in order to establish (un)reachability of the goal as efficiently as possible. The challenge of distinguishing relevant from irrelevant and the difficulty of the verification problem increases as the distance of the goal from the initial state becomes larger. This paper addresses this challenge for programs with assertions that may be nested deeply inside its call graph—thus, far away interprocedurally from the program entry point.

Deep assertions are natural in large programs. For instance, in our benchmarks (Section VI), the static nesting depth of assertions (i.e., length of an acyclic path in the call-graph from `main` to a procedure containing an assertion) ranges from 4 to 38 (Fig. 6) and the depth observed on real error traces ranges from 5 to 15 (Fig. 7). At such depths, a naïve strategy of inlining procedures to expose control locations of the assertions is infeasible for analysis because of the exponential cost of inlining.

This paper presents an approach for lifting all assertions to the entry procedure of the program, thus revealing more information about the assertions close to the initial state of the program. Our method is a source-to-source transformation that produces output whose size is a small constant times the size of the input, and applies to both sequential and concurrent programs. We empirically validate using multiple verifiers that applying this transformation before feeding a program to a verifier results in upto order-of-magnitude speedups.

Our transformation is based on the observation that any execution that descends into a call to a procedure `P` either fails inside the call (and doesn't return) or returns from it without failing. We can convert assert statements inside `P` to assume statements if (1) we make a copy of the body of `P`, (2) instrument call sites of `P` to guess whether the call will fail, and (3) either make the call in the *success* case or jump to the copy in the *failure* case. This eliminates the need for making a call in order to reach the control location of the assertion. Further, we only need to make a single copy of the body of `P` regardless of the calling context, because in the *failure* case, control does not need to return to the caller. We also lift assertions outside loops based on the observation that in any execution only the last iteration of the loop (in that execution) can fail. In the presence of concurrency, we exploit the observation that at most one thread can fail.

Contributions. The contributions of this paper are: (1) a novel program transformation that optimizes running time of goal-directed verifiers for programs with deep assertions; and (2) an extensive evaluation over real software that totals over a month of verification time, and shows up to an order of magnitude speedup for two very different verifiers.

Organization. Section II covers background and related work on goal-directed verification techniques. Section III presents an overview of our transformation. Sections IV and V formally present the transformation for a simplified programming language. Section VI presents the evaluation.

II. Background

In order to describe the intuition behind our program transformation, we first discuss some goal-directed verifiers that are based on procedure inlining strategies. We choose these kinds of verifiers for two reason: first, they form a part of our evaluation (Section VI) and second, some inlining strategies have been proposed to specifically address deeply-nested assertions, thus, we compare the effect of our transformation against them.

Bounded model-checking tools (e.g., CBMC [6], [5]) are based on an eager inlining strategy that inlines all procedure calls up to a certain depth to produce a single procedure with all assertions inside it. Eager inlining fails for moderate to large programs because the inlining can result in an exponential explosion, even for small bounds. For instance, in many

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Algorithm 1 Forward and Alternating Inlining Strategies

```
1: procedure FWD(P)
2:   if P has an error trace then
3:     Return BUG
4:   end if
5:   P_over := Replace all calls c in P with summary(c)
6:   if P_over has an error trace then
7:     Let c be a call on that trace
8:     P := Inline c in P
9:     Return FWD(P)
10:  end if
11:  Return CORRECT
12: end procedure
```

```
1: procedure ALT(P)
2:   if FWD(P) = CORRECT then
3:     Return CORRECT
4:   end if
5:   if P.head = main then
6:     Return BUG
7:   end if
8:   for all callers c of P.head do
9:     P' := Inline P in c
10:    P'.head := c
11:    if ALT(P') = BUG then
12:      Return BUG
13:    end if
14:  end for
15:  Return CORRECT
16: end procedure
```

of the benchmarks used in this paper, eager inlining ran out of memory even before the analysis was started.

To avoid the cost of eager inlining, there are several proposed *lazy* inlining strategies that inline procedure on-demand and in a goal-directed manner. Techniques such as *structural abstraction* [2], *inertial refinement* [21], and *stratified inlining* [14] are all forward-inlining strategies, described abstractly by the method FWD of Alg. 1.

Forward Inlining. FWD takes a partially-inlined program P as input. (One can think of P as a single procedure containing some procedure calls.) Initially, P is just the body of `main`. FWD checks if P contains a bug without going through a procedure call (line 2). If not, then it picks a *relevant* procedure call made by P (line 7), inlines the body of the callee (line 8) and repeats. The choice of picking relevant calls is guided using procedure summaries (that are either pre-computed or inferred on the fly): if no error trace in P_{over} goes through a call c then this proves that no error trace of the original program goes though c. A *default* summary based on mod-set information, i.e., a procedure can arbitrarily modify variables that it can touch, can always be used. FWD, even with default summaries, has been shown to be much better than eager inlining in some contexts [2], [14]. Further, one can treat loops as tail-recursive procedures to extend FWD to perform loop unrolling as well.

FWD raises two technical concerns: first, what do procedure summaries mean in the presence of assertions, and second, what does it mean to query P_{over} for error when it may not even contain an assertion? Both these question are answered using an *error-bit instrumentation*. As pre-processing, we add a Boolean global variable `err` to the program; it is set to *true* if and only if an assertion fails; and all procedures immediately return when `err` is *true*. Then procedure summaries can use `err` to distinguish failing executions from non-failing ones. Moreover, we simply query P_{over} for a trace that ends with `err` set. We note that the error-bit instrumentation results in a program with the only assertion in `main`. However, it does not reveal any information about the original assertions themselves.

We illustrate FWD using the example in Fig. 1. This program has two global variables `s` and `g`. The entry procedure `main` initializes `s` and `g` and calls `P1`. The procedure `P1` is the first

```
var s, g: int;
procedure main()
{  s := 0; g := 1;
   P1();
}
procedure P1()
{ P2(); P2(); }
...
```

```
procedure Pn()
{ while (*) {
    if (g == 1)
      Open();
    Close();
  }
}
```

```
procedure Open()
{ s := 1; }

procedure Close()
{ assert s > 0;
  s := 0;
}
```

Fig. 1: An example program

in a chain of procedures `P1`,...,`Pn` each of which (except the last) calls its successor twice. `Pn` contains a nondeterministic loop that calls `Open` and `Close` in alternation. The assert statement inside `Close` cannot fail.

Suppose we wish to explore all behaviors of this program up to R loop iterations. In this case FWD will inline $O(2^n){*}O(R)$ procedures to conclude unreachability (under R) when using default summaries because no call will be deemed irrelevant. This number comes down to $O(1)$ when FWD has the following (inductive) procedure summaries available for each `Pi`: `(old(g) == 1 && old(s) == 0) ==> (s == 0 && !err)`, where `old(v)` refers to the value of `v` at the beginning of the procedure. This says that if `g` and `s` are `1` and `0`, respectively, at the beginning of `Pi` then when `Pi` returns, the value of `s` is still `0` and `err` has not been set. Clearly, given this summary for `P1`, FWD can conclude the absence of assertion failure just looking at `main`.

Alternating Inlining. Other inlining strategies include both backward and forward search [1, Section 4.2] [22], captured abstractly using ALT in Alg. 1. It starts with P as a procedure with an assertion. It conducts a forward search (line 2) to find an error trace from the initial state of P. If such a trace is found, it picks a caller of P, inlines P inside it and repeats until the search reaches `main`. An interesting remark is that ALT does not require the error bit instrumentation. This is because it starts with the assertion that it wishes to violate, and all procedures inlined during the call to FWD are constrained to not fail. Thus, all summary computation can be done assuming fail-free executions.

On Fig. 1, ALT will inline $O(2^n) * O(R)$ procedures when using default summaries. However, using just the (inductive) fact that `g == 1` is a valid precondition of each `Pi`, this number comes down to $O(1)$. This is because when the search is at procedure `Pn`, then under this precondition, ALT can already prove the absence of assertion violations (line 3) without enumerating the calling contexts of `Pn`.

Thus, different inlining strategies can involve different amount of inlining, and put different amount of stress on invariant and summary generation.

III. Overview of our program transformation

In this section, we informally describe our novel contribution, a semantics-preserving source-to-source transformation that lifts all assert statements in a program into its entry procedure. As explained in Section I, our transformation is based on the simple observation that any execution that descends into a call to a procedure `P` either fails inside the call or returns from it. We will convert all assert statements

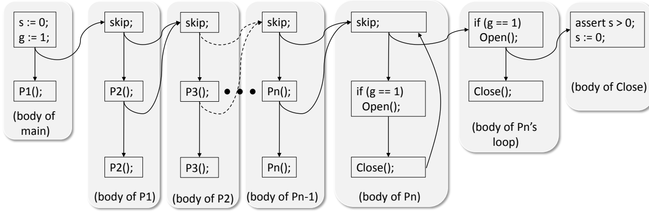


Fig. 2: Control-flow graph of transformed procedure `main`

$$
\begin{array}{rcllll}
P & \in & \mathit{Prog} & ::= & (gs, ps) & \\
p & \in & \mathit{Proc} & ::= & (x, is, os, vs, st) & \\
vs & \in & \mathit{Vars} & ::= & \cdot \mid x : t, vs & gs, is, os \in \mathit{Vars} \\
ps & \in & \mathit{Procs} & ::= & \cdot \mid p, ps & \\
st & \in & \mathit{Stmt} & ::= & l\!: \mathit{assume}\ e \mid l\!: \mathit{assert}\ e \mid l\!: xs\ :=\ es \mid & \\
 & & & & l\!: \mathit{havoc}\ xs \mid l\!: \mathit{goto}\ ls \mid l\!: \mathit{loop}\ st \mid & \\
 & & & & l\!: \mathit{call}\ xs\ :=\ x(es) \mid l\!: \mathit{async}\ x(es) \mid l\!: \mathit{yield} \mid & \\
 & & & & st; st & \\
xs & \in & \mathit{Names} & ::= & \cdot \mid x, xs & x \in \mathit{Name} \\
es & \in & \mathit{Exprs} & ::= & \cdot \mid e, es & e \in \mathit{Expr} \\
ls & \in & \mathit{Labels} & ::= & \cdot \mid l, ls & l \in \mathit{Label} \\
t & \in & \mathit{Type} & & &
\end{array}
$$

Fig. 3: Program syntax

inside `P` to assume statements and simulate failures in the body of `P` by nondeterministically jumping to a copy of the body of `P` at a call site.

Fig. 2 shows, as a control-flow graph, the result of our transformation on the `main` procedure of our running example from Fig. 1. The bodies of all other procedures remain the same except for `Close` in which `assert s > 0` is converted to `assume s > 0`. The execution of transformed `main` begins in the top-left block with the initialization of the global variables. Next, it can non-deterministically choose to call `P1` or jump to a copy of the body of `P1`. The two calls to `P2` in the body of `P1` are similarly instrumented, and so on.

The instrumentation of the body of `Pn` is interesting because it contains a loop. In addition to lifting assertions out of procedure calls, we would also like to lift them out of loops. Our insight is that it suffices to allow only the final iteration of the loop to fail. Therefore, we can make a copy of the loop body, convert assert statements inside the loop to assume statements, and then nondeterministically execute the copy of the body after the loop at most once.

It is worth noting that in Fig. 2, we did not make a copy of the body of procedure `Open`. We could do this optimization because it was possible to statically determine that a call to `Open` cannot fail.

When FWD is applied to the transformed program, it only inlines $O(1)$ number of procedures to conclude CORRECT. (In particular, it only needs to inline the call to `Open` from the new `main`.) The reason is that the value flow between the initialization of `g` and the conditional expression guarding the call to `Open` is apparent at the top-level without any intervening loops and calls, even under default summaries. In this case, inlining the call to `Open` is sufficient to discharge the assertion. Thus, no summary or invariant generation was required for this example after our transformation. This example provides intuition for the speedup on programs with an unreachable goal, however, pruning infeasible paths also translates to finding the goal faster when reachable. This is confirmed by our experiments.

While we have chosen to evaluate our program transformation against lazy inlining strategies (as each address the issue of deep assertions), our approach is more general. It is not tied to a particular analysis. It simply produces a new program that can be fed to any verifier, with the hope of speeding up the verifier. For instance, our evaluation uses the YOGI verifier for C programs that is based on predicate-abstraction and doesn't directly implement an inlining strategy. This point is further emphasized when dealing with concurrent programs, as we are not aware of inlining strategies that directly apply to concurrent programs.

Remark: Here we note that our approach is inspired by "Phase 2" of the RHS algorithm [19], [20]. RHS is the standard tabulation-based algorithm for interprocedural dataflow analysis. It works in two phases: the first phase computes procedure summaries bottom-up in the call graph. The second phase replaces procedure calls with the summaries and deletes return edges. This transformation is similar to ours, however, we do not use summaries and our target is goal-directed program verification, not dataflow analysis. Moreover, our transformation has special handling for loops and concurrency.

IV. A SIMPLE PROGRAMMING LANGUAGE

We present a core programming language, similar to Boogie [3], for formalizing our program transformation. The syntax of the language is presented in Fig. 3. A program P is a tuple comprising a set of global variable declarations gs and a set of procedure declarations ps that is assumed to contained a distinguished procedure called `main`. Each procedure is a tuple comprising its name x, input parameters is, output parameters os, local variables vs, and a statement st. As notation, for a procedure $f = (x, is, os, vs, st)$, let $\mathit{name}(f) = x$, $\mathit{input}(f) = is$, $\mathit{output}(f) = os$, $\mathit{locals}(f) = vs$, and $\mathit{code}(f) = st$. We assume, without loss of generality, that `main` is never called and it does not have output variables.

A statement st is a ";"-separated list of a label l and one of the following—*assert*, *assume*, *assignment*, *havoc*, *goto*, *loop*, *call*, *async*, or *yield*. In our presentation, we ignore the syntax of expressions and types and assume the existence of a type checker for validating that the program is well-formed. Further, we may sometimes omit writing the label of a statement, in which case it is assumed to have a fresh label that is not used elsewhere in the program. Statement labels must be unique and cannot be re-used.

The control flow in our language is straightforward. The statement $\mathit{goto}\ ls$ causes control to non-deterministically jump to some label in ls; the type checker ensures that the labels exist in the same procedure or enclosing loop. For all other statements, control implicitly moves to the next statement by following the sequential composition (";") operator. If there is no next statement, then execution of the statement terminates.

$\mathit{assert}\ e$ fails if e evaluates to false in the current state and otherwise leaves state unchanged. $\mathit{assume}\ e$ blocks if e

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

$$
\begin{array}{lcl}
[\![l\!:\!assume\ e]\!]_{\text{stmt}} & = & l\!:\!assume\ e \\
[\![l\!:\!assert\ e]\!]_{\text{stmt}} & = & l\!:\!assert\ e \\
[\![l\!:\!xs\ :=\ es]\!]_{\text{stmt}} & = & l\!:\!xs\ :=\ es \\
[\![l\!:\!havoc\ x]\!]_{\text{stmt}} & = & l\!:\!havoc\ x \\
[\![l\!:\!goto\ ls]\!]_{\text{stmt}} & = & l\!:\!goto\ ls \\
[\![st_1; st_2]\!]_{\text{stmt}} & = & [\![st_1]\!]_{\text{stmt}}; [\![st_2]\!]_{\text{stmt}} \\
[\![l\!:\!call\ xs\ :=\ x(es)]\!]_{\text{stmt}} & = &
\end{array}
$$

$$
\begin{array}{l}
\quad l\!:\!if(\star)then\{call\ xs\ :=\ x(es)\} \\
\qquad\qquad else\{input(x)\ :=\ es; havoc\ locals(x); goto\ x^{entry}\} \\
[\![l\!:\!loop\ st]\!]_{\text{stmt}} \ = \ l\!:\!loop\ \overline{st}; if(\star)then\{[\![st]\!]_{\text{stmt}}\}else\{skip\}
\end{array}
$$

$$
\begin{array}{lcl}
[\![(x, is, os, vs, st)]\!]_{\text{proc}} & = & (x, is, os, vs, \overline{st}) \\
[\![(p, ps)]\!]_{\text{proc}} & = & ([\![p]\!]_{\text{proc}}, [\![ps]\!]_{\text{proc}}) \\
[\![(gs, (\texttt{main}, is, \cdot, vs, st), ps)]\!]_{\text{prog}} & = & (gs,
\end{array}
$$

$$
\begin{array}{l}
\quad (\texttt{main}, is, \cdot, vs \cup \bigcup_{p \in ps} input(p) \cup output(p) \cup locals(p), \\
\qquad [\![st;\ die; x_1^{entry}: skip; code(x_1); die; \cdots ; x_n^{entry}: skip; code(x_n); die]\!]_{\text{stmt}}), \\
\quad [\![ps]\!]_{\text{proc}})\ \text{where}\ ps = (p_1, \cdots, p_n)\ \text{and}\ name(p_i) = x_i
\end{array}
$$

Fig. 4: Transforming sequential programs

$$
\begin{array}{lcl}
[\![l\!:\!yield]\!]_{\text{stmt}} & = & l\!:\!yield \\
[\![l\!:\!async\ x(es)]\!]_{\text{stmt}} & = & l\!:\!if(\star)then\{async\ x(es)\}
\end{array}
$$

$$
\begin{array}{l}
\qquad else\{assume\ flag = nil; a_{input(x)}\ :=\ es; flag\ :=\ c_x\}
\end{array}
$$

$$
\begin{array}{lcl}
[\![(gs, ps)]\!]_{\text{prog}} & = & (gs \cup \{flag\} \bigcup_{p \in ps} a_{input(p)},
\end{array}
$$

$$
\begin{array}{l}
\quad (\texttt{newmain}, is, \cdot, vs \bigcup_{p \in ps} input(p) \cup output(p) \cup locals(p), [\![st]\!]_{\text{stmt}}), [\![ps]\!]_{\text{proc}}) \\
\quad \text{where}\ ps = (\texttt{main}, p_1, \cdots, p_n), name(p_i) = x_i\ \text{and} \\
\quad st \stackrel{\text{def}}{=} flag\ :=\ nil; \\
\qquad if(\star)then\{flag\ :=\ c_{\texttt{main}}; goto\ \texttt{main}^{entry}; die\}else\{skip\}; \\
\qquad async\ main(is); yield; goto\ l_{x_1}, \cdots, l_{x_n}; die; \\
\qquad l_{x_1}: assume\ flag = c_{x_1}; input(x_1)\ :=\ a_{input(x_1)}; goto\ x_1^{entry}; die; \\
\qquad \ldots \\
\qquad l_{x_n}: assume\ flag = c_{x_n}; input(x_n)\ :=\ a_{input(x_n)}; goto\ x_n^{entry}; die; \\
\qquad [\![\texttt{main}^{entry}: skip; code(\texttt{main}); die]\!]_{\text{stmt}}; \\
\qquad [\![x_1^{entry}: skip; code(x_1); die; \cdots ; x_n^{entry}: skip; code(x_n); die]\!]_{\text{stmt}}
\end{array}
$$

Fig. 5: Transforming concurrent programs

evaluates to false in the current state and otherwise leaves state unchanged. $xs\ :=\ es$ is a parallel assignment that evaluates es in the current state and updates variables xs to the result. $havoc\ xs$ puts nondeterministically chosen values into each variable in xs. $loop\ st$ is a nondeterministic structured loop and executes st zero or more times. $call\ xs\ :=\ x(es)$ is call to procedure x with inputs es; the output of the procedure call is received in variables xs. $async\ x(es)$ is an asynchronous call to procedure x with inputs es; the call is executed in a new thread that executes concurrently with all existing threads. The multithreading model in our language is cooperative and nondeterministic; $yield$ yields control to a nondeterministically chosen thread.

For convenience, we also use a statement $if(\star)then\{st_1\}else\{st_2\}$ that denotes non-deterministic branching between two statements. We use it as syntactic sugar over using $goto$ statements.

V. Program transformation

We begin by presenting our transformation for sequential programs in Fig. 4 and generalize it to concurrent programs in Fig. 5. We use *skip* and *die* to compactly denote $assume\ true$ and $assume\ false$, respectively. Our transformation depends on an initial renaming of variables and labels in the program to make them globally distinct. This initial renaming is standard and we do not present it here. Further, for a statement st, let $\overline{st}$ be the same statement where all occurrences of $assert\ e$ in st are converted to $assume\ e$.

A. Transforming sequential programs

Fig. 4 describes three transformations: $[\![.]\!]_{\text{stmt}}$ for statements, $[\![.]\!]_{\text{proc}}$ for procedures and $[\![.]\!]_{\text{prog}}$ for programs. First, note that the transformation of a procedure simply disables all assertions in the procedure. The transformation on a program leaves the set of global variables unchanged and disables assertions in all procedures except `main`. The `main` procedure is transformed by absorbing the bodies of all other procedures (along with their input, output and local variables) and applying the statement transformer on them. It is easy to show that $[\![P]\!]_{\text{prog}}$ can only have assertions in `main`.

Let us now look at the statement transformer $[\![.]\!]_{\text{stmt}}$. It is non-trivial only for procedure calls and loops. It transforms a procedure call of x to a non-deterministic branch. The *then* branch simulates an execution where the procedure call succeeds. In this case, the call is left untouched. However, note that x does not have assertions in the transformed program, thus a call to it cannot fail. The *else* branch simulates an execution where the procedure call fails. In this case, we simply jump to x^{entry} where a copy of the body of x resides. Note the use of die in the $[\![.]\!]_{\text{prog}}$ transformation. This prevents the execution of, say, x_2's body to fall through onto the body of x_3. Thus, a jump to the body of a procedure cannot ever return (but it may fail).

The statement transformation for loops works by first peeling off the last iteration of the loop. ($loop\ st$ is equivalent to $loop\ st; if(\star)then\{st\}else\{skip\}$.) Next, the new loop's body is not allowed to fail ($\overline{st}$), because only the last iteration of a loop can fail. The statement transformer is applied recursively to the last iteration.

B. Transforming concurrent programs

The transformation described in the previous section, although adequate for lifting all assertions to the entry procedures of all threads, is inadequate for lifting all assertions to just the `main` block of the initial thread. This section extends the transformation described earlier to achieve this goal.

Fig. 5 defines the statement transformer for *yield* and *async* procedure calls. It also redefines the program transformation. The rest is borrowed over from Fig. 4. The main insight behind these transformations is that any erroneous execution has exactly one assertion failure which stops the execution. Therefore, it suffices to allow at most one thread, either initial or dynamically-created, to fail. The start procedure of a dynamically-created thread is one of a finite number of procedures that are targets of asynchronous procedure calls. We introduce fresh constants including the special constant nil and a constant c_x for each procedure in the input program with name x; these constants are assumed to be distinct from each other. We also introduce a fresh global variable *flag* whose value is one of these freshly introduced constants; this variable is initialized to nil. During the execution of the transformed

program its value changes at most once from nil to some constant c_x. The final value of *flag*, if different from nil, represents the entry procedure of the potentially failing thread.

The transformation of an asynchronous call $async\ x(es)$ is a non-deterministic choice. One choice is to keep the asynchronous call, but to a procedure that cannot fail (recall the procedure transformation from Fig. 4). The other choice is to atomically update *flag* from nil to c_x, which simulates the creation of a failing instance of x. (The failing instance executes in the entry procedure of the transformed program, discussed later in program transformation rule.) Blocking on the condition $flag = nil$ ensures that at most one failing instance is created. We use additional global variables a_v (where v is an input argument to some procedure) for storing the arguments of the failing thread instance.

The $[\![.]\!]_{\text{prog}}$ transformation is more sophisticated. It works by creating a new procedure, called `newmain` that is understood to be the entry procedure of transformed program. It consists of the bodies of all other procedures, including `main`. It starts by initializing *flag* to nil. Next, it decides if the `main` thread is the one that fails; if so, it jumps to `main`. Otherwise, it spawns `main` as a separate thread (which cannot fail) and non-deterministically jumps to a location l_x for some procedure x. The location l_x waits for *flag* to be set to c_x, grabs input arguments from a_v variables, and jumps to the body of x.

C. Correctness

Let P be a program where all variables and labels are globally distinct. The most important property of our transformation is that it is failure-preserving. Therefore, verifying the original program is equivalent to verifying the transformed program.

Theorem 5.1: P fails an assertion if and only if $[\![P]\!]_{\text{prog}}$ fails an assertion.

The following theorem states that we succeeded in our objective of lifting all assert statements out of loops and procedures.

Theorem 5.2: In $[\![P]\!]_{\text{prog}}$, no procedure other than the entry procedure can have assertions. Further, even loop statements in the entry procedure cannot have assertions.

Next, we state a property about the compactness of our transformation. Let $|P|$ denote the size of the program P. The *loop nesting depth* of a program is defined recursively as follows.

$$\begin{aligned} LND(ps) &= max(\{LND(p) \mid p \in ps\}) \\ LND((x, is, os, vs, st)) &= LND(st) \\ LND(st_1; st_2) &= max(LND(st_1), LND(st_2)) \\ LND(l\!:\!loop\ st) &= LND(st) + 1 \\ LND(_) &= 1 \end{aligned}$$

Theorem 5.3: $|[\![P]\!]_{\text{prog}}| = |P| \times (LND(P) + c)$ for a small constant c.

Finally, our transformation enjoys the desirable property that if the input program is recursion-free and has only structured loops, then so is the output program.

Theorem 5.4: If P is recursion-free and each procedure has an acyclic control-flow graph then $[\![P]\!]_{\text{prog}}$ is recursion-free and each procedure has an acyclic control-flow graph.

VI. Evaluation

We refer to the transformation of Section V as the *deep-assert* (DA) instrumentation. We conducted extensive experiments to evaluate its effect on the running time of two different verifiers:

1) Corral [14] is an SMT-based verifier that accepts Boogie programs [16] as input. It consists of an outer loop of abstraction refinement. Inside the loop, it verifies a program using either Fwd or Alt of Alg. 1, based on stratified inlining [14] and alternating inlining [22], respectively.
2) Yogi [4], [10] is a verifier for C programs. It alternates between test generation (for proving "reachability" information) and automated predicate abstraction (for proving "unreachability" information).

We chose Yogi because: first, it currently uses the error-bit instrumentation (Section II). Second, Yogi has been highly optimized over several years of research and development [18], [11], [4], [10], thus, any performance improvement is considered significant. Third, it is a "third-party" tool; we were never a part of the design or implementation of Yogi.

Let SI and AT refer to Corral with stratified inlining and alternating inlining, respectively, and let SI+DA refer to applying our deep-assert transformation followed by running Corral with stratified inlining. Note that once the deep-assert transformation is executed then using SI or AT is identical as all assertions would be in `main`.

Corral uses Houdini [9] for generating program invariants and procedure (and loop) summaries. Let SI+H, AT+H and SI+DA+H refer to configurations when Houdini is enabled. Houdini requires invariant templates to be supplied by the user. Invariant generation in Yogi is fully automated.

Corral and Yogi use different IR representation for programs. The implementation of the deep-assert instrumentation for Corral was 969 lines of C# code[1] and for Yogi was 166 lines of OCaml code.

All experiments were conducted on a server class machine with two Intel(R) Xeon(R) processors (16 logical cores) executing at 2.4 GHz with 32 GB RAM. Different verification instances were executed in parallel, with at most 16 instances (one per core) executing in parallel at any given time.

Static Driver Verifier. Our first set of experiments is using the Static Driver Verifier (SDV) [17]. SDV is a commercial-grade tool offered by Microsoft to third-party driver developers. We collected a set of real device drivers that have been historically challenging for SDV, shown in Fig. 6. The drivers total 115KLOC, and additionally link against libraries of size 75KLOC. Fig. 6 also gives the number of procedures (#Procs) and "Assert Depth", which is a pair consisting of the smallest and largest acyclic path in the call graph from the entry point to a procedure containing an assert. This is a static measure for how deep the assertions were in the program. The last column lists the number of verification instances for

[1] available open source at corral.codeplex.com.

Name	KLOC	#Procs	Assert Depth (min,max)	#Verif. Instances
fdc_fail	9.2	216	4-18	226
kbdclass	7.1	230	4-31	252
daytona	21.5	345	4-27	316
parport	33.9	531	4-21	169
sys	2.2	108	4-27	596
isapnp	14.1	286	4-18	94
mouser	7.4	190	4-38	600
modem	14.4	289	4-26	157
kerneldriver	5.0	183	4-34	106
Total	115+75	2378	4-38	2516

Fig. 6: Details of SDV benchmarks

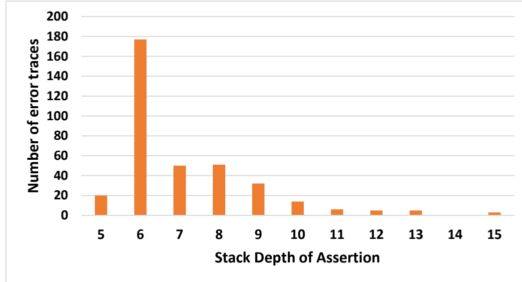


Fig. 7: Stack depth of SDV error traces

a driver: SDV verifies multiple properties of a driver and in doing so, generates multiple different verification instances. For our purpose, a verification instance is simply a program with assertions. SDV generated verification instances have no recursion (usually drivers don't have recursion, and even when they do, SDV statically unrolls the recursion up to a small bound). Moreover, all loops are structured (i.e., the control-flow graph of procedures are *reducible*), in which case we can compile loops to use our *loop* statement. Fig. 7 shows the stack depth at which the failing assert was reached among all the error traces found in the benchmark suite. It shows a reasonable range and variation.

SDV generates a set of predicates $R_1, \cdots, R_n, S_1, \cdots, S_m$ for each verification instance, based on the property that it is checking [14]. The R_is are predicates over the input state of a procedure; they serve as templates for preconditions. The S_is are templates for postconditions (summaries). SI uses the error-bit instrumentation; let `err` be the error bit summarizing if an assertion has failed or not (see Section II). SI+H uses HOUDINI to look for procedure summaries of the following form: `!err ==>` S_i (i.e., S_i is a summary when the procedure doesn't fail) and R_j `==> !err` (i.e., under R_i, the procedure doesn't fail). AT+H doesn't use the error-bit instrumentation; it looks for summaries of the form S_i and preconditions of the form R_j. While summaries can be inferred bottom-up in the call graph, inferring preconditions requires a top-down pass as well. SI+DA+H also doesn't use the error-bit instrumentation (there is no need because all assertions are lifted to `main` by DA). Further, it only looks for summaries of the form S_i; the templates R_j are dropped as our deep-assert transformation reduces the need for preconditions.

Aggregate results across all verification instances are shown in Fig. 11. The table lists the total number of instances that

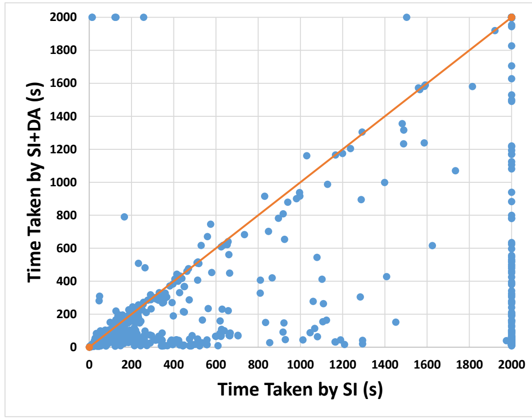
Algorithm	#TO	#Bnd	#Bugs	#Proof	Houd. (1000 s)	Time (1000 s)	
						Bug	No-bug
SI	510	477	348	1181	0	23	154
AT	314	638	345	1219	0	31	126
SI+DA	**213**	**383**	**363**	**1557**	0	**21**	**93**
SI+H	73	129	360	1954	76	35	156
AT+H	126	226	350	1814	115	47	205
SI+DA+H	**43**	**127**	**363**	**1983**	**53**	**32**	**123**

Fig. 11: Results, in aggregate, for the SDV benchmarks

timed out after 2000 seconds (#TO), hit the search bound (i.e., inconclusive) (#Bnd), produced an error trace (#Bugs), or proved the instance correct (#Proof). The other columns list the total time taken by HOUDINI (Houd), and the time spent by CORRAL (inclusive of time spent by HOUDINI) on buggy and non-buggy instances. Non-buggy instances include both bound-hit and proofs, but not timeouts. Times are reported in units of 1000 seconds. The entire table took 41 days of verification time.

The table shows advantages of the deep-assert instrumentation along several dimensions. SI+DA and SI+DA+H have much fewer timeouts, find more bugs, prove more instances correct, and take the least amount of time. Using HOUDINI significantly reduces the number of timeouts and increases the number of instances proved correct (for each of SI, AT, and SI+DA). These numbers suggest that the templates used by HOUDINI were complete to a good extent. However, the time taken by HOUDINI is a significant fraction of the total running time. Thus, optimizing HOUDINI usage is important. The table shows that the simplification of templates provided by DA improves the running time of HOUDINI. Because ALT requires preconditions for pruning, AT+H spends the maximum amount of time in HOUDINI—more than twice as much as SI+DA+H. Consequently, AT+H is the slowest among other configurations with HOUDINI. This indicates that ALT imposes a stricter demand for invariants for pruning search. SI+DA, on the other hand, does well even without invariant generation; in fact, it finds all the 363 bugs without the help of HOUDINI.

Fig. 8 presents a more detailed comparison of the running times of SI and SI+DA. The scatter plot (on the left) is the distribution of running times: each dot is a single verification instance. The chart on the right summarizes the number of instances in which DA resulted in a particular speedup (computed as a fraction of the running time). "Infinity" means that a timeout was eliminated, and "-Infinity" means that a timeout was introduced. For example, there are 54 instances in which SI+DA is at least 10 times faster than SI. The numbers on top of the bars indicate the average running time of SI (in seconds) on an instance that falls in that bar. For example, whenever SI+DA was 5 to 10 times faster than SI, the average time taken by SI was 434.2 seconds. These numbers show that the speedup was obtained on non-trivial instances. Further, only 6 timeouts were introduced, and 303 were eliminated by DA. Only 5 instances experienced a slowdown worse than a factor of 2 (see the bar "<0.5"). There are 1726 instances with speedup in the range 0.5 to 1.75. These are not shown in the figure, moreover, their average running time was just 69



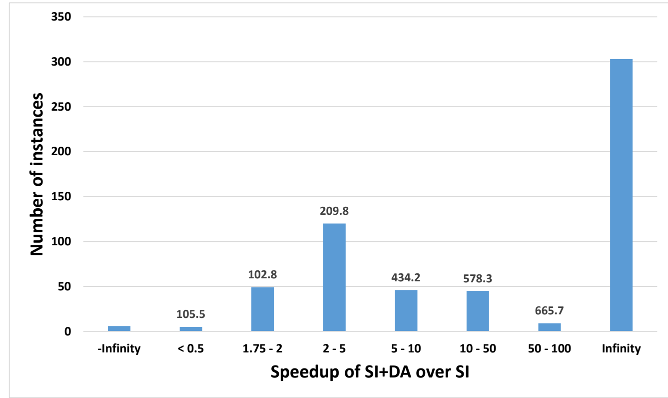
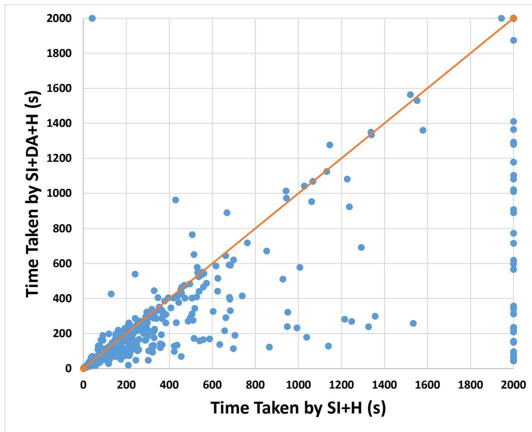


Fig. 8: Comparisons of running time between SI and SI+DA



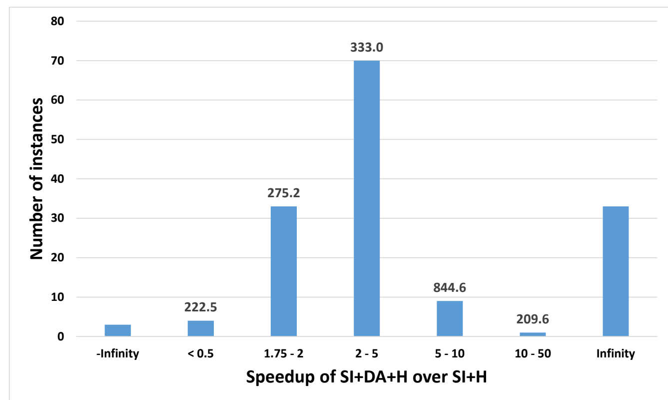


Fig. 9: Comparisons of running time between SI+H and SI+DA+H

seconds. One can also visually observe high density of dots near the origin of the scatter plot.

Fig. 9 shows similar graphs for SI+DA+H against SI+H. In this case, 33 timeouts were eliminated and only 3 introduced by DA. Only 4 instances observed a slowdown worse than a factor of 2. There are 2323 instances with speedup in the range 0.5 to 1.75 with an average running time of 76 seconds.

Fig. 10 shows the effect of DA on the running time of YOGI. The overall speedup is a modest 9% but this increases to as much as 50% (i.e, a factor of 2 faster) on harder instances that take at least 600 seconds. The benchmarks used for YOGI were the same set of drivers as mentioned in Fig. 6, but for a subset of the verification instances (total 802). Because YOGI does not support features like bitvector reasoning and arrays, we disabled some of the SDV properties when using YOGI. DA eliminated 8 timeouts and only 1 was introduced. As before, the slowdowns are mostly on trivial instances. The average running time on such instances was less than 2 minutes. The harder instances, with longer running time, usually show a speedup.

The scatter plot of Fig. 10 shows a greater spread than for CORRAL (Figs. 8 and 9). We believe this is because CORRAL uses a more powerful (SMT-based) intraprocedural analysis and this matches well with the programs produced by DA as they have a large `main` procedure.

Program	LOC	Assert Depth	#Inst-ances	CORRAL (sec)	CORRAL+DA (sec)
daytona	488	3-5	5	460.6	**407.4**
kbdclass	694	3-4	2	713.9	**641.7**
mouclass	581	3-4	7	3877.8	**2964.0**
ndisprot	592	3-5	3	**314.9**	345.9
pcidrv	449	3-5	6	**796.4**	988.5
total	2804	3-5	23	6163.9	**5347.7**

Fig. 12: Results on concurrency benchmarks

Memory Consumption: For SDV benchmarks, we observed that the ratio of $|[\![P]\!]_{\text{prog}}|$ to $|P|$ ranged from 1.1 to 1.6, which is much smaller than the worst-case mentioned in Thm. 5.3. This is because bodies of nested loops tend to be very small compared to the rest of the program. (DA copies the body of a loop as many times as its nesting depth.) Moreover, many procedures cannot statically reach an assert, thus they need not be copied into `main` by the instrumentation. Despite the increase in program size, DA still reduces memory consumption because of the decreased analysis complexity. On average, the peak memory usage of SI+H was 461MB, for AT+H it was 663MB, and for SI+DA+H it was 443MB.

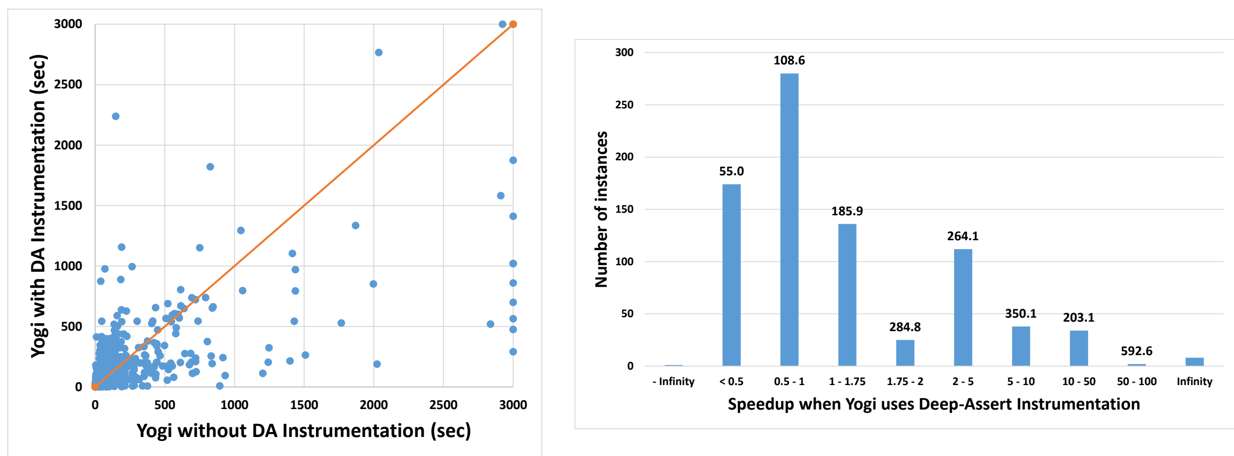


Fig. 10: Yogi with and without deep-assert instrumentation

Concurrency: One scalable approach for the analysis of large (multiple-procedure) concurrent programs is the process of *sequentialization* [13], [7], [8], [15] where a concurrent program is transformed to a sequential program and then verified using a sequential analysis tool. CORRAL supports such a sequentialization; it feeds the resulting sequential program to stratified inlining.

Remark. Sequentializations only preserve end-state reachability and require a variant of the error-bit instrumentation for assertions.[2] This implies that the generated sequential programs have an assertion only at the end of `main`. Consequently, any transformation for revealing information about deep assertions needs to be done on the concurrent program before the sequentialization, as our transformation does.

Fig. 12 reports results on concurrent programs (obtained from [13]) using CORRAL. The improvement is a modest 13% overall, and the assert depth of the benchmarks is also quite shallow. We leave further investigation on concurrent benchmarks for future work.

Summary: We note that there are several other choices of verifiers and it is possible that our program transformation may interact differently with the search heuristics of the verifier. However, our experimental evaluation shows a large potential for speedups, especially given that we do not algorithmically modify the verifier. Further, the program transformation can be applied with any verifier, and takes relatively minimal effort to implement (a few hundred lines of code).

REFERENCES

[1] D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[2] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification*, 2007.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pages 364–387, 2005.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[5] CBMC: Bounded Model Checking for ANSI-C. http://www.cprover.org/cbmc/.

[6] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[7] M. Emmi, A. Lal, and S. Qadeer. Asynchronous programs with prioritized task-buffers. In *Foundations of Software Engineering*, 2012.

[8] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Principles of Programming Languages*, 2011.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, 2001.

[10] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Principles of Programming Languages*, pages 43–56, 2010.

[11] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Foundations of Software Engineering*, 2006.

[12] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *Computer Aided Verification*, 2009.

[13] S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification*, 2009.

[14] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *Computer Aided Verification*, 2012.

[15] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1), 2009.

[16] K. R. M. Leino. Boogie: An intermediate verification language. http://research.microsoft.com/en-us/projects/boogie.

[17] Microsoft. Static driver verifier. http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx.

[18] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *International Conference on Software Engineering*, pages 355–364, 2010.

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages*, 1995.

[20] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[21] N. Sinha. Modular bug detection with inertial refinement. In *Formal Methods in Computer Aided Design*, 2010.

[22] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *Computer Aided Verification*, pages 599–615, 2012.

[2] There are sequentializations that preserve assertions [12], but these have not yet been shown to be scalable to programs in real languages like C.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Infinite-State Backward Exploration of Boolean Broadcast Programs

Peizun Liu and Thomas Wahl
Northeastern University, Boston, USA {lpzun|wahl}@ccs.neu.edu

***Abstract*—Assertion checking for non-recursive unbounded-thread Boolean programs can be performed in principle by converting the program into an infinite-state transition system such as a Petri net and subjecting the system to a *coverability* check, for which sound and complete algorithms exist. Said conversion adds, however, an additional heavy burden to these already expensive algorithms, as the number of system states is exponential in the size of the program. Our solution to this problem avoids the construction of a Petri net and instead applies the coverability algorithm directly to the Boolean program. A challenge is that, in the presence of advanced communication primitives such as broadcasts, the coverability algorithm proceeds backwards, requiring a backward execution of the program. The benefit of avoiding the up-front transition system construction is that "what you see is what you pay": only system states backward-reachable from the target state are generated, often resulting in dramatic savings. We demonstrate this using Boolean programs constructed by the SATABS predicate abstraction engine.**

I. Introduction

Infinite-state system verification continues to be an active field of research. A highly sought-after target are algorithms for the reachability of state sets "upward-closed" with respect to a given well quasi-order; a problem referred to as *coverability*. Recent years have seen intense work on designing practical coverability algorithms that attempt to defy the high computational lower bounds known for this problem.

The application of these algorithms to *programs* — with variable assignments and control flow — rather than state transition systems, is more involved. The data complexity of programs is typically addressed via predicate abstraction. Recent work has pushed the limits of this technique to encompass multi-threaded software [1]. The abstractions are finite-state "Boolean" programs executed concurrently by a possibly unbounded number of threads.

What remains is to close the gap between these programs and the framework of *well quasi-ordered systems (WQOS)* [2], for which coverability problems are decidable. In principle, this can be achieved by formally translating the (symmetric) unbounded-thread Boolean programs into transition systems such as forms of Petri nets: thread-local variable valuations become local states, which in turn are converted into unbounded counter variables, recording the number of threads occupying the corresponding local state at a given time.

In practice, however, this naive method only works for programs with few variables, since the number of states in the resulting WQOS is of course exponential in the size of the program. This explosion — before any kind of system analysis has been performed — makes subsequent coverability analysis intractable but for small programs. In finite-state model checking, the classical method to curb the explosion incured during the program-to-system translation is to avoid the translation altogether and instead build the transition system on the fly: system states are converted to program states, the program is simulated one step, and the resulting program state is converted back into a system state.

In this paper, we build on this idea and present a coverability algorithm — a variant of infinite-state backward search [2] — that operates *directly on the Boolean program*. What makes the classical on-the-fly technique challenging in this context is:

1) the WQOS constructed on the fly does not encode the simulated multi-threaded program directly, but a *counting abstraction* of it: WQOS states store numbers of threads in certain local states. This additional level of indirection must be unraveled before the program can be executed on a system state; and
2) the coverability algorithm [2] proceeds backwards. That is, after unfolding a system state into a program state, we have to execute the program backwards in order to find predecessors. Moreover, the algorithm computes preimages consisting not only of direct predecessors, but also of *cover predecessors*: predecessors of states "larger" than the current state.

The computation of cover predecessors is a consequence of the infinite-state operation of the algorithm; how to do this for Boolean programs is a main technical contribution of this paper. The backward direction of the algorithm is essential to be able to handle *broadcasts*, such as produced by a recent predicate abstraction method [1]. Alternative, forward-directed infinite-state algorithms such as the Karp-Miller procedure [3] are known not to extend naturally to broadcast programs [4].

To summarize, we present in this paper the first, to our knowledge, coverability algorithm for the broad class of Boolean broadcast programs that avoids an up-front construction of (broadcast|Petri) nets or other transition systems. The exploration cost is thus proportional to the backward-reachable system states, rather than the size of the conceivable state space. We show experimental results on 30 predicate-abstracted C programs that convincingly demonstrate how our method speeds up algorithms otherwise known to be well-performing coverability checkers.

This work is supported by NSF grant no. 1253331.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

II. PREPARATIONS

This paper presents an approach to applying Abdulla's infinite-state backward search algorithm [2], designed for well quasi-ordered transition systems (WQOS), to a Boolean program family. In this section we sketch syntax and semantics of Boolean programs, the notion of WQOS and their relation to Boolean program families, and the basics of Abdulla's algorithm to decide certain reachability questions over WQOS.

A. Boolean Broadcast Programs

Boolean programs typically arise from predicate abstractions of `C` or Java code. All variables are of type `bool`. Control flow constructs are optimized for automated analysis, rather than ease of programming.

An overview of the syntax of Boolean programs is given in Fig. 1 and mostly compatible with that used in the CPROVER toolkit[1]. A program is a top-level **decl**aration of Boolean variables — called *shared* — with compile-time computable, possibly nondeterministic initial values, followed by a list of function definitions. A function definition is an initializing **decl**aration of Boolean variables called *local*, followed by a list of labeled statements.

```
prog    ::= decl initvarlist; func*
func    ::= name (varlist) { decl initvarlist; [label: stmt;]* }
stmt    ::= seqstmt
          | start_thread label
          | atomic { [stmt;]* }
          | wait
          | broadcast
seqstmt ::= skip
          | goto labellist
          | assume (expr)
          | varlist := exprlist [constrain expr]
          | if (expr) then seqstmt else seqstmt fi
          | assert (expr)
```

Fig. 1: Boolean program syntax (partial; slightly simplified)

A formal description of the semantics of Boolean program statements is beyond the scope of this paper. We sketch here the main concepts; for some details see Table I, for more details see [5]. The **skip** statement advances the program counter (pc); **goto** *labellist* nondeterministically chooses one of the given labels as the next pc; **assume** terminates executions that do not satisfy the given expression. The **:=** statement assigns the values of the given expressions to the respective variables, in parallel, but terminates the execution if the result does not satisfy the **constrain** expression, if any. The semantics of **if** is standard; **assert** indicates assertions for verification and otherwise acts like **skip**. In all cases, *expr* is a Boolean expression over shared and local variables of the program, the constants 0 and 1, and the choice symbol $\star$; the latter nondeterministically evaluates to 0 or 1. For example, the statement `assume` $(b \wedge \star)$ behaves like `skip` in states where $b = 1$, and terminates the execution in states where $b = 0$. Function calls and **return** statements are omitted; they have standard semantics.

The remaining statements in Fig. 1 support threading in Boolean programs. Their intuitive semantics is as follows:

start_thread *label* (i) advances the pc of the executing thread to the next statement, and (ii) creates a new thread whose local variables are copied from those of the executing thread and whose pc is given by *label*.

atomic $\{$*stmt*$^*\}$ denotes atomic execution: a thread executing inside an atomic section cannot be preempted.

wait blocks the execution of a thread (see next).

broadcast advances the pc of the executing thread, and wakes up all threads currently blocked at a **wait** statement, if any, i.e. it advances their pc as well. A **broadcast** is thus non-blocking. (More general models may offer distinct pairs of **wait/broadcast** statements, using *condition variables*.)

Thread termination is omitted, as — for the purposes of reachability analysis — it can be simulated by trapping the terminating thread in a self loop. Fig. 2 (left) shows a Boolean program with an assertion. We are interested in this paper in detecting assertion violations: does there exist a multi-threaded execution of the program in which some thread reaches a failing assertion?

B. From Programs to Infinite-State Transition Systems

Let $\mathcal{B}$ be a Boolean program defined over sets of shared and local Boolean variables V_S and V_L, respectively, and let $\{1, \ldots, pc_{\max}\}$ be the set of program locations. $\mathcal{B}$ gives rise to an infinite-state transition system M^∞ as follows. The states of M^∞ have the form $(s, \ell_1, \ldots, \ell_n)$, where s is a valuation of the shared variables of $\mathcal{B}$ and is called the *shared state*. Symbol ℓ_i is a valuation of the pc and the local variables of $\mathcal{B}$ and is called the *local state of thread i*. We write $s.v$ ($\ell_i.v$) for the value of shared (local) variable v in shared (local) state s (ℓ_i), and $\ell_i.pc$ for thread i's current pc value. Finally, n is a positive integer, intuitively the number of threads currently running. The state space of M^∞ is therefore the infinite set

$$S^\infty = \{0,1\}^{|V_S|} \times \bigcup_{n=1}^{\infty} \Big(\{1, \ldots, pc_{\max}\} \times \{0,1\}^{|V_L|} \Big)^n \; .$$

A transition of M^∞ is of the form

$$(s, \ell_1, \ldots, \ell_n) \quad \rightarrow \quad (s', \ell'_1, \ldots, \ell'_{n'})$$

such that one of the following conditions holds:

1) $n' = n$ and there exists $i \in \{1, \ldots, n\}$ such that (i) the statement at $\ell_i.pc$ is a *seqstmt*; executing it *atomically* from the variable valuation given by (s, ℓ_i) results in the variable valuation given by (s', ℓ'_i), and (ii) for $j \in \{1, \ldots, n\} \setminus \{i\}$, $\ell'_j = \ell_j$.
2) $n' = n + 1$, $s' = s$ and there exists $i \in \{1, \ldots, n\}$ such that (i) the statement at $\ell_i.pc$ is of the form `start_thread` x, (ii) $\ell'_i.pc = \ell_i.pc + 1$ and $\ell'_i.v =$

[1] http://www.cprover.org/boolean-programs/grammar.pdf

$\ell_i.v$ for $v \in V_L$, (iii) $\ell'_{n'}.pc = x$ and $\ell'_{n'}.v = \ell_i.v$ for $v \in V_L$, and (iv) for every $j \in \{1,\ldots,n\} \setminus \{i\}$, $\ell'_j = \ell_j$.

3) $n' = n$, $s' = s$ and there exists $i \in \{1,\ldots,n\}$ such that (i) the statement at $\ell_i.pc$ is `broadcast`, (ii) $\ell'_i.pc = \ell_i.pc + 1$ and $\ell'_i.v = \ell_i.v$ for $v \in V_L$, (iii) for every $j \in \{1,\ldots,n\} \setminus \{i\}$ such that the statement at $\ell_j.pc$ is **wait**, $\ell'_j.pc = \ell_j.pc + 1$ and $\ell'_j.v = \ell_j.v$ for $v \in V_L$, and (iv) for every $j \in \{1,\ldots,n\} \setminus \{i\}$ such that the statement at $\ell_j.pc$ is *not* **wait**, $\ell'_j.pc = \ell_j.pc$ and $\ell'_j.v = \ell_j.v$ for $v \in V_L$.

In each case, thread i is called *active*, the others *passive*. We omit the precise formalization of **atomic** blocks, which is, however, straightforward. The initial states of M^∞ are given by (i) $n = 1$ and (ii) s and ℓ_1 determined by the (nondeterministically) initializing declarations in $\mathcal{B}$ and by $\ell_1.pc = 1$.

Transition system M^∞ thusly defined is a *well quasi-ordered transition system* (WQOS) [2]. That is, there exists a well-quasi order $\succeq$ on S^∞ that satisfies a *monotonicity* property: for states x, y, x' with $x \to x'$ and $y \succeq x$, we can find y' such that $y' \succeq x'$ and $y \to y'$. This order is the *covers* relation:

$$(\bar{s}, \bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}) \succeq (s, \ell_1, \ldots, \ell_n)$$

whenever $\bar{s} = s$ and $[\bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}] \supseteq [\ell_1, \ldots, \ell_n]$, where $[\cdot]$ denotes a *multiset*. The well-quasi orderedness follows from properties of $\supseteq$ and Dixon's lemma; the monotonicity of $\to$ with respect to $\succeq$ follows since actions of a thread in a state cannot be disabled by adding threads to the state; see semantics of M^∞. These are standard concepts.

The multi-threaded assertion violation question can now be phrased as a *coverability problem* for the derived WQOS M^∞: let Q be the set of (shared, local) state pairs (s, ℓ) such that the statement at $\ell.pc$ is an assertion that is violated by the variable valuation given by (s, ℓ). Coverability of the "bad-states set" Q asks whether a state z is reachable such that, for some $q \in Q$, $z \succeq q$. Coverability is decidable but of high complexity, e.g. *Ackermann*-complete for Petri nets with broadcasts (a form of WQOS), which means that the complexity grows as fast as the Ackermann function [6].

C. Backward Search

A sound and complete algorithm to decide coverability for WQOS is the *backward search* algorithm by Abdulla et al. [2], [7], a high-level version of which is shown in Alg. 1. In this listing, symbol $\uparrow U$ stands for the *upward closure* of U: $\uparrow U = \{\bar{u} : \exists u \in U : \bar{u} \succeq u\}$. Input to Alg. 1 is a set of initial states $I \subseteq S^\infty$, and a target set $Q \subseteq S^\infty$. The algorithm maintains a work set $W \subseteq S^\infty$ of unprocessed states, and a set $U \subseteq S^\infty$ of minimal encountered states. It successively computes minimal *cover predecessors*

$$\mathsf{CovPre}(w) = \min\{p : \exists \bar{w} \succeq w : p \to \bar{w}\} \qquad (1)$$

starting from elements in Q, and terminates either by backward-reaching an initial state (thus proving coverability of some $q \in Q$), or when no unprocessed vertex remains (thus proving uncoverability).

Algorithm 1 BWS(I, Q)

Input: initial states I, target set Q disjoint from I

1: $W := Q\,;\ U := Q$
2: **while** $\exists w \in W$
3: $\quad W := W \setminus \{w\}$
4: $\quad$ **for** $p \in \mathsf{CovPre}(w) \setminus \uparrow U$
5: $\quad\quad$ **if** $p \in I$ **then**
6: $\quad\quad\quad$ "some $q \in Q$ coverable"
7: $\quad\quad W := \min(W \cup \{p\})$
8: $\quad\quad U := \min(U \cup \{p\})$
9: "no $q \in Q$ coverable"

III. Boolean Program Backward Search: Overview

We illustrate our approach using the Boolean program $\mathcal{B}$ in Fig. 2 (left). The program is started by one thread; the nondeterministic `goto` in Line 1 determines whether to launch an additional thread in Line 2. Suppose not (we proceed in Line 3). Then the left branch starting at the node corresponding to Line 4 (Fig. 2, right) is not executable since $t = 0$ violates `assume(t)`. Along the right branch, local variable m is not modified, `assume(!t)` passes, and so does the `assert(!m)` in Line 10.

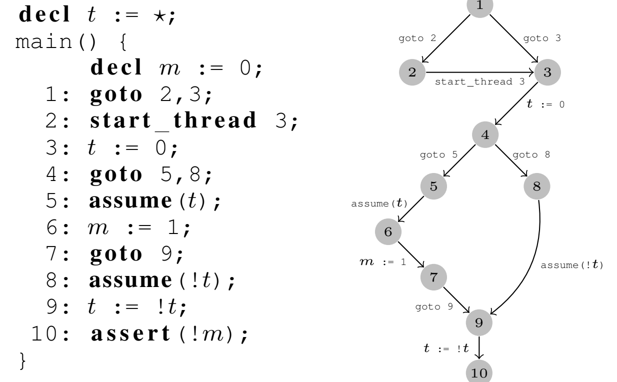


Fig. 2: A Boolean program with $V_S = \{t\}$, $V_L = \{m\}$ (left); its control flow graph (right)

Nonetheless, program $\mathcal{B}$ permits an assertion violation, as the backward trace in Fig. 3 shows. The target set is $\{(0|^{10}/_1), (1|^{10}/_1)\}$; the trace shown starts from $(0|^{10}/_1)$. Our backward search algorithm proceeds from a given state w in two steps: (i) we select a thread in w as active and compute all direct predecessors that $\mathcal{B}$ permits; (ii) we try to find *expanded* predecessors of w, explained below.

Let us first consider a direct predecessor example, using state $(1|^7/_1)$ in the first row. The algorithm first consults the control flow graph, shown in Fig. 2 (right), for possible control predecessors of $pc' = 7$. There is only one, $pc = 6$. The statement along this edge is `m := 1`. We therefore now compute the weakest precondition of state $t = 1, m = 1$ under this statement, i.e.

$$\mathsf{WP}_{m\ :=\ 1}(t \wedge m) \quad = \quad t\ ,$$

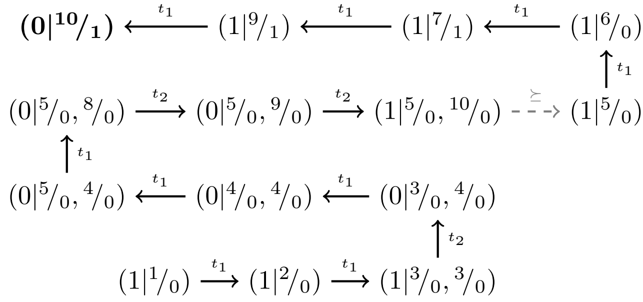


Fig. 3: Coverability analysis using backward search, applied to the program in Fig. 2 (left). Targets are the states (s, ℓ) satisfying $\ell.pc = 10, \ell.m = 1$. Notation $(t_0|^{pc_1}/_{m_1}, \ldots)$ denotes a global state with shared variable $t = t_0$ and the given values for the local variables pc and m, for the various threads. Labels atop transitions indicate the active thread; $\succeq$ indicates expansion

indicating $\{(1|^6/_0), (1|^6/_1)\}$ as the set of direct predecessors. Both are recorded in our algorithm; the trace shown in Fig. 3 continues with state $(1|^6/_0)$.

Direct predecessor computation alone will never increase the number of involved threads and thus cannot detect multi-threaded assertion violations. Alg. 1 involves a step we call *expansion* of w to a larger state $\bar{w}$, by adding to w a thread in a suitable local state not present in w. Expanded predecessors of w are then the (minimal) direct predecessors of $\bar{w}$, obtained by backward-executing the added thread. Sections IV and V present the details of this step, especially that adding a single thread to w is sufficient, what a "suitable" local state is, and that direct and expanded predecessors constitute exactly the set of all cover predecessors of w (Eq. (1)).

Consider the example of state $(1|^5/_0)$ in Fig. 3. The only direct predecessor is $(1|^4/_0)$, from which there is no further direct predecessor: the only CFG edge entering Line 4 is labeled with statement `t := 0`, which does not permit $t = 1$ in the current state. We thus try to expand. As we will see, expansion is only useful if the added thread gives rise to a predecessor *with a modified shared state*. Only few statements in $\mathcal{B}$ change t; one is `t := !t` in Line 9. Expansion therefore adds a thread with $pc = 10$, namely in local state $^{10}/_0$. The direct global predecessor state $(0|^5/_0, ^9/_0)$ has changed t by backward-executing `t := !t`. From now on the backward search proceeds with two threads until we encounter the `start_thread` command; backward-executing it eliminates thread 2. At the end, the search reaches the initial state $(1|^1/_0)$, proving reachability of the violated assertion.

IV. Boolean Program Backward Search

This section presents our infinite-state backward search algorithm, applied to an unbounded-thread Boolean program $\mathcal{B}$.

A. Data Structures and Prerequisites

While exploring $\mathcal{B}$, the algorithm builds — on the fly — the infinite-state structure M^∞. States $\tau = (s, m_1, \ldots, m_n)$ of this structure are stored in the form

$$\tau = \langle s, \{(\ell_1, n_1), \ldots, (\ell_k, n_k)\}\rangle \quad (2)$$

where $\ell_1, \ldots, \ell_k$ are the *distinct* local states occurring in τ, and for $i \in \{1, \ldots, k\}$, $n_i = |\{j : m_j = \ell_i\}|$. That is, instead of listing the local states of all n threads in τ, (2) collapses multiple occurrences of local states and lists their count. Threads in the same local state are equivalent under standard symmetry equivalence; their order in τ and their identities are immaterial. By construction, $n_i > 0$ for all i.

The algorithm assumes the control flow graph (CFG) of $\mathcal{B}$ is given as $G = (\{1, \ldots, pc_{\max}\}, E)$. The CFG is a directed graph over the program locations of $\mathcal{B}$. Each edge $e \in E$, with source and target $source(e)$ and $target(e)$, resp., is labeled with the statement $e.stmt$ of $\mathcal{B}$ that carries the control from $source(e)$ to $target(e)$. For example, Line 1 of the program in Fig. 2 (left) induces **two** edges in G, as shown on the right.

For a statement $stmt$ and states (s, ℓ) and (s', ℓ') of $\mathcal{B}$, let $\mathsf{WP}_{stmt}(s, \ell, s', \ell')$ be a Boolean formula asserting that state (s, ℓ) satisfies the *weakest precondition* for state (s', ℓ') under statement $stmt$. That is, $\mathsf{WP}_{stmt}(s, \ell, s', \ell')$ holds exactly if executing $stmt$ from state (s, ℓ) results in state (s', ℓ').[2] Examples for sequential statements are given in Table I.

Statement $stmt$	$\mathsf{WP}_{stmt}(s, \ell, s', \ell')$
`skip`	$\ell'.pc = \ell.pc + 1 \wedge invar$
`assume` $(t = m)$	$s.t = \ell.m \wedge \ell'.pc = \ell.pc + 1 \wedge invar$
`t := *`	$\ell'.m = \ell.m \wedge \ell'.pc = \ell.pc + 1$

TABLE I: Examples for weakest precondition formulas for a program $\mathcal{B}$ with $V_S = \{t\}$ and $V_L = \{m\}$. Symbol $invar$ stands for "data invariance": $s'.t = s.t \wedge \ell'.m = \ell.m$

B. Cover Predecessor Computation

Our algorithm is an instance of the high-level scheme shown in Alg. 1. The only (albeit substantial) modification is the computation of the cover predecessor function in Line 4. The definition of this function, Eq. (1), poses the following challenges for an implementation:

1) given w, expanded elements $\bar{w} \succeq w$ need to be explored;
2) given $\bar{w}$, a preimage needs to be computed.

Regarding 2), our algorithm will use the CFG of $\mathcal{B}$ and weakest precondition transformers to compute preimages of states. In order to meet challenge 1), we need an "upper bound" on the expanded elements $\bar{w}$. This is accomplished by the following lemma. We denote by $|w|$ the number of threads in state $w \in S^\infty$, e.g. $|(s, \ell_1, \ldots, \ell_n)| = n$.

Lemma 1. *For a transition relation $\rightarrow$ induced by a Boolean broadcast program, and states $p, w, \bar{w} \in S^\infty$,*

$$p \in \mathit{CovPre}(w) \wedge \bar{w} \succeq w \wedge p \rightarrow \bar{w} \quad \Rightarrow \quad |\bar{w}| \leq |w| + 1.$$

Proof: Since p is a *minimal* cover predecessor of w, there are no states $o \prec p$ and $\bar{v}$ such that $\bar{v} \succeq w$ and $o \rightarrow \bar{v}$. Note that $o \prec p$ abbreviates $o \preceq p \wedge \neg(o \succeq p)$.

[2]We note that all atomic (non-compound) statements of $\mathcal{B}$ are terminating.

We prove $|\bar{w}| \leq |w| + 1$ via contraposition: assume $|\bar{w}| \geq |w|+2$. From this assumption and $\bar{w} \succeq w$, we conclude that $\bar{w}$ contains, in some permutation, all $|w|$ local states that w also contains, and at least two more local states, say at positions j_1 and j_2 with $j_1 \neq j_2$. Let i be the index of the thread active during transition $p \to \bar{w}$. We observe that, since $j_1 \neq j_2$, there exists $j \in \{j_1, j_2\}$ such that $j \neq i$. Let now o and $\bar{v}$ be the same states as p and $\bar{w}$, respectively, except that thread j is dropped. Then: (i) $o \prec p$; (ii) $o \to \bar{v}$, because thread j is *not* active in $p \to \bar{w}$, and dropping j does not invalidate the transition; and (iii) $\bar{v} \succeq w$, since $\bar{w} \succeq w$ and $|\bar{w}| \geq |w| + 2$, and we have dropped only one thread from $\bar{w}$ to obtain $\bar{v}$. Properties (i)–(iii) contradict the minimality of p, as stated at the beginning of the proof. ■

We now turn to the main result in this paper: the procedure for cover predecessor computation (Alg. 2). Input is a state τ' in format (2) (we attach a prime $'$ to the input symbols to suggest that we are computing preimages). The results are collected in a set $\mathcal{C}$.

The computation of cover predecessors according to Eq. (1) involves finding an element $\bar{w}$ satisfying $\bar{w} \succeq w$, and then determining predecessors of $\bar{w}$. Condition $\bar{w} \succeq w$ is tantamount to $\bar{w} \preceq\succeq w \vee \bar{w} \succ w$ (where $\bar{w} \preceq\succeq w$ means *equivalence*: $\bar{w} \succeq w \wedge w \succeq \bar{w}$). The algorithm deals with the two cases $\bar{w} \preceq\succeq w$ and $\bar{w} \succ w$ separately, as follows.

1) Direct predecessors: Condition $\bar{w} \preceq\succeq w$ means that there exists a thread permutation π such that $w = \pi(\bar{w})$ (π reorders threads in the local state vector representation of $\bar{w}$). By thread symmetry, w and $\bar{w}$ therefore have the same sets of $\to$ predecessors, up to applying local state permutations. Now observe that states that are identical up to local state permutations have the same thread counter representation (2). This means that, in the case $\bar{w} \preceq\succeq w$, we can ignore the "detour" through $\bar{w}$ and directly compute predecessors of w: those are the elements of $\mathsf{CovPre}(w)$. Naturally, we call such elements *direct predecessors*.

To compute direct predecessors, Alg. 2 iterates through all local states ℓ'_i (thus, implicitly, threads) present in τ'. It then consults the CFG for edges e leading to the current program location $\ell'_i.pc$ of any of the threads in ℓ'_i (Lines 2 and 3). Reversing edge e, i.e. executing it backwards on (s', ℓ'_i), gives us the desired predecessors.

To this end, the algorithm switches over the possible types of statement $e.stmt$:

`start_thread` x: this is possible exactly if the current state τ' contains a "started" thread in some local state ℓ'_j with $\ell'_j.pc = x$ and same data as the thread in ℓ'_i (Line 6). If so, in the predecessor state τ, the thread in ℓ'_i is unchanged except that its pc is the previous program location (Line 7; notation $\ell'_i\,[pc \mapsto Y]$ returns ℓ'_i except pc replaced by Y). To construct τ, we update the thread counters: those for ℓ'_i and ℓ'_j are decremented; that of the predecessor local state ℓ_i is incremented (the thread in ℓ'_j has just been created, so going backwards it "disappears"). These updates are done in Line 8 via a function UPDATE-COUNTERS explained below. The updates are performed for all eligible local states ℓ'_j; the results are added to $\mathcal{C}$.

`broadcast`: since broadcasts are non-blocking, they are executable from any predecessor state. The broadcasting thread's pc is decreased by one; the change is recorded in a temporary variable Y. Line 13 selects all threads with program counter pc such that the statement at $pc - 1$ is `wait`: these threads *may* have just been released by the broadcast. However they may also have resided in location pc *before* the broadcast was issued — the exact subset J of indices of threads that are released cannot be determined when exploring $\mathcal{B}$ backwards.

Therefore, the algorithm iterates through all such sets $J \subseteq \mathcal{J}$ and threads $j \in J$ (Line 16): these threads are "unreleased", i.e. their pc is set back to the `wait` location. The updates to Z in Lines 18–19 (see Alg. 4 for the MERGE function) perform counter updates for the synchronous state change of all unreleased threads.

default: this case takes care of all sequential statements: using the weakest precondition function WP, we generate all possible predecessor program states (s, ℓ), update the counters, and add the results τ to $\mathcal{C}$. Solving the Boolean formula $\mathsf{WP}_{e.stmt}(s, \ell, s', \ell'_i)$ for (s, ℓ) is done with the aid of a SAT solver (see Sect. VI).

Note that the **switch** in Line 4 does not process `wait` statements: these cannot backward-execute by themselves, as releasing waiting threads happens in synchrony with broadcasts.

2) Expanded predecessors: We now consider the case $\bar{w} \succ w$. We have to expand state w by adding threads to it, followed by the computation of predecessors of the expanded state $\bar{w}$. The following observations render this step feasible:

- by Lemma 1, adding a **single** thread to w is sufficient;
- when computing predecessors p of $\bar{w}$, those obtained when the single added thread is active are sufficient: predecessors triggered by threads already present in w are handled as direct predecessors of w.

Alg. 2 implements the expanded predecessor computation along those principles. In Line 25 we determine states (s, ℓ) of $\mathcal{B}$ such that there exists a local state m' (= that of the added thread) not present in τ' such that the following holds: (i) the pc values of ℓ and m' form an edge $e \in E$, and (ii) executing $e.stmt$ from (s, ℓ) leads to (s', m').

The solutions to the constraint in Line 25 are determined using a SAT solver, which is passed the constraint as an existentially quantified Boolean formula. In this formula, all of s', $\ell'_1, \ldots, \ell'_k$ are Boolean constants; the only variables are s, ℓ (free) and m' (quantified). The solutions (s, ℓ) we are looking for are the assignments satisfying this formula, projected to s and ℓ. To avoid excessive enumeration, we discuss in Sect. V how the selection of candidate local states m' for expansion can be substantially and soundly restricted.

We conclude this algorithm description by explaining function UPDATE-COUNTERS(T, T', Z'), shown in Alg. 3. It increments/decrements the counters for all local states in T/T', using the MERGE function from Alg. 4.

Algorithm 2 CovPre(τ')

Input: $\tau' = \langle s', Z' \rangle$, where $Z' = \{(\ell'_1, n'_1), \ldots, (\ell'_k, n'_k)\}$
Output: cover predecessors of τ'

```
1:  C := ∅
2:  for each i ∈ {1,...,k}                                             ▷ direct predecessors
3:      for each e ∈ E s.t. target(e) = ℓ'_i.pc
4:          switch e.stmt:
5:          case start_thread x, for some x:
6:              for each j ∈ {1,...,k} \ {i} s.t. ℓ'_j.pc = x ∧ ∀v ∈ V_L : ℓ'_j.v = ℓ'_i.v
7:                  ℓ_i := ℓ'_i [ pc ↦ ℓ'_i.pc − 1 ]
8:                  τ := ⟨s', UPDATE-COUNTERS({ℓ_i}, {ℓ'_i, ℓ'_j}, Z')⟩
9:                  C := C ∪ {τ}
10:         case broadcast:
11:             ℓ_i := ℓ'_i [ pc ↦ ℓ'_i.pc − 1 ]
12:             Y := UPDATE-COUNTERS({ℓ_i}, {ℓ'_i}, Z')
13:             J := {j ∈ {1,...,k} \ {i} s.t. stmt. at ℓ'_j.pc − 1 is wait}
14:             for each J ⊆ J
15:                 Z := Y
16:                 for each j ∈ J
17:                     ℓ_j := ℓ'_j [ pc ↦ ℓ'_j.pc − 1 ]
18:                     Z := Z \ {(ℓ'_j, n'_j)}
19:                     MERGE(ℓ_j, n'_j, Z)
20:                 C := C ∪ {⟨s', Z⟩}
21:         default:
22:             for each (s, ℓ) s.t. WP_{e.stmt}(s, ℓ, s', ℓ'_i)
23:                 τ := ⟨s, UPDATE-COUNTERS({ℓ}, {ℓ'_i}, Z')⟩
24:                 C := C ∪ {τ}
25: for each (s, ℓ) s.t. ∃m' ∉ {ℓ'_1,...,ℓ'_k} : e := (ℓ.pc, m'.pc) ∈ E ∧ WP_{e.stmt}(s, ℓ, s', m')   ▷ expanded predecessors
26:     τ := ⟨s, UPDATE-COUNTERS({ℓ}, ∅, Z')⟩
27:     C := C ∪ {τ}
28: return C
```

Algorithm 3 UPDATE-COUNTERS(T, T', Z')

Input: T: local states whose counter is to be incremented
T': local states whose counter is to be decremented
Z': thread counter vector

```
1: for each ℓ ∈ T
2:     MERGE(ℓ, 1, Z')
3: for each ℓ' ∈ T'
4:     let n' be such that (ℓ', n') ∈ Z'          ▷ n' is unique
5:     Z := Z' \ {(ℓ', n')} ∪ ( n' > 1 ? {(ℓ', n' − 1)} : ∅ )
6: return Z
```

Algorithm 4 MERGE(ℓ, n, Z)

Input: ℓ: local state, n: counter, Z: thread counter vector

```
1: if there exists n' such that (ℓ, n') ∈ Z then
2:     Z := Z \ {(ℓ, n')} ∪ {(ℓ, n' + n)}
3: else
4:     Z := Z ∪ {(ℓ, n)}
```

V. EFFICIENCY

In Sect. IV-B2 we saw two specializations of the generic backward coverability Alg. 1 that apply to the computation of expanded cover predecessors for Boolean programs. In particular, the bound on the size of expanded states $\bar{w}$ makes CovPre(w) effectively computable. In this section we describe two improvements that are essential to, among others, curb the number of candidate local states m' in Line 25.

The first improvement is that m' can be restricted such that the statement along edge $e = (\ell.pc, m'.pc)$ changes the shared state. The justification is as follows. Suppose $e.stmt$ does not change the shared state, i.e. $s = s'$. The cover predecessor state τ is thus of the form $(s', \ldots, \ell, \ldots)$. Any cover predecessor of τ that is obtained by backward executing $\mathcal{B}$ from program state (s', ℓ) is also a cover predecessor of the original τ': we simply expand τ' by local state ℓ instead of m'.

This improvement is easy to implement: we change Line 25 in Alg. 2 to the following *two* lines:

```
for each e ∈ E s.t. e.stmt may modify the shared state
    for each (s, ℓ) s.t. s ≠ s' ∧ ∃m' ∉ {ℓ'_1,...,ℓ'_k} :
    target(e) = m'.pc ∧ WP_{e.stmt}(s, ℓ, s', m')
        τ := ...            ▷ (continue with Line 26 of Alg. 2)
```

That is, we first select edges e that have the *potential* to change the shared state. Such are edges that assign a variable

in V_S; they can be identified inexpensively up front, while building the CFG. We then determine states (s, ℓ) of $\mathcal{B}$ such that (i) shared state s is actually different from s', and (ii) there exists a local state m' not present in τ' whose pc is the target of e and such that $\mathsf{WP}_{e.stmt}(s, \ell, s', m')$.

This improvement is also highly effective: only few of the syntactically possible statements may actually change the shared state, and their frequency in Boolean programs is proportionately small, as we demonstrate in Table II (Sect. VI).

The second improvement exploits that local states m' that are not *forward-reachable* from an initial state of M^∞ can be omitted, since they obviously are not part of any legal execution. While the exact determination of reachability of a local state is of course a coverability problem in itself, we can employ very inexpensive overapproximating analyses that soundly provide *unreachability* information.

One such analysis, adapted from [8], is to execute $\mathcal{B}$ essentially as a single-threaded program. That is, statements related to multi-threading are ignored (`start_thread` in particular). Sequential statements are honored, except that:

1) assignments to shared variables are ignored
2) conditionals that depend on shared variables are replaced by $\star$, i.e. in `assume`, `constrain`, and `if` statements.

The set $\mathcal{L}$ of local states reachable in this single-threaded program is cheap to compute and overapproximates the precise set of local states reachable in the multi-threaded execution of $\mathcal{B}$. That is, local states not in $\mathcal{L}$ are unreachable. We exploit this information by adding the requirement $m' \in \mathcal{L}$ to the second **for each** statement in the above modification to Alg. 2.

In fact, this insight not only applies to the selection of states m' during expanded predecessor computation: we can simply ignore local states generated during the backward search (Lines 7, 11, 17, 22 in Alg. 2) that do not belong to $\mathcal{L}$. This technique can be seen as an instance of combining forward and backward analysis for increased efficiency (executing $\mathcal{B}$ is tantamount to forward reachability analysis). This idea was used in several other works, such as [9], where an incomplete forward-searching Karp-Miller procedure assists a slower but complete backward coverability analysis. Incidentally, the Karp-Miller implementation used in [9] may not terminate and may thus *underapproximate* the set of coverable configurations when applied to broadcast nets. In contrast, we need an overapproximation, as our goal is to prune encountered states that are guaranteed to be **un**reachable.

VI. Empirical Evaluation

In this section, we evaluate our verifier UCOB[3] on a set of 30 non-recursive concurrent C programs. Threads synchronize through diverse communication primitives, such as shared variables, mutex variables, and broadcasts. All programs contain procedures executed by an arbitrary number of threads, which are dynamically spawned by the initial thread.

[3] "**U**nbounded-thread **co**verability analysis for **b**oolean programs"

For each benchmark, we consider verification of a safety property, specified via an assertion. In total, the programs include roughly 2300 lines of code; on average they feature 3 shared and 6 local variables (cf. Table II). The programs are available from the homepage of the second author.

01–10: thread-safe algorithms: atomic counters (1–2); concurrent pseudo-random number generator (3–4); maximum element finding algorithm (5–8); stack data structure with concurrent operations (9-10).

11–17: OS code: code from the FreeBSD (11–12), NetBSD (13), Solaris (14) and Linux (15–17) open-source operating systems.

18–22: pthread programs: several programs that use the C Posix Threads library.

23–28: mutex algorithms: test-and-set lock (23); multiple locks control access to a shared resource (24–26); two ticket algorithms (27–28).

29–30: misc: two simple examples from [1].

Implementation: UCOB uses SATABS [10] to construct Boolean programs from C. To compare with coverability tools that don't accept Boolean programs as input, we also use SATABS to generate thread transition system (TTS) models as input (option `--build-tts`). Finally, we use miniSAT [11] to solve WP formulas. UCOB offers both optimizations presented in Sect. V: the shared-variable restriction, and the single-thread forward-reachable local states computation. The latter employs the tool BOOM [12], which we call a "forward oracle" in this context (a terminology suggested in [9]).

The experiments are performed on a 2.3GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux.

Comparison: For $1 \leq k \leq 30$, Fig. 4 plots the total time (log-scale) taken to solve the k easiest of our benchmark problems, for the following tools:

UCOB: our tool with shared-variable optimization;
UCOB/BOOM: UCOB with BOOM as forward oracle;
MCOV: MCOV without forward oracle [9];
MCOV/GKM: MCOV with forward oracle [9];
BOOM-KM: Karp-Miller implementation in BOOM [12].

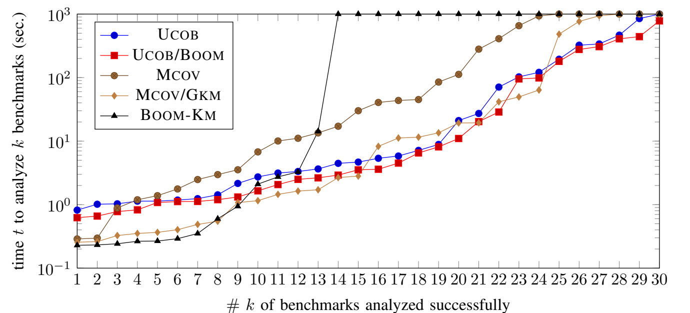


Fig. 4: Cactus plot comparing UCOB with other coverability tools. For each curve, entry (k, t) shows the time t it took to solve the k easiest — for the method associated with that curve — benchmarks (order varies across methods).

TABLE II: Benchmark characteristics and results: *SV* / *LV* / *LOC* = # of shared / local C program variables / lines of code; *Mtx?* / *Bc?* = presence of mutex vars / broadcasts (● = "yes"); $|V_S|$ / $|V_L|$ / *Its.* = # of shared / local Boolean variables / CEGAR iterations; *Mod.Sh.* = percentage of statements that may modify the shared state; *Safe?* = program safety

ID/Program	C Program					Boolean Program				Safe?
	SV	LV	LOC	Mtx?	Bc?	$\lvert V_S\rvert$	$\lvert V_L\rvert$	Its.	Mod.Sh.	
01/INC-L	2	1	46	●	○	3	1	2	7.5	●
02/INC-C	1	3	57	○	○	0	4	4	0	●
03/PRNSIMP-L	2	4	63	●	○	2	3	2	7.7	●
04/PRNSIMP-C	1	5	95	○	○	0	5	2	0	●
05/MAXSIM-L	3	3	59	●	○	1	0	2	3.7	●
06/MAXSIM-C	2	5	79	○	○	0	1	2	0	●
07/MAXOPT-L	3	4	69	●	○	1	1	2	3.1	●
08/MAXOPT-C	2	6	86	●	○	0	2	2	0	●
09/STACK-L	4	2	79	●	○	1	3	3	3.8	●
10/STACK-C	3	3	89	○	○	3	1	2	6.4	●
11/BSD-AK	1	7	90	●	●	3	1	15	11.7	●
12/BSD-RA	2	21	87	●	●	3	0	19	12.3	●
13/NETBSD	1	28	152	●	●	3	1	30	10.1	●
14/SOLARIS	1	56	122	●	●	5	1	14	10.9	●
15/BOOP	5	2	89	○	○	5	2	4	11.4	○

ID/Program	C Program					Boolean Program				Safe?
	SV	LV	LOC	Mtx?	Bc?	$\lvert V_S\rvert$	$\lvert V_L\rvert$	Its.	Mod.Sh.	
16/QRCU-2	7	6	120	○	○	3	0	16	10.1	●
17/QRCU-4	8	8	182	○	○	5	2	28	9.8	●
18/BS-LOOP	0	6	24	○	○	0	7	1	0	○
19/COND	1	3	56	●	○	0	3	2	0	●
20/FUNC-P	2	1	67	●	○	2	6	3	8.3	●
21/S-LOOP	5	0	60	●	○	4	0	20	22.8	●
22/PTHREAD	5	0	85	●	○	7	0	5	17.1	○
23/TAS-L	2	2	58	○	○	3	1	2	14.9	●
24/DOUBLE-1	3	0	70	●	○	7	1	10	16.4	●
25/DOUBLE-2	3	0	73	●	○	6	1	23	18.2	●
26/DOUBLE-3	3	0	66	●	○	4	1	3	15.3	●
27/TICKET-HC	3	1	61	○	○	5	1	5	18.4	●
28/TICKET-LO	3	1	46	○	○	5	1	5	20.8	●
29/UNVEREX	2	1	25	○	○	4	0	3	8.9	●
30/SPIN	2	0	37	●	○	3	0	2	15.5	●

The results in the chart demonstrate that UCOB solves almost all of the benchmarks (29), and does so in less time for most programs. Furthermore, the results for UCOB/BOOM show that the forward oracle, despite being an overapproximation, can accelerate the backward search by pruning unreachable cover predecessors. MCOV/GKM is the most competitive proof tool. For the small-scale benchmarks, where the TTS construction does not blow up, it takes the least amount of time. However, the efficiency drops sharply with increasing cost of either TTS generation or verification. The only other unbounded-thread on-the-fly verifier we are aware of, BOOM-KM, benefits from forward search and is thus competitive "early", but reports runtime errors for some of the more complex benchmarks. Others it cannot solve: the Karp-Miller implementation in BOOM-KM does not support broadcasts.

VII. RELATED WORK AND CONCLUDING REMARKS

The issue of the blow-up incurred when translating a program model $\mathcal{B}$ into a transition system model M is classically addressed using an on-the-fly exploration. In the context of symmetric concurrent systems, things are more complicated as the transition system (a WQOS) does not model $\mathcal{B}$ directly, but via a counting abstraction. In [13], this issue was addressed for the finite-state case, by interleaving the counting abstraction and transition system construction. We have borrowed the state representation (2) and (mostly) the counter update function UPDATE-COUNTERS (Alg. 3) from that work.

We are aware of very few attempts to address the issue in connection with (much more complex) *infinite-state* verification techniques. While these techniques have been applied to programs directly [14], [15], the application is typically preceded by a static compilation of the program into an explicit transition system, which only works for small local state spaces, for example when predicates used for abstraction are hand-picked.

An on-the-fly implementation of the Karp-Miller algorithm is available in BOOM [12]. This algorithm proceeds forward, making the implementation much easier. On the other hand, due to theoretical limitations of Karp-Miller (see e.g. [4]), this tool cannot handle broadcast programs, our target language. On non-broadcast programs, it suffers from the notoriously high space complexity of the Karp-Miller procedure. We have compared against this tool in Sect. VI.

Recent research has established that, for the rich class of Boolean broadcast programs, forward search tends to be efficient but incomplete (or unsound), while backward search guarantees correctness but lags behind. The solution is to combine both searches, which we have done here in a somewhat shallow fashion. A goal for future work is therefore to implement our on-the-fly strategy directly on an advanced WQOS coverability algorithm such as [9].

REFERENCES

[1] A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, "Counterexample-guided abstraction refinement for symmetric concurrent programs," *Formal Methods in System Design*, 2012.
[2] P. A. Abdulla, "Well (and better) quasi-ordered transition systems," *Bulletin of Symbolic Logic*, 2010.
[3] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, 1969.
[4] J. Esparza, A. Finkel, and R. Mayr, "On the verification of broadcast protocols," in *LICS*, 1999.
[5] B. Cook, D. Kroening, and N. Sharygina, "Symbolic model checking for asynchronous Boolean programs," in *SPIN*, 2005.
[6] P. Schnoebelen, "Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets," in *MFCS*, 2010.
[7] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS*, 1996.
[8] A. Emerson and T. Wahl, "Efficient reduction techniques for systems with many components," *Electr. Notes Theor. Comput. Sci.*, 2005.
[9] A. Kaiser, D. Kroening, and T. Wahl, "Efficient coverability analysis by proof minimization," in *CONCUR*, 2012.
[10] G. Basler, A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, "SATABS: A bit-precise verifier for C programs," in *TACAS*, 2012.
[11] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.
[12] G. Basler, M. Hague, D. Kroening, L. Ong, T. Wahl, and H. Zhao, "Boom: Taking Boolean program model checking one step further," in *TACAS*, 2010.
[13] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Context-aware counter abstraction," *Formal Methods in System Design*, 2010.
[14] T. Ball, S. Chaki, and S. K. Rajamani, "Parameterized verification of multithreaded software libraries," in *TACAS*, 2001.
[15] G. Delzanno, J.-F. Raskin, and L. V. Begin, "Towards the automated verification of multithreaded Java programs," in *TACAS*, 2002.

Kuai: A Model Checker for Software-defined Networks

Rupak Majumdar
MPI-SWS
Germany

Sai Deep Tetali
UC Los Angeles
USA

Zilong Wang
MPI-SWS
Germany

***Abstract*—In software-defined networking (SDN), a software controller manages a distributed collection of switches by installing and uninstalling packet-forwarding rules in the switches. SDNs allow flexible implementations for expressive and sophisticated network management policies.**

We consider the problem of verifying that an SDN satisfies a given safety property. We describe Kuai, a distributed enumerative model checker for SDNs. Kuai takes as input a controller implementation written in Murphi, a description of the network topology (switches and connections), and a safety property, and performs a distributed enumerative reachability analysis on a cluster of machines. Kuai uses a set of partial order reduction techniques specific to the SDN domain that help reduce the state space dramatically. In addition, Kuai performs an automatic abstraction to handle unboundedly many packets traversing the network at a given time and unboundedly many control messages between the controller and the switches.

We demonstrate the scalability and coverage of Kuai on standard SDN benchmarks. We show that our set of partial order reduction techniques significantly reduces the state spaces of these benchmarks by many orders of magnitude. In addition, Kuai exploits large-scale distribution to quickly search the reduced state space.

I. Introduction

Software-defined networking (SDN) is a novel networking architecture in which a centralized software controller dynamically updates the packet processing policies in network switches based on observing the flow of packets in the network [9], [5]. SDNs have been used to implement sophisticated packet processing policies in networks, and there is increasing industrial adoption [12], [9].

We consider the problem of verifying that an SDN satisfies a network-wide safety property. Since the controller code in an SDN can dynamically change how packets flow in the network, a bug in the controller code can lead to hard-to-analyze network errors at run time. We describe the design of Kuai, a distributed enumerative model checker for SDNs. The input to Kuai is a model of an SDN consisting of two parts. The first part is the controller, written in a simplified guarded-command language similar to Murphi. The second part is the description of a network, consisting of a fixed finite set of switches, a fixed set of client nodes, and the topology of the network (i.e., the connections between the ports of the clients and the switches). Given a safety property of the network, Kuai explores the state space of the SDN to check if the property holds on all executions.

Figure 1 shows a simple SDN. It consists of two switches sw_1 and sw_2 connected to two clients c_1 and c_2. Each client has a port and each switch has two ports to send and receive packets, and the figure shows how the ports are connected to each other. Each connection between ports represents a bi-directional communication channel that may reorder packets.

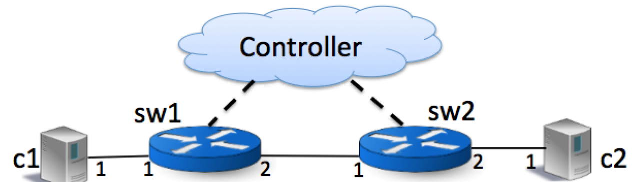


Fig. 1: SSH Example

```
1  def pktIn(pkt)
2   (sw,pt) = pkt.loc
3   if pkt.prot = SSH:
4    drop(pkt)
5   else:
6    dest = 2 if pt = 1 else 1
7    fwd(pkt, [|dest|], sw)
8   rule r1 = (5,{prot=SSH},[||])
9   rule r2 = (1,{port=1},[|2|])
10  rule r3 = (1,{port=2},[|1|])
11  message cm1 = add(r1)
12  message cm2 = add(r2)
13  message cm3 = add(r3)
14  for sw in [sw1, sw2]:
15   send_message(cm1, sw)
16   send_message(cm2, sw)
17   send_message(cm3, sw)
```

Listing 1: Controller for SSH

Moreover, the switches are connected to a controller through dedicated links. Packets are routed in the network using *flow tables* in switches. A flow table is a collection of prioritized forwarding *rules*. A rule consists of a priority, a pattern on packet headers, and a list of ports. A switch processes an incoming packet based on its flow table. It looks at the highest priority rule whose pattern matches the packet and forwards the packet to the list of ports specified in the rule, and drops the packet if the list of ports in the rule is empty. In case no rule matches a packet, the switch forwards the packet to the controller using a request queue and waits for a reply from the controller on a forward queue. The controller replies with a list of ports to which the packet should be forwarded, and optionally sends *control messages* to the control queue of one or more switches to update their flow tables. A control message can add or delete a rule in a switch.

By specifying the rules to be added or deleted, a controller can dynamically control the behaviors of all switches in an SDN network. For example, suppose we want to implement the policy that all SSH packets are dropped. The controller can update the switches with a rule that states that no SSH packets are forwarded, and another that states all non-SSH packets are forwarded. List 1 shows a possible controller that implements this policy. Essentially, the controller drops SSH packets, and adds three rules on the switches: $r1$ to drop SSH packets, $r2$ to forward packets from port 1 to port 2, and $r3$ to forward packets from port 2 to port 1. Since dropping

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

SSH packets (rule $r1$) has higher priority, it will match SSH packets, and rules $r2$ and $r3$ will only match (and forward) non-SSH packets. The controller has a subtle bug. It turns out that a switch can implement rules in arbitrary order. Thus, the switches may end up adding rules $r2$ and $r3$ before adding $r1$, thus violating the policy. Our model checker confirms the bug A possible fix in this case is to implement a *barrier* after line 15, to ensure that rule $r1$ is added before the other rules. Our model checker confirms the policy holds in the fixed version.

The verification of SDNs is challenging due to several reasons. First, even when the topology is fixed with a finite set of clients and switches, the state space is still unbounded, as clients may generate unboundedly many packets and these packets could be simultaneously progressing through the network. For example, client c_1 may send a packet to sw_1 at any point, and an unbounded number of packets can be in the network before sw_1 processes them. Similarly, there may be an unbounded number of control messages (i.e., messages sent from the controller to a switch) between the controller and the switches. While there may be a physical limit on the number of packets and control messages imposed by packet buffers in the switches, the sizes of these buffers can be large (of the order of megabytes) and precise modeling of buffers will blow up the state space.

Second, the packets may be processed in arbitrary interleaved orders, and the processing of one packet may influence the processing of subsequent ones because the controller may update flow tables based on the first packet. Similarly, control messages between the controller and the switches may be processed in arbitrary order and this may lead to potential bugs, including the bug pointed to above.

Kuai handles these challenges in the following way. First, instead of modeling unbounded multisets for packet queues, we implement a *counter abstraction* where we track, for each possible packet, whether zero or arbitrarily many instances of the packet are waiting in a multiset. This abstraction enables us to apply finite-state model checking approaches.

Second, we implement a set of partial-order reduction techniques that are specific to the SDN domain. For example, we note that while in principle a switch only processes one packet at a time, we do not lose behaviors by processing all packets at the packet queue of a switch atomically. Similarly, using the semantics of the barrier message [12], we show that a switch can atomically execute all control messages up to the last barrier in its control queue. Specifically, this optimization enables the model checker to bound the size of control queues. Additionally, we show that whenever there is a packet in a client's packet queue, the client can receive and process it immediately, so that sends from switches can be atomically processed with receives at clients. Finally, we show that we can eagerly serve requests to the controller, that is, we do not lose behaviors if we restrict the controller's request queue to size one and service these requests as soon as they appear.

We empirically demonstrate that our set of partial order reduction techniques significantly reduces the state spaces of SDN benchmarks, often by many orders of magnitude. For the simple SSH example, the number of explored states is approximately 2 million without partial order reductions, but only 13 with reductions!

To handle large state spaces, our model checker Kuai distributes the model checking over a number of nodes in a cluster, using the PReach distributed model checker [2] (based on Murphi [4]) as its back end. The large-scale distribution enables Kuai to model check large state spaces quickly.

Related Work. There is a lot of systems and networking interest in SDNs [9], [5] and standards such as Openflow [12]. From the formal methods perspective, research has focused on verified programming language frameworks for writing SDN controllers [6], [8]. Here, verification refers to correct compilation from Frenetic to executable code, or to checking composability of programs, not the correctness of invariants.

Previous model checking attempts for SDNs mostly focused either on proving a static snapshot of the network [10] or on model checking or symbolic simulation techniques for a fixed number of packets [3], [14]. Recent work extended to controller updates and arbitrary number of packets [17], but used a manual process to add non-interference lemmas. In contrast, our technique automatically deals with unboundedly many packets and, thanks to the partial-order techniques, scales to much larger configurations than reported in [17]. Program verification for SDN controllers using loop invariants and SMT solving has been proposed recently [1]. While the invariants can quantify over the network (and therefore not limited to finite topologies), the model of the network ignores asynchronous interleavings of packet and control message processing that we handle here.

Our work builds on top of distributed enumerative model checking and the PReach tool [2]. Our contribution is identifying domain specific state space reduction heuristics that enable us to explore large configurations.

II. Software-defined Networks

Preliminaries. A *multiset* m over a set Σ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over Σ. We shall write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for an empty multiset, mapping each $\sigma \in \Sigma$ to 0. We write $\{\}$ for an empty set. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

Given a set of states, a *(guarded) action* α is a pair (g, c) where g is a *guard* that evaluates the states to a boolean and c is a *command*. A action α is *enabled* in a state s if the guard of α evaluates s to true. If α is enabled in s, the command of α can execute and lead to a new state s', denoted by $s \xrightarrow{\alpha} s'$. We write $\alpha(s) = s'$ if $s \xrightarrow{\alpha} s'$. A *transition system* TS is a tuple $(S, A, \to, s_0, AP, L)$ where S is a set of states, A is a set of actions, $\to \subseteq S \times A \times S$ is a transition relation, $s_0 \in S$ is the initial state, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. We write $\to^*$ for the reflexive transitive closure of $\to$. A state s' is *reachable* from s if $s \to^* s'$. We write $s \to^+ s'$ if there is a state t such that $s \to t \to^* s'$. For a state s, let $A(s)$ be the set of actions enabled in s; we assume $A(s) \neq \emptyset$ for each $s \in S$. The *trace* of an infinite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ is defined as $\mathit{trace}(\rho) = L(s)L(s_1)\ldots$. The trace of a finite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ is defined as $\mathit{trace}(\rho) = L(s)L(s_1)\ldots L(s_n)$. An execution is *initial* if it starts in s_0. Let $\mathit{Traces}(TS)$ be the set of traces of initial executions in TS. We define invariants and invariant satisfaction in the usual way.

Syntax of Software-defined Networks We model an SDN as a network consisting of *nodes*, *connections*, and a *controller*

program. Nodes come from a finite set *Clients* of *clients* and a (disjoint) finite set *Switches* of *switches*. Each node n has a finite set of *ports* $Port(n) \subseteq \mathbb{N}$ which are connected to ports of other nodes. A *location* (n, pt) is a pair of a node and a port $pt \in Port(n)$. Let Loc be the set of locations. A connection is a pair of locations. A network is well-formed if there is a bijective function $\lambda : Loc \rightarrow Loc$, called the *topology function*, such that $\{((n, pt), \lambda(n, pt)) \mid (n, pt) \in Loc\}$ is the set of connections and no two clients are connected directly.

We model a *packet* pkt in the network as a tuple $(a_1, \ldots, a_k, loc)$, where $(a_1, \ldots, a_k) \in \{0, 1\}^k$ models an abstraction of the packet data and $loc \in Loc$ indicates the location of pkt. Let $Packet$ be the set of all packets.

Each switch contains a set of rules that determine how packets are forwarded. A *rule* is a tuple $(priority, pattern, ports)$, where $priority \in \mathbb{N}$ determines the priority of the rule, $pattern$ is a proposition over $Packet$, and $ports$ is a multiset of ports. We write $Rule$ to denote the set of all rules. Intuitively, a packet matches a rule if it satisfies $pattern$. A switch forwards a packet along $ports$ for the highest priority rule that matches.

Rules are added or deleted on a switch by the controller through a set of *control messages* $CM = \{\mathsf{add}(r), \mathsf{del}(r) \mid r \in Rule\}$. Additionally, the controller uses a *barrier* message b to synchronize.

```
type client {
 Port : set of nat
 pq : multiset of packets
}
rule "send(c, pkt)"
 true ==> send(c, pkt)
end
rule "recv(c,pkt,pkts)"
 exist(pkt:c.pq, true) ==> recv(c,pkt,pkts)
end
```

Listing 2: Client

A client $c \in Clients$ is modeled as in List 2. It consists of a finite set $Port$ of ports and a *packet queue* $pq \in \mathbb{M}[Packet]$ containing a multiset of packets which have arrived at the client. We use (guarded) actions to model behaviors of clients. An action is written as "**rule** *name guard* $\Longrightarrow$ *command* **end**." Predicate $exist(i : X, \varphi)$ asserts that there is an element i in the set (or multiset) X such that the predicate φ holds. Additionally, if $exist(i : X, \varphi)$ holds, then the variable i is bound to an element of X that satisfies φ and can be used later in the command part. In each step, a client c can (1) send a non-deterministically chosen packet pkt along some ports (rule send), or (2) receive a packet pkt from its packet queue and (optionally) send a multiset of packets $pkts$ on some ports (rule recv).

A switch sw is modeled as in List 3. It consists of a set of ports, a *flow table* $ft \subseteq Rule$, a packet queue pq containing packets arriving from neighboring nodes, a *control queue* cq containing control messages or barriers from the controller, a *forward queue* fq consisting of at most one pair $(pkt, ports)$ through which the controller tells the switch to forward packet pkt along the ports $ports$, and a boolean variable $wait$. Predicate $noBarrier(sw)$ asserts $sw.cq$ does not contain a barrier. Predicate $bestmatch(sw, r, pkt)$ asserts that r is the highest priority rule whose pattern matches the packet pkt in switch sw's flow table.

Intuitively, a switch has a normal mode and a waiting mode determined by the $wait$ variable. When the switch is in the normal mode, as long as there is no barrier in its control queue, it can either attempt to forward a packet from its packet queue

```
type switch {
 Port : set of nat
 ft : set of rules
 pq : multiset of packets
 cq : list of barriers and
      multisets of control messages
 fq : set of forward messages
 wait : boolean
}
rule "match(sw,pkt,r)"
 !sw.wait & noBarrier(sw) &
 exist(pkt:sw.pq,
   exist(r:sw.ft, bestmatch(sw,r,pkt))) ==>
 match(sw,pkt,r)
end
rule "nomatch(sw,pkt)"
 !sw.wait & noBarrier(sw) & !RqFull(controller) &
 exist(pkt:sw.pq,
   !exist(r:sw.ft,bestmatch(sw,r,pkt))) ==>
 nomatch(sw,pkt)
end
rule "add(sw,r)"
 !sw.wait & noBarrier(sw) &
 exist(add(r):sw.cq[0],true) ==>
 add(sw,r)
end
rule "delete(sw,r)"
 !sw.wait & noBarrier(sw) &
 exist(del(r):sw.cq[0],true) ==>
 delete(sw,r)
end
rule "fwd(sw,pkt,pts)"
 sw.wait & noBarrier(sw) &
 exist((pkt,pts):fq, true) ==>
 fwd(sw,pkt,pts)
end
rule "barrier(sw)"
 !noBarrier(sw) ==>
 barrier(sw)
end
```

Listing 3: Switch

based on its flow table, or update its flow table according to a control message in its control queue. When the switch cannot find a matching rule in its flow table for a packet, it can initiate a request to the controller, change to the waiting mode, and wait for a forward message from the controller telling it how to forward the packet. Once it receives a forward message (pkt, pts) and there is no barrier in the control queue, it forwards the pending packet pkt to the ports in pts, and changes back to the normal mode. If the control queue contains one or more barriers, the switch dequeues all control messages up to the first barrier from its control queue and updates its flow table.

```
type controller {
 CS : set of control states
 cs0 : CS                      cs : CS
 rq : set of packets           κ : ℕ+
 pktIn : function
}
rule "ctrl(pkt,cs)"
 exist(pkt:controller.rq, true) ==>
 ctrl(pkt,controller.cs)
end
```

Listing 4: Controller

A controller $controller$ is modeled as in List 4. It is a tuple $(CS, cs_0, cs, rq, \kappa, pktIn)$ where CS is a finite set of *control states*, $cs_0 \in CS$ is the *initial control state*, cs is the *current control state*, rq is a finite *request queue* of size $\kappa \geq 1$ consisting of packets forwarded to the controller from switches, and $pktIn$ is a function that takes a packet pkt and a control state cs_1, and returns a tuple $(\eta, (pkt, pts), cs_2)$ where η is a function from $Switches$ to $(\mathbb{M}[CM] \cup \{b\})^*$, (pkt, pts)

is a forward message, and cs_2 is a control state. Intuitively, in each step, the controller removes a packet pkt from rq and executes $pktIn(pkt, controller.cs)$. Based on the result $(\eta, (pkt, pts), cs')$, it sends back to the source of the packet the forward message (pkt, pts) that specifies pkt should be forwarded along pts, and goes to a new control state cs'. Further, for each switch sw in the network it appends $\eta(sw)$ to sw's control queue.

Semantics of Software-defined Networks The semantics of an SDN is given as a transition system. Let $\mathcal{N} = (Clients, Switches, \lambda, Packet, Rule, controller)$ be an SDN, where each component is as defined above.

A *state* s of the SDN $\mathcal{N}$ is a quadruple (π, δ, cs, rq), where π is a function mapping each client $c \in Clients$ to its packet queue pq and δ is a function mapping each switch $sw \in Switches$ to a tuple $(pq, cq, fq, ft, wait)$ consisting of its packet queue, control queue, forward queue, flow table, and the wait variable.

For a non-empty list $l = [x_1, x_2, \ldots, x_n]$, define $l.hd{=}x_1$, $l.tl{=}[x_2, \ldots, x_n]$, and $l[i]$ as the i-th element in l. Given two lists l_1 and l_2, let $l_1@l_2$ be the concatenation of l_1 and l_2. For two non-empty lists $l_1 = [x_1, \ldots, x_m]$ and $l_2 = [y_1, \ldots, y_n]$ in $(\mathbb{M}[CM] \cup \{b\})^*$, define $l_1 + l_2$ be the list $[x_1, \ldots, x_{m-1}, x_m \oplus y_1, y_2, \ldots, y_n]$ if $x_m \neq b$ and $y_1 \neq b$; $l_1@l_2$ otherwise.

Given a flow table ft and a list $l \in (\mathbb{M}[CM] \cup b)^*$, let $update(ft, l)$ be a procedure that updates ft based on l as follows. It dequeues the head of l and sets l to $l.tl$. If the head is a barrier b, then ignore it. If the head is a multiset m, it nondeterministically chooses a *fetching order* p and based on p, removes a control message cm with $m(cm) > 0$ from m. If cm is $\mathsf{add}(r)$, then add the rule r to ft, or if cm is $\mathsf{del}(r)$, then delete r from ft. It keeps updating ft based on p until m becomes empty. It repeats the above instructions on l until l becomes empty. Then it returns the resulting flow table ft.

For a function $f\colon X \to Y$, $x \in X$, and $y \in Y$, let $f[x \mapsto y]$ denote the function that maps x to y and all $x' \neq x$ to $f(x')$. Let $f[x_1 \mapsto y_1; x_2 \mapsto y_2; \ldots; x_n \mapsto y_n]$ denote the function $f[x_1 \mapsto y_1][x_2 \mapsto y_2] \ldots [x_n \mapsto y_n]$. Given a subset $X' = \{x_1, \ldots, x_n\} \subseteq X$, let $f[\text{foreach } x_i \in X' : x_i \mapsto y_i]$ be the function $f[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$ where $1 \leq i \leq n$. Given a tuple $t = (f_1, \ldots, f_n)$, let $t.f_i$ be the field f_i, for $1 \leq i \leq n$. By abuse of notation, we write $t[f_i \mapsto v]$ to be the tuple such that $t[f_i \mapsto v].f_i = v$ and for any $j \neq i$, $t[f_i \mapsto v].f_j = t.f_j$.

We define the following *basic operations* over δ and π:

1) Add or delete packets in switches or in clients. Given a set $X \subseteq Switches \times Packet^{\mathbb{N}}$, define $addPkt(\delta, X) = \delta[\text{foreach } (sw, pkt^k) \in X, sw \mapsto \delta(sw)[pq \mapsto \delta(sw).pq \oplus [\![pkt^k]\!]]]$. Given a set $Y \subseteq Clients \times Packet^{\mathbb{N}}$, define $addPkt(\pi, Y) = \pi[\text{foreach } (c, pkt^k) \in Y, c \mapsto \pi(c) \oplus [\![pkt^k]\!]]$. We define $delPkt(\delta, X)$ and $delPkt(\pi, Y)$ analogously by replacing $\oplus$ with $\ominus$ above.
2) Set the $wait$ bit of a switch sw to true or false. Define $setWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto true]]$ and $unsetWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto false]]$.
3) Add or delete a rule r in the flow table of a switch sw. Define $addRule(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{add}(r)]\!]]; sw \mapsto \delta(sw)[ft \mapsto \delta(sw).ft \cup \{r\}]]$. Define $delRule(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{del}(r)]\!]]; sw \mapsto \delta(sw)[ft \mapsto \delta(sw).ft \backslash \{r\}]]$.
4) Add or delete a forward message msg in a switch sw. Define $addFwdMsg(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[fq \mapsto \delta(sw).fq \cup \{msg\}]]$ and $delFwdMsg(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[fq \mapsto \delta(sw).fq \backslash \{msg\}]]$.
5) Flush and run all control messages up to the first barrier in a switch. Define $flush(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l; ft \mapsto update(\delta(sw).ft, [m, b])]]$ where $l = [\emptyset]$, if $\delta(sw).cq = [m, b]$; $l = l'$, if $\delta(sw).cq = [m, b]@l'$ and l' is not an empty list.
6) Flush and run all control messages up to the last barrier in a switch. Define $flushall(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l_1; ft \mapsto update(\delta(sw).ft, l_2)]]$ where $l_1 = [\emptyset]$ and $l_2 = \delta(sw).cq$ if the last element of $\delta(sw).cq$ is a barrier. Otherwise, let $\delta(sw).cq = l@[m]$. Then $l_1 = [m]$ and $l_2 = l$.
7) Add control messages and barriers to the control queues of the switches. Given a total function $f : Switches \to (\mathbb{M}[CM] \cup \{b\})^*$, define $addCtrlCmd(\delta, f) = \delta[\text{foreach } sw \in Switches : sw \mapsto \delta(sw)[cq \mapsto \delta(sw).cq + f(sw)]]$.

For a switch sw, a packet pkt, and a multiset of ports pts, let $FwdToC(sw, pkt, pts)$ be a set $\{(c, pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (c, pt') \wedge c \in Clients \wedge pkt' = pkt[loc \mapsto (c, pt')]\}$ and $FwdToSw(sw, pkt, pts)$ be a set $\{(sw', pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (sw', pt') \wedge sw' \in Switches \wedge pkt' = pkt[loc \mapsto (sw', pt')]\}$. Intuitively, when sw is about to forward pkt on its ports pts, these two sets summarize how many packets should be forwarded to its connected clients and switches.

For an SDN $\mathcal{N}$, let $Send = \{send(c, pkt) \mid c \in Clients \wedge pkt \in Packet\}$ be the set of *send actions*. We define analogously the set of *receive actions* $Recv$, the set of *match actions* $Match$, the set of *no-match actions* $NoMatch$, the set of *add actions* Add, the set of *delete actions* Del, the set of *forward actions* $Forward$, the set of *barrier actions* $Barrier$, and the set of *control actions* $Ctrl$.

Let $\pi_0 = \lambda c \in Clients.\emptyset$ and $\delta_0 = \lambda sw \in Switches.(\emptyset, [\emptyset], \{\}, \{\}, false)$. The semantics of an SDN $\mathcal{N}$ is given by a transition system $TS(\mathcal{N}) = (S, A, \to, s_0, AP, L)$. Here, S is the set of states, $s_0 = (\pi_0, \delta_0, cs_0, \{\})$ is the initial state, and $A = Send \cup Recv \cup Match \cup NoMatch \cup Add \cup Del \cup Forward \cup Barrier \cup Ctrl$. The transition relation $s \xrightarrow{\alpha} s'$ is defined as follows.

1) $\alpha = send(c, pkt)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addPkt(\delta, \{(sw, pkt)\})$ and $sw = pkt.loc.n$.
2) $\alpha = recv(c, pkt, pkts)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = delPkt(\pi, \{(c, pkt)\})$, $\delta' = addPkt(\delta, X)$ and $X = \{(sw, pkt'^k) \mid pkts(pkt') = k \wedge pkt'.loc.n = sw\}$.
3) $\alpha = match(sw, pkt, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = addPkt(\pi, FwdToC(sw, pkt, r.ports))$ and $\delta' = addPkt(\delta, FwdToSw(sw, pkt, r.ports))$.
4) $\alpha = nomatch(sw, pkt)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq')$ where $rq' = rq \cup \{pkt\}$, $\delta'' = delPkt(\delta, \{(sw, pkt)\})$, and $\delta' = setWait(\delta'', sw)$.
5) $\alpha = add(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addRule(\delta, sw, r)$.
6) $\alpha = del(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = delRule(\delta, sw, r)$.
7) $\alpha = fwd(sw, pkt, pts)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = addPkt(\pi, FwdToC(sw, pkt, pts))$, $\delta_1 = delFwdMsg(\delta, sw, (pkt, pts))$, $\delta_2 = addPkt(\delta_1, FwdToSw(sw, pkt, pts))$, and $\delta' = unsetWait(\delta_2, sw)$.
8) $\alpha = barrier(sw)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = flush(\delta, sw)$.

9) $\alpha = ctrl(pkt, cs)$. Let $pktIn(pkt, cs) = (\eta, msg, cs')$ and $sw = pkt.loc.n$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{pkt\}$, $\delta'' = addFwdMsg(\delta, sw, msg)$, and $\delta' = addCtrlCmd(\delta'', \eta)$.

An atomic proposition $p \in AP$ is an assertion over packet fields or over control states. Define an SDN specification as a safety property $\Box\phi$ where ϕ is a formula over AP and $\Box$ is the "globally" operator of linear-temporal logic. The *model checking problem for an SDN* asks, given an SDN $\mathcal{N}$ and an SDN specification $\Box\phi$, if $TS(\mathcal{N})$ satisfies $\Box\phi$. For example, blocking SSH packets can be specified as $\Box \bigwedge_{pkt \in Packet}(pkt.loc.n \in Clients \wedge pkt.src \in Clients \wedge pkt.loc.n \neq pkt.src \Rightarrow pkt.prot \neq \text{SSH})$.

III. Optimizations

We now describe partial-order reduction and abstraction techniques that reduce the state space. These techniques use the structure of SDNs and, as we demonstrate empirically, are crucial in making the model checking scale to non-trivial examples. We state the correctness theorems; the proofs are in the technical report [11].

Partial Order Reduction Let $TS = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system, i.e., $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ implies $s' = s''$. Given two actions $\alpha, \beta \in A$ with $\alpha \neq \beta$, α and β are *independent* if for any $s \in S$ with $\alpha, \beta \in A(s)$, $\beta \in A(\alpha(s))$, $\alpha \in A(\beta(s)))$, and $\alpha(\beta(s)) = \beta(\alpha(s))$. The actions α and β are dependent if α and β are not independent. An action $\alpha \in A$ is a *stutter action* if for each transition $s \xrightarrow{\alpha} s'$ in TS, we have $L(s) = L(s')$.

For $i \in \{1, 2\}$, let $TS_i = (S_i, A_i, \rightarrow_i, s_0^i, AP, L_i)$ be transition systems. Infinite executions ρ_1 of TS_1 and ρ_2 of TS_2 are *stutter-equivalent*, denoted $\rho_1 \triangleq \rho_2$, if there is an infinite sequence $A_0 A_1 A_2 \ldots$ with $A_i \subseteq AP$, and natural numbers $n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \ldots \geq 1$ such that

$$trace(\rho_1) = \underbrace{A_0 \ldots A_0}_{n_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{n_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{n_2 \text{ times}} \ldots$$

$$trace(\rho_2) = \underbrace{A_0 \ldots A_0}_{m_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{m_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{m_2 \text{ times}} \ldots$$

TS_1 and TS_2 are *stutter equivalent*, denoted $TS_1 \triangleq TS_2$, if $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$, where $\trianglelefteq$ is defined by: $TS_1 \trianglelefteq TS_2$ iff for all $\rho_1 \in Traces(TS_1)$. $\exists \rho_2 \in Traces(TS_2)$. $\rho_1 \triangleq \rho_2$.

A. Barrier Optimization

Intuitively, barrier optimization uses the observation that for any state, we can always flush out control queues of switches until there are no barriers in them. This implies that after a control action is executed, one can immediately update flow tables of switches whose control queue has barriers added by the controller. Hence a control action and successive barrier actions can be merged. We prove its correctness by viewing it as an instance of partial order reduction.

For an SDN $\mathcal{N}$, note that $TS(\mathcal{N})$ is not action-deterministic due to barrier actions. With different fetching orders, $barrier(sw)$ may lead to multiple states. Define $b(s, sw)$ as the number of transitions of the form $s \xrightarrow{barrier(sw)} s'$. Note that a barrier action from any s leads to at most $2^{|Rule|}$ states. Hence for each transition $s \xrightarrow{barrier(sw)} s_i$ where $1 \leq i \leq b(s, sw)$, we can append the action with the index i, i.e., $s \xrightarrow{barrier(sw)_i} s_i$. In the following, we redefine the set $Barrier = \{barrier(sw)_i \mid sw \in Switches \wedge 1 \leq i \leq 2^{|Rule|}\}$, and assume that $TS(\mathcal{N})$ is action-deterministic by renaming barrier actions.

A switch sw has a barrier iff there is a barrier in sw's control queue. A state s has a barrier, denoted $hasb(s)$, iff some switch $sw \in Switches$ has a barrier in s. Define the *ample set* for every state s in $TS(\mathcal{N})$ as follows: if s has a barrier, then $ample(s) = \{barrier(sw)_i \mid 1 \leq i \leq b(s, sw) \wedge sw$ has a barrier in $s\}$, that is, all barrier actions enabled in s. If s does not have a barrier, then $ample(s) = A(s)$.

Given $TS(\mathcal{N})$, we now define a transition system $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$ where $\hat{S} = S$ is the set of states, and the transition relation $\Rightarrow$ is defined as: if $s \xrightarrow{\alpha} s'$ and $\alpha \in ample(s)$, then $s \overset{\alpha}{\Rightarrow} s'$.

Theorem 1: Let $TS(\mathcal{N})$ be an action-deterministic transition system. $TS(\mathcal{N}) \triangleq \widehat{TS}$.

Intuitively, Theorem 1 holds because any barrier action is independent of other actions and is a stutter action. Hence for an infinite execution $s \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{barrier(sw)} t$ in $TS(\mathcal{N})$ where s has a barrier and α_i is not a barrier action for all $1 \leq i \leq n$, we can permute $barrier(sw)$ forward until s and obtain a stutter-equivalent execution in $\widehat{TS}$.

Since Theorem 1 holds, we can merge a control action and successive barrier actions into a single transition $s \xrightarrow{ctrl(pkt,cs)}_2 s'$ where we define the new semantics of $ctrl(pkt, cs)$ under the transition relation $\rightarrow_2$. Formally, Let $(\eta, (pkt, pts), cs') = pktIn(pkt, cs)$ and $sw = pkt.loc.n$.

Ctrl. $(\pi, \delta, cs, rq) \xrightarrow{ctrl(pkt,cs)}_2 (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{pkt\}$. Define $\delta'' = addFwdMsg(\delta, sw, (pkt, pts))$, and $\delta''' = addCtrlCmd(\delta'', \eta)$. Let $\{sw_1, \ldots, sw_n\}$ be the set of all switches whose control queue has barriers in δ'''. Let $\delta_0 = \delta'''$ and $\delta_i = \mathit{flushall}(\delta_{i-1}, sw_i)$ for all $1 \leq i \leq n$. Define $\delta' = \delta_n$.

Given $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$, define a transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$ where $S_2 \subseteq \hat{S}$ is a set of states reachable by $\rightarrow_2$, A_2 is $A \backslash Barrier$, $AP_2 = AP$, $L_2 = L$, and $\rightarrow_2$ is defined inductively as

$$\frac{s_0 \overset{\alpha}{\Rightarrow} s'}{s_0 \xrightarrow{\alpha}_2 s'} \qquad \frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} s' \wedge \alpha \notin Ctrl}{s \xrightarrow{\alpha}_2 s'}$$

$$\frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} t \Rightarrow^* s' \wedge \alpha \in Ctrl \wedge \neg hasb(s')}{s \xrightarrow{\alpha}_2 s'}$$

Since we only remove barrier actions which are stutter actions, we have $TS_2 \triangleq \widehat{TS} \triangleq TS(\mathcal{N})$. Hence we have the following theorem:

Theorem 2: Given an SDN $\mathcal{N}$ and a safety property $\Box\phi$, $TS(\mathcal{N})$ satisfies $\Box\phi$ iff TS_2 satisfies $\Box\phi$.

B. Client Optimization

Given transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$, we further reduce the state space by observing that any receive action of a client is a stutter action and is independent of other actions. Formally, we define $ample(s)$ for each state $s \in S_2$ as follows: if there is a client in s such that its packet queue is not empty, then $ample(s) = \{recv(c, pkt, pkts) \mid pkt$ is in $c.pq$ at $s\}$, that is, all receive actions enabled in s. Otherwise, $ample(s) = A(s)$. We now define a transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$ where $S_3 = S_2$,

$A_3 = A_2$, $AP_3 = AP_2$, $L_3 = L_2$, and where the transition relation $\rightarrow_3$ is defined as: if $s \xrightarrow{\alpha}_2 s'$ and $\alpha \in ample(s)$, then $s \xrightarrow{\alpha}_3 s'$.

Theorem 3: (1) $TS_2 \triangleq TS_3$. (2) Given a safety property $\Box\phi$, TS_2 satisfies $\Box\phi$ iff TS_3 satisfies $\Box\phi$.

C. $(0, \infty)$ Abstraction

The $(0, \infty)$ abstraction bounds the size of packet queues and the multiset in each control queue. The idea is as follows. One can regard a multiset as a counter that counts the number of elements in it exactly. Instead, $(0, \infty)$ abstraction abstracts a multiset so that for each element e, it either does not contain e (i.e. 0) or contains unboundedly many copies of e (i.e. ∞). Then the size of an abstracted multiset is bounded. Note that for any state s in TS_3, any switch's control queue contains exactly one multiset. Hence, the abstraction bounds the length of control queues.

Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be the extension of the natural numbers with infinity. We naturally extend the addition operation by assuming that $\infty + \infty = \infty$ and $\infty + c = \infty$ for all $c \in \mathbb{Z}$. Given a multiset $m \in \mathbb{M}[D]$ for some finite set D, define an *extended multiset* $\mathsf{over}(m)$ such that for each element $d \in D$, $\mathsf{over}(m)(d) = 0$ if $m(d) = 0$, and $\mathsf{over}(m)(d) = \infty$ otherwise. Define $\mathbb{M}[D]^\infty$ as the set of all extended multisets and multisets over D. Given a control queue cq with length n, let $\mathsf{over}(cq)$ be such that for $1 \le i \le n$, $\mathsf{over}(cq)[i] = \mathsf{over}(cq[i])$ if $cq[i] \neq b$; $\mathsf{over}(cq)[i] = b$ otherwise. For $m_1, m_2 \in \mathbb{M}[D]^\infty$, we write $m_1 \le_e m_2$ iff for all $d \in D$, $m_1(d) \le m_2(d)$ or $m_2(d) = \infty$. Given two control queues cq, cq' of same length n, define $cq \le_e cq'$ iff for each $1 \le i \le n$, $(cq[i] = b \leftrightarrow cq'[i] = b) \wedge (cq[i] \neq b \rightarrow cq[i] \le_e cq'[i])$.

Given an SDN and the transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$, Define a transition system $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$ where $S_4 = \{\mathsf{over}(s) \mid s \in S_3\}$, $A_4 = A_3$, $AP_4 = AP_3$, and $L_4 = L_3$. The definition of $\rightarrow_4$ is given in detail in [11]. We provide the intuition of $\rightarrow_4$ here: $\rightarrow_4$ is defined so that (1) whenever a packet pkt is added $k \ge 1$ times into a packet queue pq, we set pq to $\mathsf{over}(pq \oplus [\![pkt^k]\!])$, and (2) whenever $\eta(sw)$ is added into switch sw's control queue cq, we set cq to $\mathsf{over}(cq + \eta(sw))$. The following lemma claims that TS_4 simulates TS_3, which leads to Theorem 4.

Lemma 1: For any infinite initial execution $s_0 \xrightarrow{\beta_1}_3 s_1 \xrightarrow{\beta_2}_3 s_2 \ldots$ in TS_3, there is an infinite initial execution $t_0 \xrightarrow{\beta_1}_4 t_1 \xrightarrow{\beta_2}_4 t_2 \ldots$ in TS_4 such that for all $i \ge 0$, $s_i = (\pi_i, \delta_i, cs_i, rq_i)$ and $t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i)$ satisfy the following condition: for all $c \in Clients$, $\pi_i(c) \le_e \pi'_i(c)$ and for all $sw \in Switches$, $\delta_i(sw).pq \le_e \delta'_i(sw).pq$, $\delta_i(sw).cq \le_e \delta'_i(sw).cq$, $\delta_i(sw).fq = \delta'_i(sw).fq$, $\delta_i(sw).ft = \delta'_i(sw).ft$, and $\delta_i(sw).wait = \delta'_i(sw).wait$, and $cs_i = cs'_i$, and $rq_i = rq'_i$.

Theorem 4: Given a safety property $\Box\phi$, if TS_4 satisfies $\Box\phi$ then TS_3 satisfies $\Box\phi$.

D. All Packets in One Shot Abstraction

So far, a switch processes a single packet at a time. We can further reduce the reachable state space by forcing a switch to process all packets matched by some rule at a time. The intermediate states produced by successive match actions in a switch are removed. Let $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$. Define a transition system $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$ where $S_5 = S_4$, $AP_5 = AP_4$, $L_5 = L_4$, A_5 is the union of the new "multiple" match actions and A_4 excluding the old "single" match actions, and $\rightarrow_5$ is defined as:

$$\frac{s \xrightarrow{\alpha}_4 s' \wedge \alpha \text{ is not a match action}}{s \xrightarrow{\alpha}_5 s'}$$

and if $pkt_lst = [pkt_1, \ldots, pkt_n]$ and $r_lst = [r_1, \ldots, r_n]$

$$\frac{s \xrightarrow{match(sw,pkt_1,r_1)}_4 s_1 \ldots s_{n-1} \xrightarrow{match(sw,pkt_n,r_n)}_4 s'}{s \xrightarrow{match(sw,pkt_lst,r_lst)}_5 s'}$$

We prove TS_5 simulates TS_4. We define a relation $R \subseteq S_4 \times S_5$ such that $((\pi, \delta, cs, rq), (\pi', \delta', cs', rq')) \in R$ iff for all $pkt \in Packet$, for all $c \in Clients$, $\pi(c)(pkt) = \infty \rightarrow \pi'(c)(pkt) = \infty$ and for all $sw \in Switches$, $\delta(sw).pq(pkt) = \infty \rightarrow \delta'(sw).pq(pkt) = \infty$, $\delta(sw).cq = \delta'(sw).cq$, $\delta(sw).fq = \delta'(sw).fq$, $\delta(sw).ft = \delta'(sw).ft$, and $\delta(sw).wait = \delta'(sw).wait$, and $cs = cs'$, and $rq = rq'$.

Theorem 5: (1)The relation R is a simulation relation. (2)For a safety property $\Box\phi$, if TS_5 satisfies $\Box\phi$, then TS_4 satisfies $\Box\phi$.

E. Controller Optimization

We consider a restricted class of SDNs in which the size κ of the controller's request queue is one. Under this restriction, we can define a new transition system TS_6 that is stutter equivalent to TS_5 and has fewer reachable states. The idea is to observe that a no-match action is a stutter action and is independent of any actions before a corresponding control action is executed. Formally, given $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$, we define a new transition relation $\rightarrow_6$ inductively:

$$\frac{s_0 \xrightarrow{\alpha}_5 s'}{s_0 \xrightarrow{\alpha}_6 s'} \qquad \frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{nomatch(sw,pkt)}_5 s_2 \xrightarrow{ctrl(pkt,cs)}_5 s'}{s_1 \xrightarrow{nomatch_ctrl(sw,pkt,cs)}_6 s'}$$

$$\frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{\alpha}_5 s' \wedge \alpha \text{ is not a no-match action}}{s_1 \xrightarrow{\alpha}_6 s'}$$

where a new action $nomatch_ctrl(sw, pkt, cs)$ merges $nomatch(sw, pkt)$ and $ctrl(pkt, cs)$ actions. We define a transition system $TS_6 = (S_6, A_6, \rightarrow_6, s_0, AP_6, L_6)$, where $S_6 = S_5$ is the set of states, A_6 is the union of all $nomatch_ctrl(sw, pkt, cs)$ actions and $A_5 \backslash (NoMatch \cup Ctrl)$, $AP_6 = AP_5$, and $L_6 = L_5$.

Theorem 6: Given an SDN $\mathcal{N}$ where the size of the request queue of the controller is one, and a safety property $\Box\phi$. (1) $TS_5 \triangleq TS_6$. (2) TS_5 satisfies $\Box\phi$ iff TS_6 satisfies $\Box\phi$.

IV. Implementation and Evaluation

Kuai[1] is implemented on top of PReach [2], a distributed enumerative model checker built on Murphi. We model switches, clients, and the controller as concurrent Murphi processes which communicate using message passing, with the queues modeled as multisets. We manually abstract IP packets using predicates used in the controller. We implement $(0, \infty)$-counter abstraction as a library on top of Murphi multisets.

Kuai takes as input topology information such as the number of switches, clients, and their connections, (manually) abstracted packets, and the controller code written as a Murphi process, and invariants written in Murphi syntax. We found it fairly straightforward to port POX [15] controllers due to the imperative features of Murphi. Murphi allows arbitrary first order logic formulas as invariants and it is easy to specify

[1]The tool is can be downloaded at https://github.com/t-saideep/kuai

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Program	Bytes/ state	w/o optimizations		w/ optimizations	
		States	Time	States	Time
SSH 2×2	304	2,283,527	23.52s	13	6.40s
ML 3×3	320	9,109,456	89.99s	5308	6.39s
ML 6×3	748			23,926,202	604.07s
ML 9×2	1276			18,615,767	793.84s
FW(S) 1×2	332	2,110,986	26.89s	3645	5.45s
FW(M) 2×4	448			45,507	8.03s
FW(M) 3×4	560			512,439	55.06s
FW(M) 4×4	676			5,360,871	475.54s
RS 4×4	764			4998	6.60s
RS 4×5	764			590,570	82.82s
RS 4×6	764			5,112,013	327.39s
SIM 5×6	632			167	6.23s
SIM 5×8	632			167	6.34s
SIM 5×12	1108			167	6.85s

TABLE I: Experimental results. Omitted entries indicate that model checking did not terminate. The number X×Y in the Program column means that there are X switches and Y clients in the example.

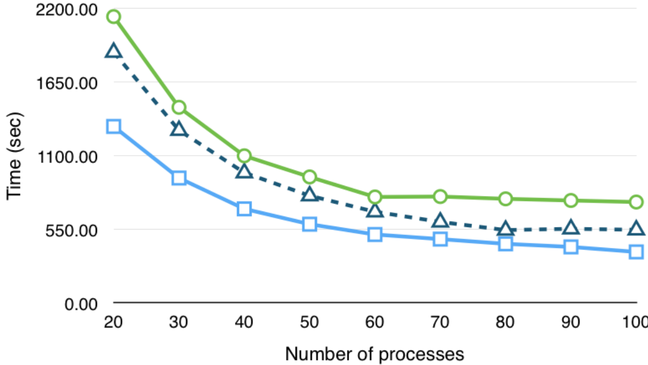


Fig. 2: Verification time vs processes ○ ML 9×2 △ ML 6×3 □ FW(M) 4×4

safety properties. Kuai compiles them into a single Murphi file and the model checking effort is then distributed across several machines using PReach. Finally the output of the tool is an error trace if the program invariant fails, or *success* otherwise.

We have evaluated Kuai on a number of real world OpenFlow benchmarks. The experiments were performed on a cluster of 5 Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments had access to a total of 150 cores and had access to 4TB of RAM.

Table I shows a summary of experimental results and compares against model checking without the optimizations from Section III. Empty rows indicate model checking did not terminate in 1 hour or ran out of memory. Figure 2 shows the scalability of model checking with increasing distribution on the three largest examples. We noticed that the performance of the distributed model checker plateaued around 70 Erlang processes on these and other large examples. Thus, times (in table I) are provided for configurations that use 70 Erlang processes. As we introduced abstractions, it is possible that we get false positives. We verified the existence of all bugs reported by Kuai manually and there were no false positives.

Besides the table, we plot the MAC learning example in Figure 3, which shows how significantly our optimization techniques reduce the state space. Though we still suffer from the state-space explosion problem, our optimizations delay it and enable us to verify SDNs with much larger configurations.

We now describe the benchmarks in detail.

SSH We run Kuai on the SSH controller from Listing 1. It finds the control message reordering bug in 0.1 seconds. By adding a barrier after line 15, Kuai proves the correctness in 6.4 seconds by exploring 13 states. In contrast, the unoptimized version explores over 2 million states.

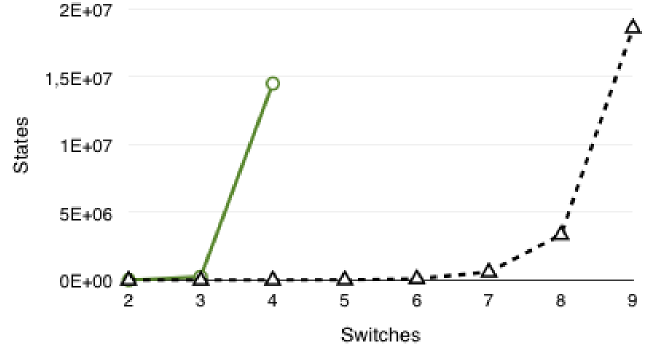


Fig. 3: State space of MAC learning controller: △: optimized, ○ unoptimized

MAC Learning Controller (ML) This is based on the POX [15] implementation of the standard ethernet discovery protocol. We checked there are no forwarding loops (similar to [17]), i.e., a packet should not reach a switch more than once. Packets are augmented with a bit for each switch which gets set when the switch processes that packet. The invariant is specified using these visit-bits (called *reached*): $\square\ \forall sw \in Switches.\ \forall pkt \in sw.pq.\ (\neg pkt.reached(sw))$.

A cycle in the topology will lead to forwarding loops as the controller does not compute the minimum spanning tree. We discover the bug in a cyclic topology of 3 switches 3 clients in 0.47 seconds. We re-ran the example on a topology containing the minimum spanning tree of the original cyclic topology and the tool is able to prove that there were no forwarding loops in 6.39 seconds. We scale the example by adding more switches. We notice that while the verification on topology with 9 switches and 2 clients has fewer states than the one with 6 switches and 3 clients, each state in the latter case is bigger than the former and hence the memory and communication overheads are higher.

Single Switch Firewall (FW(S)) This is based on an advanced GENI assignment [7] on building an OpenFlow based firewall. The controller takes as input a simple configuration file which is a list of tuples of the form (client1, port1, client2, port2). This specifies that packets originating from client1 on port1 can be forwarded to client2 on port2. We abbreviate the tuples as $(client1\colon port1 \rightarrow client2\colon port2)$. Any flow not explicitly allowed is forbidden. The flows are uni-directional and the above flow will reject traffic initiated by client2 on port2 towards client1 on port1. However, once client1 initiates a flow, the firewall should allow client2 to reply back, making the flow bi-directional until client1 closes the connection.

The naive implementation of the controller is as follows: on receiving a packet $(c1\colon p1 \rightarrow c2\colon p2)$, check if there is a tuple matching the flow in the policy. If it does, add rules $(c1\colon p1 \rightarrow c2\colon p2)$ and $(c2\colon p2 \rightarrow c1\colon p1)$ and forward the packet to $c2$. Otherwise add a rule to drop packets of the form $(c1\colon p1 \rightarrow c2\colon p2)$. The invariant to verify here is to ensure the policy of the firewall, i.e., a packet from $c1\colon p1$ should be forwarded to $c2\colon p2$ if and only if $(c1\colon p2 \rightarrow c2\colon p2)$ exists in the firewall policy or if $(c2\colon p2 \rightarrow c1\colon p1)$ exists in the policy and $c2$ has already initiated the corresponding flow. The following formula specifies that allowed packets should not be dropped: $\square \forall p \in Packet.\ on_dropped(p) \Rightarrow \neg flows[p.src][packet.src_port][packet.dest][p.dest_port]$, where *on_dropped(p)* is set if a packet-drop transition is fired on packet *p* (and reset at the beginning of every transition). *flows* is an auxiliary variable in the controller which keeps

track of allowed flows based on the firewall policy and initiating client.

We ran the experiment on a topology with 2 clients and a firewall. We found an interesting bug in our implementation which is caused by not assigning proper priorities to rules. For example, when $(c1\colon p1 \rightarrow c2\colon p2)$ is present in the policy but not $(c2\colon p2 \rightarrow c1\colon p1)$, the rule to drop flows should have a lower priority than the rules to allow flows. Otherwise, the following bug would occur. If $c2$ initiates the flow $(c2\colon p2 \rightarrow c1\colon p1)$ then the controller adds a rule to drop packets matching that flow. Later on, if $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ and the controller adds the corresponding rules to allow the flow on both directions, the switch now has two conflicting rules of the same priority. One to allow and the other to drop $(c2\colon p2 \rightarrow c1\colon p1)$. The switch may non-deterministically choose to drop the packet. Once we fixed the bug, the tool could prove the invariant in 5.45 seconds.

Multiple Switch Firewalls (FW(M)) We extend the above example to include multiple replicated firewalls for load balancing. We now allow the clients to send packets to all of these firewalls. We augment the implementation of the single switch controller to add the same rules on all firewalls. However, this implementation no longer ensures the invariant in the multi-switch setting.

Consider the case with two firewalls, $f1$ and $f2$. The tool reports the following bug: $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ on firewall $f1$. The controller adds the corresponding rules to allow flows in both directions to $f1$ and $f2$ but only sends a barrier to $f1$. Now $f2$ delays the installation of $(c2\colon p2 \rightarrow c1\colon p1)$ and $c2$ replies back to $c1$ through $f2$ which forwards the packet to the controller. The controller then drops the packet.

The fix here is to add the rules along with barriers on all switches and not just the switch from which the packet originates. With this fix the tool is able to prove the property in 8 seconds. In order to test the scalability, we tested the tool on increasing number of firewalls in the topology.

Resonance (RS) Resonance [13] is a system for ensuring security in large networks using OpenFlow switches. When a new client enters the network, it is assigned *registration* state and is only allowed to communicate with a web portal. The portal either authenticates a client by sending a signal to the controller (and the controller assigns the client an *authenticated* state), or sets the client to *quarantined* state. In the authenticated state, the client is only allowed to communicate with a scanner. The scanner ensures that the client is not infected and sends a signal to the controller and lets the controller assign it an *operational* state. If an infection is detected, it is assigned a *quarantined* state. The clients in operational state are periodically scanned and moved to the quarantined state if they are infected. Quarantined clients cannot communicate with other clients.

In our model, the web portal non-deterministically chooses to authenticate or quarantine a client and the scanner non-deterministically marks a client operational or quarantined. We check the invariant that packets from quarantined clients should not be forwarded: $\Box \forall p \in Packet.\ on_forward(p) \Rightarrow (state(p.src) \neq Quarantined)$. Similar to *on_drop*, *on_forward* is set when packet-forward transition is fired and reset before the beginning of every transition. The controller follows the Resonance algorithm [13].

We ran the experiment on a topology of two clients, one portal, one scanner and four switches. The topology is the same as in Figure 2 of [13] without DHCP and DNS clients. Kuai proves the invariant in 6.6 seconds. We scale up the example by increasing the number of clients.

Simple (SIM) Simple [16] is a policy enforcement layer built on top of OpenFlow to ensure efficient middlebox traffic steering. In many network settings, traffic is routed through several middleboxes, such as firewalls, loggers, proxies, etc., before reaching the final destination. Simple takes a middlebox policy as input and translates this to forwarding rules to ensure the policy holds. The invariant ensures that all source packets to a client will be received and forwarded by the middleboxes specified in a given policy before the packet reaches its destination.

We ran the experiment on a topology of two clients, two firewalls, one IDS, one proxy and five switches (see Figure 1 of [16]). Kuai can prove the invariant in 6.48 seconds.

We scale up the example by fixing the destination client and increasing the number of source clients that can send packets to it. Because of our "all packets in one shot" optimization (section III-D), no matter how many packets get queued initially, they are all forwarded in lock-step as the controller forwarding rule applies to all incoming packets.

References

[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. PLDI '14, pages 282–293, 2014.

[2] B. Bingham, J. Bingham, F. de Paula, J. Erickson, G. Singh, and M. Reitblatt. Industrial strength distributed explicit state model checking. In *PDMC-HIBI*, pages 28–36, Sept 2010.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. NSDI'12, pages 127–140, 2012.

[4] D. L. Dill. The Murphi verification system. CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[5] N. Feamster, J. Rexford, and E. Zegura. The road to SDN. *Queue*, 11(12):20:20–20:40, Dec. 2013.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[7] GENI Assignment. http://groups.geni.net/geni/wiki/GENIEducation/SampleAssignments/OpenFlowFirewallAssignment/ExerciseLayout/Execute.

[8] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. PLDI '13, pages 483–494, New York,USA, 2013. ACM.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. SIGCOMM13, pages 3–14, New York, NY, USA, 2013.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.

[11] R. Majumdar, S. Tetali, and Z. Wang. Kuai: A Model Checker for Software-defined Networks. Technical report. http://www.mpi-sws.org/~zilong/papers/kuai_tr.pdf.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM*, 38(2):69–74, Mar. 2008.

[13] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control for enterprise networks. WREN '09, pages 11–18, New York, NY, USA, 2009. ACM.

[14] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. A balance of power: Expressive, analyzable controller programming. HotSDN '13, pages 79–84, New York, NY, USA, 2013. ACM.

[15] POX. http://www.noxrepo.org/pox/about-pox/.

[16] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. SIGCOMM13, pages 27–38, New York, NY, USA, 2013. ACM.

[17] D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148, Oct 2013.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

ILP Modulo Data

Panagiotis Manolios, Vasilis Papavasileiou, and Mirek Riedewald
Northeastern University
{pete,vpap,mirek}@ccs.neu.edu

***Abstract*—The vast quantity of data generated and captured every day has led to a pressing need for tools and processes to organize, analyze and interrelate this data. Automated reasoning and optimization tools with inherent support for data could enable advancements in a variety of contexts, from data-backed decision making to data-intensive scientific research. To this end, we introduce a decidable logic aimed at database analysis. Our logic extends quantifier-free Linear Integer Arithmetic with operators from Relational Algebra, like selection and cross product. We provide a scalable decision procedure that is based on the BC(T) architecture for ILP Modulo Theories. Our decision procedure makes use of database techniques. We also experimentally evaluate our approach, and discuss potential applications.**

I. Introduction

In 2010, enterprises and users stored more than 13 exabytes of new data [1]. Database Management Systems (DBMS's) based on the Relational Model [3] are a key component in the computing infrastructure of virtually any organization. With big data playing a determining role in business and science, we are motivated to rethink data management and analysis.

Database systems capable of symbolic computation could enable powerful new methodologies for strategic planning, decision making, and scientific research. We propose database systems that (a) store symbolic (in addition to concrete) data, and at the same time (b) allow queries of a symbolic nature, *e.g.*, with free variables. Such database systems can be dually thought of as constraint solvers that reason in the presence of data. Symbolic data allows us to encode partially specified or entirely speculative information, *e.g.*, database entries that exist for the purpose of what-if analysis. Symbolic queries enable deductive reasoning about data.

Existing relational query languages (*e.g.*, SQL) only allow concrete data and queries. Symbolic enhancements require a formalism that combines constraints and relational queries. We address this need by introducing the Δ logic. Δ extends quantifier-free Linear Integer Arithmetic (QFLIA) with database tables and operators from Relational Algebra, like selection (σ), union ($\cup$), and cross product ($\times$). While Δ is decidable, the logic in its general form gives rise to hard satisfiability problems, primarily because it allows universal quantification over cross products of big tables. We study unrestricted Δ (for it is a natural umbrella formalism), but also provide restrictions that enable an efficient decision procedure. In other words, we identify a class of database problems that are a realistic initial target for formal analysis.

This research was supported in part by DARPA under AFRL Cooperative Agreement No. FA8750-10-2-0233 and by NSF grants CCF-1117184 and CCF-1319580.

We provide a scalable procedure based on the BC(T) architecture for ILP Modulo Theories (IMT) [10]. Our approach is dubbed *ILP Modulo Data*, because an ILP solver co-exists with a procedure that establishes a correspondence between integer variables and database tables. The latter contain a mix of concrete and symbolic data. ILP Modulo Data allows us to use a powerful ILP solver based on branch-and-cut (B&C) on the arithmetic side, while also utilizing database techniques that allow us to scale to realistic datasets.

The compositional nature of ILP Modulo Data is well-suited for potential applications. Organizations have access to vast amounts of data, but at the same time rely heavily on Mathematical Programming technology. We enhance Mathematical Programming tools with the ability to directly access data, thus assisting data-backed decision making. Such tools would also benefit scientists in fields ranging from ornithology [17] to astronomy [5], by providing immediate feedback on the consistency between models the scientists devise and datasets of observations they collect. Our paper outlines potential applications, while our experimental evaluation relies on benchmarks that characterize them. We experimentally demonstrate that our ILP Modulo Data framework provides better performance than the approach of eagerly reducing Δ to QFLIA.

Paper Structure: Section II introduces our reasoning paradigm through a motivating example. Section III presents the Δ logic, while Section IV identifies a Δ fragment that yields scalable procedures. Section V describes our decision procedure. We experimentally evaluate our approach in Section VI. We provide an overview of related work in Section VII, and conclude with Section VIII.

II. Motivating Example

Our motivating example (formalized in Figure 1) concerns the problem of optimally investing a given amount of capital. This is an appropriate application for our techniques, because (a) investments are almost always data-driven as they take historical stock prices into account, and (b) financial institutions already rely on Mathematical Programming.

The problem involves investing in a *portfolio* of n publicly traded stocks, with the goal of maximizing profit while following guidelines that minimize risk. A database provides information on these stocks, including stock prices from the New York Stock Exchange (NYSE). We would like to pick the n stocks that would have yielded the highest profit over a period of time in the recent past, *e.g.*, over the preceding year. This optimization problem is subject to risk-mitigation constraints that require us to pick companies from a variety of sectors. While investing in the exact solver-generated portfolio

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Id	Cap	Sector
1 (EMC)	large	tech
2 (FII)	medium	financials
3 (AKR)	small	retail
...	...	...

(a) `stocks`

Id	Diff
1	128
2	117
3	89
...	...

(b) `quotes`

maximize
$\Sigma_{1 \le i \le n} a_i \cdot d_i$
subject to
$(x_i, c_i, s_i) \in \texttt{stocks}$, $\quad 1 \le i \le n$
$(x_i, d_i) \in \texttt{quotes}$, $\quad 1 \le i \le n$
$x_i \neq x_j$, $\quad 1 \le i < j \le n$
$\Sigma_{\{i \ \mid \ 1 \le i \le n, s_i = s\}} a_i \le \Sigma_{1 \le i \le n} a_i / 3$, $\quad$ for every sector s
$\Sigma_{\{i \ \mid \ 1 \le i \le n, c_i = \mathsf{small}\}} a_i \le \Sigma_{1 \le i \le n} a_i / 4$

(c) Constraints

Fig. 1. Portfolio Management with ILP Modulo Data

(which relies only on past performance) is not necessarily a good strategy, such a portfolio provides useful information for the analysts who make the final investment decisions.

The data is given in tables `stocks` and `quotes` (Figures 1a and 1b). Each company in `stocks` is described by a unique ID (with the associated NYSE symbol parenthesized), its capitalization (small, medium, or large), and its sector (*e.g.*, tech, retail, financials, automotive, energy, emerging-markets). While Figure 1 uses human-readable names, we can encode these fields with bounded integer quantities. Each entry in `quotes` describes the observed movement of a certain stock in a given timeframe, assuming that dividends were reinvested. For example, the first row describes an increase of 28% in the price of EMC. `quotes` is an application-specific abstraction, *i.e.*, the actual database contains past stock prices and `quotes` is a *view* produced by comparing data for two time periods.

The ith stock in the portfolio is characterized by a unique ID x_i that corresponds to entries in the dataset, *i.e.*, there exist entries $(x_i, c_i, s_i) \in \texttt{stocks}$ and $(x_i, d_i) \in \texttt{quotes}$. To minimize risk, we force the n IDs x_i to be distinct, and allow no single sector to account for more than a third of the total capital. Additionally, no more than a fourth of the capital goes to smallcap companies. The objective function maximizes the capital at the end of the period, and thus the profit.

Note that if the amounts a_i are variables, the objective function is non-linear. The problem can be circumvented by providing integer constants for a_i, *i.e.*, by specifying how the capital will be partitioned. With constants for a_i, the non-table constraints are essentially in QFLIA. (The summations for i that satisfy conditions like $s_i = s$ and $c_i = \mathsf{small}$ are easy to encode as sums of if-then-else terms.) Conversely, the problem is essentially satisfiability of an arithmetic instance, where certain variables correspond to database contents. This is the kind of problem that we propose new techniques for. We cannot use a standalone DBMS, since DBMS's do not handle constraints and optimization. Neither are existing solvers up to the task, since they do not provide ways of managing data.

The constraints we have described are meant to be representative. Clearly investors also have to consider other options, including investing in index funds, bonds, debt securities and derivative contracts. These financial instruments may have other characteristics that need to be modeled. Our constraints are also based on simplifying assumptions, *e.g.*, that we can invest an arbitrary amount in any given stock at any time. It is not within the scope of our paper to model investment problems comprehensively. What matters is that these additional concerns also mix arithmetic with data, thus reinforcing the need for data-aware solving.

III. The Logic Δ

$$F ::= T_1 \le T_2 \mid \exists\, D \mid \neg F \mid F_1 \vee F_2$$

$$D ::= \{T^+\} \mid \langle \sigma\, x : F : D \rangle \mid D_1 \times D_2 \mid D_1 \cup D_2$$

$$T ::= (T_1, T_2) \mid \mathsf{left}(T) \mid \mathsf{right}(T) \mid x \mid K \mid K \cdot T \mid T_1 + T_2$$

$$K ::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$$

Fig. 2. Grammar of Δ

This Section introduces the logic Δ. Δ combines arithmetic with queries over tabular data. Δ thus encompasses database problems like our motivating example of Section II.

The grammar of Δ is given in Figure 2. K, T, D, and F are the non-terminal symbols for integer constants, terms, tables, and formulas, respectively. The first line of productions for T corresponds to pairs and their accessors; the second line is for variable symbols (x) and integer expressions. A table (non-terminal symbol D) is either an *input table*, a *selection*, a *cross-product*, or a *union*. The selection $\langle \sigma\, x : F : D \rangle$ is a table that consists of only those entries in D that satisfy F, *i.e.*, the variable x ranges over the table entries; σ binds x in F, but not in D. For formulas (non-terminal symbol F), $\exists\, D$ should be read as "D is not empty". All other constructs bear the obvious meaning. We assume that all variables not bound by σ are integer. We will freely use derived operators, *e.g.*, conjunction and integer equality.

Δ is typed. Each term is either of type `int` or of type $s * t$, where s and t are types. left and right are only permissible when applied to a term of type $s * t$ for some type s and some type t; if x is of type $s * t$, then $\mathsf{left}(x)$ is of type s and $\mathsf{right}(x)$ is of type t. The integer constants are of type `int`. The arithmetic operators ($+$, $\cdot$, and $\le$) only apply to terms of type `int`; $+$ and $\cdot$ produce integers. Each table has a *schema*, which is the type of its entries. (Schemas are the table-level counterpart of types.) An input table is comprised of entries of the same type. If table D_1 has schema s_1 and table D_2 has schema s_2, then $D_1 \times D_2$ has schema $s_1 * s_2$. For $\langle \sigma\, x : F : D \rangle$ to be properly typed, F should be a properly-typed formula under the assumption that the type of x is the schema of D; the schema of $\langle \sigma\, x : F : D \rangle$ is the same as the schema of D. Union expects tables of the same schema and preserves it.

Clearly, Δ is at least as powerful as QFLIA. At the same time, Δ encompasses most features one would expect from a relational query language. We have left out certain operators usually present in query languages. First, note that projection

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

(π) would not provide additional power, since it is possible to refer to any subset of the columns, without producing an intermediate table that leaves out the irrelevant ones. Also, the set difference $A \setminus B$ can be encoded as $\langle \sigma\, a : \neg \exists \langle \sigma\, b : a = b : B \rangle : A \rangle$, assuming that the schema of A and B has exactly one column; otherwise, in place of $a = b$ we would have a conjunction of equalities over all columns. Additionally, Δ can express many forms of aggregation, including count (when compared to a constant), min, and max.

Example 1. *The portfolio encoded by Figure 1 can be represented as the input table*

$$\texttt{portfolio} = \{(1, (x_1, a_1)), \ldots, (n, (x_n, a_n))\}.$$

`portfolio` *contains symbolic data, something which is not allowed by DBMS's. The first column ensures that the n entries are distinct, irrespective of the assignment.* `portfolio` *is of schema* `int` $*$ (`int` $*$ `int`). *Consider the following constraint:*

$$\neg \exists \boxed{\begin{array}{lll} \langle \sigma\, x : & \mathsf{left}(\mathsf{left}(x)) \neq \mathsf{left}(\mathsf{right}(x)) \wedge & \\ & \mathsf{left}(\mathsf{right}(\mathsf{left}(x))) = \mathsf{left}(\mathsf{right}(\mathsf{right}(x))) & \\ : & \boxed{\texttt{portfolio}} \times \boxed{\texttt{portfolio}} & \rangle \end{array}}$$

The constraint states that there are no entries $(i, (x_i, a_i))$ and $(j, (x_j, a_j))$ in `portfolio` *such that $i \neq j$ and $x_i = x_j$,* i.e., `portfolio` *references n distinct stocks (as was our intention in Figure 1). The constraint essentially involves universal quantification over* `portfolio` $\times$ `portfolio`.

A. Decidability

Δ satisfiability can be reduced to QFLIA satisfiability. We explain the reduction briefly. We represent a table expression D of schema s as a set $[\![D]\!]$ consisting of pairs $r \oslash b$, where r is a term of type s and b is a QFLIA formula, with the intended meaning that r is present in the table iff b is true. We use the operator $\oslash$ to distinguish the auxiliary pairs used for the reduction from the ones allowed by the syntax of Δ. For a formula F, $[\![F]\!]$ denotes the corresponding formula in QFLIA; similarly for integer terms. $F[x/r]$ stands for substituting x with r in F, with appropriate care for occurrences of the symbol x bound by σ inside F. We define $[\![\cdot]\!]$ for tables and formulas below as two mutually recursive functions.

$$\begin{aligned} [\![\{r_1, \ldots, r_n\}]\!] &= \{r_1 \oslash \mathsf{true}, \ldots, r_n \oslash \mathsf{true}\} \\ [\![\langle \sigma\, x : F : D \rangle]\!] &= \{r \oslash (b \wedge [\![F[x/r]]\!]) \mid r \oslash b \in [\![D]\!]\} \\ [\![D_1 \times D_2]\!] &= \{(r_1, r_2) \oslash (b_1 \wedge b_2) \mid \\ &\qquad r_1 \oslash b_1 \in [\![D_1]\!], r_2 \oslash b_2 \in [\![D_2]\!]\} \\ [\![D_1 \cup D_2]\!] &= [\![D_1]\!] \cup [\![D_2]\!] \end{aligned} \tag{1}$$

$$\begin{aligned} [\![T_1 \leq T_2]\!] &= [\![T_1]\!] \leq [\![T_2]\!] \\ [\![\exists\, D]\!] &= \bigvee_{r \oslash b \in [\![D]\!]} b \\ [\![\neg F]\!] &= \neg [\![F]\!] \\ [\![F_1 \vee F_2]\!] &= [\![F_1]\!] \vee [\![F_2]\!] \end{aligned} \tag{2}$$

For encoding Δ integer terms as QFLIA terms (*e.g.*, $[\![T_i]\!]$ in Equation 2), all that needs to be done is elimination of pair constructors and accessors via the rules $\mathsf{left}((x, y)) = x$ and $\mathsf{right}((x, y)) = y$. The reduction suffices to establish decidability of Δ. The reduction also provides formal semantics for Δ by specifying its meaning in terms of QFLIA.

B. Complexity

Theorem 1. *The satisfiability problem for Δ is in NEXPTIME.*

Proof Sketch. The reduction to QFLIA (Equations 1 and 2) produces a formula exponentially larger than the input. Since QFLIA is in NP, the reduction provides a non-deterministic exponential time procedure for Δ-satisfiability. □

Theorem 2. *The satisfiability problem for Δ is PSPACE-hard.*

Proof Sketch. We reduce the (PSPACE-complete) QBF problem to Δ satisfiability in polynomial time. We deal with Boolean quantification by quantifying over the input table $\mathcal{B} = \{0, 1\}$. For example, the formula $\forall x \exists y (x \vee \neg y)$ becomes $\neg \exists \boxed{\langle \sigma\, x : \neg \exists \boxed{\langle \sigma\, y : x = 1 \vee y = 0 : \mathcal{B} \rangle} : \mathcal{B} \rangle}$. □

Complexity analysis of Δ beyond Theorems 1 and 2 is not within the scope of this paper, and has mostly theoretical significance. In practice, query size is orders of magnitude smaller than data size. Conversely, it is meaningful to study *data complexity* [19], *i.e.*, complexity where only the amount of data varies. Instead of assuming a query of constant size, we provide a stronger result by limiting the number of tables that can participate in a cross product. (We also limit nested quantifiers, because the latter can simulate cross products.) We define below the rank function that characterizes this number.

$$\begin{aligned} \mathrm{rank}(\{r_1, \ldots, r_n\}) &= 1 \\ \mathrm{rank}(\langle \sigma\, x : F : D \rangle) &= \mathrm{rank}(F) + \mathrm{rank}(D) \\ \mathrm{rank}(D_1 \times D_2) &= \mathrm{rank}(D_1) + \mathrm{rank}(D_2) \\ \mathrm{rank}(D_1 \cup D_2) &= \max(\mathrm{rank}(D_1), \mathrm{rank}(D_2)) \end{aligned} \tag{3}$$

$$\begin{aligned} \mathrm{rank}(T_1 \leq T_2) &= 0 \\ \mathrm{rank}(\exists\, D) &= \mathrm{rank}(D) \\ \mathrm{rank}(\neg F) &= \mathrm{rank}(F) \\ \mathrm{rank}(F_1 \vee F_2) &= \max(\mathrm{rank}(F_1), \mathrm{rank}(F_2)) \end{aligned} \tag{4}$$

Definition 1 (k-Δ)**.** *For any natural number k, k-Δ is the set of formulas $\{F \mid F \in \Delta$ and $\mathrm{rank}(F) \leq k\}$.*

Theorem 3. *For any natural number k, k-Δ is NP-complete.*

Proof Sketch. k-Δ is NP-hard , because any QFLIA formula can be reduced to a $0{-}\Delta$ formula in polynomial time ($0{-}\Delta \subseteq k$-Δ). We obtain membership in NP from the reduction defined by Equation 2, which produces polynomially-sized QFLIA formulas. □

Given the class of formulas k-Δ for some k, the reduction produces QFLIA formulas of size $O(n^{k+1})$, where n is the input size. While the reduction is polynomial (since k is fixed), it may not be practical even for $k = 2$, given that datasets of millions of entries are common. Conversely, we propose restrictions that yield a lazy solving architecture.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

IV. The Existential Fragment of Δ

We proceed to study the *existential fragment* of Δ, which we denote by $\exists\Delta$.

Definition 2 ($\exists\Delta$)**.** *A Δ formula belongs to $\exists\Delta$ if the $\exists$ operator always appears below an even number of negations,* i.e.*, $\exists$ only appears with positive polarity.*

The motivation for studying $\exists\Delta$ is as follows. Universal quantification pushes for an approach similar to quantifier instantiation, *e.g.*, Example 1 (which is not in $\exists\Delta$) inherently requires instantiating a constraint for every element in $\texttt{portfolio} \times \texttt{portfolio}$. This can be done incrementally by applying patterns that are standard in verification tools. In contrast, we are not aware of techniques that would be a good match for the kind of existential quantification that arises in Δ. Therefore, the rest of this paper focuses on $\exists\Delta$.

Formulas in $\exists\Delta$ can be transformed into formulas in a convenient intermediate logic without cross products, selections, or unions. We rephrase $\exists$ in terms of a new membership operator. Each formula of the form $\exists\, D$ is viewed as $x \in D$, where $\in$ has the obvious semantics and x is a properly shaped row comprised of fresh integer variables. We will refer to rows like x that serve as witnesses for $\exists$ as *witness rows*. The next step is to translate membership in arbitrary table expressions to membership in input tables. $(x, y) \in D \times E$ becomes $x \in D \wedge y \in E$, while $x \in D \cup E$ becomes $x \in D \vee x \in E$. Finally, $x \in \langle \sigma\, y : F : D\rangle$ becomes $F[y/x] \wedge x \in D$. We eliminate all cross products, selections, and unions by repeated application of the above transformations.

Example 2. *The tables of Figures 1a and 1b can be easily encoded as Δ input tables of schemas* $\texttt{int} * (\texttt{int} * \texttt{int})$ *and* $\texttt{int} * \texttt{int}$. *Let* small *capitalization be represented by the constant* 0. *Consider the following constraint:*

$$\exists \boxed{\begin{array}{ll} \langle \sigma\, x : & \mathsf{left}(\mathsf{left}(x)) = \mathsf{left}(\mathsf{right}(x)) \wedge \\ & \mathsf{left}(\mathsf{right}(\mathsf{left}(x))) = 0 \wedge \\ & \mathsf{right}(\mathsf{right}(x)) \geq 150 \\ \quad : & \boxed{\texttt{stocks}} \times \boxed{\texttt{quotes}} \qquad \rangle \end{array}}$$

The constraint asserts the existence of some tuple $((x_1, (x_2, x_3)), (x_4, x_5)) \in \texttt{stocks} \times \texttt{quotes}$ *that satisfies* $\Phi = [x_1 = x_4 \wedge x_2 = 0 \wedge x_5 \geq 150]$. *(We have eliminated the accessors* left *and* right*.) This is equivalent to asserting that* $(x_1, (x_2, x_3)) \in \texttt{stocks} \wedge (x_4, x_5) \in \texttt{quotes} \wedge \Phi$.

The procedure we outlined produces a *decomposed* formula consisting of a QFLIA part and *membership constraints*. We proceed to define these notions formally.

Definition 3 ((Conditional) Membership Constraint)**.** *A membership constraint is a constraint of the form*

$$(x_1, \ldots, x_k) \in \{(y_{1,1}, \ldots, y_{1,k}), \ldots, (y_{l,1}, \ldots, y_{l,k})\} \quad (5)$$

for positive integers k and l and variable symbols x_i, $y_{j,i}$. A constraint of the form $b = 1 \Rightarrow m$, where b is a variable symbol and m is a membership constraint, is called a conditional *membership constraint.*

A membership constraint may hold conditionally, either because it arises from an $\exists$-atom that appears under propositional structure (and therefore holds conditionally), or because of a disjunction introduced by the union operator. We use conditions of the form $b = 1$ because ILP necessitates $[0, 1]$-bounded integer variables in place of Boolean variables. Implication in the opposite direction is never needed, since $\exists$ always appears with positive polarity (as per Definition 2).

Membership constraints do not contain arbitrary arithmetic expressions, but only variable symbols. "Variable abstraction" [9] eliminates richer expressions. While variable abstraction allows for compositional reasoning and helps with theoretical analysis, a limited fragment of arithmetic in membership constraints yields more efficient implementation. Part of our discussion will involve tables that contain integer constants and terms of the form $v + c$, where v is a variable symbol and c is an integer constant. (Everything we present is easy to generalize for such terms.) For convenience, we flatten out rows constructed using the pair constructor of Figure 2, and instead deal with k-tuples of integers. This is only a matter of presentation and has no impact on the algorithms.

Definition 4. *A* decomposed *formula is a conjunction $F \wedge M$, where (a) F is a QFLIA formula and (b) M is a conjunction of possibly conditional membership constraints.*

Theorem 4. *$\exists\Delta$ satisfiability is NP-complete.*

Proof. $\exists\Delta$ satisfiability is NP-hard, because $\exists\Delta$ is at least as powerful as QFLIA. $\exists\Delta$ satisfiability is in NP, because we can reduce $\exists\Delta$ to QFLIA in polynomial time. The reduction first produces a formula in decomposed form (Definition 4). Equation 5 is equivalent to $\bigvee_{j=1,\ldots,l} \bigwedge_{i=1,\ldots,k} x_i = y_{j,i}$; therefore, the membership operator can be eliminated. The result is a formula in QFLIA. $\square$

The polynomial size of the reduction relies on the fact that Δ does not allow tables to be named and referenced from multiple places, *i.e.*, table expressions are not DAG-shaped. Despite the polynomial reduction, a lazy scheme remains relevant. The reason is that QFLIA solvers are not meant for long disjunctions that essentially encode database tables.

V. BC(T) for Δ

The decomposed form of Definition 4 is particularly suited for a scheme that combines separate procedures for QFLIA and table membership. Given that the QFLIA part can be encoded as a conjunction of integer linear constraints [10], it becomes possible to solve instances in decomposed form (and by extension $\exists\Delta$ instances) by instantiating the BC(T) framework for IMT [10]. An ILP solver deals with the QFLIA constraints, and exchanges information with a procedure that checks membership in finite sets. Since database queries typically have simple propositional structure, we do not expect encoding the latter with linear constraints to be a bottleneck.

The membership procedure is confronted with a conjunction of membership constraints (Definition 3). Dealing with conditional constraints is essentially a matter of Boolean search. The membership procedure needs to understand equality atoms,

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

equality being a primitive. (Our setting is standard first-order logic with equality.) In particular, the procedure keeps track of truth assignments to the equalities in:

$$\{x_i = y_{j,i} \mid j \in [1,l], i \in [1,k]\} \tag{6}$$

The symbols x_i and $y_{j,i}$ have the same meaning as in Definition 3. In the presence of multiple membership constraints, the union of sets, like in Equation 6, is relevant. Given that membership constraints can be checked in isolation, our discussion proceeds with a single constraint. The variables x_i and $y_{j,i}$ also appear in linear constraints. It simplifies our design to assume that all of them appear in ILP, even if they are unconstrained there. The BC(T) framework provides a mechanism ("difference constraints" [10]) for notifying background procedures about atoms like the ones in Equation 6. Given truth values for these atoms, we check that a membership constraint is satisfied by simply traversing the table and looking for a tuple that is column-wise equal to the witness row. The constraint is violated if for every $j \in [1,l]$, there exists some $i \in [1,k]$ such that $x_i \neq y_{j,i}$, *i.e.*, there is no candidate tuple.

The arithmetic and membership parts share variables. It is vital that we systematically explore the space of (dis)equalities between these variables. This exchange of information resembles the non-deterministic Nelson-Oppen scheme (NO) for combining decision procedures [15]. We demonstrate that NO can accommodate membership constraints.

Definition 5 (Arrangement)**.** *Let E be an equivalence relation over a set of variables V. The set*

$$\alpha(V,E) = \{x = y \mid xEy\} \cup \{x \neq y \mid x,y \in V \text{ and not } xEy\}$$

is the arrangement *of V induced by E.*

Definition 6 (Stably-Infinite Theory)**.** *A Σ-theory T is called stably-infinite if for every T-satisfiable quantifier-free Σ-formula F there exists an interpretation satisfying $F \wedge T$ whose domain is infinite.*

Fact 1 (Nelson-Oppen for Stably-Infinite Theories [15, 9])**.** *Let T_i be a stably-infinite Σ_i-theory, for $i = 1,2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Also, let Γ_i be a conjunction of Σ_i-literals. $\Gamma_1 \cup \Gamma_2$ is $(T_1 \cup T_2)$-satisfiable iff there exists an equivalence relation E of the variables V shared by Γ_1 and Γ_2 such that $\Gamma_i \cup \alpha(V,E)$ is T_i-satisfiable, for $i = 1,2$.*

Lemma 1 (Nelson-Oppen for Membership Constraints)**.** *Let T be a stably-infinite Σ-theory. Also, let Γ be a conjunction of Σ-literals, and M be a conjunction of possibly negated membership constraints. $\Gamma \cup M$ is T-satisfiable iff there exists an equivalence relation E of the variables V shared by Γ and M such that $\Gamma \cup \alpha(V,E)$ is T-satisfiable and $M \cup \alpha(V,E)$ is satisfiable.*

A longer version of this paper [11] provides a proof of Lemma 1. Note that Lemma 1 allows negated membership constraints. While the latter do not pose algorithmic difficulties, our discussion is limited to the positive occurrences needed for $\exists\Delta$. The statement of Lemma 1 is structurally similar to that of Fact 1, with membership constraints replacing the constraints of some participating stably-infinite theory. It follows that a membership procedure can participate in NO as a black box, much like a theory solver, even though we have not formalized membership constraints by means of a theory. We can thus combine a form of set reasoning with any stably-infinite theory.

BC(T) guarantees completeness for the combination of ILP with a stably-infinite theory [10] by ensuring that the branching strategy explores all possible arrangements. We established that membership can be used much like a stably-infinite theory. All that is needed for completeness is a membership procedure capable of checking consistency of its constraints conjoined with a given arrangement (that contains all literals of Equation 6). As we have seen, this operation is simple and involves no arithmetic. In pursuit of efficiency, we proceed to describe branching and propagation techniques based on table contents. Meaningful branching and propagation involve the integer bounds of variables, *i.e.*, necessitate limited arithmetic reasoning on the membership side.

A. Propagation

B&C-based ILP solvers keep track of variable lower and upper bounds, and heavily rely on bounds propagation algorithms. We describe how to enhance such propagation to exploit the structure of membership constraints.

We denote by $\text{lb}(v)$ an $\text{ub}(v)$ the current lower and upper bounds on variable v. $\text{lb}(v)$ (respectively $\text{ub}(v)$) is either an integer constant, or $-\infty$ (resp. $+\infty$) if no bound is known. We use the notation $\text{lb}'(v)$ and $\text{ub}'(v)$ for bounds on v that the membership procedure infers. We proceed with a membership constraint as per Definition 3. Let $x = (x_1, \ldots, x_k)$; similarly, we denote by y_j the tuple $(y_{j,1}, \ldots, y_{j,k})$. Let $\text{match}(x, y_j)$ be true if and only if for all $i \in [1,k]$, the sets $[\text{lb}(x_i), \text{ub}(x_i)]$ and $[\text{lb}(y_{j,i}), \text{ub}(y_{j,i})]$ intersect.

$$\text{lb}'(x_i) = \max(\text{lb}(x_i), \min\{\text{lb}(y_{j,i}) \mid j \in [1,l], \text{match}(x, y_j)\}) \tag{7}$$

$$\text{ub}'(x_i) = \min(\text{ub}(x_i), \max\{\text{ub}(y_{j,i}) \mid j \in [1,l], \text{match}(x, y_j)\}) \tag{8}$$

We over-approximate the values of the variables x_i by considering all candidate entries (inner $\min$ and $\max$). The outer $\max$ and $\min$ guarantee that we do not weaken bounds. If there exists exactly one value j such that $\text{match}(x, y_j)$, it is sound to deduce the equalities $x_i = y_{j,i}$, for all $i \in [1,k]$. If there is no candidate entry, inconsistency is reported.

Example 3 (Interleaved Propagation)**.** *Consider the decomposed formula $x = y \wedge (x,y) \in \{(1,2),(2,4),(3,6),(4,8)\}$. The formula corresponds to a query over concrete tuples that any DBMS can evaluate in linear time. It is thus vital that our techniques yield acceptable performance. Equations 7 and 8 bound x to* $[\min\{1,2,3,4\}, \max\{1,2,3,4\}] = [1,4]$ *and y to* $[\min\{2,4,6,8\}, \max\{2,4,6,8\}] = [2,8]$. *Given the equality $x = y$, ILP propagation deduces that* $x, y \in [2,4]$, *since* $[2,4]$ *is the intersection of permissible ranges for x and y. The membership procedure detects that* match *now only holds for* $(2,4)$, *and fixes x to 2 and y to 4. The ILP solver in turn deduces unsatisfiability, since $x = y$ is violated. No branching was needed. Encoding the formula in QFLIA would hide its structure, leading to search. The example generalizes to other lengths and bounded symbolic data.*

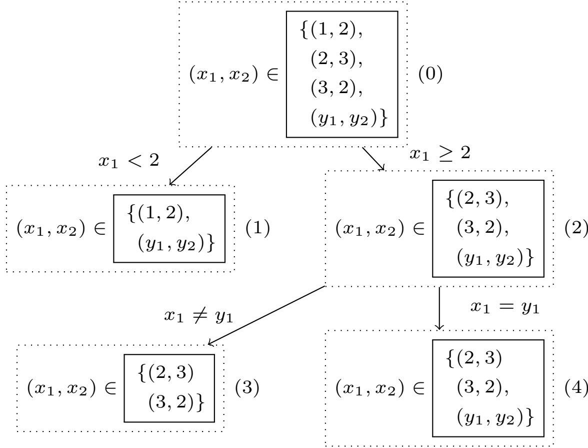


Fig. 3. Data-Driven Branching

B. Branching and Arrangement Search

It follows from Lemma 1 that a branching strategy which exhaustively explores all possible arrangements of the shared variables guarantees completeness. To achieve better performance, we have to branch with the tabular structure of databases in mind, without overlooking symbolic data.

Figure 3 provides an example. The root node (Node 0) describes a single membership constraint, which we assume to be part of a larger decomposed formula. We maintain integer constants in the table, instead of performing variable abstraction which would introduce auxiliary variables for them. According to Equation 6, the membership procedure needs truth assignments for the equalities in $\{x_1 = 1, x_1 = 2, x_1 = 3, x_1 = y_1, x_2 = 2, x_2 = 3, x_2 = y_2\}$. It would not be wise for the search strategy to overlook that this set originates from a table containing numbers, and treat the set members as if they were atomic propositions unrelated to each other.

In our example, branching on the condition $x_1 < 2$ produces two subproblems. Node 1 shows only the tuples that still apply under the condition $x_1 < 2$, *i.e.*, the ones that still satisfy the predicate match; similarly for Node 2. $x_1 < 2$ is a choice informed by the tabular structure. Since 2 as the value of the first column is close to the "middle" of the table, branching on $x_1 < 2$ rules out approximately half of the candidates. (y_1, y_2) is present in both subproblems (Nodes 1 and 2). Branching based on constant bounds is therefore not enough, for we will possibly have to deal with symbolic tuples. Figure 3 demonstrates further branching on $x_1 = y_1$ to determine whether (y_1, y_2) is a suitable witness for the membership constraint.

The example demonstrates the dual nature of the search strategy needed. The problem naturally pushes towards branch-and-bound (which is a restriction of B&C), *e.g.*, branching on $x_1 < 2$ is meaningful. It remains necessary to also branch on equalities between shared variables (*e.g.*, $x_1 = y_1$), just like in any practical implementation of NO. (To be precise, in ILP we would have two separate nodes for $x_1 < y_1$ and $x_1 > y_1$ in place of $x_1 \neq y_1$.) Implementing NO with B&C enables both kinds of branching.

Branching is organically tied to propagation. Initially (Node 0), assuming no previously known bounds for x_1, the table contents only allow us to bound x_1 to the range $[\min(\mathrm{lb}(y_1), 1), \max(\mathrm{ub}(y_1), 3)]$; if y_1 is unbounded, x_1 remains unbounded. The decisions $x_1 \geq 2$ and $x_1 \neq y_1$ (*i.e.*, Node 3) tighten x_1 to $[2, 3]$. We also obtain the range $[2, 3]$ for x_2, *i.e.*, branching on some column potentially leads to propagation across other columns.

C. Discussion

The analysis of this Section indicates that Δ formulas can be decomposed in such a way that a procedure for table lookup assumes part of the workload. BC(T) is particularly suited for implementing such a combination. BC(T) can easily accommodate data-aware propagation (Section V-A) and branching (Section V-B). Our techniques would be harder to implement within a DPLL(T)-style solver [16], given that the toplevel search of DPLL(T) is over the Booleans (and not the integers). A DPLL(T)-based implementation of our techniques would essentially require integrating branch-and-bound in DPLL(T), which is beyond the scope of our work.

The table lookup procedure can be thought of as a small database engine within the solver. The employed database engine can be an actual DBMS, storing the concrete part of tables and possibly bounds on symbolic fields. A DBMS would provide multiple opportunities for improvements. Equations 7 and 8 essentially describe database aggregation, and thus provide a starting point for the kinds of queries that apply. DBMS queries can be over multiple tables at a time, and can involve conditions other than bounds. As a matter of fact, the match predicate of Equations 7 and 8 can be strengthened with any condition on the data that follows from the formula (*e.g.*, $x = y$ in Example 3), thus computing tighter bounds. Different kinds of database optimizations apply, *e.g.*, materializing queries for better incremental behavior and smarter indexing based on user input.

$\exists\Delta$ (and its decomposed form) formally characterizes a relevant class of problems that can be solved by a compositional scheme which employs a database engine. Our scheme may actually apply to a superset of $\exists\Delta$.

VI. Applications and Experiments

We have implemented support for databases on top of the Inez constraint solver.[1] Inez is our implementation of the BC(T) architecture for IMT on top of the SCIP (M)ILP solver [2]. We refer to the version of Inez that provides database extensions as Inez*DB*. Inez*DB* supports existential database constraints by means of the BC(T)-based combination described in Section V, but also universal quantification by eager instantiation. Inez*DB* (like Inez) additionally supports objective functions.

We have produced a collection of Inez*DB* input files that have the structure we expect in applications. Our benchmark suite is publicly available and can be used as a starting point towards a richer benchmark suite of problems that involve data and constraints.[2] We provide a brief overview of the application areas that inspire our benchmarks.

[1]https://github.com/vasilisp/inez

[2]http://www.ccs.neu.edu/home/vpap/fmcad-2014.html

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

A. How-To Analysis

Research in the general direction of reverse data management [12] proposes ways of obtaining the desired results out of a database query. We outline this class of problems through an example, which gives rise to some of our benchmarks.

Example 4 (`emp_join.ml`)**.** *The management of a company is surprised to find out that (according to the corporate database) there is no employee younger than 30 whose yearly income exceeds $60000.* Why not *is not obvious, since income is a complicated function of multiple quantities including a base salary, benefits based on age, employee level (junior, middle, or senior), and bonuses.*

The management consults the database administrator on how to *[13] ameliorate the seeming injustice. Together, they explore bonuses that would allow young employees to exceed the $60000 limit. This amounts to synthesizing tuples for the table of bonuses. An alternative is to adjust various parameters in the income computation,* i.e.*, to modify the query instead of the data [18]. This can be done by replacing constants with variables, and letting the solver come up with suitable values.*

B. Test-Case Generation

Test case generation is relevant for databases [20]. A family of benchmarks in our collection demonstrate test data generation by concretizing tables initially containing symbolic data.

Example 5 (`emp_keys.ml`)**.** *The problem involves two tables, named* `incomes` *and* `employees`*.* `incomes` *has an ID column constrained to reference existing entries in* `employees`*,* i.e.*, there is a foreign key constraint.* `incomes` *contains thousands of tuples with symbolic IDs. A satisfying assignment corresponds to a generated database that meets the foreign key constraint, thus serving as meaningful test input.*

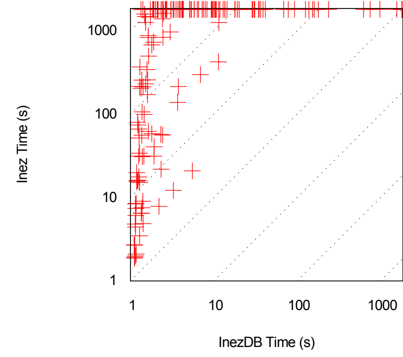
C. Scientific Applications

Studying big datasets is a key aspect of scientific research in fields ranging from ornithology [17] to astronomy [5]. To demonstrate the applicability of our techniques, we provide benchmarks inspired by queries that ornithologists perform.

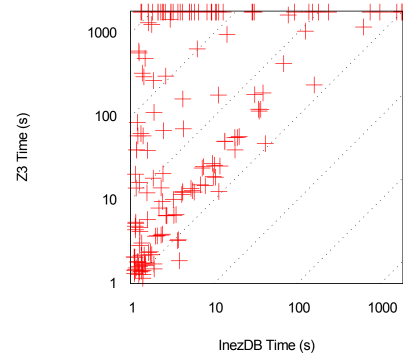
Example 6 (`birds_box.ml`)**.** *An ornithologist wants to see a rare species in person, but has not decided on a good location. She has access to a database of observations. Each observation describes a bird and the geographic coordinates where it was seen. An area can be described as a symbolic rectangle* $B = [\mathtt{longitude}_{\mathtt{min}}, \mathtt{longitude}_{\mathtt{max}}] \times [\mathtt{latitude}_{\mathtt{min}}, \mathtt{latitude}_{\mathtt{max}}]$*. Our techniques allow the ornithologist to simply ask for* n *observations of the species of interest that lie in* B*. The query effectively concretizes* B*.*

D. Portfolio Management

We experimented with the portfolio optimization example of Section II. Our exact instance (`portfolio.ml`) encodes a more complex variant of the formalization in Section II. An additional table contains stock dividends; dividends are taken into account in the objective function. We tried a range of parameters with a timeout of one hour, and obtained a range of solutions. Notably, picking an optimal portfolio of 5 out of 50 stocks took 161 seconds; 5 out of 4000 stocks took 1510 seconds; and 6 out of 2000 stocks took 1172 seconds. Such table sizes are realistic, given that NYSE lists approximately 2800 companies.



(a) Inez*DB* versus Inez



(b) Inez*DB* versus Z3

Fig. 4. Experiments: Inez*DB* versus the eager approach

E. Overview of Results

We compare Inez*DB* against an Inez frontend that solves Δ formulas by eagerly translating them to QFLIA via the encoding of Theorem 4. Inez in turn solves QFLIA formulas by reducing them to constraints that SCIP understands. (These constraints are not strictly ILP, since we utilize specialized constraint handlers [2].) We refer to this configuration simply as Inez, since the only addition to Inez is a new frontend. We also produce SMT-LIB versions of our QFLIA formulas, and run them against the latest available version of Z3 (4.3.1).

We provide 8 benchmark generators that allow different modes of operation (*e.g.*, some of them are able to produce both satisfiable and unsatisfiable benchmarks), and are able to output benchmarks with different table sizes. Our input table sizes range from 60 tuples to 640000 tuples. In total, our parameters give rise to 166 benchmarks. We run all three solvers with a timeout of 1800 seconds and a memory limit of 12GB on a machine that provides 2 Intel Xeon X5677 CPUs of 4 cores each and 96GB of RAM. Figure 4 visualizes our experiments. Inez solves 25 satisfiable and 47 unsatisfiable benchmarks. Inez*DB* solves 74 satisfiable and 81 unsatisfiable benchmarks. Finally, Z3 solves 57 satisfiable

ISBN: 978-0-9835678-4-4.
Copyright owned jointly by the authors and FMCAD Inc.

and 58 unsatisfiable benchmarks. Among the failures for Inez (resp. Z3), 37 (resp. 27) are due to the memory limit. Inez*DB* runs out of memory only once. If we turn off the memory limits, the total numbers of failures don't change much.

Figure 4a indicates that Inez*DB* outperforms Inez by a significant margin. This margin can be attributed to two factors. First, Inez*DB* exploits the structure of database problems (*e.g.*, for branching and propagation), while Inez has no knowledge of this structure. Second, our reduction to QFLIA (in the case of Inez) produces patterns that SCIP is not optimized for, since the latter is designed for MILP and not for QFLIA.

Figure 4b compares Inez against a leading solver for QFLIA (Z3), and thus characterizes the tool's performance in absolute terms. There is a cluster of 40 benchmarks for which Inez*DB* is 2-8 times faster than Z3. (Note that the scale is logarithmic.) Inez*DB* is at least 8 times faster for 31 of the benchmarks that both tools solve, and solves many benchmarks for which Z3 times out. All failures for Inez*DB* are failures for Z3. Z3 outperforms Inez*DB* for only 7 out of the 166 benchmarks, none of which take Inez*DB* more than 4 seconds to solve.

We conclude the evaluation by pointing out that there is significant room for improvement in Inez*DB*. As is the case with almost every first implementation of a new decision procedure, there is room for improvement, *e.g.*, Inez*DB* can benefit from better preprocessing and more sophisticated branching. Inez*DB* can also be improved by adopting database techniques (as we outlined in Section V), or by integrating a DBMS. Our promising experimental results even without such optimizations constitute sufficient evidence that ILP Modulo Data is a viable design for data-enabled reasoning tools.

VII. Related Work

The Constraint Database framework [6] provides a database perspective on constraint solving. The framework encompasses relations described by means of constraints, but not relations comprised of concrete tuples.

"Table constraints" [8, 4], as studied in Constraint Programming, resemble our membership constraints. Such tables are not meant as database tables. Our work differs in significant ways, *e.g.*, our setup allows symbolic table contents. Also, the algorithms presented for table constraints rely on table contents from small domains (*i.e.*, not the reals or the integers). This aligns with the overall emphasis of Constraint Programming, but conflicts with our intended applications.

Veanes et al. describe the Qex technique and tool that uses Z3 to generate tests for SQL queries [20]. Qex essentially encodes the relational operators via axioms, which are later instantiated via E-matching [14]. E-matching is a generic scheme that is not optimized in any way for database problems. Qex is geared towards relatively small tables that suffice as test cases, while our target applications involve bigger tables.

Other approaches tackle constraints arising in database applications with off-the-shelf generic solvers (via eager reductions). Notably, Khalek et al. use Alloy [7], while Meliou and Suciou use MILP [13]. In neither of these approaches does the core of the solver exploit the structure of database instances, *e.g.*, for branching or propagation.

VIII. Conclusions and Future Work

We introduced the ILP Modulo Data framework for marrying data with symbolic reasoning. To that end, we introduced the decidable logic Δ. We identified a fragment of Δ that can be solved efficiently by instantiating the BC(T) architecture. We developed a solver for Δ, and evaluated this solver on a set of benchmarks that we made publicly available.

There are many interesting research directions to be explored in future work, including: (a) the design and implementation of solvers that include an actual DBMS, (b) efficiently handling universal quantification over big tables, say by partitioning input tables and using parallelization, (c) extending our techniques to allow mixed integer, real arithmetic, and other first-order theories, and (d) solving interesting business and scientific applications using the ILP Modulo Data framework.

References

[1] Challenges and Opportunities with Big Data, 2012. Computing Community Consortium White Paper.

[2] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universitat Berlin, 2007.

[3] Edgar Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6):377–387, 1970.

[4] Ian Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data Structures for Generalised Arc Consistency for Extensional Constraints. In *AAAI*, 2007.

[5] Jim Gray, Alex Szalay, Ani Thakar, Peter Kunszt, Christopher Stoughton, Don Slutz, and Jan vandenBerg. Data Mining the SDSS SkyServer Database. *arXiv preprint cs/0202014*, 2002.

[6] Paris Kanellakis, Gabriel Kuper, and Peter Revesz. Constraint Query Languages (Preliminary Report). In *PODS*, 1990.

[7] Shadi Abdul Khalek, Bassem Elkarablieh, Yai Laleye, and Sarfraz Khurshid. Query-Aware Test Generation Using a Relational Constraint Solver. In *ASE*, 2008.

[8] Christophe Lecoutre and Radoslaw Szymanek. Generalized Arc Consistency for Positive Table Constraints. In *CP*, 2006.

[9] Zohar Manna and Calogero Zarba. Combining Decision Procedures. In *10th Anniversary Colloquium of UNU/IIST*, 2002.

[10] Panagiotis Manolios and Vasilis Papavasileiou. ILP Modulo Theories. In *CAV*, 2013.

[11] Panagiotis Manolios, Vasilis Papavasileiou, and Mirek Riedewald. ILP Modulo Data. *CoRR*, abs/1404.5665, 2014.

[12] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Reverse Data Management. In *VLDB*, 2011.

[13] Alexandra Meliou and Dan Suciu. Tiresias: The Database Oracle for How-To Queries. In *SIGMOD*, 2012.

[14] Leonardo De Moura and Nikolaj Bjorner. Efficient E-matching for SMT solvers. In *CADE-21*, 2007.

[15] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.

[16] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.

[17] Daria Sorokina, Rich Caruana, Mirek Riedewald, Wesley Hochachka, and Steve Kelling. Detecting and Interpreting Variable Interactions in Observational Ornithology Data. In *DDDM*, pages 64–69. IEEE, 2009.

[18] Quoc Trung Tran and Chee-Yong Chan. How to ConQueR Why-Not Questions. In *SIGMOD*, 2010.

[19] Moshe Vardi. The Complexity of Relational Query Languages. In *STOC*, 1982.

[20] Margus Veanes, Nikolai Tillmann, and Peli de Halleux. Qex: Symbolic SQL Query Explorer. In *LPAR-16*, 2010.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Turbo-Charging Lemmas on Demand with Don't Care Reasoning

Aina Niemetz, Mathias Preiner, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***Abstract*—Lemmas on demand is an abstraction/refinement technique for procedures deciding Satisfiability Modulo Theories (SMT), which iteratively refines full candidate models of the formula abstraction until convergence. In this paper, we introduce a dual propagation-based technique for optimizing lemmas on demand by extracting partial candidate models via don't care reasoning on full candidate models. Further, we compare our approach to a justification-based approach similar to techniques employed in the context of model checking. We implemented both optimizations in our SMT solver Boolector and provide an extensive experimental evaluation, which shows that by enhancing lemmas on demand with don't care reasoning, the number of lemmas generated, and consequently the solver runtime, is reduced considerably.**

I. Introduction

Procedures for deciding satisfiability of first order formulas w.r.t. first order theories, also known as Satisfiability Modulo Theories (SMT), are usually divided into so-called *eager* and *lazy* approaches. Eager SMT approaches eagerly encode an SMT formula into an equisatisfiable Boolean formula, which then serves as input for a SAT solver. Lazy approaches, on the other hand, are generally based on a tight integration of a SAT solver and one or more theory solvers. The SAT solver typically enumerates Boolean truth assignments satisfying a Boolean abstraction of the input formula, whereas the theory solver(s) not only check if those assignments are consistent w.r.t. the first order theorie(s), but guide the SAT solver through its search. The majority of state-of-the-art SMT solvers employ lazy SMT approaches, where the *lemmas on demand* procedure as introduced for the extensional theory of arrays in [7] is one extreme variant thereof [20]. The core idea of lemmas on demand is similar to the Counterexample-Guided Abstraction Refinement (CEGAR) approach introduced in [9] and goes back to [11], while at the same time, a related technique was proposed in the context of bounded model checking, where all-different constraints are lazily encoded over bit vectors (see also [5]). Recently, in [19] we introduced a generalization of the lemmas on demand decision procedure in [7] to lazily handle λ terms.

Similar to other lazy SMT approaches, lemmas on demand as in [7][19] enumerates truth assignments (so-called *candidate models*) of the bit vector abstraction of the (preprocessed) input formula and iteratively refines those assignments with lemmas until convergence. Each of these candidate models is a full truth assignment of the formula abstraction, which subsequently needs to be checked for consistency w.r.t. the theory of bit vectors with arrays. A full candidate model, however, includes parts of the formula abstraction irrelevant to its satisfiability under the current assignment and might therefore be over-determined.

In this paper we aim at exploiting *a posteriori observability don't cares*, i.e., parts of the formula abstraction irrelevant under the current assignment. We show that don't care reasoning on full candidate models to extract partial candidate models subsequently reduces the cost for consistency checking by focusing on the relevant parts of the formula, only. Motivated by *dual propagation* techniques in the context of quantified boolean formulas (QBF) [15][16], we propose an optimization of the lemmas on demand procedure in [19] and compare our approach to a technique based on *justification* heuristics in ATPG [18]. We implemented both techniques in our SMT solver Boolector and analyse the results in comparison to the version of Boolector that won the QF_AUFBV track of the SMT competition 2012.

Note that in this paper, our justification-based approach mainly serves as a basis for comparison to our dual propagation-based approach. In the context of SMT, Barrett and Donham [3] and De Moura and Bjørner [10] applied justification-based techniques to prune the search space of DPLL(T). In the context of model checking, justification-based techniques have been previously employed to identify a posteriori observability don't cares. Bingham and Hu [6], e.g., prune the search space of their simulation-based bounded model checking engine by means of a justification-based generalization mechanism (*skip cubes*) similar to learning and non-chronological backtracking of conventional SAT procedures. Eén et al. [13] employ a related approach when generalizing proof obligations by *ternary simulation* for property directed reachability (PDR), whereas Chockler et al. [8] use a variant of offline dual propagation for SAT. The verification tool Reveal [2][1], on the other hand, employs a CEGAR approach for model checking complex hardware designs and generalizes candidate counter examples by justification techniques similar to our justification-based method. Their (and our) justification-based approach, however, is only applicable on structural (non-clausal) problems. In contrast, our dual propagation-based approach generalizes full candidate models by exploiting the duality of the Boolean layer of the input formula and is not restricted to structural formula abstractions.

This work was funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

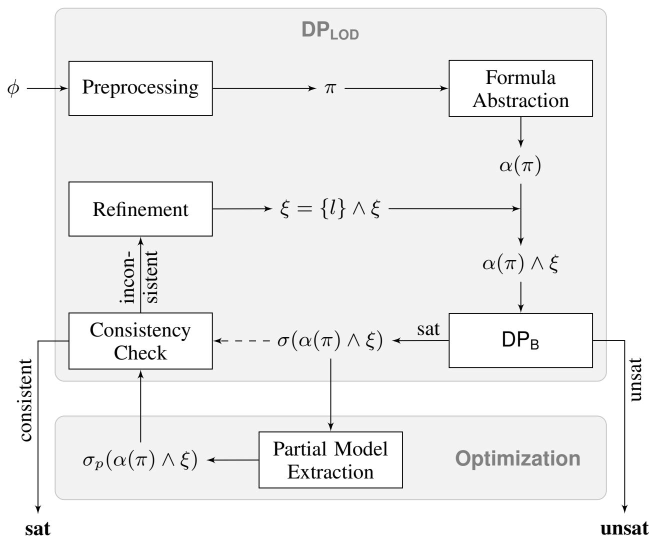


Fig. 1: The workflow of the *lemmas on demand* decision procedure $\mathsf{DP}_{\mathsf{LODopt}}$ in Boolector. The original procedure $\mathsf{DP}_{\mathsf{LOD}}$ (indicated by the dashed line) works on full candidate models, whereas the optimized procedure $\mathsf{DP}_{\mathsf{LODopt}}$ extracts partial candidate models prior to consistency checking.

II. Lemmas on Demand at a Glance

The *lemmas on demand* decision procedure as implemented in Boolector is an iterative abstraction/refinement approach for the quantifier-free theory of fixed-sized bit vectors and arrays. Figure 1 gives a high-level view of the procedure and introduces both the original, unoptimized approach $\mathsf{DP}_{\mathsf{LOD}}$ and our optimized approach $\mathsf{DP}_{\mathsf{LODopt}}$ as follows.

Given a formula ϕ, $\mathsf{DP}_{\mathsf{LOD}}$ uses a *bit vector* skeleton of the preprocessed formula π as formula abstraction $\alpha(\pi)$. In each iteration, an underlying decision procedure DP_{B} determines the satisfiability of the refined formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$ by encoding Γ to SAT and determining its satisfiability by means of a SAT solver. Note that initially, formula refinement ξ is $\top$. As Γ is an overapproximation of ϕ, $\mathsf{DP}_{\mathsf{LOD}}$ immediately concludes with *unsat* if Γ is unsatisfiable. If Γ is satisfiable, the current (full) candidate model $\sigma(\alpha(\pi) \wedge \xi)$ is checked for consistency w.r.t. the preprocessed input formula π. If $\sigma(\alpha(\pi) \wedge \xi)$ is consistent, $\mathsf{DP}_{\mathsf{LOD}}$ immediately concludes with *sat*. Otherwise, $\sigma(\alpha(\pi) \wedge \xi)$ is *spurious* and a lemma l is added to formula refinement ξ.

As indicated in Fig. 1, $\mathsf{DP}_{\mathsf{LOD}}$ iteratively refines $\alpha(\pi)$ by consistency checking *full* candidate models, which usually include parts of the bit vector skeleton irrelevant to its satisfiability under the current assignment. In the following section, we will introduce an optimization to extract a partial candidate model $\sigma_p(\alpha(\pi) \wedge \xi)$ from the full candidate model $\sigma(\alpha(\pi) \wedge \xi)$ in order to guide the consistency check towards the relevant parts of $\alpha(\pi)$ only.

III. Partial Model Extraction

In terms of runtime, abstraction refinement usually is the most costly part of the lemmas on demand procedure $\mathsf{DP}_{\mathsf{LOD}}$, where cost generally correlates with the number of lemmas generated. During refinement, procedure DP_{B} (and consequently the call to the underlying SAT solver) constitutes the majority of the overall runtime per iteration, which adds up when a great number of refinement iterations is needed. Hence, optimizing $\mathsf{DP}_{\mathsf{LOD}}$ in terms of runtime directly translates to reducing the number of lemmas generated.

In contrast to other lazy SMT approaches [20], formula abstraction in $\mathsf{DP}_{\mathsf{LOD}}$ does not produce a pure Boolean skeleton, but a bit vector skeleton, where each function application $f(a_0, \ldots, a_n)$ in the preprocessed formula π is mapped to a fresh bit vector variable. Consequently, consistency checking in $\mathsf{DP}_{\mathsf{LOD}}$ is performed on *all* function applications in the bit vector skeleton (for details see [19]). A high level view of the consistency checking algorithm consistent in $\mathsf{DP}_{\mathsf{LOD}}$ is given in Fig. 2 and proceeds as follows. Given the refined formula abstraction Γ and the full candidate model σ, search_initial_applies collects all function applications in Γ that need to be checked for consistency (line 2) and iteratively checks each APPLY $f(a_0, \ldots, a_n)$ w.r.t. the current assignment σ (lines 4-5). If check_consistency encounters an inconsistency, consistent immediately returns with $\perp$. Else, search_applies_for_consistency_check instantiates function f with arguments $a_0, \ldots, a_n$, which yields term t, and subsequently collects all function applications in formula abstraction $\alpha(t)$ for consistency checking (lines 7-8). If all applies in S have been checked without inconsistencies, procedure consistent concludes that current candidate model σ is consistent and returns $\top$.

```
1 procedure consistent (Γ, σ)
2   S ← search_initial_applies (Γ)
3   while S ≠ ∅
4       f(a_0, ..., a_n) ← pop (S)
5       consistent ← check_consistency (f(a_0, ..., a_n), σ)
6       if not consistent return ⊥
7       S' ← search_applies_for_consistency_check (f(a_0, ..., a_n))
8       push (S, s' ∈ S')
9   return ⊤
```

Fig. 2: Procedure consistent in pseudo-code.

Consistency checking all function applications in formula abstraction Γ corresponds to checking the *full* candidate model σ for consistency, with the order in which applies are checked as the only way to positively influence the number of refinement iterations (by coincidentally finding lemmas that shortcut the search, early on). Checking the full candidate model, however, is often not required, as only a small subset of the full candidate model is responsible for actually satisfying the formula abstraction. As a consequence, parts of the formula abstraction irrelevant to its satisfiability under the current assignment are checked, which subsequently produces lemmas that do not necessarily prune the search space and therefore mainly cost runtime.

Example 1: As a running example, consider the formula

$$\psi_1 \equiv i \neq k \wedge (f(i) = e \vee f(k) = v) \wedge v = ite(i = j, e, g(j))$$

as given in Fig. 3. Its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$

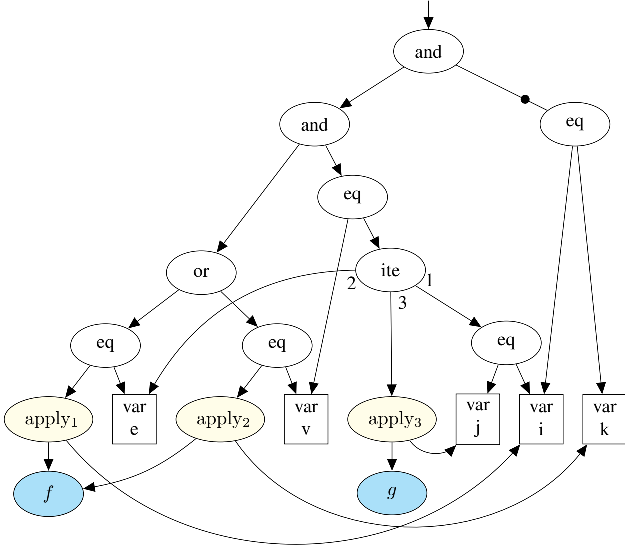


Fig. 3: DAG representation of formula ψ_1 (running example).

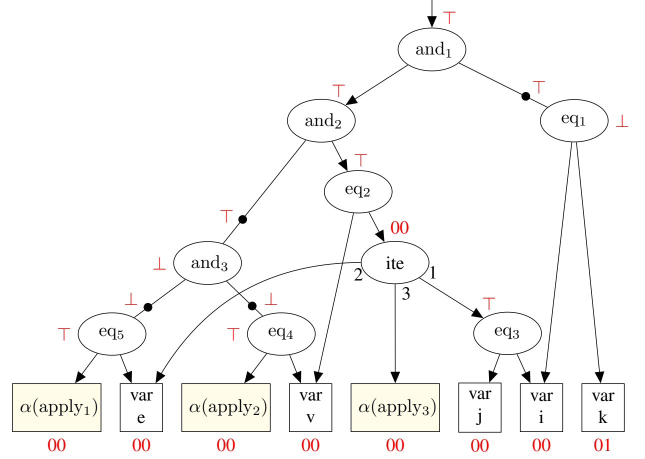


Fig. 4: Formula abstraction Γ_{ψ_1} of formula ψ_1 with candidate model $\sigma(\Gamma_{\psi_1})$ indicated in red (running example).

and a (possible) initial full candidate model $\sigma(\Gamma_{\psi_1})$ (indicated in red) is given in Fig. 4. In the following, we assume that all variables in ψ_1 are bit vector variables of size 2 and Γ_{ψ_1} is a bit vector skeleton. For the sake of simplicity, we further assume that functions f and g represent uninterpreted functions, i.e., we concentrate on consistency checking of the *full* versus a *partial* candidate model (via procedure search_initial_applies) and do not bother with details of the internals of the actual consistency check (for details, see [19]). Procedure search_initial_applies initially collects all function applications in Γ_{ψ_1} (apply_1, apply_2, apply_3) to be checked for consistency. During consistency checking, however, no further applies are identified as required to being checked (procedure search_applies_for_consistency_check) as both f and g do not make subsequent calls to other functions. Note that given $\sigma(\Gamma_{\psi_1})$, instead of checking all applies in ψ_1, either checking apply_1 or apply_2 would be sufficient.

In the following, we consider two techniques for identifying irrelevant parts of the formula abstraction by extracting partial candidate models, which subsequently reduces the number of refinement iterations, and therefore, the overall runtime of the lemmas on demand procedure.

A. *Justification-Based Partial Model Extraction*

In the context of ATPG [18], sets of don't care conditions are usually divided into *observability don't cares (ODC)* and *controllability don't cares (CDC)*. The former denotes lines that do not influence the primary outputs (independent from the current assignment to the primary inputs), and the latter identifies line values that can not be justified and are therefore illegal under any assignment to the primary inputs. Given a concrete assignment to the primary inputs, however, we can determine what we call *a posteriori* observability don't cares, i.e., lines that do not influence the output of a gate under its current assignment. In the context of model checking, such *a posteriori* ODC have already been exploited by Bingham and Hu [6], Eén et al. [13], and Andraus et al. [2][1].

In this section, we introduce a technique similar to [2][1] and extract partial candidate models by identifying parts of the formula abstraction Γ that are irrelevant to its satisfiability under the current assignment σ. As indicated above, this directly translates to collecting and checking function applications in relevant parts of Γ only. In the following, we assume that Γ is represented as a directed acyclic graph (DAG) with exactly one root, where all Boolean operations are expressed by means of NOT and (two-input) AND gates. In place of procedure search_initial_applies, we introduce search_initial_applies$_{\text{just}}$ (Fig. 5), which collects function applications while traversing all relevant paths in Γ as follows.

Given Γ and a full candidate model σ, starting from the root, search_initial_applies$_{\text{just}}$ iteratively traverses Γ towards its primary inputs (bit vector variables and function applications) in depth first search (DFS) order. That is, initially, root(Γ) is pushed onto stack X (line 2) and for each node $x \in X$ we determine the paths to be skipped as follows. If a node x is an AND node and its output is assigned to $\bot$, we follow (one of) its controlling input(s), i.e., one of its inputs with controlling value ($\bot$ for an AND) [18], and skip the other (lines 7-14). Similarly, if x is an IF-THEN-ELSE (ITE) node and its condition is assigned to $\top$ (resp. $\bot$), we follow both its condition and its *then* (resp. *else*) branch (lines 15-20). In any other case where x is not an APPLY node, we follow all inputs of node x (line 22). However, if x is an APPLY node, we collect x (line 6) and cut off the traversal, as function applications are treated as fresh bit vector variables in the formula abstraction.

Note that in the case that *both* inputs of an AND node are controlling, we can skip *either* one of them (lines 9-10). Hence, we choose to follow the input with minimum cost in terms of consistency checking, where the cost of a node x is defined as the minimum number of (unique) applies along a path from x to the primary inputs in the preprocessed formula π. Similar as controllability measures in ATPG [18],

```
1  procedure search_initial_applies_just (Γ, σ)
2    S ← ∅,  X ← {root (Γ)}
3    while  X ≠ ∅
4        x ← pop (X)
5        if  is_apply (x)
6            push (S, x)
7        elif  is_and (x)  and  σ(x) = ⊥
8            l ← left_input (x),  r ← right_input (x)
9            if  is_controlling (l)  and  is_controlling (r)
10               push (X, choose (l, r))
11           elif  is_controlling (l)
12               push (X, l)
13           else
14               push (X, r)
15       elif  is_ite (x)
16           push (x, condition (x))
17           if  σ(condition (x)) = ⊤
18               push (X, then (x))
19           else
20               push (X, else (x))
21       else
22           push (X, i ∈ inputs (x))
23   return  S
```

Fig. 5: Procedure search_initial_applies$_{\text{just}}$ in pseudo-code.

we recursively define a cost function cost (x) as follows.

$$\text{cost}\,(x) = \begin{cases} 0 & \text{if is_var}\,(x) \\ \min \{\text{cost}\,(i) \mid i \in \text{inputs}\,(x)\} & \text{if is_and}\,(x) \\ \text{sum} \{\text{cost}\,(i) \mid i \in \text{inputs}\,(x)\} + 1 & \text{if is_apply}\,(x) \\ \text{sum} \{\text{cost}\,(i) \mid i \in \text{inputs}\,(x)\} & \text{otherwise} \end{cases} \quad (1)$$

Given formula π, a bit vector variable is a primary input, hence its cost is defined as 0. Function applications, on the other hand, are not primary inputs but define the cost of a path from input x to the primary inputs. Hence, the cost of an APPLY is defined as the sum of the costs of its inputs increased by one. In case of an AND node, we want to choose the input with minimum cost if both inputs are controlling, hence cost is defined as the minimum cost of its inputs. In any other case, all input paths have to be followed and cost (x) is defined as the sum of the costs of all inputs of x.

Example 2: Consider formula ψ_1, formula abstraction Γ_{ψ_1}, and a full candidate model $\sigma(\Gamma_{\psi_1})$ as given in Example 1. Starting from the root (and_1), procedure search_initial_applies$_{\text{just}}$ traverses Γ_{ψ_1} in DFS order while identifying (and skipping) all paths irrelevant w.r.t. assignment $\sigma(\Gamma_{\psi_1})$. Note that in Fig. 3 and 4, inverted nodes are indicated by black dots. In the following, however, we will interpret an inverted node as two distinct nodes (with resp. distinct assignments), i.e., $\neg\text{and}_3$ with $\sigma(\neg\text{and}_3) = \top$ in Fig. 4, for example, is treated as a NOT (assigned to $\top$) in front of an AND (assigned to $\bot$). Starting with root and_1, which is assigned to $\top$, neither of its inputs may be skipped and we first travel down towards eq_1, whose inputs are both bit vector variables. Hence, we immediately continue with and_2 (also assigned to $\top$) and follow its input eq_2, where we encounter an ite with its condition assigned to $\top$. We skip the *else* branch, no APPLY is collected, and we continue down the input path leading to and_3, which is assigned to $\bot$. Both inputs of and_3 are controlling (i.e., assigned to $\bot$), hence we choose one of them heuristically. The minimum cost for both paths, however, is 0 (as the body of function f does not contain any further applies), hence we may choose either. We decide on the path to apply_1 and conclude with $S = \{\text{apply}_1\}$, which corresponds to the partial model to be subsequently checked for consistency.

B. Dual Propagation-Based Partial Model Extraction

Exploiting the duality of QBF by propagating a dual set of values through a QBF ϕ and its negation $\neg\phi$, also referred to as *dual propagation*, has successfully been employed in [15] to significantly prune, and therefore speed up the search in circuit-based QBF solvers. The core idea of *dual propagation*, however, is neither restricted to circuit-based representations [16] nor to QBF and is based on the fact that assignments satisfying an input formula ϕ (the *primal* channel), falsify its negation $\neg\phi$ (the *dual* channel) and vice versa. Given a Boolean formula $\psi_2 \equiv (a \wedge b) \vee (c \wedge d)$, for example, assignment $\{\sigma(a) = \top, \sigma(b) = \top, \sigma(c) = \top, \sigma(d) = \top\}$ satisfies ψ_2, but falsifies its negation $\neg\psi_2 \equiv (\neg a \vee \neg b) \wedge (\neg c \vee \neg d)$.

The duality of formula ψ_2, however, can be further exploited. Assume, for example, that given ψ_2 and $\sigma(\psi_2)$ as above, we fix the values of all input variables assigned in $\sigma(\psi_2)$ by making assumptions $\{a{=}\top, b{=}\top, c{=}\top, d{=}\top\}$ to a SAT solver maintaining its negation $\neg\psi_2$. All assumptions inconsistent with $\neg\psi_2$, also called *failed assumptions* [14], identify all input assignments sufficient to falsify $\neg\psi_2$, hence sufficient to satisfy ψ_2. This set of failed assumptions, for example $\{a = \top, b = \top\}$, therefore represents a *partial model* satisfying ψ_2. Note that our approach does not require a structural SAT solver—structural don't care reasoning is simulated via the dual solver, which maintains $\neg\psi_2$ in CNF. Consequently, given a CNF representation of ψ_2 (where structural information of ψ_2 is essentially lost), we extract a partial model (disregarding structural don't cares w.r.t. assignment σ) that satisfies ψ_2 but not necessarily its encoding to CNF. Consider, for example, the Tseitin encoding CNF(ψ_2) $\equiv (\neg o \vee x \vee y) \wedge (\neg x \vee o) \wedge (\neg y \vee o) \wedge (\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg y \vee c) \wedge (\neg y \vee d) \wedge (\neg c \vee \neg d \vee y)$. Our previous partial model $\{a = \top, b = \top\}$ satisfies ψ_2 (and therefore identifies those parts of ψ_2 relevant to its satisfiability) but does not satisfy all clauses in CNF(ψ_2). This is in contrast to other partial model extraction techniques based on iterative removal of unnecessary assignments on the CNF level (e.g. [12]), which do not enable structural don't care reasoning and therefore need to satisfy all clauses in CNF(ψ_2).

In this section, we lift the approach sketched above to the word level by means of a dual SMT solver and introduce a technique to extract partial candidate models via dual propagation-based don't care reasoning. Given a formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$, we use a single *dual* solver instance to maintain $\neg\Gamma$ over all refinement iterations in combination with the *primal* (or *main*) solver. However, since in each iteration i a new lemma l_i is added to $\xi \equiv l_1 \wedge ... \wedge l_{i-1}$, we set up the dual solver to maintain $\neg\Gamma \equiv \neg(\alpha(\pi) \wedge l_1 \wedge ... \wedge l_{i-1} \wedge l_i)$

```
1  procedure search_initial_applies_dp (Γ, σ)
2      S ← ∅,  A ← ∅
3      assume (dual_solver, ¬Γ)
4      X ← collect_primary_inputs (Γ)
5      for x ∈ X
6          a ← x = σ(x),  A ← A ∪ a
7          assume (dual_solver, a)
8      res ← DP_B (dual_solver)
9      assert res = UNSAT
10     for a ∈ A
11         x, σ(x) ← a
12         if is_failed (a) and is_apply (x)
13             push (S, x)
14     return S
```

Fig. 6: Procedure search_initial_applies$_{\text{dp}}$ in pseudo-code. Solver instance $dual_solver$ simulates the dual channel and is maintained globally.

as assumption rather than assertion. As illustrated in Fig. 6, we introduce search_initial_applies$_{\text{dp}}$ in place of procedure search_initial_applies as follows.

Given Γ and a full candidate model σ, procedure search_initial_applies$_{\text{dp}}$ initializes the dual solver by assuming $\neg\Gamma$ (line 3). The value of all primary inputs in $\neg\Gamma$ is then fixed by making assumptions of the form $x = \sigma(x)$, where x is either a bit vector variable or an abstracted function application, and $\sigma(x)$ is its assignment in the current full candidate model σ (lines 4-7). Candidate model σ represents a satisfying assignment for Γ, hence decision procedure DP_B must conclude that assuming σ, $\neg\Gamma$ is unsatisfiable (lines 8-9). The resulting set of failed assumptions identifies all relevant parts of Γ w.r.t. assignment σ, and all function applications in the set of failed assumptions are subsequently collected for consistency checking (lines 10-13).

Example 3: Again, consider formula ψ_1, its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$, and a (possible) full candidate model $\sigma(\psi_1)$ as given in Example 1. Procedure search_initial_applies$_{\text{dp}}$ initializes the dual solver by assuming $\neg\Gamma_{\psi_1} \equiv \neg(i \neq k \wedge (\alpha(\text{apply}_1) = e \vee \alpha(\text{apply}_2) = v) \wedge v = ite(i = j, e, \alpha(\text{apply}_3)))$, and subsequently collects all bit vector variables i, j, k, e, v and abstracted function applications $\alpha(\text{apply}_1)$, $\alpha(\text{apply}_2)$, $\alpha(\text{apply}_3)$ in Γ_{ψ_1} onto stack X. All primary inputs $x \in X$ are then fixed by making assumptions $\{i = 00, j = 00, k = 01, e = 00, v = 00, \alpha(\text{apply}_1) = 00, \alpha(\text{apply}_2) = 00, \alpha(\text{apply}_3) = 00\}$ to the dual SMT solver instance, which concludes that under the current set of assumptions, $\neg\Gamma_{\psi_1}$ is unsatisfiable. Assumption $\alpha(\text{apply}_1) = 00$ is identified as failed assumption and we conclude with $S = \{\text{apply}_1\}$ to be subsequently checked for consistency.

Note that in a sense, our dual propagation-based approach as discussed above simulates dual propagation as introduced in the context of QBF [15][16] rather than literally lifting it to bit vectors with arrays. Dual propagation as in [15][16] is done *eagerly* by means of one single solver instance maintaining a primal and a dual channel without additional overhead. Primary inputs are shared between both channels, which enables symmetric propagation between the primal and dual channel and allows to detect partial models early—even before a full assignment has been generated. In our approach, however, propagation is not interleaved, but consecutive—the primal solver generates a full assignment before the dual solver enables partial model extraction based on the primal full assignment. Further, primary inputs are not physically shared as the dual solver discretely maintains $\neg\phi$ (while mapping primary inputs back to the primal solver and vice versa). Hence we have to simulate shared inputs via fixing input values by means of assumptions to the dual solver, which simply acts as "slave" for partial model extraction to the primal solver. In order to adopt a more eager approach to enable early partial model extraction while reducing the dual solver overhead, interleaved execution between the primal and dual solver similar to "SAT modulo SAT" [4] would be required. Integrating such an interleaved decision process into an existing SMT solver has high potential, however, is rather involved to implement and left to future work.

	Solver	Solved (sat/unsat)	TO	MO	Time [s]	DS [s]
SMT'12	Boolector$_{sc}$	140 (83/57)	9	0	15882	-
	Boolector$_{ba}$	141 (83/58)	8	0	19312	-
	Boolector$_{ju}$	142 (84/58)	7	0	15709	-
	Boolector$_{dp}$	142 (84/58)	7	0	20992	5045
Selected	Boolector$_{sc}$	116 (72/44)	50	7	85863	-
	Boolector$_{ba}$	121 (76/45)	45	7	76104	-
	Boolector$_{ju}$	130 (85/45)	36	7	63202	-
	Boolector$_{dp}$	130 (85/45)	36	7	66991	4705

TABLE I: Overall results on sets *SMT'12* and *Selected*.

IV. Experimental Evaluation

We implemented justification-based and dual propagation-based partial model extraction in our SMT solver Boolector and provide a comparison of the following four configurations:

- Boolector$_{sc}$: The version that won the QF_AUFBV track of the SMT competition 2012.
- Boolector$_{ba}$: Our current base version of Boolector, a slightly optimized version of [19], with partial model extraction disabled.
- Boolector$_{ju}$: Our base version of Boolector with justification-based partial model extraction enabled.
- Boolector$_{dp}$: Our base version of Boolector with dual propagation-based partial model extraction enabled.

We compiled two benchmarks sets for our experimental evaluation: (1) *SMT'12* (149 instances), which consists of all non-extensional benchmarks used for the SMT competition 2012 and (2) *Selected* (173 instances), which includes all non-extensional benchmarks from the QF_AUFBV category of SMT-LIBfor which Boolector$_{sc}$ required at least 10 seconds (CPU time) for solving (incl. timeouts and memouts). Note that we had to exclude extensional benchmarks as Boolector$_{ba}$ and its optimized versions Boolector$_{ju}$ and Boolector$_{dp}$ do not yet support extensionality on arrays. Further note that 58 instances of the benchmark set *SMT'12* are included in *Selected*. All experiments were performed on 2.83Ghz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 12.04.

	Solver	Time [s]			Sat [s]			DS overhead [s]			LOD			Array Model Size		
		Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.
SMT'12	Boolector$_{sc}$	4129	29	2	3662	26	0	-	-	-	30741	221	0	184032	2272	20
	Boolector$_{ba}$	8564	61	6	7262	52	1	-	-	-	33013	237	0	33310	411	20
	Boolector$_{ju}$	6362	45	4	5226	37	0	-	-	-	23660	170	0	19751	243	13
	Boolector$_{dp}$	10145	72	5	4700	33	0	4109	29	0	33492	240	0	27912	344	12
Selected	Boolector$_{sc}$	15037	133	35	12836	113	34	-	-	-	104646	926	175	512225	7645	1257
	Boolector$_{ba}$	10001	88	35	8330	73	22	-	-	-	31752	280	88	136681	2040	212
	Boolector$_{ju}$	8182	72	29	6639	58	19	-	-	-	28215	249	28	122763	1832	154
	Boolector$_{dp}$	10838	95	30	6164	54	15	3036	26	0	24866	220	29	130440	1946	170

TABLE II: Results for commonly solved instances on sets *SMT'12* (139 benchmarks, 82 sat, 57 unsat) and *Selected* (113 benchmarks, 70 sat, 43 unsat). Commonly solved satisfiable instances for determining array model size were 81 (out of 82) for *SMT'12* and 67 (out of 70) for *Selected*. Array model size is measured in terms of number of index/value pairs.

The memory and time limits for each solver instance were set to 7GB and 1200 seconds, respectively.

A. Results Overview

The overall results of all four solver configurations on both benchmark sets *SMT'12* and *Selected* are shown in Table I, which summarizes the number of solved instances (Solved), timeouts (TO), memouts (MO), total CPU time (Time), and the overhead produced by the dual solver in terms of CPU time (DS). Note that the overhead introduced by our justification-based approach is negligible. Further note that in case of a timeout or memout, a penalty of 1200 seconds was added to the total CPU time. On the *SMT'12* benchmark set, in terms of solved instances, Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$ perform slightly better than Boolector$_{sc}$. In terms of runtime, however, only Boolector$_{ju}$ shows a significant improvement (of about 20%), while Boolector$_{dp}$ appears to even perform worse than Boolector$_{ba}$, which is mainly due to the runtime overhead introduced by the dual solver. If we disregard this overhead, the overall runtime of Boolector$_{dp}$ is competitive with the runtime of Boolector$_{ju}$. It is conceivable that an eager implementation of dual propagation would perform equally well, i.e., at least as fast as Boolector$_{dp}$ without the overhead.

Interestingly, Boolector$_{sc}$ clearly outperforms all other three solver configurations on the benchmark family "platania strcmp" (9 instances). Boolector$_{sc}$ solved these benchmarks in about 31 seconds, whereas the other solvers required 4416 seconds (Boolector$_{ba}$), 2308 seconds (Boolector$_{ju}$), and 4527 seconds (Boolector$_{dp}$, incl. 2277 seconds dual solver overhead), respectively. The base version Boolector$_{ba}$, and consequently both Boolector$_{ju}$ and Boolector$_{dp}$, obviously struggle on these benchmarks, which needs further investigation.

Note that benchmark set *SMT'12* is not necessarily representative for lemmas on demand in Boolector, as 79 (53%) out of a total of 149 instances are immediately solved by Boolector$_{sc}$ without a single refinement iteration. Benchmark set *Selected*, on the other hand, has been compiled based on the runtime performance of the SMT competition 2012 winner Boolector$_{sc}$ (incl. timeouts and memouts) and represents a set considered to be "harder" for Boolector. As indicated in Table I, on set *Selected* both Boolector$_{ju}$ and Boolector$_{dp}$ clearly outperform their base version Boolector$_{ba}$ as well as the competition configuration Boolector$_{sc}$. More precisely, both our justification-based and dual propagation-based optimizations considerably reduce the overall runtime while solving 14 (9) additional instances compared to Boolector$_{sc}$ (Boolector$_{ba}$), where 13 (9) out of 14 (9) are satisfiable instances. Again, Boolector$_{dp}$ is slowed down by the dual solver overhead, but still manages to solve as many instances as Boolector$_{ju}$. Disregarding the dual solver overhead, Boolector$_{dp}$ even outperforms Boolector$_{ju}$ in terms of runtime. Note that the dual solver overhead in general correlates with the number of lemmas generated. This is due to the fact that in each refinement iteration a partial candidate model is extracted from the full candidate model, which requires an additional call to the dual solver. On set *Selected*, for 10 out of 130 instances, the dual solver overhead constitutes about 50-70% of the total runtime per instance, whereas for 83 instances it does not exceed 10%.

B. Results Commonly Solved Instances

Table II summarizes all instances in each benchmark set that could be solved by all four solver configurations and gives an overview of the runtime required for solving (Time), the runtime required by the underlying SAT solver (Sat), the dual solver overhead (DS), the number of lemmas generated (LOD), and the size of the array models for satisfiable instances (Array Model Size). For all four solver configurations, we identified 139 common instances (82 sat, 57 unsat) on benchmark set *SMT'12* and 113 common instances (70 sat, 43 unsat) on benchmark set *Selected*. Array model size is measured in terms of the number of index/value pairs identified by each solver with model generation enabled. However, unlike Boolector$_{ba}$ (and consequently Boolector$_{ju}$ and Boolector$_{dp}$), Boolector$_{sc}$ requires additional overhead for model generation, which has a negative impact on the overall number of solved instances. As a consequence, Boolector$_{sc}$ effectively "lost" 1 (resp. 3) satisfiable instance(s) on set *SMT'12* (resp. *Selected*). We therefore compiled all columns except column Array Model Size with model generation disabled.

On the 139 common instances in the *SMT'12* benchmark set, Boolector$_{sc}$ is still the fastest solver, albeit only due to the "platania strcmp" benchmarks mentioned above—on those nine instances, Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$ spent 50%, 35% and 45% of the overall runtime, respectively. A similar picture emerges when comparing the number of refinement iterations required for these nine instances, which constitutes 59%, 47%, and 60% of the total number of lemmas generated by Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$, respectively. In comparison to the base version Boolector$_{ba}$, however, Boolector$_{dp}$ shows the most notable

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

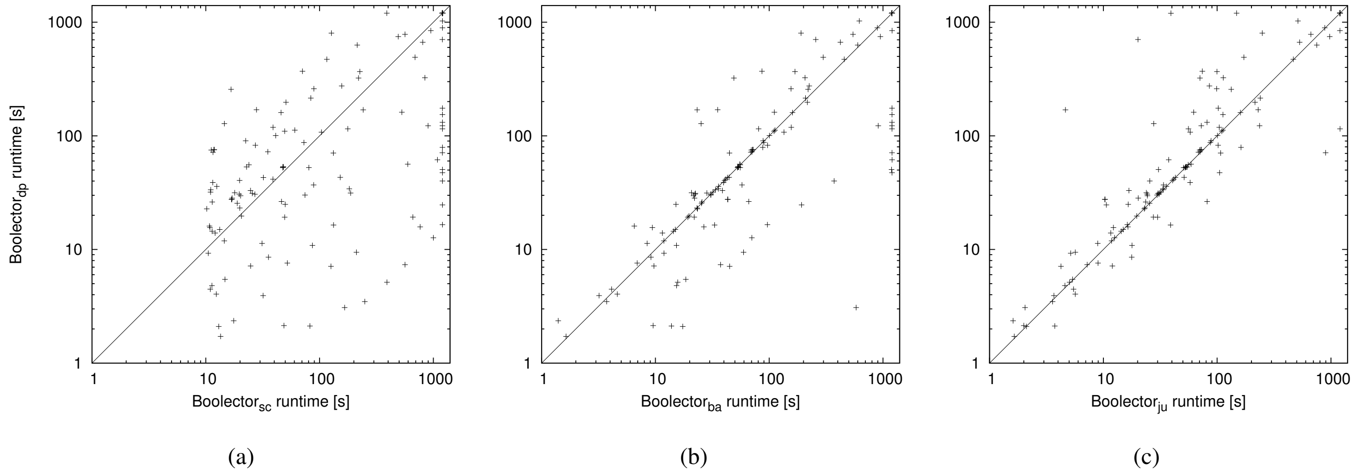


Fig. 7: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead included.

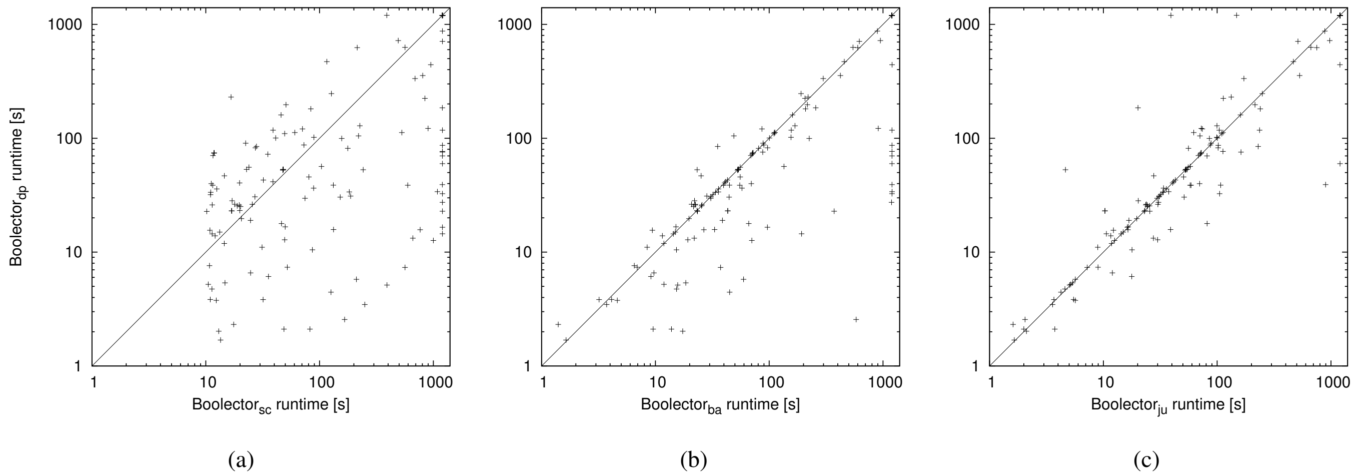


Fig. 8: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead *not* included.

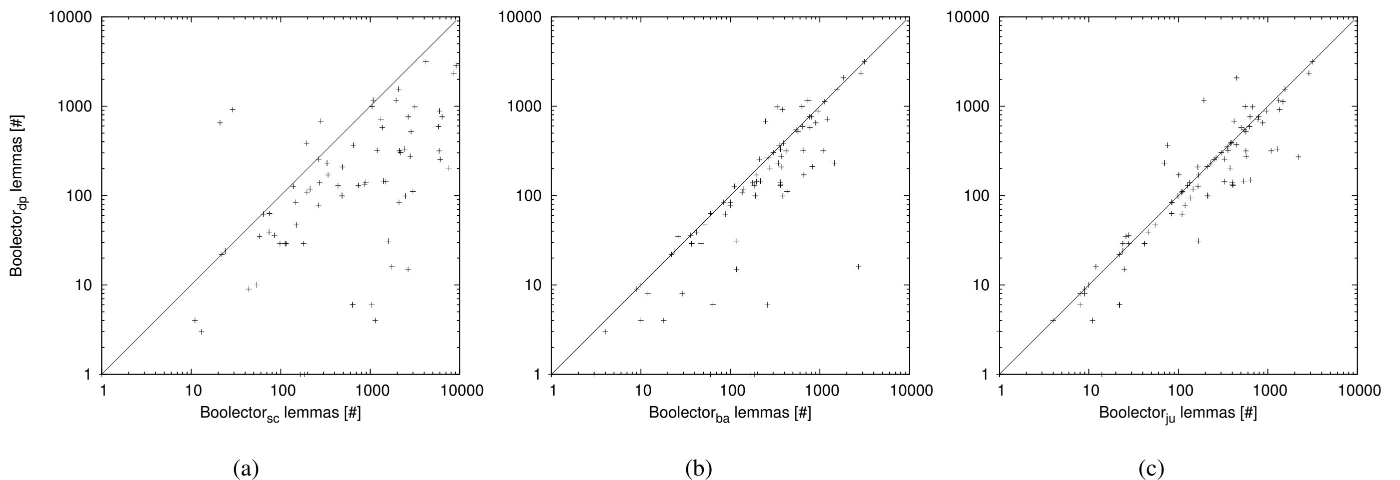


Fig. 9: Comparison of the number of lemmas generated by Boolector$_{dp}$ vs. Boolector$_{sc}$ (9a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (9b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (9c) on benchmark set *Selected*.

improvement (about 26%) in terms of runtime required by the underlying SAT solver on the 139 common instances in *SMT'12*. Disregarding the dual solver overhead, Boolector$_{dp}$ even outperforms Boolector$_{ju}$ in terms of overall runtime. Interestingly, in terms of the number of lemmas generated, Boolector$_{dp}$ requires slightly more lemmas than the base version, which is in stark contrast to Boolector$_{ju}$. However, in case of Boolector$_{dp}$, this can be contributed to a relative small number of instances. On 14 instances, Boolector$_{dp}$ generates 1.5 to 2.6 times more lemmas than Boolector$_{ba}$, whereas on all other instances, Boolector$_{ba}$ requires considerably more refinement iterations than Boolector$_{dp}$. This might indicate that in some cases, Boolector$_{ba}$ coincidentally generates lemmas that shortcut the search early on. In terms of array model size, both optimized configurations Boolector$_{ju}$ and Boolector$_{dp}$ clearly show a reduction in the number of array index/value pairs compared to the base version Boolector$_{ba}$.

Note that the considerable difference in array model size between Boolector$_{sc}$ and Boolector$_{ba}$ is due to an optimization of procedure search_applies_for_consistency_check (see Section III) introduced subsequent to [19]. In essence, given a function application $f(a)$, this optimization aims at consistency checking APPLY nodes reachable while traversing in DFS order from $f(a)$ to the primary inputs, only. In contrast, prior to that optimization it was possible that function applications irrelevant to consistency checking $f(a)$ were pulled in. The effect of this optimization is even more notable on the *Selected* benchmark set, where Boolector$_{ba}$ clearly outperforms Boolector$_{sc}$ in every aspect.

On the 113 common instances in set *Selected*, Boolector$_{dp}$ clearly outperforms Boolector$_{ju}$ and Boolector$_{ba}$ not only in terms of runtime required by the underlying SAT solver, but in the number of lemmas generated. Disregarding the dual solver overhead, Boolector$_{dp}$ shows even more improvement in terms of overall runtime than Boolector$_{ju}$. Note that without the optimization of procedure search_applies_for_consistency_check mentioned above, the difference in terms of overall runtime between Boolector$_{ba}$ and both optimized versions Boolector$_{ju}$ and Boolector$_{dp}$ would be even greater, i.e., comparable to the difference between both optimized versions and Boolector$_{sc}$.

C. Results Dual Propagation-Based Optimization

A more detailed overview of the instance-based results of our dual propagation-based approach Boolector$_{dp}$ on benchmark set *Selected* is given in Fig. 7-9. Figure 7 compares the overall runtime of Boolector$_{dp}$ (incl. the overhead introduced by the dual solver) with the runtime of Boolector$_{sc}$ (7a), Boolector$_{ba}$ (7b), and Boolector$_{ju}$ (7c). Even though the dual solver overhead constitutes 31% of the total runtime of Boolector$_{dp}$, it still outperforms Boolector$_{sc}$ and Boolector$_{ba}$ on a majority of the instances and is even competitive with Boolector$_{ju}$. Disregarding the overhead of the dual solver (Fig. 8), Boolector$_{dp}$ even outperforms Boolector$_{ju}$ on a majority of the instances (Fig. 8c). In terms of the number of lemmas generated (Fig. 9), in comparison to all three solver configurations Boolector$_{sc}$, Boolector$_{ba}$, and Boolector$_{ju}$, our dual propagation-based solver Boolector$_{dp}$ clearly shows the most notable improvement.

V. Conclusion

In this paper we introduced a dual propagation-based optimization of the lemmas on demand procedure for bit vectors with arrays as implemented in Boolector. We compared our approach with a justification-based approach similar to [2][1]. We showed that don't care reasoning on full candidate models improves the performance of lemmas on demand considerably, Our current simulation of dual propagation is competitive with our justification-based optimization and clearly outperforms the winner of the SMT competition 2012, even though the dual solver introduces a considerable amount of overhead to the overall runtime. Adopting a more eager dual propagation approach promises to render the dual solver overhead negligible, while further improving the overall performance by enabling partial model extraction even before a full candidate model has been generated. However, this would require an interleaved execution between the primal and the dual solver, which is rather involved to implement and subject of future work. Further, our current version of dual propagation-based partial model extraction heavily relies on incremental SAT solving under assumptions, which can benefit from dedicated data structures [17]. The integration of such SAT solver level optimization techniques is also left to future work.

Binaries of Boolector and all log files of our experimental evaluation can be found at http://fmv.jku.at/dpjust.

References

[1] Z. S. Andraus. *Automatic Formal Verification of Control Logic in Hardware Designs*. PhD thesis, University of Michigan, 2009.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for Verilog designs. In *LPAR'08*, volume 5330 of *LNCS*. Springer, 2008.

[3] C. Barrett and J. Donham. Combining sat methods with non-clausal decision heuristics. *ENTCS*, 125(3), 2005.

[4] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *FMCAD'13*. IEEE, 2013.

[5] A. Biere and R. Brummayer. Consistency checking of all different constraints over bit-vectors within a SAT solver. In *FMCAD'08*. IEEE.

[6] J. D. Bingham and A. J. Hu. Semi-formal bounded model checking. In *CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[7] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3), 2009.

[8] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *FMCAD'11*. FMCAD Inc., 2011.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[10] L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[11] L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE'02*, volume 2392 of *LNCS*. Springer, 2002.

[12] D. Déharbe, P. Fontaine, D. L. Berre, and B. Mazure. Computing prime implicants. In *FMCAD'13*. IEEE, 2013.

[13] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD'11*. FMCAD Inc., 2011.

[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT'04*, volume 2919 of *LNCS*. Springer, 2004.

[15] A. Goultiaeva and F. Bacchus. Exploiting QBF duality on a circuit representation. In *AAAI'10*. AAAI Press, 2010.

[16] A. Goultiaeva, M. Seidl, and A. Biere. Bridging the gap between dual propagation and CNF-based QBF solving. In *DATE'13*. ACM, 2013.

[17] J.-M. Lagniez and A. Biere. Factoring out assumptions to speed up MUS extraction. In *SAT'13*, volume 7962 of *LNCS*. Springer, 2013.

[18] Z. Navabi. *Digital Sytem Test and Testable Design*. Springer, 2011.

[19] M. Preiner, A. Niemetz, and A. Biere. Lemmas on demand for lambdas. In *DIFTS'13*, volume 1130 of *CEUR Workshop Proceedings*, 2013.

[20] R. Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4), 2007.

Reduction for Compositional Verification of Multi-Threaded Programs

Corneliu Popeea
Technische Universität München

Andrey Rybalchenko
Microsoft Research Cambridge and
Technische Universität München

Andreas Wilhelm
Technische Universität München

***Abstract*—Automated verification of multi-threaded programs requires keeping track of a very large number of possible interactions between the program threads. Different reasoning methods have been proposed that alleviate the explicit enumeration of all thread interleavings, e.g., Lipton's theory of reduction or Owicki-Gries method for compositional reasoning, however their synergistic interplay has not yet been fully explored. In this paper we explore the applicability of the theory of reduction for pruning of equivalent interleavings for the automated verification of multi-threaded programs with infinite-state spaces. We propose proof rules for safety and termination of multi-threaded programs that integrate into an Owicki-Gries based compositional verifier. The verification conditions of our method are Horn clauses, thus facilitating automation by using off-the-shelf Horn clause solvers. We present preliminary experimental results that show the advantages of our approach when compared to state-of-the-art verifiers of C programs.**

I. Introduction

Development of practical verification tools for multi-threaded programs requires dealing with the explosion of the number of thread interleavings that need to be taken into consideration. However, while one can easily construct a contrived program in which every interleaving leads to a different outcome, often enough different interleavings produce equal outcome, and hence can be considered equivalent. Such an equivalence between interleavings suggests that only representatives of each equivalence class need to be considered when verifying a multi-threaded program.

One way to exploit such equivalence is called partial order reduction (POR) [1]. This technique is used in combination with model checking and amounts to restricting the successor computation to representative interleavings, which is performed on-the-fly during the exploration of the model. Explicit-state [1], [2] as well as symbolic [3], [4] model checking algorithms can be effectively combined with POR. Furthermore, recent work shows that POR can also boost interpolation based verification [5], which makes it applicable for the verification of programs with infinite-state spaces.

Alternatively, one can exploit equivalence by transforming a multi-threaded program such that it only produces representative interleavings, or a sufficiently small superset thereof. Such transformation summarises and replaces certain sequences of statements within threads by their composition into so-called *reducible blocks*. Following Lipton's theory of reduction [6], executing these code blocks without any preemption produces representative interleaving when their building blocks commute. Reducible blocks can greatly simplify deductive verification of multi-threaded programs using proof assistants, see e.g. [7]. For finite state systems, reducible blocks (also called transactions in the literature) can be effectively identified and created on-the-fly during model checking [8], [9], [10]. Unfortunately, this does not benefit automatic verification tools for multi-threaded programs with infinite-state spaces. In particular, if reducible blocks contain loops then their invariants are required to replace reducible blocks by their summaries.

In this paper we explore the applicability of reduction for pruning of equivalent interleavings for the automated verification of multi-threaded programs with infinite-state spaces. Since reducible blocks rarely contain all statements of a thread, i.e., there are multiple reducible blocks in each thread as well as some statements that do not belong to any reducible block, we integrate compositional reasoning into our exploration as a complementary technique for avoiding the explicit exploration of all interleavings. That is, our method relies on reduction whenever possible, while statements outside of reducible blocks are subject to compositional reasoning.

Technically, our paper makes the following contributions: 1) a Horn constraint based method for identifying commutativity (mover annotations) of program statements, 2) compositional proof rules for safety and termination that integrate reduction and Owicki-Gries reasoning [11], 3) an efficient implementation based on these ingredients. Our design decisions were directed by the following considerations. Commutativity inference, the first building block of our approach, serves as a preliminary step for a final constraint based verification run. We allow it to be more precise and data dependent in comparison with type based approaches, e.g. [12]. Even though being potentially more expensive, the ability to infer larger transactions at step 1 may lead to dramatic reduction in verification time when dealing with step 2. Our proof rules are also inspired by the use of procedure summaries, see e.g. [13], however instead of being driven by calls/returns to mark start/finish points of summaries, we use transitions that enter/exit from reducible blocks. Note that loops can be part of reducible blocks and summarisation constraints defer reasoning about them to the final solving step.

In summary, this paper shows that reducible blocks can be identified without requiring deep and intricate modification of the underlying verification techniques. Our experimental evaluation shows that the conceptual separation of concerns, i.e., treatment of equivalence between interleavings via reducible blocks and keeping track of interleavings using compositional proof system, compares favourably with state-of-the-art verification approaches.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

```
int x=2, y=2, mx=0, my=0;
// Thread-1
    int a;
 0: acquire(mx);
 1: a = x;
 2: acquire(my);
 3: y = y+a;
 4: release(my);
 5: a = a+1;
 6: acquire(my);
 7: y = y+a;
 8: release(my);
 9: x = 2*x+a;
10: release(mx);
11:

 // Thread-2
 0: acquire(mx);
 1: x = x+2;
 2: release(mx);
 3:

 // Thread-3
 0: acquire(my);
 1: y = y+2;
 2: release(my);
 3:
```

$$
\begin{aligned}
V_G &= (\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my}),\ V_1 = (\mathtt{a},\mathtt{pc}_1),\ V_2 = (\mathtt{pc}_2),\ V_3 = (\mathtt{pc}_3)\\
\mathit{init}(V) &= (\mathtt{x}=2 \wedge \mathtt{y}=2 \wedge \mathtt{mx}=0 \wedge \mathtt{my}=0 \wedge \mathtt{pc}_1=\mathtt{pc}_2=\mathtt{pc}_3=\ell_0)\\
\mathit{next}_1(VV') &= (\mathit{move}_1(\ell_0,\ell_1) \wedge \mathtt{mx}=0 \wedge \mathtt{mx}'=1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{my},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_1,\ell_2) \wedge \mathtt{a}'=\mathtt{x} \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my})) \vee\\
&\quad (\mathit{move}_1(\ell_2,\ell_3) \wedge \mathtt{my}=0 \wedge \mathtt{my}'=1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_3,\ell_4) \wedge \mathtt{y}'=\mathtt{y}+\mathtt{a} \wedge \mathit{skip}(\mathtt{x},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_4,\ell_5) \wedge \mathtt{my}'=0 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_5,\ell_6) \wedge \mathtt{a}'=\mathtt{a}+1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my})) \vee\\
&\quad (\mathit{move}_1(\ell_6,\ell_7) \wedge \mathtt{my}=0 \wedge \mathtt{my}'=1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_7,\ell_8) \wedge \mathtt{y}'=\mathtt{y}+\mathtt{a} \wedge \mathit{skip}(\mathtt{x},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_8,\ell_9) \wedge \mathtt{my}'=0 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_9,\ell_{10}) \wedge \mathtt{x}'=2*\mathtt{x}+\mathtt{a} \wedge \mathit{skip}(\mathtt{y},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee\\
&\quad (\mathit{move}_1(\ell_{10},\ell_{11}) \wedge \mathtt{mx}'=0 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{my},\mathtt{a}))\\
\mathit{next}_2(V,V') &= (\mathit{move}_2(\ell_0,\ell_1) \wedge \mathtt{mx}=0 \wedge \mathtt{mx}'=1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{my})) \vee\\
&\quad (\mathit{move}_2(\ell_1,\ell_2) \wedge \mathtt{x}'=\mathtt{x}+2 \wedge \mathit{skip}(\mathtt{y},\mathtt{mx},\mathtt{my})) \vee\\
&\quad (\mathit{move}_2(\ell_2,\ell_3) \wedge \mathtt{mx}'=0 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{my}))\\
\mathit{next}_3(V,V') &= (\mathit{move}_3(\ell_0,\ell_1) \wedge \mathtt{my}=0 \wedge \mathtt{my}'=1 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx})) \vee\\
&\quad (\mathit{move}_3(\ell_1,\ell_2) \wedge \mathtt{y}'=\mathtt{y}+2 \wedge \mathit{skip}(\mathtt{x},\mathtt{mx},\mathtt{my})) \vee\\
&\quad (\mathit{move}_3(\ell_2,\ell_3) \wedge \mathtt{my}'=0 \wedge \mathit{skip}(\mathtt{x},\mathtt{y},\mathtt{mx}))\\
\mathit{error}(V) &= (\mathtt{x}=11 \wedge \mathtt{pc}_1=\ell_{11} \wedge \mathtt{pc}_2=\ell_3 \wedge \mathtt{pc}_3=\ell_3)
\end{aligned}
$$

Fig. 1. Program P1-1 and its representation as a transition system. The notation $skip(v_1, \ldots, v_k)$ abbreviates the constraint $(v_1' = v_1 \wedge \cdots \wedge v_k' = v_k)$, while $move_i(\ell_j, \ell_k)$ stands for $(pc_i = \ell_j \wedge pc_i' = \ell_k)$.

II. Illustration

Our proposed method consists of two steps, inference of reducible blocks and compositional verification with summarization of the reducible blocks.

We illustrate our verification approach with a multi-threaded program that uses locks $(\mathtt{mx}, \mathtt{my})$ to protect accesses to the shared variables $(\mathtt{x}, \mathtt{y})$. See Figure 1 for the program P1-1. The program state is given by a valuation of program variables $V = (V_G, V_1, V_2, V_3)$, where V_G are global (shared) variables and V_i are thread-local variables for a thread $i \in \{1, 2, 3\}$. Each thread has a thread-local program counter variable $\mathtt{pc}_i \in V_i$ that holds location values ℓ_p for a program line labeled p. We denote by PC_i the set of locations from thread i, i.e., $PC_1 = \{\ell_0, \ldots, \ell_{11}\}$, $PC_2 = \{\ell_0, \ldots, \ell_3\}$ and $PC_3 = \{\ell_0, \ldots, \ell_3\}$. The assertion $init(V)$ gives the initial states of the program. We model the effect of program statements using a thread transition relation $next_i(V, V')$ corresponding to thread i. The primed version V' of the program variables are the next state valuations of V.

Our example uses a thread-synchronization model based on locks, acquire and release statements. We assume that a lock is initially not taken, e.g., $\mathtt{mx} = 0$ and $\mathtt{my} = 0$, and that the `acquire(mx)` statement waits until the lock is released $(\mathtt{mx} = 0)$ and then assigns the value 1 to `mx`. The release statement sets the lock value back to 0.

Reducible block boundaries The objective of the inference of reducible blocks is to minimize the number of explored interleavings during verification. For this illustration, we only show how reducible blocks are encoded in our approach. (See Section V for a formal description of a constraint based method for identifying commutativity of program statements and its application on the P1-1 example.)

Since global variables are consistently accessed while holding a corresponding lock, the statements between acquire and release for both `thread-2` and `thread-3` correspond to a reducible block. For `thread-1`, our inference obtains two reducible blocks, the first one from location ℓ_0 to ℓ_5 and the second one from ℓ_6 to ℓ_{10}. Informally, we shall refer to the four reducible blocks with labels (a), (b), (c) and (d).

(a)	`thread-1` $\{\ell_0 - \ell_5\}$	(b)	`thread-1` $\{\ell_6 - \ell_{10}\}$
(c)	`thread-2` $\{\ell_0 - \ell_2\}$	(d)	`thread-3` $\{\ell_0 - \ell_2\}$

Formally, the result of reduction is encoded using a partitioning of the transition relation of each thread into four categories: $next_out_out_i(V, V')$ describes transitions of thread i having both $\mathtt{pc}_i$ and $\mathtt{pc}_i'$ set to locations outside reducible blocks, $next_in_in_i(V, V')$ and $next_out_in_i(V, V')$ describe transitions with target location $\mathtt{pc}_i'$ inside a reducible block, while $next_in_out_i(V, V')$ describes transitions having $\mathtt{pc}_i$ set to a location inside a reducible block and target $\mathtt{pc}_i'$ outside a reducible block. We also make sure that transitions that target error locations are not part of reducible blocks.

Compositional proof rule with reduction The crux of our verification approach is a proof rule that uses both reduction and compositional reasoning. The proof rule lists conditions over three kinds of auxiliary assertions (or program invariants):

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

$IR_i(V)$ describes reachable states outside reducible blocks that should be accounted for interference by different threads; $LStep_i(V_G, V_i, V'_G, V'_i)$ is a binary relation representing steps inside the same reducible block that are visible only to thread i; $IStep_i(V, V')$ represents steps of thread i that are visible to other threads including steps outside reducible blocks and summaries of reducible blocks.

Let us consider an interleaving of program statements that starts with those statements from the reducible block (c). Verification based on our proof rule continues exploring non-deterministically reducible blocks from either `thread-1` or `thread-3`, i.e., $c{-}a{-}b{-}d$ or $c{-}a{-}d{-}b$ or $c{-}d{-}a{-}b$. Few interleavings are effectively explored due to the coarse-grained nature of reducible blocks. Overall, the following list contains possible block-interleavings that are explored for P1-1.

(I1)	$a-b-c-d$	(I7)	$c-a-b-d$
(I2)	$a-b-d-c$	(I8)	$c-a-d-b$
(I3)	$a-c-b-d$	(I9)	$c-d-a-b$
(I4)	$a-c-d-b$	(I10)	$d-a-b-c$
(I5)	$a-d-b-c$	(I11)	$d-a-c-b$
(I6)	$a-d-c-b$	(I12)	$d-c-a-b$

The effect of all these interleavings is captured by the auxiliary assertions from our proof rule.

For illustration, we aim to prove that the value of the variable `x` is not equal to 11 at the end location. (The variable `x` could have value 11 at the end of the program only following the interleaving $a-c-b-d$. However, as the reader may observe, this interleaving corresponds to an infeasible execution.) For safety, we require that the auxiliary assertions corresponding to the reachable states do not intersect error states. The proof rule has premises over the auxiliary assertions that are expressed as universally quantified Horn clauses. (See Section IV for the formal details.) We compute solutions for the auxiliary assertions using a Horn solver based on abstraction refinement and interpolation over the linear arithmetic domain [14].

For the illustration example, the solution for the reachable state assertion is computed as follows.

$$
\begin{aligned}
&\mathtt{pc}_1 = \ell_0 \wedge \mathtt{mx} = 0 \wedge \\
&\quad (\mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} = 2 \vee \\
&\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge 4 \le \mathtt{x} \le 7) \vee \\
&\mathtt{pc}_1 = \ell_6 \wedge \mathtt{mx} = 1 \wedge \\
&\quad (\mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} = 2 \wedge 2\mathtt{x} + \mathtt{a} = 7 \vee \\
&\quad\ \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge 4{\le}\mathtt{x}{\le}7 \wedge 2\mathtt{x}{+}\mathtt{a}{\ge}13) \vee \\
&\mathtt{pc}_1 = \ell_{11} \wedge \mathtt{mx} = 0 \wedge \\
&\quad (\mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \le 7 \vee \\
&\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \le 9 \vee \\
&\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \ge 13 \wedge 2\mathtt{x} + \mathtt{a} \ge 13)
\end{aligned}
$$

All the states have the location of `thread-1`, $\mathtt{pc}_1 \in \{\ell_0, \ell_6, \ell_{11}\}$. The lock `mx` is held only at ℓ_6, otherwise it is available. The different cases for each program location result from varied interleavings of `thread-1` and `thread-2`, since statements of `thread-3` have no influence on the value of `x`. At states with $\mathtt{pc}_1 = \ell_{11}$, we observe three possibilities: `thread-2` has not yet started ($\mathtt{x} \le 7$), `thread-2` may have been executed after `thread-1` ($\mathtt{x} \le 9$), or `thread-2` may have been executed before `thread-1` ($\mathtt{x} \ge 13$). Note that our method over-approximates the set of reachable states, e.g., constraints on value of `y` are not present in the above solution.

III. Preliminaries

Multi-threaded programs A multi-threaded program P consists of $N \ge 1$ threads. We assume that the program variables $V = (V_G, V_1, \ldots, V_N)$ are partitioned into global variables V_G shared by all threads and local variables $V_1, \ldots, V_N$, which are only accessible by the respective threads.

The set of global states G consists of the valuations of global variables, and the sets of local states $L_1, \ldots, L_N$ consist of the valuations of the local variables of respective threads. A program state is a valuation of the global variables and the local variables of all threads. We represent sets of program states using assertions over program variables. Binary relations between sets of program states are represented using assertions over unprimed and primed variables. The set of initial program states is denoted symbolically by $init(V)$. For each thread i we have a finite set of transitions. Each transition is a binary relation between sets of program states. Furthermore, each transition can only change the values of the global variables and the local variables of the thread i (local variables of other threads do not change). This fact is captured in constraint form using the abbreviation $next_i^{=} := \bigwedge_{j \in 1..N \setminus \{i\}} V'_j = V_j$. We write $next_i(V, V')$ for the union of the transitions of the thread i . The transition relation of the program is $next(V, V') = next_1(V, V') \wedge next_1^{=}(V, V') \vee \cdots \vee next_N(V, V') \wedge next_N^{=}(V, V')$. In the subsequent sections, we abbreviate $next_i(V, V') \wedge next_i^{=}(V, V')$ to $next_i(V, V')$.

We distinguish two special types of variables, program counter variables and lock variables. Firstly, each thread has a program counter pc_i that is a local variable with values in the set PC_i. As a convention, we use labels $\ell_0, \ell_1, ..$ to denote some elements from the previous set. Secondly, some global variables are used for thread synchronization via acquire (acq) and release (rel) primitives. The set of lock variables is denoted by $Locks$, we have $Locks \subseteq V_G$ and we use m, mx, my to denote some elements from the set of locks.

Computations Let $\models$ denote the satisfaction relation between (pairs) of states and assertions over program variables (and their primed versions). A computation of P is a sequence of program states $s_1, s_2, \ldots$ such that s_1 is an initial state, i.e., $s_1 \models init$, and each pair of consecutive states s_i and s_{i+1} in the sequence is connected by a transition ρ of some program thread, i.e., $(s_i, s_{i+1}) \models \rho$. A path is a sequence of transitions.

A program state is reachable if it appears in some computation. Let φ_{reach} be the symbolic representation of the set of all reachable states. The set of error states of a program is denoted using $error(V)$. The program is safe if none of its error states is reachable, i.e., $\varphi_{reach}(V) \wedge error(V) \rightarrow false$. The program is *terminating* if it does not have any infinite computations.

Constraints and queries Let $\mathcal{T}$ be a first-order theory in a given signature and $\models_{\mathcal{T}}$ be the entailment relation with respect to $\mathcal{T}$. We refer to formulas in the given signature as constraints, and let $c(v)$ denote a constraint over the variables v. For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \le 2$, $y \le 1 \wedge x - y \le 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities over rationals/reals. The entailment $y \le 1 \wedge x - y \le 0 \models_{\mathcal{T}} x \le 2$ is valid.

For assertions IR_i, $LStep_i$ and $IStep_i$,

$$
\begin{array}{lll}
\text{(S1)} & init(V) & \rightarrow IR_i(V) \\
\text{(S2)} & IR_i(V) \wedge next_out_in_i(V,V') & \rightarrow LStep_i(V_G,V_i,V'_G,V'_i) \\
\text{(S3)} & LStep_i(V_G,V_i,V'_G,V'_i) \wedge next_in_in_i(V',V'') & \rightarrow LStep_i(V_G,V_i,V''_G,V''_i) \\
\text{(S4)} & IR_i(V) \wedge LStep_i(V_G,V_i,V'_G,V'_i) \wedge next_i^{=}(V,V') \ \wedge next_in_out_i(V',V'') & \rightarrow IStep_i(V,V'') \wedge IR_i(V'') \\
\text{(S5)} & IR_i(V) \wedge next_out_out_i(V,V') & \rightarrow IStep_i(V,V') \wedge IR_i(V') \\
\text{(S6)} & IR_i(V) \wedge (\bigvee_{j \in 1..N \setminus \{i\}} IStep_j(V,V')) & \rightarrow IR_i(V') \\
\text{(S7)} & (\bigwedge_{i=1}^{N} IR_i(V)) \wedge error(V) & \rightarrow false \\
\hline
\end{array}
$$

multi-threaded program P is safe

Fig. 2. Proof rule RULESAFETY.

We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is assumed to be encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries.

Horn-like clauses Let $h(v)$ range over queries and constraints with variables in v. We define a Horn-like clause to be an implication $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$. The left-hand side of the implication is called the body and the right-hand side is called the head. To support efficient verification, our Horn-like clauses slightly deviate from the standard notion of Horn clauses since constraints occurring in our clauses can contain disjunctions and conjunctions.

Solving Horn-like clauses We use a solver for Horn clauses over a first-order theory $\mathcal{T}$ that is invoked as follows.

$$\Sigma := \text{HSF}(HC, \mathcal{Q}, Preds)$$

The solver takes as input a set of clauses HC over queries $\mathcal{Q}$ with optional predicates $Preds$. The function $Preds$ assigns a finite set of predicates to each query symbol q from $\mathcal{Q}$ and defines the abstract domain of a data-flow analysis or predicate abstraction. The solver returns a solution function Σ that maps each query from $\mathcal{Q}$ to a constraint from $\mathcal{T}$.

IV. PROOF RULES

In this section we present proof rules that combine reduction and compositional reasoning.

A. Proof rule for safety

See Figure 2 for our proof rule RULESAFETY that lists conditions for program safety over the following assertions.

- $IR_i(V)$: interfering state assertions that represent state reachability information outside reducible blocks for thread $i \in 1..N$.
- $LStep_i(V_G, V_i, V'_G, V'_i)$: non-interfering step assertions that represent steps of thread i that are only locally-visible for thread $i \in 1..N$.
- $IStep_i(V, V')$: interfering step assertions that represent steps of thread i that are visible to other threads (interfering steps) for thread $i \in 1..N$.

The clauses (S1) to (S6) are replicated for each thread i. The clause (S1) considers that initial states are reachable states. The clauses (S2) and (S3) do thread-modular reasoning inside reducible blocks - (S2) initiates relations with target locations inside reducible blocks and (S3) transitively extends these relations. The clause (S4) makes the effect of a reducible block visible to other threads, as well as in the interfering reachable states. The clauses (S5) and (S6) perform compositional reasoning outside reducible blocks by using single transitions and reducible block relations, respectively. The last clause (S7) checks that states reachable outside reducible blocks do not intersect the error states.

Theorem 1. *The proof rule* RULESAFETY *is sound, i.e., if an error state is reachable the constraint system consisting of clauses (S1) to (S7) has no solution.*

A correctness argument of the proof rule is omitted for space constraints. (A soundness proof for a rule based on reduction and compositional reasoning is included in the thesis of one of the authors [15, (Section 3.5)].)

Example 1. The first clause from the proof rule states that all initial states are included in the $IR_i(V)$ assertions. For the example from Section II a solution of the reachable-states assertion will include at least the initial states:

$$(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \wedge \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)$$

Clause (S2) initiates a binary relation $LStep_i$ for a thread i whenever a transition $next_out_in_i(V, V')$ targeting a location from a reducible block is enabled. Once inside a reducible block, the clause (S3) uses relational composition to include relations in $LStep_i$ as long as further transitions $next_in_in_i(V, V')$ are enabled. We illustrate the application of these clauses using the transitions corresponding to `thread-2` that start from the previously computed initial states.

$$move_2(\ell_0, \ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{my}) \ \vee$$
$$move_2(\ell_0, \ell_2) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge \mathtt{x}' = \mathtt{x} + 2 \wedge skip(\mathtt{y}, \mathtt{my})$$

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

For assertions IR_i, $LStep_i$ and $IStep_i$ satisfying (S1), ...,(S6) and assertions $LRound_i$, $IRound$,

$$\begin{array}{ll}(\text{T1}) & (\bigwedge_{i=1}^{N} IR_i(V)) \wedge LStep_i(V_G, V_i, V'_G, V'_i) \wedge next_in_in_i(V', V'')) \rightarrow LRound_i(V'_G, V'_i, V''_G, V''_i) \\ (\text{T2}) & well\text{-}founded(LRound_i) \\ (\text{T3}) & (\bigwedge_{i=1}^{N} IR_i(V)) \wedge (\bigvee_{j=1}^{N} IStep_j(V, V')) \rightarrow IRound(V, V') \\ (\text{T4}) & well\text{-}founded(IRound) \\ \hline \end{array}$$

multi-threaded program P terminates

Fig. 3. Proof rule RULETERMINATION.

Clause (S4) generates a summary relation for the reducible block of `thread-2` from the previous relation and the assertion $next_in_out_2(V, V')$:

$$move_2(\ell_0, \ell_3) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 0 \wedge \mathtt{x}' = \mathtt{x} + 2 \ \wedge skip(\mathtt{y}, \mathtt{my})$$

Besides clause (S1), the clauses (S4) and (S5) generate reachable states $IR_i(V)$ by applying enabled reducible block relations or transitions outside reducible blocks, respectively. For our example, the following formula represents additional reachable states generated from these clauses.

$$\begin{array}{l}(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \wedge \\ \quad \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0) \vee \\ (\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_0 \wedge \\ \quad \mathtt{x} = 4 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)\end{array}$$

B. Proof rule for termination

See Figure 3 for our proof rule RULETERMINATION that lists conditions to ensure that a program is terminating. The conditions are over assertions IR_i, $LStep_i$, $IStep_i$ that satisfy the clauses from the rule RULESAFETY and the following additional assertions:

- $LRound_i$: binary relation assertions that represent thread-modular transition relations inside reducible blocks for $i \in 1..N$.
- $IRound$: binary relation assertion that represents transition relations outside of reducible blocks together with summary relations of reducible blocks.

For each thread i the clause (T1) together with clause (T2) guarantees that there is no infinite computation executing within some reducible block. Clause (T3) together with clause (T4) guarantees that there is no infinite computation that keeps alternating between reducible blocks of the program infinitely often.

Theorem 2. *The proof rule* RULETERMINATION *is sound, i.e., the constraint system consisting of clauses (S1)..(S6) and (T1)..(T4) has a solution only if the program is terminating.*

The premises of RULETERMINATION can be solved using the Horn solver HSF [14], since the premises can be represented as Horn clauses with disjunctive well-foundedness constraints. We write $well\text{-}founded(\varphi(v, v'))$ if $\varphi(v, v')$ is a well-founded relation, i.e., there is no infinite sequence $s_1, s_2, \dots$ such that $\varphi(s_i, s_{i+1})$ for all $i > 1$. A relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations, i.e., there exist well-founded $\varphi_1(v, v'), \dots, \varphi_n(v, v')$ such that $\varphi(v, v') \rightarrow \varphi_1(v, v') \vee \dots \vee \varphi_n(v, v')$ is a valid implication.

Example 2. We extend `thread-1` from Figure 1 with a loop that spans over newly inserted locations ℓ_{1b} and ℓ_{8b}.

```
// Thread-1
 ...
 1:  a = x;
 1b: while a<=4
 ...
 8:  release(my);
 8b: endwhile
 9:  x = 2*x+a;
 ...
```

Since this change does not introduce any additional non-mover transitions, the reducible block boundaries of `thread-1` remain the same, i.e., $\{\ell_0 - \ell_5\}$ and $\{\ell_6 - \ell_{10}\}$.

The check corresponding to clause (T2) succeeds immediately, since the example does not contain looping executions in a reducible block. For (T3), consider the following formula that is computed by the Horn solver for the body of the clause.

$$(\bigwedge_{i=1}^{N} IR_i(V)) \wedge \Big(\begin{array}{l}(move_1(\ell_0, \ell_6) \wedge \mathtt{a} \leq 4 \wedge ..) \vee \\ (move_1(\ell_6, \ell_6) \wedge \mathtt{a} \leq 4 \wedge \mathtt{a}' = \mathtt{a} + 1 \wedge ..) \vee \\ (move_1(\ell_6, \ell_{11}) \wedge \mathtt{a} > 4 \wedge ..) \vee \\ (move_2(\ell_0, \ell_3) \wedge ..) \vee \\ (move_3(\ell_0, \ell_3) \wedge ..)\end{array} \Big)$$

The only disjunct that could potentially permit infinite state sequences corresponds to $move_1(\ell_6, \ell_6)$. The HSF solver concludes that this relation is well-founded, since variable a is incremented and has an upper bound, and thus the example program is proven terminating.

V. INFERENCE OF REDUCIBLE BLOCKS

This section depicts the computation steps for obtaining reducible block boundaries. We consistently use a constraint-based approach to solve the data-flow problems for every step. Our formalization is based on the theory of reduction [6] and follows the approach used to infer transactions for finite-state model checking [9]. We illustrate our method using the program from Figure 1.

A. Locks-held and mover information

Reducible block inference requires for each reachable program transition specific mover information which highly depends on the held locks. We use data-flow analysis to compute $lh_i(\ell)$, an approximation of the set of locks held by a thread $i \in 1..N$ at location $\ell \in PC_i$. The following set of Horn clauses over queries $\mathcal{Q}_1 := \{LR_1(V), \dots LR_N(V)\}$ allows us to obtain reachable states of thread i without considering any thread context switches.

$$HC_1 := \{ \; init(V) \rightarrow LR_i(V), \\ LR_i(V) \wedge next_i(V, V') \rightarrow LR_i(V') \mid i \in 1..N\}$$

The abstract domain of the static analysis is defined by a predicate function. It is initialized with predicates over program counter and lock variables as follows.

$$Preds_1(LR_i(V)) := \; \{pc_i = \ell_i \mid \ell_i \in PC_i\} \cup \\ \{m = 0, m = 1 \mid m \in Locks\}$$

We invoke the HSF solver: $\Sigma_1 := \mathrm{HSF}(HC_1, \mathcal{Q}_1, Preds_1)$. Note that without a clause involving error states, the solver computes only one over-approximation of the reachable states, i.e. no abstraction refinement is performed in this phase. $lh_i(\ell)$ contains the set of held locks for location ℓ by utilizing Σ_1.

$$lh_i(\ell) := \{m \in Locks \mid \forall V{:}\Sigma_1(LR_i(V)) \wedge pc_i = \ell \rightarrow m = 1\}$$

Example 3. The solution corresponding to the first thread from Figure 1, $\Sigma_1(LR_1(V))$, follows.

$$(pc_1 \in \{\ell_0, \ell_{11}\} \wedge mx = 0 \wedge my = 0) \vee \\ (pc_1 \in \{\ell_1, \ell_2, \ell_5, \ell_6, \ell_9, \ell_{10}\} \wedge mx = 1 \wedge my = 0) \vee \\ (pc_1 \in \{\ell_3, \ell_4, \ell_7, \ell_8\} \wedge mx = 1 \wedge my = 1)$$

The locks-held information derived at location ℓ_3 is $lh_1(\ell_3) := \{mx, my\}$.

We represent transition-mover information using four boolean functions defined over pairs of program locations: $rm_i(pc_i, pc_i')$, $lm_i(pc_i, pc_i')$, $nm_i(pc_i, pc_i')$, $bm_i(pc_i, pc_i')$. Following the theory of reduction [6], an acquire transition is a right-mover (i.e., it commutes to the right with every transition from other threads) and a release transition is a left-mover (i.e., it commutes to the left with every transition from other threads). A transition $\rho_i(pc_i, pc_i')$ is a non-mover if there exists a transition from another thread $\rho_j(pc_j, pc_j')$ that accesses some common global variable (at least one thread performing a write access) and the intersection of the sets of locks held by thread i at pc_i and those held by thread j at pc_j is empty. Transitions that are neither left-movers, right-movers nor non-movers are both-movers.

Example 4. For the first thread we obtain:

$$rm_1 := \{(\ell_0, \ell_1), (\ell_2, \ell_3), (\ell_6, \ell_7)\} \\ lm_1 := \{(\ell_4, \ell_5), (\ell_8, \ell_9), (\ell_{10}, \ell_{11})\} \\ nm_1 := \emptyset \\ bm_1 := \{(\ell_1, \ell_2), (\ell_3, \ell_4), (\ell_5, \ell_6), (\ell_7, \ell_8), (\ell_9, \ell_{10})\}$$

B. In-Out information

Let $n, m \in \mathbb{Z}^+$, $i \in 1..n$, and $j \in 1..m$. A reducible block is a non-empty sequence of transition relations $a_1, \dots, a_n, [c], b_1, \dots, b_m$ where each a_i (b_j) is a right-mover (left-mover) and c is an optional non-mover. We use the transition-mover information from Section V-A to group program locations into two phases; a *pre-commit-phase* and a *post-commit-phase*. The former phase contains target locations of right-mover (a_i) or initial locations. The latter phase contains target locations of all other transitions (c, b_j).

We utilize Horn clauses over the following set of queries: $\mathcal{Q}_2 := \{Ph_1(V, p), \dots, Ph_N(V, p)\}$ representing reachable state queries extended by a boolean phase variable p that indicates either the *pre-commit-phase* (p has value 1) or the *post-commit-phase* (p has value 0). The set HC_2 contains the following clauses replicated for $i \in 1..N$.

$$init(V) \wedge p = 1 \rightarrow Ph_i(V, p) \\ Ph_i(V, p) \wedge next_i(V, V') \wedge rm_i(pc_i, pc_i') \wedge p'{=}1 \rightarrow Ph_i(V', p') \\ Ph_i(V, p) \wedge next_i(V, V') \wedge \\ (lm_i(pc_i, pc_i') \vee nm_i(pc_i, pc_i')) \wedge p' = 0 \rightarrow Ph_i(V', p') \\ Ph_i(V, p) \wedge next_i(V, V') \wedge bm_i(pc_i, pc_i') \wedge p'{=}p \rightarrow Ph_i(V', p')$$

To define the abstract domain of the data-flow analysis, we initialize the predicate function with predicates over program counter and phase variables.

$$Preds_2(Ph_i(V, p)) := \{pc_i = \ell_i \mid \ell_i \in PC_i\} \cup \{p{=}0, p{=}1\}$$

We invoke the HSF solver: $\Sigma_2 := \mathrm{HSF}(HC_2, \mathcal{Q}_2, Preds_2)$. Reducible block information is extracted from the solution Σ_2 and represented using boolean functions defined over program locations. $In_i(pc_i)$ holds when pc_i is a location inside a reducible block, while $Out_i(pc_i)$ holds when pc_i is a location outside any reducible block. A location is inside a reducible block if it is contained in the *pre-commit-phase* or if every enabled transition left-commutes with transitions from other threads. Otherwise, a location is outside any reducible block.

$$In_i(pc_i) \; := \Sigma_2(Ph_i(V, p)) \wedge \neg init(V) \wedge (p = 1 \vee p = 0 \wedge \\ \forall pc_i' : lm_i(pc_i, pc_i') \vee bm_i(pc_i, pc_i')) \\ Out_i(pc_i) := \neg In_i(pc_i)$$

Example 5. The solution corresponding to the first thread follows.

$$\Sigma_2(Ph_1(V, p)) := (pc_1 \in \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\} \wedge p = 1 \vee \\ pc_1 \in \{\ell_5\} \wedge p = 0 \vee \\ pc_1 \in \{\ell_6, \ell_7, \ell_8\} \wedge p = 1 \vee \\ pc_1 \in \{\ell_9, \ell_{10}, \ell_{11}\} \wedge p = 0)$$

We obtain the following results for the first thread: $Out_1 := \{\ell_0, \ell_6, \ell_{11}\}$ and $In_1 := PC_1 \setminus Out_1$. The results for the second and third thread are computed similarly: $Out_2 := \{\ell_0, \ell_3\}, In_2 = \{\ell_1, \ell_2\}$ and $Out_3 := \{\ell_0, \ell_3\}, In_3 = \{\ell_1, \ell_2\}$.

Given the in-out information, we partition the transition relation of a thread depending on whether the target of a transition is a state in/outside a reducible block. We also make sure that transitions that target error locations are not part of reducible blocks. We obtain the following four relations corresponding to $next_in_in_i(V, V')$, $next_out_in_i(V, V')$, $next_in_out_i(V, V')$ and respectively to $next_out_out_i(V, V')$.

$$next_i(V, V') \wedge In_i(pc_i) \wedge In_i(pc_i') \wedge \neg error(V') \\ next_i(V, V') \wedge Out_i(pc_i) \wedge In_i(pc_i') \wedge \neg error(V') \\ next_i(V, V') \wedge In_i(pc_i) \wedge (Out_i(pc_i') \vee error(V')) \\ next_i(V, V') \wedge Out_i(pc_i) \wedge (Out_i(pc_i') \vee error(V'))$$

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

TABLE I. Results for verification of safety properties. A ✓-mark (×-mark) indicates a safe (unsafe) program. Experiments were run on an Intel Xeon machine, clocked at 3.47GHz with 8 GB RAM. A T/O-mark represents a time-out after 5400s.

Program	LOC	Threads	Safe	Impara	Threader	Comp	RedComp
P1-1	48	3	✓	**1s**	6s	7s	2s
P1-5	64	3	✓	110s	281s	101s	**32s**
P1-10	84	3	✓	T/O	T/O	840s	**64s**
P1-50	244	3	✓	T/O	T/O	T/O	**2400s**
P2-5	65	3	✓	**83s**	617s	270s	140s
P2-10	85	3	✓	T/O	T/O	1020s	**220s**
P2-50	245	3	✓	T/O	T/O	T/O	**3778s**
stack-safe-5	50	3	✓	115s	**5s**	96s	17s
stack-safe-10	50	3	✓	635s	127s	224s	**75s**
stack-unsafe-5	48	3	×	2s	**1s**	9s	2s
stack-unsafe-10	48	3	×	62s	**2s**	9s	3s
pbzip2-safe	283	4	✓	T/O	T/O	T/O	**840s**
twostage-3-unsafe	129	4	×	T/O	843s	T/O	**17s**

VI. Implementation and Experimental Evaluation

The implementation of our approach consists of three conceptual modules. The first module is a frontend implemented in the OCaml language that relies on the CIL library. It translates C programs to corresponding transition systems in Horn clause form. Our frontend relies on a number of static analyses: thread-scope inference for dynamic thread creation, a pointer analysis that is context-sensitive and malloc-sensitive, optional array expansion for bounded arrays and restricted quantified invariants for unbounded arrays.

A second module that infers reducible block boundaries following the approach from Section V can be automated using HSF, but is not yet integrated in our implementation. The lock analysis can be realised through solving the clauses HC1 from Section 5.1, while identification of left/right-movers reduces to solving the clauses HC2 from Section 5.2.

The third module is a model checker implemented using the HSF approach [14]. This module is given as input a proof rule written as Horn clauses and the program generated by our frontend with thread transition relations partitioned as $next_in_in$, $next_in_out$, $next_out_in$ and $next_out_out$.

A. Evaluation

In this section, we give details on an experimental evaluation of our approach. We compare results from our implementation with two state-of-the-art verifiers: Threader [16], the winner in the Concurrency category of SV-COMP 2013 and Impara [5], a verifier that combines partial-order-reduction and interpolation [17]. Binaries and test programs used for evaluation are made publicly available [18].

In general, our method benefits from the datarace-free nature of statements to infer coarse-grained reducible blocks. For evaluation purposes, we test how our verifier works on programs with race conditions on shared variables. Consider P2-1, a variation of P1-1 from Figure 1 such that `thread-3` has an additional statement that accesses the shared variable `y` without holding the lock `my`. For this modified program, our reducible block inference computes more reducible blocks for `thread-1` and `thread-3` than it is the case for the original program P1-1. However, the reduction phase still significantly reduces the number of interleavings to be explored.

See Table I for verification results of safety properties. We report on variations of four programs. P1-1 and P2-1 were described in the previous sections of the paper, stack-safe-5 is part of SV-COMP 2013 and is challenging to the partial-order reduction method implemented in Impara [5] and stack-unsafe-5 is the modified stack example that does not satisfy its safety assertion. As variations, P1-x, P2-x have "x" statements in each of their reducible blocks. For stack-safe-x and stack-unsafe-x, we vary the number of elements stored in the stack. Lastly, we include two benchmarks that are challenging to Impara and Threader, twostage-3-unsafe from SV-COMP and pbzip2-safe, a multi-threaded implementation of a compression algorithm.

For each test program from Table I, we report the number of lines of C code in Column 2, the expected verification result in Column 4 and statistics on four verification methods. Column 5 presents results from Impara [5], while Column 6 presents timings from the best performing compositional proof rule implemented in Threader. Column 7 presents results from our implementation based on a rule that uses compositional reasoning but not reduction. Column 8 presents timings for our new verification method (RedComp stands for the Reduction-Compositional verification).

For the same implementation, we observe that reduction improves the performance of a compositional reasoning verifier, i.e., RedComp in Column 7 versus Comp in Column 6. When comparing our synthesis-driven implementation Comp with Threader, a verifier optimized for the same proof rule, we observe some overhead for test programs that are less favorable for reduction, i.e., P1-1, stack-safe-5 and stack-unsafe-5. However, for the variations of these programs that are more favorable for reduction, we observe reduction in verification time for our proposed method RedComp.

See Table II for results on verification of termination properties. The program fig2-tacas12 has a complex termination proof based on disjunctive well-founded transition invariant [19]. sync01-safe is a benchmark from SV-COMP 2014 that is marked as safe for assertion violations and suffers from a non-termination bug. One thread may block waiting for a signal on a condition variable, a bug that is uncovered using RedComp. Finally, we include a C program modeling the dining philosophers problem.

Due to the not-yet integrated block inference, we present in this section only a limited experimental evaluation on selected examples that are challenging for Impara and Threader. In principle our current approach (reduction + compositional reasoning) subsumes the compositional algorithms from Threader. For the most imprecise inference of reducible blocks, i.e., with $In_i = \emptyset$ and $Out_i = PC_i$, the proof rule from Figure 2

TABLE II. Results for verification of termination properties.

Program	LOC	Terminates	Comp	RedComp
fig2-tacas12	24	✓	**2s**	3s
sync01-safe-fixed	62	✓	308s	**4s**
dining-philo	108	✓	T/O	**7s**

reduces immediately to the Owicki-Gries rule automated in Threader. (Threader already delivers conclusive results for most of the other "Concurrency" benchmarks from SV-COMP.)

VII. Related work

The reduction principle, as formulated by Lipton [6], has been used in program analysis for checking or inferring whether a method is atomic, i.e., whether the body of the method corresponds to a reducible block. These program analyses were formalized either using a type system [12], [20], or as a dynamic analysis [21]. Going one step further and using the result of atomicity analysis for verification has been proposed only in the context of finite-state verification algorithms [8], [9] where the algorithms that benefit from reduction are quite different than approaches like ours based on interpolation-based verification. Reduction can greatly simplify deductive verification of multi-threaded programs using proof assistants [7]. Our current work can be viewed as a step towards an integration of reduction in interpolation-based verification.

Apart from works based directly on Lipton's theory of reduction, there have been other verification methods aiming to avoid exploring interleavings that are equivalent.

One approach stems from compositional reasoning proof rules, i.e., the Owicki-Gries method [11] or rely-guarantee reasoning [22]. These compositional proof methods have been automated for verification of finite-state models [23] and infinite-state models [24] using counter-example guided abstraction refinement [25]. Since compositional reasoning avoids exploring many equivalent interleavings, Threader [16], an implementation of the previous algorithms, has been able to compete with success in the Concurrency category of the verification competition held at TACAS [26]. Our current work can be viewed as an extension of Threader's algorithms with a reduction-based static analysis that avoids exploring even more redundant interleavings.

Another approach to the state explosion problem is partial order reduction [1] that has been used for finite-state verification, e.g., [2]. Recent work shows that POR can also boost interpolation based verification [5], which makes it applicable for the verification of programs with infinite-state spaces. This approach has been implemented in a tool called Impara.

We emphasize the connection between procedure summarization [13] and our approach. Rather than summarizing procedures in sequential programs, our current work summarizes reducible blocks in the context of multi-threaded program verification. Our approach has been inspired by a work on summarization of concurrent programs [9], with the distinguishing feature that our work is applicable for infinite-state spaces. While procedure summarization allowed software analysis tools like Slam and Saturn to perform composable analysis of large code bases, our work aims to use summarization of reducible blocks to allow verification to scale to large multi-threaded programs.

Acknowledgments

We thank Klaus von Gleissenthall for comments and suggestions. This research was supported in part by the ERC project 308125.

References

[1] P. Godefroid, "Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem," Ph.D. dissertation, University of Liege, Computer Science Department, 1994.

[2] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005.

[3] F. Lerda, N. Sinha, and M. Theobald, "Symbolic model checking of software," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 3, pp. 480–498, 2003.

[4] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *CAV*, 1997.

[5] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with Impact," in *FMCAD*, 2013.

[6] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[7] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *POPL*, 2009.

[8] C. Flanagan and S. Qadeer, "Transactions for software model checking," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 3, pp. 518–539, 2003.

[9] S. Qadeer, S. K. Rajamani, and J. Rehof, "Summarizing procedures in concurrent programs," in *POPL*, 2004.

[10] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *CAV*, 2004.

[11] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, 1976.

[12] C. Flanagan and S. Qadeer, "Types for atomicity," in *TLDI*, 2003.

[13] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995.

[14] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.

[15] A. Wilhelm, "Efficient verification of multi-threaded programs," Master's thesis, 2013, available from http://www.model.in.tum.de/~popeea/research/wilhelm.msc13.pdf/.

[16] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *CAV*, 2011.

[17] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.

[18] C. Popeea, "Redcomp webpage," http://www.model.in.tum.de/~popeea/research/redcomp, accessed: 09-Feb-2014.

[19] C. Popeea and A. Rybalchenko, "Compositional termination proofs for multi-threaded programs," in *TACAS*, 2012.

[20] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*, 2003.

[21] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *POPL*, 2004.

[22] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, 1983.

[23] A. Cohen and K. S. Namjoshi, "Local proofs for global safety properties," in *CAV*, 2007.

[24] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*, 2011.

[25] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000.

[26] D. Beyer, "Second competition on software verification - (summary of SV-COMP 2013)," in *TACAS*, 2013.

Finding Conflicting Instances of Quantified Formulas in SMT

Andrew Reynolds
The University of Iowa

Cesare Tinelli
The University of Iowa

Leonardo de Moura
Microsoft Research

***Abstract*—In the past decade, Satisfiability Modulo Theories (SMT) solvers have been used successfully in a variety of applications including verification, automated theorem proving, and synthesis. While such solvers are highly adept at handling ground constraints in several decidable background theories, they primarily rely on heuristic quantifier instantiation methods such as E-matching to process quantified formulas. The success of these methods is often hindered by an overproduction of instantiations which makes ground level reasoning difficult. We introduce a new technique that alleviates this shortcoming by first discovering instantiations that are in conflict with the current state of the solver. The solver only resorts to traditional heuristic methods when such instantiations cannot be found, thus decreasing its dependence upon E-matching. Our experimental results show that our technique significantly reduces the number of instantiations required by an SMT solver to answer "unsatisfiable" for several benchmark libraries, and consequently leads to improvements over state-of-the-art implementations.**

I. Introduction

Many recent formal methods applications rely heavily on Satisfiability Modulo Theories (SMT) solvers for answering logical queries required to solve complex tasks. These systems typically are composed of multiple cooperating decision procedures, or *theory solvers*, each specialized on sets of ground constraints over some background theory. Thanks to the widespread success and applicability of SMT solvers, there has been a push to use them to handle queries based on richer encodings that include quantified formulas. Handling such formulas in a general way has been an ongoing challenge in the SMT community.

To date, E-matching, first described in [13], is the most popular and successful method used by SMT solvers for handling quantified formulas. In this method, instances of a quantified formula are generated by matching selected terms in the formula (called *matching patterns*) with ground terms in the rest of the problem. While solvers based on E-matching have had widespread success over many applications, their power is often difficult to wield. One reason is that E-matching often produces a very large number of instances, which may exhaust a solver's memory or generally cause its performance to degrade. The problem is often compounded by instances that introduce new ground terms, which subsequently trigger even more instantiations. This can lead to non-terminating *matching loops* in the worst case, in which a repeating pattern of terms causes an infinite chain of instantiation steps.

It is thus important to limit the number of instances produced as a result of E-matching. Past research has addressed this issue in various ways, including the use of user-provided matching patterns (or *triggers*) [7], and methods for recognizing or avoiding matching loops [9]. We present a new quantifier instantiation procedure that aims at decreasing the number of produced instances by decreasing the dependency of SMT solvers on E-matching. This is done by looking for instantiations that lead directly to ground conflicts or to relevant new constraints. In this scheme, the solver resorts to E-matching only when it cannot perform instantiations of this sort. Our goal is to enable the sub-module that handles quantified formulas in a SMT solver to behave more like an efficient theory solver for ground constraints. In particular, our method enables the quantifier module to influence the search performed by the main engine by reporting conflicts and propagating relevant ground constraints, as typically done by efficient theory solvers based on the DPLL(T) [14] framework.

The instantiation procedure described in this paper applies to arbitrary SMT inputs containing quantified formulas. However, it is not intended to be a comprehensive solution for handling such formulas. Instead, it is meant to supplement existing instantiation techniques in a principled manner, so that those, such as E-matching, which are currently cumbersome and expensive, are invoked as little as possible.

1) Contributions: This paper presents a new technique for quantifier instantiation in DPLL(T)-based SMT solvers that on average significantly reduces the number of instantiations required to prove a formula unsatisfiable. We give a formal argument for various properties of the technique and the instances it produces. We describe an optimized implementation that is efficient in practice. Finally, we provide detailed evidence that our implementation leads to significant improvements, according to several metrics, over state-of-the-art SMT solvers handling quantified formulas.

2) Related Work: Various works have focused on methods for discovering the unsatisfiability of quantified formulas in SMT. The first implementation of E-matching was given in the solver Simplify [7], which included various techniques such as mod-time and pattern-element optimization. These techniques were used by the SMT solver Z3 [6] and enhanced further, as described in [5]. Quantifier instantiation in DPLL(T) as implemented in the SMT solver CVC3 [3] is described in [9]. Specifying decision procedures with quantified formulas through the use of triggers is described in [8]. Techniques also exist for discovering the satisfiability of quantified formulas in SMT, including reasoning in local theory extensions [11], complete instantiation [10] and finite model finding [15].

II. Formal Preliminaries

We assume the usual notions from many-sorted first-order logic with equality (denoted by $\approx$). We fix a set $\mathbf{S}$ of *sort*

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

symbols and for every $S \in \mathbf{S}$ an infinite set of $\mathbf{X}_S$ of *variables of sort* S. We assume the sets $\mathbf{X}_S$ are pairwise disjoint and let $\mathbf{X}$ be their union. A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq \mathbf{S}$ of sort symbols and a set Σ^{f} of *(sorted) function symbols* $f^{S_1 \cdots S_n S}$, where $n \geq 0$ and $S_1, \ldots, S_n, S \in \Sigma^{\mathrm{s}}$. We drop the sort superscript from function symbols when it is clear from context or unimportant. We assume that signatures always include a Boolean sort Bool and constants $\top$ and $\bot$ of that sort (respectively, for true and false).

Given a many-sorted signature Σ, well-sorted terms, atoms, literals, clauses, and formulas with variables in $\mathbf{X}$ are defined as usual and referred to respectively as Σ-*terms*, Σ-*atoms* and so on.[1] A *ground term/formula* is a Σ-term/formula with no variables. When $\mathbf{x} = (x_1, \ldots, x_n)$ is a tuple of variables and Q is either $\forall$ or $\exists$, we write $Q\mathbf{x}\,\varphi$ as an abbreviation of $Qx_1 \cdots Qx_n\,\varphi$. If e is a Σ-term or formula and $\mathbf{x}$ has no repeated variables, we write $e[\mathbf{x}]$ to denote that e's free variables are from $\mathbf{x}$; if $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{t} = (t_1, \ldots, t_n)$ are term tuples, we write $e[\mathbf{t}]$ for the term or formula obtained from e by simultaneously replacing each occurrence of x_i in e by t_i; we write $\mathbf{s} \approx \mathbf{t}$ for the set $\{s_1 \approx t_1, \ldots, s_n \approx t_n\}$.

A Σ-*interpretation* $\mathcal{I}$ maps: each $S \in \Sigma^{\mathrm{s}}$ to a non-empty set S^I, the *domain* of S in $\mathcal{I}$, with $\mathsf{Bool}^I = \{\top, \bot\}$; each $x \in \mathbf{X}$ of sort S to an element $x^{\mathcal{I}} \in I_S$; and each $f^{S_1 \cdots S_n S} \in \Sigma^{\mathrm{f}}$ to a total function $f^{\mathcal{I}} : S_1^I \times \cdots \times S_n^I \to S^I$. A satisfiability relation between Σ-interpretations and Σ-formulas is defined inductively as usual.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only for how it interprets the variables is also in $\mathbf{I}$) and isomorphism. A Σ-formula $\varphi[\mathbf{x}]$ is T-*satisfiable* (resp., T-*unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas T-*entails* a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is T-*satisfiable* if $\Gamma \not\models_T \bot$. For a given signature Σ the *theory of equality (with uninterpreted functions)* or E, consists of the set of all Σ-interpretations. Informally, we refer to the sort and function symbols in this theory as *uninterpreted*.

A substitution σ is a mapping from variables to terms of the same sort, such that the set $\{x \mid \sigma(x) \neq x\}$, the *domain* of σ, is finite. We say that σ is a *grounding substitution* for a tuple $\mathbf{x} = (x_1, \ldots, x_n)$ of variables if σ maps each element of $\mathbf{x}$ to a ground term. If $\mathbf{t} = (t_1, \ldots, t_n)$, we write $\mathbf{x} \mapsto \mathbf{t}$ to denote the substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$; for a term or formula $e[\mathbf{x}]$, we write $e\sigma$ to denote the expression $e[\mathbf{t}]$. This notation extends to sets of formulas/terms as expected.

III. Finding Conflicts for Quantified Formulas

To handle quantified formulas, DPLL(T) solvers typically divide the input set of formulas into a set Q of quantified formulas and a set G of ground ones. To determine if $Q \cup G$ is unsatisfiable in the background theory T, they heuristically add to G selected ground instances of formulas from Q, and succeed when they have added enough instances to make G T-unsatisfiable. When G is T-satisfiable, they build a truth assignment for the atoms in G that satisfies all the formulas in G and is consistent with T. The truth assignment is represented as a set M of all the ground literals it satisfies, which we will call a *context*. In this case, a possible quantifier instantiation heuristic is to add, when possible, ground instances φ of formulas from Q that are in conflict with the current context M, in the sense that $M \cup \{\varphi\}$ is T-unsatisfiable. Adding such an instance to G will effectively force the solver to discard M and look for another context, if one exists.

This section presents a new quantifier instantiation procedure that, as described above, searches for instances of universally quantified formulas that are in conflict with the context maintained by the solver. For simplicity, we describe only a basic version of the procedure here. A more practical implementation is discussed in the next section.

For the rest of the section we fix a theory T of signature Σ, a Σ-formula $\forall\mathbf{x}\,\psi \in Q$ with $\psi[\mathbf{x}]$ quantifier-free, and a context M consisting of a T-satisfiable set of ground Σ-literals. We will use $\mathbf{T}_M$ to denote the set of all terms occurring in M.

A. Conflict Finding Instantiation Procedure

Our instantiation procedure tries to construct grounding substitutions σ for $\mathbf{x}$ such that $M \models_T \neg\psi\sigma$. We refer to σ as a *conflicting substitution for* (M, ψ). Conflicting substitutions are of interest since they suffice to show that there is no model of T that satisfies both M and $\forall\mathbf{x}\,\psi$.

Example 1: If M is $\{f(a) \not\approx g(b), b \approx h(a)\}$, then $\{x \mapsto a\}$ is a conflicting substitution for $(M, f(x) \approx g(h(x)))$. □

To simplify its presentation, we assume our procedure is run on the flat form of quantified formulas $\forall\mathbf{x}\,\psi$, defined as follows.

Definition 1: A flat form of a quantified formula $\forall\mathbf{x}\,\psi$ is an equivalent formula $\forall\mathbf{x},\mathbf{y}\,(\mu \Rightarrow \varphi)$ where

- *μ is a conjunction of equalities $x_0 \approx f(x_1, \ldots, x_n)$, which we will call the matching constraints, where $n \geq 0$ and $x_0, \ldots, x_n$ are variables from $\mathbf{x}, \mathbf{y}$;*
- *φ is a quantifier-free formula, which we will call the flattened body, whose non-ground atoms are all equalities between variables from $\mathbf{x}, \mathbf{y}$.*

A flat form of $\forall\mathbf{x}\,\psi$ can be computed by starting with $\mu = \top$ and $\varphi = \psi$ and repeatedly replacing selected terms t in $\mu \Rightarrow \varphi$ by a fresh variable x_t and adding the equation $x_t \approx t$ to μ until all non-ground terms have the form x or $f(x_1, \ldots, x_n)$.

Definition 2: Let $\mathbf{z}$ be a tuple of variables. An assignment over $\mathbf{z}$ is a set of equations of the form $z \approx t$ with z in $\mathbf{z}$ and $t \in \mathbf{T}_M$. A constrained assignment over $\mathbf{z}$ is a set $E \cup C$ where E is an assignment over $\mathbf{z}$ and C is a set of equalities and disequalities over $\mathbf{z}$. A constrained assignment A is M-feasible if $M \cup A$ is T-satisfiable.

Given the context M and a flat form $\forall\mathbf{x},\mathbf{y}\,(\mu \Rightarrow \varphi)$ of $\forall\mathbf{x}\,\psi$, our instantiation procedure will attempt to construct a constrained assignment A over the variables $\mathbf{x}, \mathbf{y}$ that summarizes the conditions under which one can build a conflicting

[1] In this formalization all atoms have the form $s \approx t$ with s and t of the same sort. Having $\approx$ as the only predicate symbol causes no loss of generality as other predicate symbols can be modeled as function symbols with return sort Bool.

proc $\mathsf{falsify}(\varphi_0, b_0)$
  if φ_0 is ground
    if $M \models_T \varphi_0 \Leftrightarrow \bar{b}_0$ **then** $\{\emptyset\}$ **else** $\emptyset$
  else if φ_0 is $x_1 \approx x_2$
    if b_0 is $\top$ **then** $\{\{x_1 \not\approx x_2\}\}$ **else** $\{\{x_1 \approx x_2\}\}$
  else if φ_0 is $\neg\varphi_1$ **then**
    $\mathsf{falsify}(\varphi_1, \bar{b}_0)$
  else if φ_0 is $\varphi_1 \vee \varphi_2$
    if b_0 is $\top$ **then**
      $\{A_1 \cup A_2 \mid A_1 \in \mathsf{falsify}(\varphi_1, b_0), A_2 \in \mathsf{falsify}(\varphi_2, b_0)\}$
    else
      $\mathsf{falsify}(\varphi_1, b_0) \cup \mathsf{falsify}(\varphi_2, b_0)$

Fig. 1. The falsify procedure. It returns a set $\mathcal{A}$ of constrained assignments such that $M \cup A \models_T (\varphi_0 \Leftrightarrow \bar{b}_0)$ for each $A \in \mathcal{A}$, where $\bar{b}_0$ denotes the complement of b_0.

proc $\mathsf{match}(S_0)$
  if S_0 is $\{y \approx f(\mathbf{z})\} \cup S_1$ **then**
    $\{A \cup \{y \approx f(\mathbf{t})\} \cup \mathbf{z} \approx \mathbf{t} \mid A \in \mathsf{match}(S_1),\ f(\mathbf{t}) \in \mathbf{T}_M\}$
  else
    $\{\emptyset\}$

Fig. 2. The match procedure. It returns a set $\mathcal{A}$ of constrained assignments such that $M \cup A \models_T S_0$ for each $A \in \mathcal{A}$.

substitution for (M, ψ). When it succeeds in building A, the procedure is also able to return one such substitution.

1) Quantifier Instantiation Procedure: A basic, unoptimized version of the procedure consists of three steps. The first step returns constrained assignments A which by construction falsify the flattened body φ; more precisely, constrained assignments A such that $M \cup A \models_T \neg\varphi$. The second step returns constrained assignments A' which by construction entail the matching constraints μ, that is, $M \cup A' \models_T \mu$. The third step considers unions of the constrained assignments $A \cup A'$ constructed in steps one and two, and tries to extract from $A \cup A'$ a grounding substitution $\mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$ such that $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A \cup A'$. If such a substitution exists, the procedure returns $\mathbf{x} \mapsto \mathbf{s}$ as a conflicting substitution for (M, ψ); otherwise, it fails. We discuss these three steps in more detail in the following.

a) Step 1: Construct constrained assignments conflicting with the flattened body φ: This step is executed by the recursive subprocedure falsify shown in Figure 1 (where for brevity we assume that the only Boolean connectives in φ are $\neg$ and $\vee$), which takes as input a subformula φ_0 of the flattened body φ, and a Boolean constant $b_0 \in \{\top, \bot\}$ indicating the polarity of φ_0 in φ, and returns a set of constrained assignments computed according to that polarity.[2] Its initial inputs are $(\varphi, \top)$.

b) Step 2: Construct constrained assignments that entail the matching constraints μ: This step constructs a set $\mathcal{A}$ of constrained assignments each of which entails μ. It does so by using the subprocedure match shown in Figure 2, which is called on the set of all the constraints S_μ in μ. For each matching constraint $z \approx f(z_1, \ldots, z_n) \in S_\mu$, the subprocedure considers all terms of the form $f(t_1, \ldots, t_n) \in \mathbf{T}_M$, and adds to A the constraints $z \approx f(t_1, \ldots, t_n), z_1 \approx t_1, \ldots, z_n \approx t_n$.

[2]Formula φ_0 has positive polarity in φ (indicated by $\top$) if and only if it occurs below an even number of $\neg$ symbols.

c) Step 3: Extract a conflicting substitution from constrained assignment: This step tries to generate a conflicting substitution for (M, φ), if there exists one. To do so, it considers all M-feasible constrained assignments $A' = A'_f \cup A'_m$, where $A'_f \in \mathsf{falsify}(\varphi, \top)$ and $A'_m \in \mathsf{match}(S_\mu)$. It partitions A' into two sets B' and C' such that the equivalence closure of B' contains at most one ground term per equivalence class. Using B', the procedure constructs a grounding substitution $\sigma = (\mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t})$, which we call a *completion of* A', by computing the equivalence closure of B', and then mapping every variable in the same equivalence class to the ground term in that class if there is one, or to an arbitrary one from $\mathbf{T}_M$ otherwise. If it succeeds in constructing a completion σ such that $M \models C'\sigma$, the procedure ends, returning the substitution $\mathbf{x} \mapsto \mathbf{s}$. Otherwise, it tries to extract a conflicting substitution from a different constrained assignment in $\mathcal{A}'$.

Example 2: To see how substitutions like σ above are computed, suppose T is E, the theory of equality, $M = \{f(a) \not\approx f(b)\}$, $B' = \{x \approx y, z \approx a, z \approx w\}$, and $C' = \{x \not\approx w\}$. Note that $A' = B' \cup C'$ is an M-feasible constrained assignment. The set B' induces the equivalence relation $\{\{x, y\}, \{w, z, a\}\}$. Adding b to the equivalence class of x leads to the grounding substitution $\sigma = \{x \mapsto b, y \mapsto b, z \mapsto a, w \mapsto a\}$ which is such that $M \models_{\mathrm{E}} C'\sigma$. □

We remark that, in our experience, guessing ground terms to add to the equivalence classes in the equivalence closure of B' in the third step of the procedure is rarely needed. The reason is that B' typically contains a *grounding equation* $z \approx t$ (with $t \in \mathbf{T}_M$) for each variable z in it. When this is not the case, it is because either z does not occur as an argument of a function symbol in the flattened form $\forall \mathbf{x}, \mathbf{y}\, (\mu \Rightarrow \varphi)$, or it is not relevant to the falsification of that formula.

We illustrate our procedure as a whole with a simple example where T is again the theory E of equality.

Example 3: Say M is $\{f(a) \not\approx g(b), b \approx h(a)\}$ and consider the formula $\forall x\, \psi$ where ψ is $f(x) \approx g(h(x))$. A flattened form of $\forall x\, \psi$ is

$$\forall x, y_1, y_2, y_3\ \underbrace{(y_1 \approx f(x) \wedge y_2 \approx h(x) \wedge y_3 \approx g(y_2))}_{\mu} \Rightarrow \underbrace{y_1 \approx y_3}_{\varphi}$$

If we run our procedure on this formula, $\mathsf{falsify}(y_1 \approx y_3, \top)$ returns the set of constrained assignments $\{\{y_1 \not\approx y_3\}\}$. The procedure then invokes $\mathsf{match}(S_\mu)$ where S_μ is $\{y_1 \approx f(x), y_2 \approx h(x), y_3 \approx g(y_2)\}$. The recursive calls of match when processing each equality in S_μ are as follows:

equation	output
	$\{\emptyset\}$
$y_3 \approx g(y_2)$	$\{\{y_3 \approx g(b), y_2 \approx b\}\}$
$y_2 \approx h(x)$	$\{\{y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a\}\}$
$y_1 \approx f(x)$	$\{\{y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a, y_1 \approx f(a)\}\}$

Let A' be the union of the (single) constrained assignments produced by falsify and match. Notice that A' is M-feasible. Splitting A' into $B' = \{x \approx a, y_1 \approx f(a), y_2 \approx h(a), y_3 \approx g(b)\}$ and $C' = \{y_2 \approx b, y_1 \not\approx y_3\}$, say, the procedure can generate (in this case only) the substitution $\sigma = \{x \mapsto a, y_1 \mapsto f(a), y_2 \mapsto h(a), y_3 \mapsto g(b)\}$. Since $M \models_{\mathrm{E}} C'\sigma$, the procedure returns the substitution $\{x \mapsto a\}$. Note that $M \models_{\mathrm{E}} f(a) \not\approx g(h(a))$, that is, $M \models_{\mathrm{E}} \neg\psi[a]$, which shows that the returned substitution is indeed conflicting. □

One can show by structural induction that the subprocedures falsify and match have the following properties.

Lemma 1: For all $A \in \mathsf{falsify}(\varphi, \top)$ and $A' \in \mathsf{match}(S_\mu)$, $M, A \models_T \neg\varphi$, and $M, A' \models_T \mu$.

Lemma 2: Let $\forall \mathbf{x}, \mathbf{y}\, (\mu \Rightarrow \varphi)$ be the flat form of $\forall \mathbf{x}\, \psi[\mathbf{x}]$. Let $A'_f \in \mathsf{falsify}(\varphi, \top)$, $A'_m \in \mathsf{match}(S_\mu)$, $A'[\mathbf{x}, \mathbf{y}] = A'_f \cup A'_m$, and $\sigma = \mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$. If $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T C$, then $M, \psi[\mathbf{s}] \models_T \neg(A' \setminus C)[\mathbf{s}, \mathbf{t}]$.

Proof: Let σ, A'_f, A'_m and A' be as above. By Lemma 1, $M, A'_f \models_T \neg\varphi$ and $M, A'_m \models_T \mu$. Thus, we have that $M, A' \models_T \mu \wedge \neg\varphi$ or, equivalently, $M, A' \models_T \neg(\mu \Rightarrow \varphi)$. By our assumption, we have that $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T C$. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t}, (A' \setminus C) \models_T \neg(\mu \Rightarrow \varphi)$ which implies that $M, (\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}] \models_T \neg(A' \setminus C)[\mathbf{s}, \mathbf{t}]$. The claim then follows by the equivalence of $(\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}]$ and $\psi[\mathbf{s}]$. ■

This justifies the correctness result for our procedure.

Proposition 1: Every substitution returned by the instantiation procedure is conflicting for (M, ψ).

Proof: Let σ, A'_f, A'_m and A' be as in Lemma 2. Recall our instantiation procedure in Step 3 partitions A' into $B' \cup C'$. We have that $M \models_T B'\sigma$ due to our construction of σ. Furthermore, by assumption the procedure returns σ only such that $M \models_T C'\sigma$. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A'$, and by Lemma 2 with $C = A'$, we have that $M, \psi[\mathbf{s}] \models_T \neg(A' \setminus A')[\mathbf{s}, \mathbf{t}]$, thus, $M, \psi[\mathbf{s}] \models_T \bot$. ■

2) Constraint-Inducing Substitutions: Even when no conflicting substitutions exist for (M, ψ), it may be useful to find other substitutions that help the solver deduce useful information about the terms in M. This can be done by relaxing one of the requirements on the substitutions returned by our instantiation procedure. Let $\sigma = \mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$ and $A' = B' \cup C'$ be as in Step 3 of the procedure, except that $M \models_T D\sigma$ does not hold for a non-empty subset $D \subseteq C'$. Since the proof of Lemma 2 does not rely on that entailment, we still have $M \cup \psi[\mathbf{s}] \models_T \neg D[\mathbf{s}, \mathbf{t}]$, even though σ is no longer conflicting for (M, ψ). We refer to σ as a *constraint-inducing substitution* for (M, ψ). If D is a conjunction of disequalities, we refer to σ as an *equality-inducing substitution* for (M, ψ). Observe that since each predicate symbol in D is applied to variables, and $\mathbf{s}$ and $\mathbf{t}$ are tuples of terms from $\mathbf{T}_M$, the entailed formula $\neg D[\mathbf{s}, \mathbf{t}]$ is a disjunction of constraints over terms in $\mathbf{T}_M$. As a consequence, it may be beneficial to generate the instance $\psi[\mathbf{s}]$ anyway since it causes the solver to deduce constraints over terms from $\mathbf{T}_M$. This contrasts with instantiations produced by E-matching, which often introduce constraints over fresh terms.

Example 4: Consider the quantified formula $\forall x\, \psi[x]$ from Example 3, and say M is $\{f(a) \approx c, d \approx g(b), b \approx h(a)\}$. Our procedure produces the same constrained assignment A' as in that example. In this case too, A' is M-feasible. However, the completion $\sigma = \{x \mapsto a,\, y_1 \mapsto f(a),\, y_2 \mapsto h(a),\, y_3 \mapsto g(b)\}$, corresponding to the partition $B' \cup C'$ of A' with $C' = \{y_2 \approx b,\, y_1 \not\approx y_3\}$, is *not* such that $M \models_{\mathrm{E}} (y_1 \not\approx y_3)\sigma$. In fact, it is not difficult to see there are no conflicting substitutions for ψ. However, M together with the instance $\psi[a]$, i.e. $f(a) \approx g(h(a))$, allows the solver to deduce that the terms $f(a)$ and $g(b)$ from $\mathbf{T}_M$ are equal. □

3) An Instantiation Strategy: A strategy can be used that produces both conflicting and constraint-inducing substitutions for a given context M and set of quantified formulas Q. First, if a conflicting substitution can be found for one quantified formula in Q, add the corresponding instance to the set of ground clauses G. This will cause the solver to backtrack some decision in M. Otherwise, if no conflicting substitution can be found, add instances corresponding to *every* constraint-inducing substitution found for each quantified formulas in Q.

IV. Practical Implementation

For greater clarity, the description of the instantiation procedure given in Section III favors simplicity over efficiency. Our actual implementation relies on one major restriction and numerous enhancements, briefly discussed in the following.

A. Restriction to the Theory of Equality

In our current implementation, the instantiation procedure does not reason modulo the actual background theory T but only modulo the theory E of equality. Concretely, this means that all function symbols in M and $\forall \mathbf{x}\, \psi$ (including arithmetic symbols) are treated as uninterpreted. This is done both for uniformity and efficiency since checking T-entailment/satisfiability is generally expensive for theories other than E. Since every theory T is a refinement of E (in the sense that it allows less interpretations), this restriction is sound: any conflicting substitution with respect to E is also conflicting with respect to a stronger theory. The obvious downside of this naïve approach is that for stronger theories the procedure returns only a coarse under-approximation of the set of conflicting substitutions for (M, φ).

Example 5: Let $M = \{f(a) \approx b,\, (g(a) \geq b{+}1) \approx \top\}$ and let $\forall \mathbf{x}\, \psi$ be $\forall x\, f(x) \approx g(x)$ where f, g, a, b are uninterpreted symbols and $\geq, +, 1$ are from the theory A of integer arithmetic. In this case, the background theory T is the union of E and A. Consider the following flat form of $\forall x\, f(x) \approx g(x)$:

$$\forall x, y_1, y_2\, (y_1 \approx f(x) \wedge y_2 \approx g(x)) \Rightarrow y_1 \approx y_2\ .$$

By treating the arithmetic symbols as symbols of E, our procedure will not discover any conflicting substitutions in this example. To see this, note that equating y_1 to $f(a)$ and y_2 to $g(a)$ in match (the only possibility) would produce the M-feasible constrained assignment $\{y_1 \not\approx y_2,\, y_1 \approx f(a),\, y_2 \approx g(a),\, x \approx a\}$. The corresponding substitution $\sigma = \{y_1 \mapsto f(a),\, y_2 \mapsto g(a),\, x \mapsto a\}$ is not conflicting for (M, ψ) in E because $M \not\models_{\mathrm{E}} (f(x) \not\approx g(x))\sigma$, so our current implementation of the procedure will return no substitutions in this case. In contrast, $M \models_{\mathrm{E} \cup \mathrm{A}} (f(x) \not\approx g(x))\sigma$ when $\geq, +, 1$ are treated as symbols of A. Hence, if our procedure did so and were able to determine the latter entailment it would be able to return the substitution $\{x \mapsto a\}$. □

We point out that reasoning modulo the actual background theory instead of E is not enough in general to return all possible conflicting substitutions, since the match sub-procedure is in fact incomplete for general theories T. To see this, observe that in A, an assignment containing $y \approx x + y_1$, $y_1 \approx 2$ will match with the term $3{+}2$, but fail to match with the equivalent term $2 + 3$. That said, for our purposes, using incomplete yet efficient theory matching and entailment tests may lead to the

proc $\mathsf{falsify}_i(\varphi_0, b_0, A, S)$
  if φ_0 is ground
    if $M \models_T \varphi_0 \Leftrightarrow \bar{b}_0$ **then** $\{(A, S)\}$ **else** $\emptyset$
  else if φ_0 is $x_1 \approx x_2$
    if b_0 is $\top$ **then**
      $\mathsf{match}_i(S\mid_{\{x_1,x_2\}}, A \cup \{x_1 \not\approx x_2\}, S \setminus S\mid_{\{x_1,x_2\}})$
    else
      $\mathsf{match}_i(S\mid_{\{x_1,x_2\}}, A \cup \{x_1 \approx x_2\}, S \setminus S\mid_{\{x_1,x_2\}})$
  else if φ_0 is $\neg\varphi_1$ **then**
    $\mathsf{falsify}_i(\varphi_1, \bar{b}_0, A, S)$
  else if φ_0 is $\varphi_1 \vee \varphi_2$
    if b_0 is $\top$ **then**
      $\bigcup_{(A',S') \in \mathsf{falsify}_i(\varphi_1, b_0, A, S)} \mathsf{falsify}_i(\varphi_2, b_0, A', S')$
    else
      $\mathsf{falsify}_i(\varphi_1, b_0, A, S) \cup \mathsf{falsify}_i(\varphi_2, b_0, A, S)$
proc $\mathsf{match}_i(S_0, A, S)$
  if S_0 is $\{y \approx f(\mathbf{z})\} \cup S_1$ **then**
    $S_0' := S_1 \cup S\mid_{\mathbf{z}};\ S' := S \setminus S\mid_{\mathbf{z}};$
    $\bigcup_{f(\mathbf{t}) \in \mathbf{T}_M} \mathsf{match}_i(S_0', A \cup \{y \approx f(\mathbf{t})\} \cup \mathbf{z} \approx \mathbf{t}, S')$
  else
    $\{(A, S)\}$

Fig. 3. The $\mathsf{falsify}_i$ and match_i procedures. We have that $M, A \models_T \neg((S_0 \setminus S) \Rightarrow \varphi_0)$ for each $(A, S) \in \mathsf{falsify}_i(\varphi_0, \top, \emptyset, S_0)$. $S\mid_V$ denotes the set of matching constraints from S whose left hand side is in V.

best performance, where conflicting substitutions are found only when it is reasonably easy for the procedure to do so.

B. Enhancements to the Basic Procedure

The most important enhancement with respect to the basic procedure described in Section III is that its three main steps are interleaved, as demonstrated in Figure 3. With respect to the basic procedure, $\mathsf{falsify}_i$ and match_i take two additional arguments: a constrained assignment A and a set of matching constraints S. Intuitively, A is the current constrained assignment we are building, and S is the matching constraints that are left to process. When considering a quantified formula with flat form $\forall \mathbf{x}.\, \mu \Rightarrow \varphi$, we initially call $\mathsf{falsify}_i$ with arguments $(\varphi, \top, \emptyset, S_\mu)$, where S_μ is the set of matching constraints from μ. This builds a set of pairs $\mathcal{A}$, such that for each $(A, S) \in \mathcal{A}$, we have $M, A \models_T \neg((S_\mu \setminus S) \Rightarrow \varphi)$. It can be shown that when $S \neq \emptyset$, the matching constraints in S do not need to be entailed when constructing a completion for A.

This procedure has several important advantages over the basic one. First, constrained assignments are built incrementally, which (although not shown here) allows us to discard a constrained assignment A as soon as it becomes M-infeasible. Second, matching constraints are processed for a variable x as soon as any constraint involving x is added to A, as in the second branch of $\mathsf{falsify}_i$ and in match_i, allowing us to eagerly determine cases where the current constrained assignment will not lead to a conflicting substitution. Third, we compute the set $\mathcal{A} = \mathsf{falsify}_i(\varphi, \top, \emptyset, S_\mu)$ lazily, which allows us to check whether there exists a conflicting substitution for a returned constrained assignment before producing the entire set $\mathcal{A}$.

C. Implementation Details

The ground theory solver maintains an equivalence relation $\equiv_M$ over the terms in $\mathbf{T}_M$ induced by the constraints in M (whereby $s \equiv_M t$ only if $M \models_{\mathrm{E}} s \approx t$). For each $t \in \mathbf{T}_M$, let $[t]_M$ denote the equivalence class of t in $\equiv_M$ and let $[\mathbf{t}]_M$ denote $([t_1]_M, \ldots, [t_n]_M)$ if $\mathbf{t} = (t_1, \ldots, t_n)$.[3] For every function symbol f of arity n in the input formula, we build an index $\mathcal{I}_f$ containing entries of the form $[\mathbf{t}]_M \mapsto f(\mathbf{t})$, mapping an n-tuple $[\mathbf{t}]_M$ of equivalence classes to some term $f(\mathbf{t}) \in \mathbf{T}_M$. The index is functional, that is, if $f(\mathbf{s}), f(\mathbf{t}) \in \mathbf{T}_M$ with $\mathbf{s} \equiv_M \mathbf{t}$ at most one of $f(\mathbf{s})$ and $f(\mathbf{t})$ is in $\mathcal{I}_f$. This data structure is used by the falsify procedure when checking entailment of ground equalities thanks to the following invariant maintained within the solver:

$$M \models_{\mathrm{E}} f(\mathbf{t}) \approx g(\mathbf{s}) \quad \text{iff} \quad \begin{cases} [\mathbf{t}]_M \mapsto f(\mathbf{u}) \in \mathcal{I}_f, \\ [\mathbf{s}]_M \mapsto g(\mathbf{v}) \in \mathcal{I}_g, \text{ and} \\ [f(\mathbf{u})]_M = [g(\mathbf{v})]_M. \end{cases}$$

To process matching constraints we build an extended index $\mathcal{J}_f$ with entries of the form $([f(\mathbf{t})]_M, [\mathbf{t}]_M) \mapsto f(\mathbf{t})$ for terms $f(\mathbf{t}) \in \mathbf{T}_M$. When considering a matching constraint $x \approx f(x_1, \ldots, x_n)$, the match procedure enumerates, modulo $\equiv_M$, the terms in $\mathbf{T}_M$ with top symbol f by traversing the index $\mathcal{J}_f$ — and backtracking whenever it determines that the constrained assignment it is constructing is not M-feasible.

Constrained assignments are represented as a pair (U, C), where U is a partial map from variables $\mathbf{x}, \mathbf{y}$ to a term they are equated to (either a representative term from $\mathbf{T}_M$ or another variable), and C is a set of flat constraints over $\mathbf{x} \cup \mathbf{y}$. Finally, formulas $\forall \mathbf{x}\, \psi$ are not actually flattened. Instead of replacing a term t in ψ with a fresh variable y, we treat t itself as y when needed.

V. Results

We implemented our instantiation procedure with the restrictions and enhancements mentioned in Section IV within the SMT solver CVC4 [1] (version 1.3). In this section, we compare the performance of our implementation against various state-of-the-art SMT solvers.[4]

We considered three different configurations of CVC4 that vary on the instantiation strategy they use. All of them apply quantifier instantiation *lazily*, that is, after the solver produces a T-satisfiable context M that propositionally satisfies the set G of current ground formulas. Given a set of *active* quantified formulas Q, each configuration of CVC4 runs one or more of the following steps in succession until a ground instance is added to G.

1) Add the instance $\psi[\mathbf{t}]$ if there exists a conflicting substitution $\mathbf{x} \mapsto \mathbf{t}$ for (M, ψ) for some $\forall \mathbf{x}\, \psi \in Q$.
2) Add the instances $\psi[\mathbf{t}]$ for a subset of the equality-inducing substitutions $\mathbf{x} \mapsto \mathbf{t}$ for (M, ψ), for each $\forall \mathbf{x}\, \psi \in Q$.
3) Add all instances based on E-matching for (M, Q).

The first configuration, which we will refer to as **cvc4**, performs Step 3 only. The second configuration, **cvc4+c**, performs Step 1 and Step 3. The third, **cvc4+ci**, performs all three steps. In Step 2, configuration **cvc4+ci** considers at most one equality-inducing substitution for each constrained assignment

[3]In the implementation, $[t]_M$ is represented by a distinguished term in it.
[4]Details can be found at http://cvc4.cs.nyu.edu/papers/FMCAD2014-qcf/.

TABLE I. NUMBER OF SOLVED UNSATISFIABLE BENCHMARKS.

Set	Class	cvc3	z3	cvc4	cvc4+c	cvc4+ci
TPTP	**EPR**	596	**840**	809	768	769
	NEQ	910	**1,406**	1,346	1,374	1,373
	PEQ	641	656	668	690	**824**
	SEQ	3,087	3,366	3,277	3,581	**3,650**
	Sub-Total	5,234	6,268	6,100	6,413	**6,616**
Isabelle	**ArrowOrder**	321	178	307	339	**371**
	FFT	**296**	277	288	291	288
	FTA	**1,124**	917	990	1,012	1,018
	Hoare	607	549	563	579	**621**
	NS_Shared	105	108	117	140	**143**
	QEpres	297	325	360	361	**362**
	StrongNorm	207	241	242	251	**253**
	TwoSquares	643	620	708	712	**719**
	TypeSafe	227	291	283	298	**307**
	Sub-Total	3,827	3,506	3,858	3,983	**4,082**
SMT-LIB	**boogie**	653	**741**	678	692	706
	simplify	2,070	**2,478**	2,334	2,358	2,360
	why	380	**385**	369	371	373
	other	304	**379**	299	300	308
	Sub-Total	3407	**3,983**	3,680	3,721	3,747
	Total	12,468	13,757	13,638	14,117	**14,445**

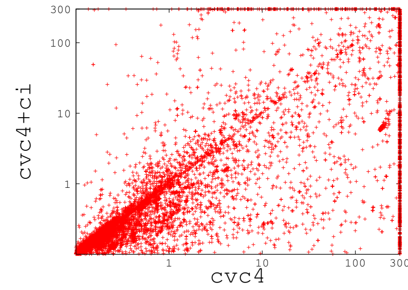
produced by the first two steps of our instantiation procedure; that is, it does not add instances for multiple completions of the same constrained assignment. Configurations **cvc4+c** and **cvc4+ci** use the naïve approach for handling interpreted theory symbols described in Section IV-A. A single run of these steps we will refer to as an *instantiation round*.

A. Comparison with SMT solvers

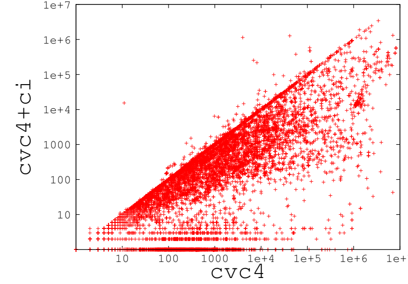
We compared these three configurations of CVC4 with the SMT solvers Z3 (version 4.3.2) [6] and CVC3 [3], both of which rely on quantifier instantiation to reason about quantified formulas. We report results on *unsatisfiable* benchmarks from various collections from the verification and automated theorem proving communities: the TPTP library (version 6.0.0) [17]; a set of benchmarks produced as proof obligations from Isabelle [4]; and SMT-LIB [2]. We considered 12,406 unsatisfiable benchmarks from TPTP which contain primarily quantified formulas and are all over the theory of equality.[5] We considered 13,041 Isabelle benchmarks (many of whom are classified as satisfiable or unknown) which also primarily contain quantified formulas, but also include both integer and real arithmetic constraints. Many of the SMT-LIB benchmarks represent software verification conditions, and make heavy use of symbols over several theories. We considered all 26,320 benchmarks from SMT-LIB that contained quantified formulas but no non-linear arithmetic constraints, which CVC4 does not yet support. Of all of these SMT-LIB benchmarks, we report results only for the 4,633 that were *non-trivial*, which we define here as taking more than 0.1 seconds to solve for at least one configuration of one solver. We ran all the experiments with a 300 second timeout per benchmark and analyzed the results according to two metrics: the performance of all solvers in terms of time and number of (unsatisfiable) benchmarks solved, and their efficiency in terms of the number of instantiations needed to answer unsatisfiable.

1) Problems Solved: Table I reports the number of benchmarks solved by the solvers for the three benchmark sets. For TPTP benchmarks, **cvc4+ci** is the overall winner, solving

[5]We did not consider TPTP benchmarks having TFF syntax (which includes theory constraints), since Z3 and CVC3 do not have a parser for this format, and no translator from this format was available.



(a) Runtime (in seconds).



(b) Reported number of instances.

Fig. 4. **cvc4+ci** vs **cvc4** over all benchmarks. Data shown on a log-log scale.

6,616 within the time limit. This is 347 more than **z3** and 516 more than **cvc4**. At least one configuration of CVC4 solves 34 unsatisfiable problems from TPTP with current rating 1.0, which is given to benchmarks that no ATP system can solve. In particular, 15 of these problems were solved using the new techniques (configurations **cvc4+c** and **cvc4+ci**) only. For Isabelle benchmarks, **cvc4+ci** is again the overall winner, solving noticeably more problems than the other solvers (4,082 vs. 3,858 for **cvc4**, 3,827 for **cvc3**, and 3,506 for **z3**). This shows that our techniques are quite effective on problems with mostly uninterpreted symbols. For SMT-LIB benchmarks, **z3** is the clear winner, with 3,983 solved problems. The new techniques yield a small improvement in performance, as **cvc4+ci** solves 67 more problems than **cvc4**. However, their performance still trails **z3**'s significantly, by 236 benchmarks. We conjecture that this is partially due to the fact that our procedure handles interpreted symbols naïvely, although several implementation differences exist between CVC4 and Z3.[6]

Overall, over the three benchmark sets, **cvc4+ci** solves more problems than any other configuration. In particular, it consistently outperforms **cvc4+c** (14,445 vs. 14,117), solving 404 problems that **cvc4+c** cannot, while **cvc4+c** only solves 76 that **cvc4+ci** cannot. This shows that computing constraint-inducing substitutions in addition to conflicting substitutions is beneficial. The scatter plot in Figure 4(a) shows that the new instantiation techniques (**cvc4+ci**) typically improve the runtime performance of CVC4—although there are several cases where they do not. Over the benchmarks they both solve, **cvc4+ci** solves 4,419 benchmarks at least 20% faster than **cvc4**, whereas **cvc4** solves 1,845 benchmarks at least 20% percent faster than **cvc4+ci**. We believe the improvement in performance is due to the reduction in the number of instances produced by **cvc4+ci**, as discussed later. Over all

[6]In particular, CVC4 does not use eager quantifier instantiation, clause deletion, or relevancy (see Section 7 of [5]) for SMT-LIB benchmarks.

TABLE II. NUMBER OF REPORTED INSTANTIATIONS FOR SOLVED UNSATISFIABLE BENCHMARKS.

	TPTP		Isabelle		SMT-LIB	
	Solved	Inst	Solved	Inst	Solved	Inst
cvc3	5,245	627.0M	3,827	186.9M	3,407	42.3M
z3	6,269	613.5M	3,506	67.0M	**3,983**	6.4M
cvc4	6,100	879.0M	3,858	119.0M	3,680	60.7M
cvc4+c	6,413	190.8M	3,983	54.0M	3,721	41.0M
cvc4+ci	**6,616**	150.9M	**4,082**	28.2M	3,747	32.4M

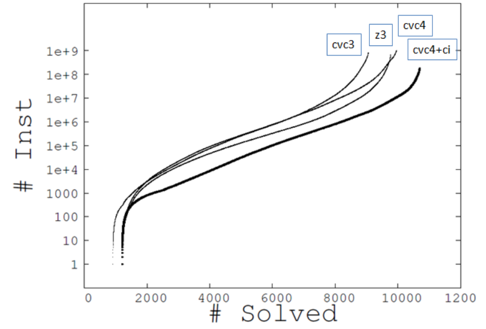
benchmark sets, **cvc4** solves 235 that **cvc4+ci** cannot solve, while **cvc4+ci** solves 1,042 benchmarks that **cvc4** cannot. At least one configuration of either **cvc4+ci** or **cvc4+c** solves 359 benchmarks that no implementation of E-matching (either Z3, CVC3, CVC4) can solve, indicating that our techniques can be used to improve the precision of SMT solvers for unsatisfiable problems containing quantified formulas.

2) Instances Generated: Table II gives the cumulative number of generated instances reported by each solver for the three benchmarks sets. For both the TPTP and the Isabelle set, in addition to solving the most benchmarks, configuration **cvc4+ci** requires by far the least number of instantiations to do so. For TPTP, **cvc4+ci** produces about 151 million instances to solve 6,616 problems, which is 5.8 times fewer than what **cvc4** requires for solving 6,100 problems. Similarly for Isabelle, **cvc4+ci** requires 28M instantiations to solve 4,082 problems, which is 4.2 times fewer than what **cvc4** requires for solving 3,858 problems. For SMT-LIB, **z3** is by far the most efficient solver, solving 3,983 problems while requiring only 6.4M instantiations. The new techniques in CVC4 reduce the instantiations by approximately half, which is less dramatic than the improvements seen on TPTP and Isabelle. This is again likely due to the prevalence of theory symbols in the encodings used by SMT-LIB benchmarks.

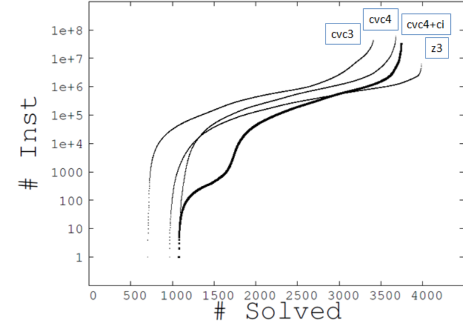
The scatter plot in Figure 4(b) compares the reported number of instances produced by configurations **cvc4** and **cvc4+ci** on the benchmarks they both solve. The plot clearly shows that **cvc4+ci** consistently requires many fewer instantiations, confirming that the instances it produces are generally effective at contributing towards finding refutations.

Figure 5 shows the cumulative number of instances reported by each of the solvers on the benchmarks they solve. For benchmarks with low theory content (from the TPTP and Isabelle libraries), the configuration **cvc4+ci** consistently produces fewer instances while solving more benchmarks than the other solvers and configurations. For SMT-LIB benchmarks, the plot shows that the configuration **cvc4+ci** uses considerably fewer instances than **z3** to solve its first 1,750 benchmarks. However, **cvc4+ci** requires more instantiations overall to solve fewer benchmarks than **z3**. This suggests that our techniques are highly effective at handling a subset of the SMT-LIB benchmarks, but require further enhancements to account for the encodings used by these benchmarks.

Table III shows a detailed view of the instances produced by the three configurations of CVC4. The first column (IR) gives the cumulative number of instantiation rounds each configuration requires for the benchmarks it solves. The remaining six columns give the percentage of instantiation rounds where they produce instances based respectively on E-matching, conflicting substitutions, and constraint-inducing substitutions; and the total number of instances produced for each of



(a) On TPTP and Isabelle benchmarks.



(b) On SMT-LIB benchmarks.

Fig. 5. Cactus plot showing the cumulative number of instantiations reported by all solvers on the benchmarks they solve.

TABLE III. DETAILS ON INSTANCES PRODUCED BY THREE CONFIGURATIONS OF CVC4.

		E-matching		Conflicting Sub.		C-Inducing Sub.	
	IR	%IR	# Inst	%IR	# Inst	%IR	# Inst
TPTP							
cvc4	71.6K	100.0	879.0M				
cvc4+c	202.0K	21.7	190.6M	78.3	158.2K		
cvc4+ci	209.0K	20.3	150.4M	76.4	159.7K	3.3	415.8K
Isabelle							
cvc4	7.0K	100.0	119.0M				
cvc4+c	18.2K	28.9	54.0M	71.1	12.9K		
cvc4+ci	21.8K	22.4	28.2M	64.0	13.9K	13.6	130.9K
SMT-LIB							
cvc4	14.0K	100.0	60.7M				
cvc4+c	51.7K	24.3	41.0M	75.7	39.1K		
cvc4+ci	58.0K	20.0	32.3M	71.6	41.5K	8.4	51.5K

these types. We can see that while configurations **cvc4+c** and **cvc4+ci** require significantly more instantiation rounds on average to answer unsatisfiable on each benchmark library, they require much fewer instances overall. Overall, a conflicting substitution was found on 77.3% of the instantiations rounds performed by **cvc4+c** and on 74.5% of the instantiation rounds performed by **cvc4+ci**. These percentages are fairly consistent across the three benchmark classes, indicating that a majority of satisfying assignments found at the ground level can be ruled out by an instance from a conflicting substitution. For **cvc4+ci**, a conflicting substitution was found on 78.5% of the instantiation rounds where a constraint-inducing substitution was not produced, which is slightly higher than the percentage found by **cvc4+c** alone (77.3%). This suggests that constraint-inducing substitutions help the solver find conflicting substitutions. In total, E-matching was called 1.57 fewer times by **cvc4+ci** than by **cvc4**, which led to a factor of 5 fewer instances produced as a result of such calls.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Overall, 12,165 of the 14,445 benchmarks that **cvc4+ci** solved required at least one instantiation round by all configurations of CVC4, and 2,520 of these 12,216 benchmarks (20.7%) could be solved by **cvc4+ci** using *only* instances resulting from conflicting and constraint-inducing substitutions. In other words, for 20.7% of the benchmarks it solves, **cvc4+ci** did not rely on E-matching at all to answer unsatisfiable. Moreover, 94 of these 2,251 benchmarks could not be solved by **cvc4** within the timeout, showing that difficult benchmarks can be solved solely by the techniques mentioned in this paper.

B. Comparison with Automated Theorem Provers

We do not give a detailed comparison with automated theorem provers, which are capable of handling benchmarks from the TPTP library, but do so using entirely different methods than SMT solvers. For a brief and informal overview, a recent (multi-strategy) run script for iProver [12] solves 6,508 unsatisfiable benchmarks from the TPTP library, while a recent run script for E [16] solves 9,751. A version of both of these scripts as well as the systems themselves were used in CASC 24, the latest competition for automated theorem provers. Using a run script devised for a similar purpose, which incorporates several configurations of E-matching as well as the techniques described here, CVC4 solves 7,227 unsatisfiable TPTP benchmarks, making CVC4 highly competitive with a state-of-the-art instantiation-based prover like iProver.

VI. CONCLUSION

We have presented a technique for quantifier instantiation in SMT that increases the ability of an SMT solver to detect unsatisfiable problems containing quantified formulas. The method relies on a more principled heuristic for choosing instances, focusing on those that communicate conflicts or relevant constraints to the ground-level sub-solver. It handles any set of quantified formulas by treating theory symbols (at worst) as uninterpreted. Our experiments show that the number of instantiations necessary to solve unsatisfiable benchmarks is on average decreased by almost an order of magnitude when compared to implementations using E-matching only. As a result, our implementation shows a noticeable improvement in performance in terms of average runtime and overall number of unsatisfiable benchmarks solved.

In future work, we plan to implement a more incremental version of our instantiation procedure to recognize conflicts while the SMT is reasoning at the ground level, which has been shown to lead to performance improvements in other implementations of quantifier instantiation in SMT [5], [9]. We also plan to extend the procedure beyond its naïve treatment of interpreted symbols to increase the number of conflicting substitution found for formulas containing such symbols. As discussed in Section IV-A, doing so requires devising fast, if incomplete, T-satisfiability tests for theories other than equality. Finally, we would like to identify language fragments and investigate extensions of our techniques that are *complete*, that is, guaranteeing the existence of a model for the input set when they fail to produce additional instances. A main challenge for this will be to ensure that the extension is also as efficient (or better) than competitive implementations of E-matching when the input problem is unsatisfiable.

ACKNOWLEDGEMENTS

We would like to thank Tim King for implementing preliminary infrastructure in CVC4 for extending these techniques to quantified formulas containing linear arithmetic.

REFERENCES

[1] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of CAV'11*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[2] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[4] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Børner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[5] L. de Moura and N. Bjørner. Efficient E-Matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[6] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[8] C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Reasoning with triggers. In P. Fontaine and A. Goel, editors, *SMT 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2013.

[9] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Bremen, Germany*, volume 4603 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2007.

[10] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[11] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 265–281. Springer, 2008.

[12] K. Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer Berlin Heidelberg, 2008.

[13] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[14] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[15] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer Berlin Heidelberg, 2013.

[16] S. Schulz. E-a brainiac theorem prover. *Ai Communications*, 15(2):111–126, 2002.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

Using Interval Constraint Propagation for Pseudo-Boolean Constraint Solving

Karsten Scheibler and Bernd Becker
University of Freiburg, Georges Koehler Allee 51, 79110 Freiburg, Germany
{scheibler,becker}@informatik.uni-freiburg.de

***Abstract*—This work is motivated by (1) a practical application which automatically generates test patterns for integrated circuits and (2) the observation that off-the-shelf state-of-the-art pseudo-Boolean solvers have difficulties in solving instances with huge pseudo-Boolean constraints as created by our application.**

Derived from the SMT solver iSAT3 we present the solver iSAT3p that on the one hand allows the efficient handling of huge pseudo-Boolean constraints with several thousand summands and large integer coefficients. On the other hand, experimental results demonstrate that at the same time iSAT3p is competitive or even superior to other solvers on standard pseudo-Boolean benchmark families.

I. Introduction

Boolean satisfiability (SAT) and extensions thereof have gained increased importance also in the area of digital circuit testing – in particular since they allow the generation of so-called high quality tests [1], [2], [3]. On the other hand, it turns out that the test pattern generation for more complex physical defects demands for abilities going beyond the boolean level. In this paper we deal with pseudo-Boolean constraints (PB constraints) arising among others in this context and corresponding solution methods. Before going into solver details, we want to give a short introduction to our test pattern generation application. More details on the general context can be found e.g. in [4]. Some more specific information on the application considered here are provided in [5].

Assume the design of an integrated circuit (IC) is given and should now go into production. When manufacturing ICs, many things may go wrong. Thus, at the end of the production process it is necessary to test if the produced ICs behave according to their specification. I.e. one applies input patterns to an IC and compares the output with an expected result. If there is a difference, the IC is faulty. Obviously, the major aim is to recognize (hopefully all) faulty circuits from a given set of freshly produced ICs. Furthermore, the test procedure for one IC should be very fast in order to be able to test many circuits in a short period of time. Therefore, testing all possible input patterns is infeasible for circuits with a reasonable number of inputs.

To be able to generate a small set of test patterns, it is necessary to make assumptions about what can go wrong during the production process. Usually, the visible effects of a specific physical defect are described in a so-called fault model. Of course there exists a bunch of different fault models – each focussing on different aspects. The stuck-at fault model [6] assumes that a faulty line on a chip always carries a logical zero or one. Although it is one of the oldest models it is still widely-used, because of its simplicity. Nonetheless, due to to latest nanoelectronic technology [7], more complex fault models have become more and more important recently – in particular the open fault model. The open fault model looks at broken lines on a chip. Here, it is assumed that the voltage of the disconnected part is determined by the voltages of the surrounding lines. This voltage is then mapped to a logical value. In our application we focus on the generation of test patterns for this kind of fault.

When generating test patterns for a circuit one starts with a set of all possible faults regarding the underlying fault model. Then a fault is taken from this set and it is tried to generate a test pattern for it. Regarding the open fault model a set of boolean and PB constraints is created. The basic idea is to encode a fault-free and a faulty version of the circuit and demanding a difference at at least one output. In the faulty version additional PB constraints are used to describe the influence of the surrounding lines[1] (as given by the layout of the circuit) which induce faulty values to the disconnected part. If this set of constraints is satisfiable, the values of the variables representing the inputs of the circuit constitute a test pattern which discovers the considered fault.

One way to solve a set of PB constraints is to translate them into a SAT instance and to employ a SAT solver to solve it. The PB constraints generated within our application may contain up to several thousand summands. As our results show, such PB constraints pose a hard problem for solvers solely relying on SAT translation techniques. Therefore, we decided to utilize the SMT solver iSAT3, which is able to handle PB constraints directly in the solver. iSAT3 supports boolean, integer- and real-valued variables and uses interval constraint propagation (ICP) to handle boolean combinations of linear and non-linear constraints.

Compared to other solvers iSAT3 performs superior on our benchmark class. On the other hand one could expect that this is not the case for other benchmark classes as well, because ICP is a general deduction mechanism not tailored for PB constraints. In order to create a solver performing superior on all benchmark classes, we decided to develop a hybrid approach which (1) uses all the merits provided by SAT translation techniques and (2) exploits the abilities of ICP to do reasoning on the arithmetic level – in particular by introducing a preprocessing technique which is not applicable on the boolean level.

The paper is structured as follows. After giving some preliminaries in Section II, we present the extensions done to the solver in Section III. In Section IV we discuss the experimental results and conclude with a summary and outlook in Section V.

II. Preliminaries

Most modern SAT solvers operate on a *conjunctive normal form* (CNF). A CNF consists of a conjunction of clauses with each clause being a disjunction of literals and a literal being a boolean variable x or its negation $\overline{x}$. One core component of a SAT solver is the *boolean constraint propagation* (BCP) [8] which is used to detect implied assignments. Everytime a clause

[1] Each summand in the PB constraint (consisting of a large integer coefficient and a boolean variable) represents the logic value of a surrounding line (boolean variable) and its influence on the disconnected part (large integer coefficient).

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

with n literals contains $n-1$ literals being already assigned to `false`, the remaining literal has to be `true` in order to retain a chance to satisfy the formula. Furthermore, today's SAT solvers add conflict clauses to the CNF to prune the search space even further – so-called *conflict-driven clause learning* (CDCL) [9].

In *SAT Modulo Theory* (SMT) the CDCL working principle is lifted to a higher level. The CNF is just a boolean abstraction of the real problem to be solved. Each literal may now represent a theory atom, e.g. $(x+y<10)$. The SAT solver works on this boolean abstraction and assigns `true` or `false` to the literals – and thus also to the theory atoms. If the SAT solver does not find a solution the underlying SMT problem is unsatisfiable – but if it finds a satisfying assignment a theory solver has to be used to check if the conjunction of theory atoms satisfying the clauses is indeed satisfiable within the theory. If this is not the case, the boolean abstraction is refined with a conflict clause which forbids the conflicting theory atoms. This is the classical scheme for handling SMT formulas. It is also abbreviated as DPLL(T) or CDCL(T) – with T being the theory used within the atoms.

iSAT3 [10] is the third implementation of the iSAT algorithm [11], [12] and uses *interval constraint propagation* (ICP, see e.g. [13]) to check the consistency of the theory atoms. But unlike classical SMT, the iSAT algorithm does not separate the consistency check of the theory atoms from the search for a satisfying assignment in the boolean abstraction. Instead, ICP is tightly integrated into the CDCL framework. The iSAT algorithm allows theory atoms to contain linear and non-linear arithmetic as well as transcendental functions, e.g. $(x^2+y^2=z^2)$, $(|v-w|<\min(v,w))$ or $(\sqrt[3]{x}+\sin y<e^z)$. Three variable types are natively supported: boolean, integer- and real-valued variables. Furthermore, ICP demands each integer- and real-valued variable to be declared with an initial interval.

iSAT3 uses an abstract syntax graph (ASG) to preprocess the given formula. In contrast to an abstract syntax tree (AST) an ASG-node may have multiple parent nodes. This allows structural hashing to natively share sub-expressions. The Tseitin-transformation [14] is used to convert the input formula to a CNF. Additionally, arithmetic constraints are decomposed into sub-expressions and simple bounds (a simple bound is a comparison between an integer- or real-valued variable and a constant). The solver core of iSAT3 is a SAT solver – extended in two directions: (1) it is able to create new literals on-the-fly during the solving process in order to map every newly deduced simple bound to a literal, (2) it executes ICP in addition to BCP. For more details refer to [10].

In the context of this paper we concentrate on constraints with pseudo-Boolean arithmetic. The linear form of such constraints has the form: $\sum_i c_i x_i \sim C$ where c_i and C are integer coefficients, x_i boolean variables and $\sim$ a relational operator with $\sim \in \{<,\leq,\geq,>\}$. Non-linear PB constraints additionally allow variables to be multiplied: $\sum_i (c_i \Pi_j x_j) \sim C$. The PB constraint $2x_1+4x_2+x_3<5$ is an example for the linear form, while $3x_1x_4+3x_2+x_3x_5<5$ represents a non-linear PB constraint.

Especially when translating PB constraints to SAT it is desired that the resulting CNF enables BCP to infer all the implications present in the original PB constraint – also denoted as *maintaining generalized arc consistency* (maintaining GAC). This means if a constraint C implies literal l under the partial assignment A then the constraint encoded in CNF C_{CNF} should allow BCP do the same: $C \wedge A \vDash l \;\Leftrightarrow\; C_{CNF} \wedge A \vdash_{BCP} l$.

In [15] BDDs, sorting networks and adder circuits were utilized to translate PB constraints into CNF. The proposed BDD-based encoding creates a BDD which describes the set of satisfying assignments of the PB constraint. Then each inner BDD-node is translated into CNF as an *if-then-else* (ITE) gate. While the BDD-based CNF encoding maintains GAC, sorting networks and adder circuits do not – this means possible implications are not recognized as early as possible which leads in most cases to a worse SAT solver performance. On the other hand the latter two encodings are compact, whereas BDD representations could have exponential size in worst case [16]. The authors of [17] proposed a different encoding which is also able to maintain GAC but stays polynomial in size. A PB constraint with n variables and the maximum integer coefficient c_{max} is encoded with $O(n^2 log(n) log(c_{max}))$ variables in $O(n^3 log(n) log(c_{max}))$ clauses. For PB constraints containing several hundreds or even thousands of variables this encoding method would generate billions of clauses and is therefore not applicable for PB constraints originating from our application. Additionally, BDDs are able to represent certain PB constraint types in linear size, while the encoding proposed in [17] stays in $O(n^3 log(n) log(c_{max}))$.

ICP operates on interval valuations and is used in iSAT3 to reason about linear and non-linear arithmetic constraints. Basically, ICP checks if a constraint is still consistent under the current (partial) assignment and tries to shrink the interval valuations of the variables occuring in the constraint if possible. In the following we illustrate the basic steps done by ICP when evaluating the PB constraint $C: 4x_1+2x_2+7x_3<10$ under the partial assignment $A: x_1=1$. With A these interval valuations are examined: $I_1=4x_1=$ [4,4], $I_2=2x_2=$ [0,2], $I_3=7x_3=$ [0,7], $I_4=$ [0,10). According to the current interval valuations C looks like this: [4,4] + [0,2] + [0,7] = [0,10). C is consistent under A, because there are still values in the intervals I_2 and I_3 such that the intersection between $I_1+I_2+I_3$ and I_4 is not empty. Furthermore, ICP is able deduce a new upper bound for I_3 (because of I_1). In order to prune definitive non-solutions I_3 is shrunk from [0,7] down to [0,6). In a next step the new upper bound for I_3 is propagated to x_3. With $I_3=7x_3 \;\wedge\; I_3=[0,6) \;\wedge\; x_3\in\mathbb{B}$ we can deduce $x_3=0$. The sum of the lower (upper) bounds of the left-hand side exceeds (falls below) the upper (lower) bound of the right-hand side, whenever the constraint is inconsistent under a partial assignment. Furthermore, the sum of the upper (lower) bound of interval I_i and the lower (upper) bounds of intervals $I_{j\neq i}$ exceeds (falls below) the upper (lower) bound of the right-hand side, whenever $x_i=0$ ($x_i=1$). Therefore, ICP is able to maintain GAC. In fact this is not surprising, because ICP does reasoning on the arithmetic level. On the other hand ICP is a general deduction mechanism and not optimized for PB constraints. Especially PB constraints like $x_1+x_2+x_3\geq 1$ are handled more efficently if their CNF translation is used – in this extreme case this would be just one clause: $(x_1 \vee x_2 \vee x_3)$. Therefore we combine ICP and BDD-based CNF translations as described in the next section.

III. iSAT3p = iSAT3 + PB Extensions

iSAT3p1: This variant is nearly identical to the underlying SMT solver iSAT3. We just extended the rewrite rules in the ASG formula preprocessing in order to normalize PB constraints to have positive coefficients on the left-hand side ($-c_i x_i \sim C$ can be rewritten to $c_i\overline{x_i} \sim C+c_i$ with $\sim\in\{<,\leq,\geq,>\}$).

iSAT3p2: We extend iSAT3p1 by adding the ability to represent PB constraints as BDDs similar to [15]. The boolean variables are ordered according to their coefficients – from the largest to the smallest. This also determines the static variable order of the BDD. The variable with the largest coefficient will be the top level variable. We use the ASG already present in iSAT3 to store the BDD as a directed acyclic graph of ITE-

nodes. These ITE-nodes are then converted to a CNF – which is handled efficiently by iSAT3p, because of its SAT solver core. A heuristic collects some statistics during BDD creation (i.e. number of created ITE-nodes, number of reused ITE-nodes because of structural hashing), estimates the expected size and decides whether the BDD creation should be aborted. If this is the case the PB constraint will be kept in its arithmetic form. The solver core will then use ICP as deduction mechanism.

iSAT3p3: On the one hand ICP is not as efficient as CNF translations for certain kinds of PB constraints. On the other hand ICP operates on the arithmetic level and therefore allows us to apply preprocessing techniques which are not applicable for CNF translations. We build on iSAT3p2 and extend it with symbolic gaussian elimination (SGE). The basic idea behind SGE is to generate helpful lemmas and add them to the formula before solving in order to strengthen ICP. The generated lemmas are not limited to PB constraints. In fact it does not matter, whether the variables occuring in a constraint are boolean, integer- or real-valued. We illustrate the idea with a small example with two constraints C_1, C_2 in the $\mathbb{R}^2$ space: $(y \geq 2.00001 \cdot x + 0.25) \wedge (y \leq 2 \cdot x)$, the initial intervals are: $x, y \in [0{,}1000000]$. Within the initial intervals C_1 and C_2 have no intersection. Therefore, the formula is unsatisfiable. ICP will continously shrink the intervals of x and y and may need millions of deductions until it finally discovers the conflict and deduces contradicting bounds for one variable. Geometrically, ICP constructs wrapping boxes around each constraint and calculates the intersection of those boxes. These boxes are parallel to the coordinate axes. Here, the idea is to generate an additional lemma which enables ICP to use an alternate coordinate axis for its wrapping box. A good choice for such an alternate axis is one of the constraints itself.

To generate such lemmas we re-use the auxiliary variables introduced during the decomposition of the original constraints into sub-expressions and simple bounds. Regarding our example the original constraints would be decomposed as follows.

$$\begin{array}{ll} C_1 \rightsquigarrow & (h_1 = y - 2.00001 \cdot x) \wedge (h_1 \geq 0.25) \\ C_2 \rightsquigarrow & (h_2 = y - 2 \cdot x) \wedge (h_2 \leq 0) \end{array}$$

Clearly, the following two equations are tautological and could be added to the formula without harm, because they just rephrase the equations above:

$$\begin{array}{l} y - 2.00001 \cdot x - h_1 = 0 \\ y - 2 \cdot x - h_2 = 0 \end{array}$$

In a system of equations, gaussian elimination replaces the problem variables step-by-step. We apply the same principle to the two tautolgies above. Assume we replace y in the second tautology with $2.00001 \cdot x + h_1$. This yields the following lemma: $0.00001 \cdot x + h_1 - h_2 = 0$. If we add it to the formula, ICP is able to deduce the conflict in a few steps: assume there is an additional auxiliary variable $(h_4 = -h_2)$ and we rewrite the lemma to $(0.00001 \cdot x + h_1 + h_4 = 0)$. Because of $(h_2 \leq 0)$ it directly follows that $(h_4 \geq 0)$. With $(h_1 \geq 0.25) \wedge (h_4 \geq 0)$ a new upper bound for x is deduced: $(x \leq -25000)$. This contradicts with the initial lower bound $(x \geq 0)$.

So in general SGE creates for every constraint a tautology containing the left-hand side of the constraint and the auxiliary variable introduced during Tseitin-transformation. Then, one of these tautologies is selected and redirected to a problem variable in order to replace this variable in all remaining tautologies. This process is repeated until no further replacements are possible. The current implementation processes the tautologies in the order of their creation in the ASG. Depending on the structure of the constraints this may result in one or more lemmas. On the one hand SGE needs enough constraints to construct useful lemmas, but on the other hand with increasing size and number of the constraints, SGE could become expensive. Therefore, a heuristic is used to decide if SGE should be aborted.

If the auxiliary variable representing the left-hand side of a constraint is used in a lemma, then this constraint will be kept – even if a BDD representation for this constraint is created later on. This allows ICP and BCP to reason about the same constraint simultaneously.

IV. Experimental Results

Solver		DBL (14)	DSL (355)	DSN (30)	OF10 (321)	$\sum$
Minisatp	SAT	8	136	-	8	
	UNS	**1**	93	-	0	
	S+U	9	229	-	8	243
SAT4JPB	SAT	9	129	[5]	221	
	UNS	0	90	**[5]**	**12**	
	S+U	9	219	[10]	233	461 [471]
Clasp	SAT	5	**138**	[8]	275	
	UNS	0	96	**[5]**	**12**	
	S+U	5	234	[13]	287	526 [539]
iSAT3p1	SAT	2	92	**[15]**	301	
	UNS	0	63	**[5]**	**12**	
	S+U	2	155	**[20]**	313	470 [490]
iSAT3p2	SAT	**13**	118	**[15]**	**307**	
	UNS	**1**	90	**[5]**	8	
	S+U	**14**	208	**[20]**	**315**	537 [557]
iSAT3p3	SAT	**13**	116	**[15]**	**307**	
	UNS	**1**	**122**	**[5]**	8	
	S+U	**14**	**238**	**[20]**	**315**	**567 [587]**
DEC-BIGINT-LIN=DBL, DEC-SMALLINT-LIN=DSL, DEC-SMALLINT-NLC=DSN, OPENFAULTS-DIV10=OF10						

Figure 1. Comparing Minisatp, Clasp and three variants of iSAT3p over a set of four benchmark families. The experiments were conducted on an Intel Xeon with 3.3 GHz with a timeout of 900 seconds and a memory limit of 8 GB.

We compared all three variants of iSAT3p against Minisatp [15] (git d91742bcd1), SAT4JPB [18] (version 2.3.5) and Clasp [19] (version 2.1.4). All three solvers were among the best solvers in the pseudo-Boolean competition 2012. Minisatp relies on the SAT solver Minisat (git 37dc6c67e2) and translates all PB constraints into SAT – either via BDD representations, sorting networks or adder circuits. SAT4JPB utilizes dedicated deduction mechanisms for PB constraints. Clasp is an answer set solver for (extended) normal logic programs.

From the pseudo-Boolean competition 2012 we selected those benchmark families containing satisfiability problems, namely: DEC-BIGINT-LIN (with 14 benchmark instances), DEC-SMALLINT-LIN (with 355 instances) and DEC-SMALLINT-NLC (with 30 instances). The first two families contain linear PB constraints, while the third contains non-linear ones. Additionally, we created a fourth benchmark family OPENFAULTS-DIV10 with 321 converted instances originating from our application. During test pattern generation we directly created the instance to be solved with ASG-nodes via the library interface of iSAT3p. In order to obtain a conjunction of PB constraints, we introduced additional auxiliary variables when needed. Furthermore, for PB constraints containing large numbers we had to divide all integer constants in the constraint by 10 – otherwise Clasp was unable to parse the benchmarks.

To compare the solvers we used an Intel Xeon with 3.3 GHz. The results are shown in Figure 1. For each benchmark family and for each solver the table shows the number of solved satisfiable (SAT) and unsatisfiable (UNS) instances as well as the sum of both (S+U). The best numbers in each category are

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

marked bold. Minisatp did not handle benchmarks with non-linear PB constraints properly and immediately returned UNKNOWN for those benchmarks. Therefore, we list the numbers for the benchmark family DEC-SMALLINT-NLC for SAT4JPB, Clasp and iSAT3p in square brackets.

The results show that the baseline solver iSAT3p1 already outperforms Minisatp, SAT4JPB and Clasp on the benchmark families DEC-SMALLINT-NLC and OPENFAULTS-DIV10, but falls somewhat behind for DEC-BIGINT-LIN and DEC-SMALLINT-LIN. To a large extent this is due to the fact that the ICP routines borrowed from iSAT3 were written to handle generic linear and non-linear arithmetic constraints and are not optimized for PB constraints. iSAT3p2 is able to close the gap for DEC-BIGINT-LIN and DEC-SMALLINT-LIN. For these two benchmark families iSAT3p2 performs equally well as SAT4JPB with its dedicated PB deduction routines. Regarding OPENFAULTS-DIV10 and DEC-SMALLINT-NLC iSAT3p2 has significant lower runtimes compared to iSAT3p1. Finally, iSAT3p3 with SGE is able to outperform the other solvers on all benchmark families. As mentioned earlier, SGE needs on the one hand enough constraints to create useful lemmas, but on the other hand may become too expensive with increasing size and number of the constraints. Therefore, SGE is only applicable to a subset of the benchmark instances – in particular those in DEC-SMALLINT-LIN. The benchmark instances in DEC-BIGINT-LIN contain between 50-100 variables, but only two constraints. OPENFAULTS-DIV10 contains constraints with several hundred variables, so SGE will be too expensive and is aborted. DEC-SMALLINT-NLC contains non-linear PB constraints and is therefore not suitable for SGE.

The results for OPENFAULTS-DIV10 emphasize that solvers solely reying on a translation into SAT are not competitive for applications which require PB constraints with many summands and large integer coefficients. While Minisatp solves only 8 instances, SAT4JPB and Clasp solve 233 and 287 instances. All variants of iSAT3p solve almost all of the 321 instances.

To sum up: the results approve the efficacy of the extensions made to iSAT3. Resorting to BDD representations, the performance especially for the two benchmark families DEC-BIGINT-LIN and DEC-SMALLINT-LIN is improved. Here we see that a BDD-based CNF translation allows more efficient deductions. Additionally, SGE strengthens ICP and improves the overall performance further such that iSAT3p3 shows superior performance on all benchmark families.

V. Conclusion and Outlook

We presented an approach for solving PB constraints with interval constraint propagation – and when possible with BDD representations of the constraints. The experimental results confirmed the efficiency of our approach. Over the complete benchmark set iSAT3p3 was able to solve 587 instances – compared to the second best solver Clasp, this is a gain of 48 instances or 8.9%. The gain is even higher if iSAT3p3 is compared to SAT4JPB and Minisatp, namely 27.3% and 133% more benchmark instances are solved compared to these two solvers. Furthermore, we observed that methods only relying on a translation to SAT fail for our benchmark class. Therefore, it is clearly beneficial to keep the ability to handle constraints in an arithmetic way.

The fact that iSAT3p has a SAT solver in its core enables us to use all the merits of a BDD-based SAT translation. At the same time, the tight integration of ICP in iSAT3p allows us to opt out for BDD creation individually for each constraint. Additionally, we presented a preprocessing technique which generates lemmas to strengthen ICP reasoning. It improves the overall performance of the solver and is therefore a good starting point for future work in this direction. Furthermore, going beyond satisfiability checking and adding the capability to optimize solutions is a challenging task we want to address as well.

Acknowledgements

The authors thank Leonore Winterer and Felix Neubauer as well as Dominik Erb and Linus Feiten for supporting this work. This work has been partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (DFG, SFB/TR 14 AVACS, `http://www.avacs.org/`) and by the Cluster of Excellence BrainLinks-BrainTools (DFG, grant number EXC 1086)

References

[1] M. Sauer, A. Czutro, I. Polian, and B. Becker, "Small-delay-fault atpg with waveform accuracy," in *ICCAD*. IEEE, 2012, pp. 30–36.

[2] D. Erb, M. A. Kochte, M. Sauer, S. Hillebrecht, T. Schubert, H.-J. Wunderlich, and B. Becker, "Exact logic and fault simulation in presence of unknowns," *Accepted for publication in ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2014.

[3] S. Eggersglüß, R. Wille, and R. Drechsler, "Improved sat-based atpg: more constraints, better compaction," in *ICCAD*, J. Henkel, Ed. IEEE/ACM, 2013, pp. 85–90.

[4] N. K. Jha and S. K. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2003.

[5] D. Erb, K. Scheibler, M. Sauer, and B. Becker, "Efficient smt-based atpg for interconnect open defects," in *DATE*, 2014, pp. 125:1–125:6.

[6] R. D. Eldred, "Test routines based on symbolic logical statements," *Journal of the ACM*, vol. 6, no. 1, pp. 33–36, 1959.

[7] V. H. Champac, R. Rodríguez-Montañés, J. A. Segura, J. Figueras, and J. A. Rubio, "Fault modelling of gate oxide short, floating gate and bridging failures in CMOS circuits," in *European Test Conf.*, 1991, pp. 143–148.

[8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[9] J. P. M. Silva and K. A. Sakallah, "Grasp - a new search algorithm for satisfiability," in *ICCAD*, 1996, pp. 220–227.

[10] K. Scheibler, S. Kupferschmid, and B. Becker, "Recent improvements in the smt solver isat," in *MBMV*, C. Haubelt and D. Timmermann, Eds. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013, pp. 231–241.

[11] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *Journal on Satisfiability, Boolean Modeling, and Computation*, vol. 1, no. 3-4, pp. 209–236, 2007.

[12] C. Herde, "Efficient solving of large arithmetic constraint systems with complex boolean structure: proof engines for the analysis of hybrid discrete-continuous systems," Ph.D. dissertation, 2011.

[13] F. Benhamou and L. Granvilliers, "Continuous and Interval Constraints," in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, 2006, pp. 571–603.

[14] G. S. Tseitin, "On the complexity of derivations in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, A. Slisenko, Ed., 1968.

[15] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into sat," *JSAT*, vol. 2, no. 1-4, pp. 1–26, 2006.

[16] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger, "A new look at bdds for pseudo-boolean constraints," *J. Artif. Int. Res.*, vol. 45, no. 1, pp. 443–480, Sep. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2444851.2444862

[17] O. Bailleux, Y. Boufkhad, and O. Roussel, "New encodings of pseudo-boolean constraints into cnf," in *SAT*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 181–194.

[18] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, pp. 59–6, 2010.

[19] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *LPNMR*, ser. Lecture Notes in Computer Science, C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483. Springer, 2007, pp. 260–265.

Patient-Specific Models from Inter-Patient Biological Models and Clinical Records

E. Tronci*, T. Mancini*, I. Salvo*, S. Sinisi*, F. Mari*, I. Melatti*, A. Massini*, F. Davì*,
T. Dierkes†, R. Ehrig†, S. Röblitz†, B. Leeners‡, T.H.C. Krüger§, M. Egli¶, F. Ille¶
*Computer Science Department Sapienza University of Rome
†Computational Systems Biology Group Zuse Institute Berlin
‡Division of Reproductive Endocrinology University Hospital Zurich
§Dpt. of Psychiatry, Social Psychiatry and Psychotherapy Hannover Medical School
¶CC Aerospace Biomedical Science & Tech. Luzern Univ. of Appl. Sciences & Arts

***Abstract*—One of the main goals of systems biology models in a health-care context is to *individualise* models in order to compute patient-specific predictions for the time evolution of species (e.g., hormones) concentrations. In this paper we present a statistical model checking based approach that, given an *inter-patient* model and a few clinical measurements, computes a value for the model parameter vector (*model individualisation*) that, with high confidence, is a global minimum for the function evaluating the mismatch between the model predictions and the available measurements. We evaluate effectiveness of the proposed approach by presenting experimental results on using the *GynCycle* model (describing the feedback mechanisms regulating a number of reproductive hormones) to compute patient-specific predictions for the time evolution of blood concentrations of E2 (Estradiol), P4 (Progesterone), FSH (Follicle-Stimulating Hormone) and LH (Luteinizing Hormone) after a certain number of clinical measurements.**

I. Introduction

Systems biology models aim at providing quantitative information about time evolution of biological species. Depending on the system at hand, many modelling approaches are currently investigated. For example, see [21], [19] for an overview on discrete as well as continuous modelling approaches, and [43] for a survey on stochastic modelling approaches. In this paper we focus on biological networks modelled with a system of *Ordinary Differential Equations (ODEs)* depending on a set of parameters as in, e.g., [33], [44], [36].

A. Motivations

One of the main goals of systems biology models in a health-care context is to individualise models in order to compute patient-specific predictions (see, e.g., [23]) for the time evolution of species of interest (e.g., hormones). In our setting, this can be done by assigning suitable values to the model parameters.

Biological models typically depend on many (easily hundreds of) parameters, whose values cannot be chosen arbitrarily because of *inter-dependency* constraints among them (see, e.g., [25]). If model parameter values are chosen ignoring such constraints, then the resulting model behaviour is biologically meaningless. Unfortunately, such constraints are usually not explicitly known and thus are not modelled.

Model identification (see, e.g., [26]) techniques are typically used to estimate model parameters by minimising mismatch with respect to experimental data. In our setting, model identification is typically accomplished by computing a value for the model parameter vector (*parameter estimation*) so that a suitable error function measuring mismatch between model predictions and experimental data is minimised. If such a value exists and is unique the model (as well as its parameter vector univocally defining the model [26]) is said *identifiable*.

Model identification techniques require availability of many measurements (see, e.g., [7]). This is difficult to achieve in a scientific trial, let alone in a clinical setting. For example, model identification for our *GynCycle* case study has been done in [36] (with the approach described in [9]) using a Pfizer database comprising 20–25 measures for each of the 4 observed hormones for 12 healthy women. This amounts to more than 1000 overall measurements. This is a typical state of affairs: in order to gather enough experimental data, model identification is carried out using measurements from several patients. This leads to the computation of a value (*default value*) for the model parameters that averages among the behaviours of many patients (see, e.g., [7], [36]). As a result, although in principle model identification techniques could be used to compute *patient-specific* model parameters, in practice, because of the large amount of measurements needed, they are typically used to compute *inter-patient* model parameters.

In a clinical setting, for each patient, only a few (say, 3) measurements are available, since measurements can be costly, invasive and time-consuming. This is far from the hundreds of measurements used in model identification. Furthermore, a fast response time is needed, since decisions resting upon our patient-specific predictions must be taken within a time compatible with the health problem being addressed.

The above considerations motivate investigation on methods and tools that can support model individualisation in a clinical setting where measurements are at a premium and a fast response time is needed.

B. Main contributions

We present a statistical model checking based approach that given an ODE based model for a biological system and a few clinical measurements for a patient, computes a patient-specific model. This enables patient-specific predictions for the time evolution of each species of interest.

As discussed in Section I-A, the above cannot be done using *model identification* approaches, since we do not have enough measurements available to attain identifiability. *Parameter estimation* approaches cannot be used either, since with such a few data they would not take into due consideration inter-dependencies among model parameters [25], thereby leading to biologically meaningless model behaviours.

We overcome such an obstacle as well as that of getting a fast on-line response time, by splitting our computation into two phases. First, an *off-line* phase that accounts for

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

parameter inter-dependencies [25] and narrows our search space to vectors of parameter values leading to *biologically meaningful* model behaviours. Second, an *on-line* phase that computes a patient-specific model by selecting a vector of parameter values in our search space. Our contributions can be summarised as follows.

Formalisation of biological admissibility: In general, to decide if time evolution of species concentration is biologically meaningful takes a domain expert. However, our goal is to build a general purpose tool that can automatically search through millions of model parameter values. Thus, we need a criterion to automatically filter out (*most* of) the parameter values leading to time evolutions that are not biologically meaningful. We provide such a criterion by defining, as *Biologically Admissible (BA)* parameter values, those entailing time evolution with a second order statistics *close enough* to that of the model default parameter values.

Off-line computation of all Biologically Admissible (BA) parameters: Our goal is to compute a set of BA values for the model parameters that encompasses as many biologically meaningful behaviours as possible, but at the same time is not too large, in order to speed up our on-line computation. Thus, taking into account that differences in values below a certain threshold are meaningless from a biological point of view, we discretise the range of values for each model parameter. In such a framework, we present a statistical model checking based algorithm that computes a set S containing *only* and (with arbitrarily high confidence) *all* BA values for our model parameters. Note that such an algorithm does not depend on patient-specific data. Thus it can be run once and for all *off-line* and its output (the set S) can be stored for further processing.

On-line computation of patient-specific predictions: Given the set S computed by our off-line algorithm above and patient-specific clinical measurements, we compute a parameter λ^* that globally minimises the mismatch between species concentrations computed using parameter λ^* and those actually measured from the patient. Simulating our model with parameter λ^* yields the patient-specific predictions we are looking for. Note that, by looking at such predictions, a domain expert can easily disregard them (and thus λ^*) if they are not biologically meaningful. Thus, returning BA parameter values that do not yield biologically meaningful time evolutions is harmless, but returning too many of them makes our tool useless. Thanks to the off-line pre-computation of the set S, our on-line algorithm has a fast response time and allows us to compute a patient-specific model from very few (say, 3) patient measurements.

Experimental evaluation: We evaluate effectiveness of our approach by presenting experimental results on using it on the *GynCycle* model in [36]. The computation time of our off-line algorithm (computing set S above) ranges from about a week to more than a month, depending on the thresholds used to check biological admissibility of model parameters and on the degree of confidence required (0.999 in our case). Starting from the set S above and from clinical measurements for E2, P4, FSH and LH, our on-line algorithm computes in a matter of *minutes* patient-specific predictions for the concentrations of all 33 species in the model (that is, also for those for which no clinical measurements are available). Our results show that: 1) most patient-specific predictions stemming from our computed BA model parameters in S are biologically meaningful (*soundness*); 2) most of the measurements in our data sets (from Pfizer database logs, [36]) can be reproduced by selecting a suitable parameter in S (*completeness*); 3) the average error of our patient-specific predictions with respect to experimental data is smaller that the one yielded by predictions based on the default model parameter.

C. Overview of the paper

Biological systems as dynamical systems: We model (Section II) a biological system with a system of ODEs defining a dynamical system (see, e.g., [37]) whose state variables comprise species concentration and whose outputs are the species that we can actually measure. Our approach is *black-box*. Accordingly we use a solver (namely, *Limex* [11]) to compute a solution to the ODEs modelling our system.

Biologically Admissible (BA) model parameters: Section III gives our notion of *biological admissibility*. First, we note that a biological model is equipped with a default value λ_0 for the (vector of the) model parameters. Such a default value is provided by the model authors and summarises the biological behaviour of many patients (*inter-patient model*). We say that a model parameter λ is BA if the model behaviours that λ entails are *highly correlated* (in a signal processing sense, [41]) to the model behaviours entailed by the model default parameter λ_0. Our approach can be easily generalised to account for models which define multiple different admissible behaviours (modelling, e.g., both healthy patients and patients with different pathologies) by providing a *set* Λ_0 of default parameters (one per behaviour class) and by considering as BA any λ entailing a model behaviour highly correlated to the behaviour entailed by at least one default parameter $\lambda_0 \in \Lambda_0$. In this paper, for simplicity of presentation, we focus on models equipped with a single default parameter (as it happens in the *GynCycle* model).

Patient Logs and Parameter Fitness: Section IV describes how we model patient data (*clinical records* or just *logs*) and our measure of fitness. Given a patient log $\mathcal{L}$ and a model parameter λ, we define the error $\eta(\mathcal{L}, \lambda)$ as the mismatch between the species concentrations computed from our model using parameter λ and those in log $\mathcal{L}$.

Off-line computation of the set of BA parameters: Along the lines of [16], we use statistical hypothesis testing to compute off-line, with high statistical confidence, the set S of BA values for the model parameters. To this end, Section V first defines our sampling space and our sampling strategy. Our sampling space is the set $\hat{\Lambda}$ of discretised values for the model parameters. Our off-line algorithm initialises S to the singleton set $\{\lambda_0\}$ containing only the default parameter, and then samples $\hat{\Lambda}$ adding all found BA parameter values to S until S stays stable for *long enough*. Upon termination, we are guaranteed that, with high statistical confidence, all BA parameter values are in S.

Individualising a Biological Model: Section VI gives our main algorithm that computes, with arbitrarily high statistical confidence, a BA parameter value λ^* which globally minimises error $\eta(\mathcal{L}, \lambda)$ when λ is constrained to take BA values. Our algorithm consists of two phases: an *off-line* phase computing, as outlined above, the set S of BA parameter values, followed by an *on-line* phase, computing a value λ^* such that $\eta(\mathcal{L}, \lambda^*)$ attains its global minimum in S. The off-line phase is computationally quite heavy. However it has to be run only once and does not depend on the patient-specific data in $\mathcal{L}$. The on-line phase is our fast response time algorithm (since S is usually quite small) to be deployed in a clinical setting.

Experimental results: Section VII describes our case study, namely the *GynCycle* model described in [36], and presents experimental results evaluating effectiveness of our approach.

D. Related work

The input to our off-line algorithm consists of a system model along with the *default value* for its parameters. The *GynCycle* model considered in our case study has been presented in [36] and the default value for its parameters has been computed in [9] using model identification (often referred to as *parameter identification* in our setting) techniques [26].

A key feature of parameter identification approaches is their ability to give information about parameter *identifiability* (see, e.g., [7] and citations thereof). For example, the parameter identification approach in [9] provides information about parameter identifiability. Gradient-based methods, as, e.g., the classical one in [24], provide a local optimum solution to the parameter estimation problem, without giving any information about parameter identifiability. Global methods, such as [27], provide a global optimum solution without any information about parameter identifiability. Heuristic approaches as evolutionary algorithms (see, e.g., [5], [40]), provide near-global optimal solutions without information about parameter identifiability. When observations are scarce, parameters usually become non-identifiable. Studying the correlation among system parameters can reduce the number of data needed for identifiability (see, e.g., [34], [25]). Our goal here is to support model individualisation from clinical measurements. This means that we need to compute model parameters from a few (say, 3) observations about a small subset (4 in our case study) of the species occurring in the model (33 in our case). Unfortunately, as discussed in Section I-A, because of scarcity of measurements, neither model identification approaches nor parameter estimation approaches can be used in our setting.

Model checking based parameter estimation approaches have been investigated for example in [18], [10], [35], [20]. Such approaches differ from ours, since they do not address the problem of automatically restricting the search to parameters leading to biologically meaningful model trajectories. This is a fundamental step in complex models as ours.

The works closest to ours are those in [38], [6] and citations thereof, where the problem of computing all (discretised) model parameter values meeting given LTL properties has been investigated. We extend such works in two directions. First, the above mentioned papers focus on piecewise affine ODE systems, whereas we can handle any (possibly) non-linear ODE system (as is the case for our *GynCycle* model [36]). Second, the above mentioned papers aim at computing a maximal set of parameters satisfying a given LTL property describing the typical behaviour for the biological system at hand. Thus, when the model changes, a new LTL property has to be provided by domain experts. Our approach infers such a system property by the default value for the model parameters using the notion of biological admissibility of Section III. This decreases the amount of input needed from domain experts, thereby alleviating one of the main problems in such a framework: formalising the properties that biologically meaningful system trajectories must satisfy.

We note that computing the set of *all* model parameter values that satisfy a given property is closely related to that of computing *all* control strategies satisfying a given property. In a discrete time setting this problem has been addressed, for piecewise affine systems and safety properties, in [30], [2], [1], [31], [4], [3], [32], [8].

Model checking techniques have been widely used in systems biology, in order to verify time behaviours. Examples are in [22], [17], [12], [14], [33]. Such approaches focus on verifying a given property for the model trajectories, whereas our main problem here is to compute *all* biologically plausible values for the model parameters.

II. Parametric Dynamical Systems

We model biological systems using dynamical systems (see, e.g., [37]). In this section we give the formal background on which our approach rests. Throughout the paper, we denote with $[n]$ the set $\{1, 2, \ldots, n\}$ of the first n natural numbers and with $\mathbb{R}^+$, $\mathbb{R}^{\geq 0}$ and $\mathbb{R}$ the sets of, respectively, positive, non-negative and all real numbers. We also denote with $(\mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0})^*$ the set of pairs $(a, b) \in \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ such that $a \geq b$.

Definition 1 (Parametric Dynamical System): A *Parametric Dynamical System* (or, simply, a Dynamical System) $\mathcal{S}$ is a tuple $(\mathcal{X}, \mathcal{Y}, \Lambda, \varphi, \psi)$, where:

- $\mathcal{X} = X_1 \times \ldots \times X_n$ is a non-empty set of *states*, called the *state space* of $\mathcal{S}$;
- $\mathcal{Y} = Y_1 \times \ldots \times Y_p$ is a non-empty set of *outputs*, called the *output value space*;
- Λ is a non-empty set of *parameters*, called the *parameter value space*;
- $\psi : \mathbb{R}^{\geq 0} \times \mathcal{X} \to \mathcal{Y}$ is the *observation function* of $\mathcal{S}$;
- $\varphi : (\mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0})^* \times \mathcal{X} \times \Lambda \to \mathcal{X}$ is the *transition map* of $\mathcal{S}$. Intuitively, $\varphi(t_2, t_1, x, \lambda)$ is the state reached by the system (with parameter values λ) at time t_2 starting from the state $x \in \mathcal{X}$ at time $t_1 \leq t_2$. Function φ must satisfy the following properties:
  - *semigroup*: for each $t_1, t_2, t_3 \in \mathbb{R}^{\geq 0}$ such that $t_1 < t_2 < t_3$, for each $\lambda \in \Lambda$, we have that $\varphi(t_3, t_1, x, \lambda) = \varphi(t_3, t_2, \varphi(t_2, t_1, x, \lambda), \lambda)$;
  - *consistency*: for each $t \in \mathbb{R}^{\geq 0}$, $x \in \mathcal{X}$ and $\lambda \in \Lambda$, we have $\varphi(t, t, x, \lambda) = x$.

Remark 1: Usually, a dynamical system comes equipped with a function space $\mathcal{U}$ that models both *controllable* inputs (e.g., treatments) as well as *uncontrollable* inputs (*disturbances*). In this paper, we do not address treatments or disturbances. Accordingly, for sake of simplicity, we omit inputs from Definition 1.

Remark 2: To simplify notation, unless otherwise stated, we assume that the set of parameters Λ has the form $\mathcal{X} \times \Gamma$ (where Γ is a non-empty set). Therefore, a parameter $\lambda = (x_0, \gamma) \in \Lambda$ embodies information about the initial state x_0 of a system trajectory. Such a system trajectory is a function of time $x(\lambda)(t)$, which, for each $t \in \mathbb{R}^{\geq 0}$, evaluates to $\varphi(t, 0, x_0, \gamma)$. In the following, abusing notation as usual, we write $x(\lambda, t)$ instead of $x(\lambda)(t)$. Analogously, we write $x_i(\lambda, t)$ $[y_i(\lambda, t)]$ for the time evolution $x_i(\lambda)(t)$ $[y_i(\lambda)(t)]$ of the ith state [output] component with parameters γ starting in x_0 from time 0.

Example 1: Dynamical systems whose dynamics is described by a system of Ordinary Differential Equations (ODEs) depending on parameters are currently of great interest as a mathematical model for biological networks (see, e.g., [13], [36]). In this paper, we will use as a case study the *GynCycle* model presented in [36]. It is a differential equation

model for the feedback mechanisms between Gonadotropin-Releasing Hormone (GnRH), Follicle-Stimulating Hormone (FSH), Luteinizing Hormone (LH), development of follicles and corpus luteum, and the production of Estradiol (E2), Progesterone (P4), Inhibin A (IhA), and Inhibin B (IhB) during the female menstrual cycle. The model aims at predicting blood concentrations of LH, FSH, E2, and P4 during different stages of the menstrual cycle. The model is intended as a tool to help in preparing and monitoring clinical trials with new drugs that affect GnRH receptors (*quantitative and systems pharmacology*). To get simulations of hormone concentrations, the system of differential equations is solved numerically.

In our *black-box* approach, the system transition map models our call to a solver (namely, *Limex* [11]) computing a solution to the ODEs defining dynamical systems in our context. This is along the lines of simulation based system level formal verification as in [42], [28], [29].

III. Biological admissibility

In general, given a value λ for the (vector of) model parameters, it takes a domain expert to decide if it holds that for each species x_i in the model, the time evolution $x_i(\lambda, t)$ is *biologically meaningful*. This stems from the fact that many parameter values lead to time evolutions for the model species that are not compatible with the laws of biology. However, our goal is to build a general purpose tool that automatically searches through millions of model parameter values. Thus, we need a criterion to automatically filter out parameter values leading to time evolutions that are not biologically meaningful. We provide such a criterion by asking that the time evolution of $x(\lambda, t)$ is *similar enough* (modulo bounded *stretch* and/or *time-shifts*) to that of $x(\lambda_0, t)$, that is the one entailed by the model default parameter value λ_0. To this end, in the following definition, we consider three measures of how similar two trajectories are (modulo bounded stretch and/or time-shift).

Given a function f from $\mathbb{R}$ to $\mathbb{R}$ and $\alpha, \tau \in \mathbb{R}$, we denote with $f^{\alpha,\tau}$ the function defined by $f^{\alpha,\tau}(t) = f(\alpha(t + \tau))$ for all t. Here, α and τ are used to model, respectively, a stretch and a shift of f. Given two functions f and g from $\mathbb{R}$ to $\mathbb{R}$, the *cross-correlation* (see, e.g., [41]) $\langle f, g\rangle(\xi)$ between f and g is a function of ξ (where $\xi \in \mathbb{R}$ is the *time lag*) defined as: $\langle f, g\rangle(\xi) = \int_{-\infty}^{+\infty} f(t)g(t + \xi)\mathrm{d}t$. We consider the *normalised zero-lag cross-correlation* function $\rho_{f,g}$, defined as $\rho_{f,g} = \frac{\langle f,g\rangle(0)}{\|f\|\|g\|}$, where $\|f\|$ and $\|g\|$ are the L^2 norms of f and g, i.e., $\sqrt{\langle f, f\rangle(0)}$ and $\sqrt{\langle g, g\rangle(0)}$. The higher $\rho_{f,g}$ the more *similar* are f and g (e.g., f and g have the same peaks). In particular, $\rho_{f,g}$ is 1 if f is equal to g up to an amplification factor.

Given a dynamical system $\mathcal{S}$ with n state variables, two parameter values λ, λ_0 for $\mathcal{S}$, and a finite horizon $h \in \mathbb{R}^{\geq 0}$, let $x_i(\lambda_0, t)$ and $x_i(\lambda, t)$ be the time evolutions of species x_i (for each $i \in [n]$) under parameters λ_0 and λ respectively. Being time evolutions, both $x_i(\lambda_0, t)$ and $x_i(\lambda, t)$ are defined for $0 \leq t \leq h$. Anyway, to easily match the above general definition of cross-correlation, we define such functions on the whole set of real numbers, as being 0 for any $t < 0$ or $t > h$.

In order to model biological admissibility, we define the following three functions (i ranges over $[n]$, $\alpha, \tau \in \mathbb{R}$):

1) normalised zero-lag cross-correlation:
$$\rho_{\lambda_0,\lambda,i}(\alpha, \tau) = \rho_{x_i(\lambda_0),x_i^{\alpha,\tau}(\lambda)}$$
2) normalised average differences:
$$\mu_{\lambda_0,\lambda,i}(\alpha, \tau) = \left|\frac{\int_0^h (x_i(\lambda_0, t) - x_i^{\alpha,\tau}(\lambda, t))\mathrm{d}t}{\int_0^h x_i(\lambda_0, t)\mathrm{d}t}\right|$$
3) normalised squared norm differences:
$\chi_{\lambda_0,\lambda,i}(\alpha) = \left|(\|x_i(\lambda_0)\|^2 - \|x_i^{\alpha,\tau}(\lambda)\|^2)\right| \;/\; \|x_i(\lambda_0)\|^2.$

The *normalised zero-lag cross-correlation* $\rho_{\lambda_0,\lambda,i}(\alpha, \tau)$ measures the similarity of the trajectories $x_i(\lambda_0, t)$ and $x_i(\lambda, t)$ as for qualitative aspects (for example, if they have the same peaks), when $x_i(\lambda, t)$ is subject to stretch α and time-shift τ. Analogously, the *normalised average differences* $\mu_{\lambda_0,\lambda,i}(\alpha, \tau)$ and the *normalised squared norm differences* $\chi_{\lambda_0,\lambda,i}(\alpha, \tau)$ are two measures of the average distance between $x_i(\lambda_0, t)$ and $x_i(\lambda, t)$, when $x_i(\lambda, t)$ is subject to stretch α and time-shift τ.

In the following, we use these functions to formalise the notion of Biologically Admissible (BA) parameter λ with respect to a default parameter λ_0. Intuitively, Definition 2 considers λ as BA if the three measures above are all above or below certain thresholds.

Definition 2 (Biologically Admissible parameter): Let λ_0, $\lambda \in \mathcal{X} \times \Lambda$ be two parameters. Let $\mathbb{A} \subseteq \mathbb{R}^+$, $\mathbb{B} \subseteq \mathbb{R}$ be two sets of real numbers such that $1 \in \mathbb{A}$ and $0 \in \mathbb{B}$. Given a tuple $\Theta = (\theta_1, \theta_2, \theta_3)$ of positive real numbers, we say that λ is Θ-*biologically admissible* with respect to λ_0, notation $adm_{\mathbb{A},\mathbb{B}}(\lambda_0, \lambda, \Theta)$, if there exist $\alpha \in \mathbb{A}$ and $\tau \in \mathbb{B}$ such that, for all $i \in [n]$: $(\rho_{\lambda_0,\lambda,i}(\alpha, \tau) \geq \theta_1) \wedge (\mu_{\lambda_0,\lambda,i}(\alpha, \tau) \leq \theta_2) \wedge (\chi_{\lambda_0,\lambda,i}(\alpha, \tau) \leq \theta_3)$.

IV. Patient logs and parameter fitness

In order to evaluate model predictions with respect to clinical records, we first formally define the notion of system log. System logs model experimental results that we get by taking system measurements. A system log consists of a sequence of time instants for each output under consideration, and, for each time instant, the corresponding measured value. This definition is motivated by the fact that, in clinical practice, different species may be measured in different time instants.

Definition 3 (System log): Let $\mathcal{S}$ be a dynamical system as in Definition 1, and $\mathcal{Y} = Y_1 \times \ldots \times Y_p$ be its p-component output value space.

An *output time set* T for $\mathcal{S}$ is the Cartesian product $T_1 \times \ldots \times T_p$, where each T_i is a finite subset (possibly empty) of $\mathbb{R}^{\geq 0}$. A T-*output log* is a map from T to $\mathcal{Y}$.

A *system log* $\mathcal{L}$ for $\mathcal{S}$ is a pair (T, z), where T is an output time set for $\mathcal{S}$, and z is a T-output log.

Example 2: As an example of system log, here we briefly describe a typical patient log for monitoring women menstrual cycle (see Example 1) that we use in our case study. Logs from 12 women from a Pfizer database considered in [36] contain measurements regarding only four hormones: Estradiol (E2), Progesterone (P4), Follicle-Stimulating Hormone (FSH), and Luteinizing Hormone (LH). These hormone concentrations are measured mostly every day from day 5 to day 28 of the menstrual cycle. In such a case, we have $T_{E2} = T_{P4} = T_{FSH} = T_{LH} = \{5, 6, 7, \ldots, 28\}$ (time here is in days). In everyday clinical practice, even a smaller set of measurements is taken. For example, in clinical treatments of fertility, only three to five blood samples (measurements) are performed during a cycle and some hormone concentrations are measured only twice. As an instance, the output time set for Estradiol could be $T_{E2} = \{1, 7, 9, 12, 23\}$ and the output time set for Progesterone could be $T_{P4} = \{1, 6\}$.

To evaluate how well a model prediction fits a system log, we consider an *error function* $\eta(\mathcal{S}, \mathcal{L}, \lambda)$, which is a real-valued map that measures to what extent predictions computed with the model $\mathcal{S}$ with parameter λ differ from measurements in the patient log $\mathcal{L}$. When the system $\mathcal{S}$ under consideration is clear from the context, we will write just $\eta(\mathcal{L}, \lambda)$ for $\eta(\mathcal{S}, \mathcal{L}, \lambda)$.

In our case study, we consider the *GynCycle* model as in Example 1 and a system log $\mathcal{L} = (T, z)$ as in Example 2. Our error function is defined as the average (over the $p = 4$ measured species) of the average error of model predictions $[y_i(\lambda, t)]$ with respect to all measurements in the patient log $[z_i(t)]$: $\eta(\mathcal{L}, \lambda) = \frac{1}{p} \sum_{i \in [p]} \frac{1}{|T_i|} \sum_{t \in T_i} \frac{|y_i(\lambda,t) - z_i(t)|}{\max\{|z_i(t)|, \zeta\}}$. Note that, as we need to average the errors for different species, we need *normalised* error functions. To this end, we consider the log observations as reference values (*relative* error), and to avoid abnormal situations, if an observation is 0, we normalise it with respect to a given small positive constant ζ. An alternative option would have been to normalise the error with respect to the length of the range of legal values for each species. Unfortunately, this option is unviable in our context, as the range of legal values for many (unobservable) species is unknown.

V. Computation of admissible parameters

The first phase of our procedure finds the set S of (with high confidence) all Biologically Admissible (BA) parameter values with respect to a default parameter λ_0 validated by the model designer as biologically meaningful. The set S is computed by checking parameter values in a finite (grid-shaped) subset $\hat{\Lambda}$ of Λ (*discretised parameter space*). This approach is justified by the fact that small differences in values are meaningless from a biological point of view.

Since the number of parameters to identify is large (75 in our case study), the discretised parameter space is huge (10^{75} if we consider 10 possible values for each parameter), thus making an exhaustive search on the discretised parameter space $\hat{\Lambda}$ unfeasible. To overcome such an obstruction, we follow an approach inspired by statistical model checking [16], [15].

A. Algorithm Outline

Algorithm 1 incrementally computes the set S of Biologically Admissible (BA) parameter values trying to find at each iteration of the **repeat** loop (lines 5–13) new BA parameter values. To do so, Algorithm 1 iteratively selects a random parameter value $\lambda \in \hat{\Lambda}$ (line 8), tests if it is BA (i.e., if $adm_{\mathbb{A},\mathbb{B}}(\lambda_0, \lambda, \Theta)$ holds) and, if this is the case, adds it to the set S of already computed BA parameter values (lines 10–11).

To check $adm_{\mathbb{A},\mathbb{B}}(\lambda_0, \lambda, \Theta)$ we compute the functions defined in Section III by numerical integration over a finite number of points. To do this, we invoke the simulator just once for any parameter value λ: given the requested output time set T and the sets $\mathbb{A}$ and $\mathbb{B}$ for the allowed stretch and time-shift factors, function $simulate(\mathcal{S}, T_{\mathbb{A},\mathbb{B}}, \lambda)$ in line 9 simulates the system $\mathcal{S}$ computing points $(t, x(\lambda, t))$ of the system trajectory for all time points in $T_{\mathbb{A},\mathbb{B}}$. Set $T_{\mathbb{A},\mathbb{B}}$ (line 4) contains all time instants for which function $adm_{\mathbb{A},\mathbb{B}}$ needs species values in order to evaluate whether parameter λ satisfies Definition 2. Function $simulate(\mathcal{S}, T_{\mathbb{A},\mathbb{B}}, \lambda)$ returns as a result a finite domain function L, such that, for any time instant $t \in T_{\mathbb{A},\mathbb{B}}$, $L_i(t)$ is the value of species x_i at time t.

Our sampling strategy selects a parameter value λ from $\hat{\Lambda} \setminus S$ with probability $\mathbf{Pr}^S[\lambda] > 0$. To speed up our procedure, we give a higher probability to parameter values "close" to those already in S (see Section V-C).

Algorithm 1 Computing the set S of BA parameters

Input: A dynamical system $\mathcal{S} = (\mathcal{X}, \mathcal{Y}, \Lambda, \varphi, \psi)$, a finite subset $\hat{\Lambda}$ of Λ, a default parameter λ_0, two real numbers $\varepsilon, \delta \in (0, 1)$, a tuple Θ of BA thresholds, two finite sets of real numbers $\mathbb{A}$ and $\mathbb{B}$ (with $1 \in \mathbb{A}$ and $0 \in \mathbb{B}$), and an output time set T

function $bioAdmPars(\mathcal{S}, \hat{\Lambda}, \lambda_0, \varepsilon, \delta, \Theta, \mathbb{A}, \mathbb{B}, T)$

1. $N \leftarrow \lceil \ln(\delta)/\ln(1-\varepsilon) \rceil$
2. $S' = \{\lambda_0\}$
3. $L_0 \leftarrow simulate(\mathcal{S}, T, \lambda_0)$
4. $T_{\mathbb{A},\mathbb{B}} \leftarrow T \cup \{t' \mid t' = \alpha(t + \tau), t \in T, \alpha \in \mathbb{A}, \tau \in \mathbb{B}\}$
5. **repeat**
6. $\quad S \leftarrow S'$
7. $\quad$ **for** $i \leftarrow 1$ **to** N **do**
8. $\quad\quad \lambda \leftarrow chooseNextParameter(\hat{\Lambda}, S)$
9. $\quad\quad L \leftarrow simulate(\mathcal{S}, T_{\mathbb{A},\mathbb{B}}, \lambda)$
10. $\quad\quad$ **if** $adm_{\mathbb{A},\mathbb{B}}(L, L_0, \Theta) \wedge \lambda \notin S$ **then**
11. $\quad\quad\quad S' \leftarrow S' \cup \{\lambda\}$
12. $\quad\quad\quad$ **break**
13. **until** $S' = S$
14. **return** S

We use Statistical Hypothesis Testing to compute S, much along the lines of [16]. Let δ and ε be two real numbers in $(0, 1)$ and $N = \lceil \frac{\ln(\delta)}{\ln(1-\varepsilon)} \rceil$. The algorithm stops when N attempts fail to find a BA parameter. Our null hypothesis $H_0(S)$ states that the probability of selecting a BA parameter value outside S is greater than ε. In other words, $H_0(S)$ states that S does not contain *all* BA parameter values. Upon termination, the algorithm rejects H_0 with statistical confidence $1 - \delta$. This means that the probability of a Type-I error (i.e., to reject H_0 when it holds) is less than $1 - \delta$. Rejecting H_0 means that the probability of selecting a BA parameter value outside $S \subseteq \hat{\Lambda}$ is less than ε.

B. Algorithm Correctness

The above considerations are the key argument to prove the following.

Theorem 1: Given a dynamical system $\mathcal{S}$ as in Definition 1, a finite subset $\hat{\Lambda}$ of Λ, a value $\lambda_0 \in \hat{\Lambda}$, a tuple Θ of biological admissibility thresholds, two real numbers ε and δ in $(0, 1)$, and two finite sets of real numbers $\mathbb{A}$ and $\mathbb{B}$ (with $1 \in \mathbb{A}$ and $0 \in \mathbb{B}$), Algorithm 1 is such that:

1) it terminates in $\mathcal{O}(N|\hat{\Lambda}|)$ steps, where $N = \lceil \frac{\ln \delta}{\ln(1-\varepsilon)} \rceil$;

2) upon termination, it computes a set $S \subseteq \hat{\Lambda}$ of Θ-Biologically Admissible parameter values;

3) set S is such that, with confidence $1 - \delta$: $\mathbf{Pr}^S[\{\lambda \in \hat{\Lambda} \setminus S \mid adm_{\mathbb{A},\mathbb{B}}(\lambda_0, \lambda, \Theta)\}] < \varepsilon$.

The computational complexity of Algorithm 1 depends on the fact that, in order to find a BA parameter, we make at worst N attempts and, in principle, all discretised parameter values can be BA. As a consequence, the worst running time of Algorithm 1 is worse than an exhaustive search over $\hat{\Lambda}$. We remark, however, that the *average* running time is, in general, much better than that of an exhaustive search, since the set of BA parameters is very small compared with the size of the whole discretised parameter space. As a matter of fact, the algorithm stops with high probability in a reasonable time (see Section VII-B) by failing to find a new BA parameter value.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

C. Parameter Probability Space

The probability distribution that we consider over the parameter space $\hat{\Lambda}$ is parametric to the set S of BA parameter values computed so far, and it is defined in such a way that parameter values that are close to values in S are most likely to be chosen. This speeds up (with respect to, e.g., uniform sampling) the finding of new BA parameter values.

Given a set S, we choose the next value λ to examine as follows:

1) We randomly choose $\lambda' \in S$ uniformly at random.

2) We randomly choose the maximum number h of components in which λ will differ from λ'. In this case, the set $[n]$ is considered distributed as a power-law of the form $\mathbf{Pr}[h] = ah^{-b}$, with $b > 1$ and a being a normalisation constant. This implies that, with high probability, λ will differ from λ' in a small number of components.

3) We randomly choose a subset of h different components in $[n]$, assuming a uniform distribution over the set of subsets of cardinality h, $\mathcal{P}_h([n])$, that is $\{X \subseteq [n] \mid |X| = h\}$.

4) For each component i, we choose a value $\lambda_i \in \hat{\Lambda}_i$ uniformly at random.

This sampling technique defines a probability space $(\hat{\Lambda}, \mathcal{P}(\hat{\Lambda}), \mathbf{Pr}^S)$ parametric with respect to a set $S \subseteq \hat{\Lambda}$. By multiplying the (conditional) probabilities of steps 1)–4) above, we have: $\mathbf{Pr}^S[\lambda] = \frac{1}{|S|} \sum_{\lambda' \in S} a \, |d(\lambda, \lambda')|^{-b} \binom{n}{|d(\lambda,\lambda')|}^{-1} \prod_{i \in d(\lambda,\lambda')} \frac{1}{|\hat{\Lambda}_i|}$, where $d(\lambda, \lambda')$ is the set of the components on which λ and λ' differ. Note that $\mathbf{Pr}^S[\lambda]$ is non-zero for all λ.

VI. Computation of patient-specific parameters

Once the set S of (almost) all Biologically Admissible (BA) parameters has been computed by the *off-line* procedure described in Section V, *patient-specific parameters* can be efficiently computed. Given a patient log $\mathcal{L}$, the patient-specific parameter for $\mathcal{L}$ is the parameter λ^* that minimises $\eta(\mathcal{L}, \lambda)$, that is the parameter that minimises model prediction errors with respect to the patient measurements in $\mathcal{L}$.

Since S contains with *arbitrary high confidence* all BA parameters, we just compute the value $\lambda^* = \mathit{argmin}_{\lambda \in S} \eta(\mathcal{S}, \mathcal{L}, \lambda)$ to get, with the same confidence, a BA parameter value λ^* that minimises $\eta(\lambda, \mathcal{L})$ over $\hat{\Lambda}$. This procedure is intended to be an *on-line* computation to be used in everyday clinical practice.

Theorem 2: Let S be the set of BA parameters computed by Algorithm 1 taking as input a dynamical system $\mathcal{S}$, a tuple Θ of biological admissibility thresholds, a finite subset $\hat{\Lambda}$ of the parameter space Λ, a default parameter value $\lambda_0 \in \hat{\Lambda}$, a probability threshold ε, a confidence level δ, and finite sets $\mathbb{A}$ and $\mathbb{B}$. Given a patient log $\mathcal{L} = (T, z)$, the parameter value $\lambda^* = \mathit{argmin}_{\lambda \in S} \eta(\mathcal{S}, \mathcal{L}, \lambda)$ is such that, with confidence $(1-\delta)$, $\mathbf{Pr}^S[\{\lambda \in \hat{\Lambda} \setminus S \mid \eta(\mathcal{L}, \lambda) < \eta(\mathcal{L}, \lambda^*)\}] < \varepsilon$.

Remark 3: Once the *off-line* set S of (almost) all BA parameters has been computed (once and for all), the computation of $\lambda^* = \mathit{argmin}_{\lambda \in S} \eta(\mathcal{L}, \lambda)$ is linear in the size of S, which in turn is very small with respect to $\hat{\Lambda}$.

VII. Experimental results

The effectiveness of our approach has been evaluated on the *GynCycle* model in [36]. Such a model has 114 parameters, 75 of which are patient-specific (at least for our purposes), and consists of 41 differential equations defining the time evolution of 33 species. We implemented our tool in the C programming language and connected it with the *Limex* solver [11] integrating the Ordinary Differential Equations (ODEs) defining our model.

A. Experimental setting

All experiments have been carried out on a cluster of Linux machines each one equipped with two Intel(R) Xeon(R) CPU @ 2.27GHz and 24GB of RAM.

We set the probability threshold ε and the confidence level δ to 10^{-3}. Set $\mathbb{A}$ (see Definition 2 in Section III) comprises all stretch factors α multiple of 0.1, from 0.9 to 1.1. Set $\mathbb{B}$ (see Definition 2 in Section III) comprises all time-shifts τ multiple of 2 hours, from -5 days to $+5$ days. We set constant ζ (see Section IV) to 10^{-4} to avoid division by zero during normalisation. The discretisation $\hat{\Lambda}$ of Λ has been obtained by uniformly discretising the range of each parameter into 10 or 3 values. Cross-correlations, averages and L^2 norms are computed on a discretisation of the time evolutions with values every 15 minutes. As for the individualisation of our model we used the very same Pfizer data in [36] about 12 women.

B. Experimental results

1) Off-line computation of admissible parameters: Table I shows the computation time and the size of the set S of computed Biologically Admissible (BA) parameters for different runs of our off-line algorithm, using different configurations for biological admissibility thresholds $\theta_1, \theta_2, \theta_3$ (see Section III).

run id	θ_1	θ_2	θ_3	discr. steps	$\lvert S \rvert$	CPU time
r1	0.6	0.5	0.5	10	3940	~ 31 days
r2	0.6	0.4	0.4	10	3504	~ 29 days
r3	0.5	0.7	0.7	10	6989	~ 147 days
r4	0.5	0.5	0.5	10	6406	~ 167 day
r5	0.7	0.3	0.3	3	126	~ 6 days

TABLE I: Off-line: Size of the set of BA parameters and computation time.

Parts of such runs have been executed with a parallel version of our algorithm, which is still under development. Other parts have been executed with our stable sequential algorithm. In order to allow comparisons, we ensure homogeneity by reporting in Table I all times as if we were running our sequential algorithm. Data in Table I should be read with some caution since, being generated by a probabilistic algorithm implementing the sampling process described in Section V, different runs may yield different results as for computation time and size of set S.

As we can see from Table I, the off-line computation may take several days of intensive computation. On the other hand, it only has to be run once, since it does not depend on the patient log being considered. The RAM usage is negligible and the disk storage requirements are perfectly reasonable (tens of GB) for today standards.

2) On-line computation of patient-specific parameters: To evaluate the improvement that we obtain in species predictions, we consider patient $p2$ in the Pfizer data set and its associated log $\mathcal{L}_2$. The average error $\eta(\lambda_0, \mathcal{L}_2)$ obtained by using the default parameter λ_0 is 61.9%.

Table II shows results when *only three* observations (at days 8, 11, and 15 of the patient menstrual cycle) are used to compute our predictions for patient $p2$.

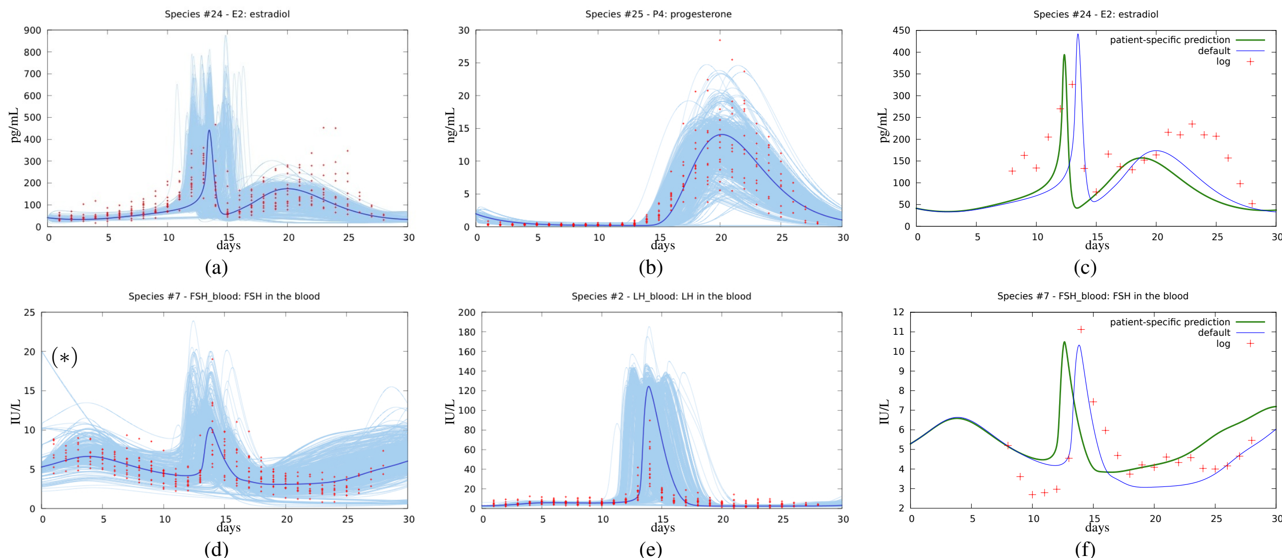


Fig. 1: (a), (b), (d), (e): all system trajectories under admissible parameters, as computed in run $r3$ for, respectively, E2, P4, FSH, LH (dark blue curves denote trajectories under default parameter). (c), (f): patient-specific prediction (green curves) for patient $p2$ vs. default prediction (blue curves) for, respectively, E2 and FSH.

run id	CPU time	avg. error	error red.	error red.%	biol. meaningful
r1	8m35s	56.0%	5.9	9.5%	yes
r2	5m06s	55.7%	6.1	9.9%	yes
r3	39m20s	55.0%	6.9	11.2%	yes
r4	36m5s	55.4%	6.5	10.5%	yes
r5	0m23s	61.9%	0.0	0.0%	yes

TABLE II: On-line: Error reduction using λ^* for patient $p2$.

The table shows CPU time and effectiveness of our on-line algorithm, when run with the same configurations for biological admissibility thresholds $\theta_1, \theta_2, \theta_3$ as in Table I. Column "average error" gives the minimum value of $\eta(\lambda, \mathcal{L}_2)$ for $\lambda \in S$, where S is the set of BA parameters computed by the off-line algorithm (as shown in the corresponding rows of Table I). Column "error reduction" shows the value of $(\eta(\lambda_0, \mathcal{L}_2) - \eta(\lambda, \mathcal{L}_2)) / \eta(\lambda_0, \mathcal{L}_2)$. Column "biologically meaningful" shows always "yes", as all trajectories we found are biologically meaningful, even though we cannot ensure a priori that all BA parameters will yield biologically meaningful trajectories.

Results show that the on-line computation completes within minutes, thereby yielding a fast on-line response time as required in a clinical setting. Runs $r3$ and $r4$ have been executed on a machine with an external storage device: their longer computation times are due to slower I/O. RAM requirements are negligible.

C. Discussion

1) Experimental soundness and completeness of biological admissibility: We experimentally evaluate *soundness* and *completeness* of our notion of biological admissibility, using reference values from the literature (e.g., [39]). To this end, Figures 1a, 1b, 1d and 1e show the trajectories for hormones E2, P4, FSH and LH (for which measurements are available in our Pfizer data-set) obtained by running the *GynCycle* model on all parameter values computed by our *off-line* algorithm in run $r3$. We see that most of such trajectories are biologically meaningful, being in agreement with the trajectories in [39]. This shows (experimentally) *soundness* of our biological admissibility notion. Furthermore, most of our Pfizer measurement data (red crosses in Figures 1a, 1b, 1d and 1e) lie within the region covered by our trajectories. This shows (experimentally) *completeness* of our biological admissibility notion.

An example of biologically *not* meaningful trajectory is denoted with $(*)$ in Figure 1d. Also, Figure 1a shows that not all Pfizer data are covered by our trajectories. This state of affairs is to be expected, since both biological admissibility and our off-line algorithm are based on statistical notions (signal second order statistics and statistical model checking, respectively), and clinical measurements might be noisy.

2) Error reduction in patient-specific predictions: The error reductions reported in Table II show that our proposed approach enables effective patient-specific predictions even in a clinical setting, where the measurements are at a premium (we used only three observations). Figures 1c and 1f give an example of the predictions of, respectively, E2 (Estradiol) and FSH for patient $p2$, and compare them with the default predictions and actual measurements in the patient log. The achieved error reduction is of about 10%. This value has a relevant impact from a clinical standpoint, as it can move hormone peaks (which are among the main fertility/infertility indicators) by several days (see Figures 1a, 1b, 1d and 1e).

The lack of error reduction shown in the single case where the minimum cross-correlation is 0.7 is due to the fact that the only BA parameters found by our off-line algorithm are very close to the default parameter. On the other hand, the first row of Table I is more liberal in considering parameters as BA. As a result, that process was able to find more parameter values in less time (possibly including model parameters leading to model behaviours which are not biologically meaningful).

VIII. Conclusions

We have presented a method to effectively compute patient-specific predictions from an ODE-based biological model and

clinical records. We overcome the main obstacles in our clinical setting (scarcity of measurements and fast response time) with an approach resting on three main pillars: first, a formalisation of the notion of *biological admissibility* that allows us to automatically filter out most parameter values that do not lead to biologically meaningful system trajectories; second, a statistical model checking algorithm that, with arbitrarily high confidence, computes *off-line* the set S of *all* (discretised) Biologically Admissible parameter values; third, an *on-line* algorithm that computes from S the best prediction with the available data. We are currently developing a parallel version for the presented algorithms.

Acknowledgments

This work has been partially supported by the EC FP7 project PAEON (Model Driven Computation of Treatments for Infertility Related Endocrinological Diseases, 600773). We are grateful to the anonymous reviewers for their comments.

References

[1] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci. Automatic control software synthesis for quantized discrete time hybrid systems. In *Proc. of 51th CDC*, pages 6120–6125. IEEE, 2012.

[2] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci. On model based synthesis of embedded control software. In *Proc. of 12th EMSOFT*, pages 227–236. ACM, 2012.

[3] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci. A map-reduce parallel approach to automatic synthesis of control software. In *Proc. of SPIN*, volume 7976 of *LNCS*, pages 43–60, 2013.

[4] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci. On-the-fly control software synthesis. In *Proc. of SPIN*, volume 7976 of *LNCS*, pages 61–80, 2013.

[5] E. Balsa-Canto, M. Peifer, J. R. Banga, J. Timmer, and C. Fleck. Hybrid optimization method with general switching strategy for parameter estimation. *BMC Systems Biology*, 2:26, 2008.

[6] J. Barnat, L. Brim, D. Šafránek, and M. Vejnár. Parameter Scanning by Parallel Model Checking with Applications in Systems Biology. In *Proc. of HiBi/PDMC*, pages 95–104. IEEE, 2010.

[7] O-T. Chis, J. R. Banga, and E. Balsa-Canto. Structural identifiability of systems biology models: A critical comparison of methods. *PLoS ONE*, 6(11), 2011.

[8] G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Synchronized regular expressions. *Acta Inf.*, 39(1):31–70, 2003.

[9] T. Dierkes, S. Röblitz, M. Wade, and P. Deuflhard. Parameter identification in large kinetic networks with bioparkin. *CoRR*, abs, 2013.

[10] R. Donaldson and D. Gilbert. A model checking approach to the parameter estimation of biochemical pathways. In *Proc. of 6th CMSB 2008*, volume 5307 of *LNCS*, 2008.

[11] R. Ehrig, U. Nowak, L. Oeverdieck, and P. Deuflhard. Advanced extrapolation methods for large scale differential algebraic problems. In *High Performance Scient. and Eng. Comp.*, LNCSE, 1999.

[12] H. Gong, P. Zuliani, A. Komuravelli, J. R. Faeder, and E. M. Clarke. Analysis and verification of the hmgb1 signaling pathway. *BMC Bioinformatics*, 11(S-7):S10, 2010.

[13] H. Gong, P. Zuliani, A. Komuravelli, J. R. Faeder, and E. M. Clarke. Computational modeling and verification of signaling pathways in cancer. In *Proc. of 4th ANB*, volume 6479, pages 117–135, 2010.

[14] H. Gong, P. Zuliani, Q. Wang, and E. M. Clarke. Formal analysis for logical models of pancreatic cancer. In *Proc. of 50th CDC*, pages 4855–4860. IEEE, 2011.

[15] R. Grosu and S. A. Smolka. Quantitative model checking. In *Preliminary Proc. of ISoLA*, pages 165–174, 2004.

[16] R. Grosu and S. A. Smolka. Monte carlo model checking. In *Proc. of TACAS*, pages 271–286, 2005.

[17] J. Heath, M. Z. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theor. Comput. Sci.*, 391(3):239–257, 2008.

[18] F. Hussain, R. G. Dutta, S. K. Jha, C. J. Langmead, and S. Jha. Parameter discovery for stochastic biological models against temporal behavioral specifications using an sprt based metric for simulated annealing. In *Proc. of 2nd ICCABS*, pages 1–6. IEEE, 2012.

[19] B. Ingalls and P. Iglesias. *Control Theory and Systems Biology*. MIT Press, 2009.

[20] S. Jha, A. Donze, R. Khandpur, J. Dutta-Moscato, Q. Mi, Y. Vodovotz, G. Clermont, and C. Langmead. Parameter estimation and synthesis for systems biology: New algorithms for nonlinear and stochastic models. *Journal of Critical Care*, 26(2), 2011.

[21] H. De Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9:67–103, 2002.

[22] M. Kwiatkowska, G. Norman, and D. Parker. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):14–21, 2008.

[23] C. J. Langmead. Generalized queries and bayesian statistical model checking in dynamic bayesian networks: Application to personalized medicine. In *Proc. of CSB*, pages 201–212, 2009.

[24] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Math*, 2:164–168, 1944.

[25] Pu Li and Quoc D. Vu. Identification of parameter correlations for parameter estimation in dynamic biological models. *BMC Systems Biology*, 7(1):91+, 2013.

[26] Lennart Ljung. *System Identification (2Nd Ed.): Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[27] S. Stahl M. Brusco. *Branch-and-Bound Applications in Combinatorial Data Analysis*. Statistics and Computing. Springer, 2005.

[28] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci. System level formal verification via model checking driven simulation. In *Proc. 25th CAV*, volume 8044 of *LNCS*, pages 296–312, 2013.

[29] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. System level formal verification via distributed multi-core hardware in the loop simulation. In *Proc. of PDP*, 2014.

[30] F. Mari, I. Melatti, I. Salvo, and E. Tronci. Synthesis of quantized feedback control software for discrete time linear hybrid systems. In *Proc. of 23rd CAV*, volume 6174 of *LNCS*, pages 180–195, 2010.

[31] F. Mari, I. Melatti, I. Salvo, and E. Tronci. Undecidability of quantized state feedback control for discrete time linear hybrid systems. In *Proc. of ICTAC*, volume 7521 of *LNCS*, pages 243–258, 2012.

[32] F. Mari, I. Melatti, I. Salvo, and E. Tronci. Model based synthesis of control software from system level formal specifications. *ACM TOSEM*, 23(1):6:1–6:42, 2014.

[33] N. Miskov-Zivanov, P. Zuliani, E. M. Clarke, and J. R. Faeder. Studies of biological networks with statistical model checking: Application to immune system cells. In *Proc. of BCB*, pages 728–729. ACM, 2007.

[34] A. Raue, C. Kreutz, T. Maiwald, J. Bachmann, M. Schilling, U. Klingmüller, and J. Timmer. Structural and practical identifiability analysis of partially observed dynamical models by exploiting the profile likelihood. *Bioinformatics*, 25(15):1923–1929, August 2009.

[35] A. Rizk, G. Batt, F. Fages, and S. Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In *Proc. of 6th CMSB*, pages 251–268, 2008.

[36] S. Röblitz, C. Stötzel, P. Deuflhard, H. M. Jones, D.-O. Azulay, P. van der Graaf, and S. W. Martin. A mathematical model of the human menstrual cycle for the administration of gnrh analogues. *Journal of Theoretical Biology*, 321:8–27, 2013.

[37] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems. (2nd Edition)*. Springer, New York, 1998.

[38] A. Streck, A. Krejci, L. Brim, J. Barnat, D. Safranek, M. Vejnar, and T. Vejpustek. On parameter synthesis by parallel model checking. *IEEE/ACM Trans. on Comput. Biology and Bioinf.*, 9(3):693–705, 2012.

[39] R. Stricker, R. Eberhart, M.C. Chevailler, F. A. Quinn, P. Bischof, and R. Stricker. Establishment of detailed reference values for luteinizing hormone, follicle stimulating hormone, estradiol, and progesterone during different phases of the menstrual cycle on the abbott architect analyzer. *Clin. Chem. Lab. Med.*, 44(7):883–887, 2006.

[40] J. Sun, J. M. Garibaldi, and C. Hodgman. Parameter estimation using metaheuristics in systems biology: A comprehensive review. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 9(1):185–202, 2012.

[41] Saeed V. Vaseghi. *Advanced Digital Signal Processing and Noise Reductio*. John Wiley & Sons, 2006.

[42] G. Verzino, F. Cavaliere, F. Mari, I. Melatti, G. Minei, I. Salvo, Y. Yushtein, and E. Tronci. Model checking driven simulation of sat procedures. In *Proc. of 12th SpaceOps*, 2012.

[43] D. J. Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman and Hall/CRC, 1 edition, April 2006.

[44] P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367, 2013.

Reducing CTL-live Model Checking to First-Order Logic Validity Checking

Amirhossein Vakili and Nancy A. Day
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{avakili,nday}@uwaterloo.ca

***Abstract*—Temporal logic model checking of infinite state systems without the use of iteration or abstraction is usually considered beyond the realm of first-order logic (FOL) reasoners because of the need for a fixpoint computation. In this paper, we show that it is possible to reduce model checking of a finite or infinite Kripke structure that is expressed in FOL to a validity problem in FOL for a fragment of computational tree logic (CTL), which we call *CTL-live*. CTL-live includes the CTL connectives that are traditionally used to express liveness properties. Our reduction can form the basis for methods that use FOL reasoning techniques directly to accomplish model checking of CTL-live properties without the need for fixpoint operators, transitive closure, abstraction, or induction.**

I. Introduction

Model checking is the problem of checking whether a Kripke structure satisfies a temporal logic formula [1]. Model checking has been used extensively to verify and find bugs in finite state systems. To deal with the growing complexity of software and hardware systems, we need methods that can analyze more abstract models so that we can discover errors earlier in the development process. The progress in SMT (satisfiability modulo theories) solvers [2] has turned first-order reasoners into powerful, efficient verification tools. In this paper, we examine the challenge of using first-order logic (FOL) to express the temporal logic model checking problem for models described in FOL.

Existing model checking methods that use first-order reasoners can be divided into two major categories: 1) bounded model checking (e.g., [3], [4]) and 2) unbounded model checking (e.g., [5], [6]). Bounded methods check whether a property holds for a certain length of execution path by creating a formula consisting of the transition relation expanded to the desired bound. Since the bound is finite, the problem can be expressed in FOL, therefore, FOL reasoners can be used to solve the entire bounded (and therefore incomplete) model checking problem at one time. Unbounded methods call a FOL reasoner multiple times iteratively to traverse the reachable state space. This iteration can result in parts of the reasoning being redone multiple times. These methods are mostly used for safety properties; for infinite systems, termination (without approximation) is guaranteed only in the case where the property is violated. FOL reasoners, such as SMT solvers, have not been used to solve an entire unbounded model checking problem in one call because model checking is a question of reachability within a graph (in this case a Kripke structure), and the reachability relation (transitive closure) is not expressible in FOL. Therefore, temporal logic model checking for infinite state systems without the use of iteration or abstraction is usually considered beyond the realm of FOL reasoners.

Our contribution is to show that model checking an interesting fragment of computational tree logic (CTL) [7], which we call *CTL-live*, is reducible to validity checking in FOL; in other words, model checking a CTL-live property of a Kripke structure can be done completely using deductive techniques of FOL. Thus, some reachability queries can be answered using a FOL reasoner even though the reachability relation itself is not expressible in FOL. CTL-live includes the CTL connectives that are often used to express liveness properties (e.g., **AF**, **AU**, etc.). Our result holds for any Kripke structure expressible symbolically in FOL. Since FOL validity checking is recursively enumerable (r.e.) [8], if a Kripke structure satisfies a CTL-live property our reduction can be used to generate a proof automatically. This is the opposite of iterative unbounded methods, such as [6], which guarantee termination only if the property is not satisfied.

Model checking a CTL formula φ requires checking whether the set of initial states of a Kripke structure is included in the set of states that satisfy φ. Validity in FOL is defined using a universal quantifier over interpretations, which is not a first-order quantifier. The key insight in our approach is to use this implicit higher-order quantifier to quantify over sets that include every state that satisfies φ and possibly more; these sets along with this higher-order quantifier are sufficient to solve the model checking problem for a CTL-live formula.

Our result can form the basis for using first-order reasoners directly for model checking CTL-live properties of infinite Kripke structures expressed symbolically in FOL. By avoiding external iteration, we allow the reasoning tool to work at its maximum efficiency with respect to reusing parts of the deduction. By avoiding manual abstraction, we have removed a large burden on the user to justify the validity of the abstraction.

II. Preliminaries

We use standard first-order logic with equality (FOL) [8]. The syntax and semantics of FOL is defined using the concepts

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

of signatures and interpretations. A *signature* is a set of *functional* and *relational symbols* where each symbol has a corresponding *arity*, which is a natural number. For a given signature, an *interpretation* consists of a *domain* (a non-empty set), and a *mapping*, which determines the content of each functional and relational symbol in the signature. We use the notation $X^{\mathcal{I}}$ to denote the value that the symbol X is mapped to under the interpretation $\mathcal{I}$.

We denote the *satisfiability* relation for FOL by $\Vdash$, where $\mathcal{I} \Vdash \Phi$ means that the interpretation $\mathcal{I}$ *satisfies* the FOL formula Φ, and $\mathcal{I} \nVdash \Phi$ denotes otherwise. If Γ is a set of FOL formulae and $\mathcal{I}$ is an interpretation, the notation $\mathcal{I} \Vdash \Gamma$ means that $\mathcal{I}$ satisfies every formula in Γ. *Validity* (or semantic entailment) in FOL is denoted by $\Gamma \models \Phi$.

The subset relation symbol ($\subseteq$) is overloaded in this paper: suppose X and Y are relational symbols with arity 1; the formula $X \subseteq Y$ is a short form for $\forall s : \ X(s) \rightarrow Y(s)$.

Computational tree logic (CTL) is a temporal logic to specify properties over time [7]. A temporal connective of CTL consists of two parts: a path and a state quantifier. A path quantifier is either **E** (there exists a path) or **A** (for all paths). The state quantifiers are **X** (next state), **F** (eventually), **G** (globally), and **U** (strong until). The semantics of CTL formulae is defined using Kripke structures. A *Kripke structure* is a four tuple, $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$, where: $\mathcal{S}$ is a set of states; $\mathcal{S}_0$, the set of initial states, is a non-empty subset of $\mathcal{S}$; $\mathcal{N}$, the next-state relation, is a total binary relation over $\mathcal{S}$; $\mathbb{P}$ is a finite set of unary predicates over states. Predicates represent the local properties of the states, and are called *labelling predicates*.

The notation $\mathcal{K}, s \Vdash_c \varphi$ denotes that the state s of the Kripke structure $\mathcal{K}$ satisfies the CTL formula φ and $\mathcal{K}, s \nVdash_c \varphi$ denotes otherwise. We use the standard semantics of CTL [1].

The set of states of a Kripke structure $\mathcal{K}$ that satisfies a CTL formula φ is denoted by $[\varphi]_{\mathcal{K}}$:

$$[\varphi]_{\mathcal{K}} = \{ s \in \mathcal{S} \mid \quad \mathcal{K}, s \Vdash_c \varphi \}$$

The Kripke structure $\mathcal{K}$ satisfies the CTL formula φ, denoted by $\mathcal{K} \Vdash_c \varphi$, iff for all $s \in \mathcal{S}_0$ we have $\mathcal{K}, s \Vdash_c \varphi$:

$$\mathcal{K} \Vdash_c \varphi \iff \mathcal{S}_0 \subseteq [\varphi]_{\mathcal{K}}$$

III. Overview

The goal of this work is to reduce the model checking problem ($\Vdash_c$) to validity checking in FOL ($\models$). The first step is to represent a Kripke structure symbolically in FOL. For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$, its symbolic representation (*symbolic*(K)) is a set of FOL formulae over the signature $K = \{S_0, N, P_1, .., P_n\}$ where relational symbol N has arity 2 and every other symbol has arity 1.

Since *symbolic*(K) is a set of FOL formulae, it can have multiple satisfying interpretations (each of which is a Kripke structure) that are not isomorphic because it may use uninterpreted functions and relations, and it may underconstrain the model.

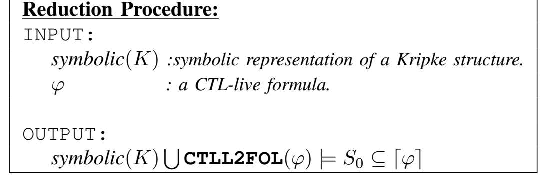


Fig. 1. Reduction Procedure

We define *symbolic*$(K) \models_c \varphi$ to mean that every satisfying interpretation $\mathcal{K}$ of *symbolic*(K) satisfies the CTL formula φ:

$$\textit{symbolic}(K) \models_c \varphi \Longleftrightarrow \forall \mathcal{K} : \mathcal{K} \Vdash \textit{symbolic}(K) \Longrightarrow \mathcal{K} \Vdash_c \varphi$$

If *symbolic*(K) has only one satisfying interpretation up to isomorphism, then *symbolic*$(K) \models_c \varphi$ is equivalent to $\mathcal{K} \Vdash_c \varphi$. However, we do not require *symbolic*(K) to have only one satisfying interpretation up to isomorphism.

Our main contribution is to identify a fragment of CTL such that its model checking problem for a symbolic representation of a Kripke structure is reducible to validity checking in FOL. We call this fragment CTL-live. We show that there exists a Γ (set of FOL formulae) and Φ (FOL formula) such that:

$$\textit{symbolic}(K) \models_c \varphi \quad \Longleftrightarrow \quad \Gamma \models \Phi$$

for φ in CTL-live. We present a function **CTLL2FOL** that takes a CTL-live φ formula as input and generates a finite set of FOL formulae that represent the satisfiability of φ. The function **CTLL2FOL** introduces a new relational symbol with arity 1 for every sub-formula of φ including φ itself. We use the notation $\lceil \varphi \rceil$ to refer to the relational symbol introduced by **CTLL2FOL** for the formula φ. The formulae generated by **CTLL2FOL** are constraints over these new relational symbols. Figure 1 is an overview of our reduction. The input of the reduction is a symbolic representation of a Kripke structure(s) (*symbolic*(K)) and a CTL-live formula (φ). The reduction procedure asserts whether the union of *symbolic*(K) with the formulae generated by **CTLL2FOL**(φ) entails $S_0 \subseteq \lceil \varphi \rceil$.

IV. Reducing CTL-live Model Checking to FOL

In this section, first, we present the intuition behind reducing model checking to FOL validity checking. Then, we define CTL-live and **CTLL2FOL**(φ).

Suppose *symbolic*(K) is a symbolic representation with a unique satisfying Kripke structure $\mathcal{K}$, and $P \in \mathbb{P}$ is a labelling predicate. We are interested in checking whether $\mathcal{K}$ satisfies **EF** P. From the semantics of CTL and its encoding in the mu-calculus [1], we know that the set of states that satisfy **EF** P, $[\mathbf{EF}\ P]_{\mathcal{K}}$, is the **smallest** set, $\lceil \mathbf{EF}\ P \rceil$, such that the following two FOL formulae hold:

$$\begin{array}{ll} A_1) & P \subseteq \lceil \mathbf{EF}\ P \rceil \\ A_2) & \forall s, s' : \big(N(s, s') \wedge \lceil \mathbf{EF}\ P \rceil(s')\big) \rightarrow \lceil \mathbf{EF}\ P \rceil(s) \end{array} \tag{1}$$

Formula A_1 states that every state that satisfies P also satisfies **EF** P, and Formula A_2 states that if a state s has a next state that satisfies **EF** P, then s also satisfies **EF** P. We use the symbol $\lceil \mathbf{EF}\ P \rceil$ rather than $[\mathbf{EF}\ P]$ because there are multiple

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

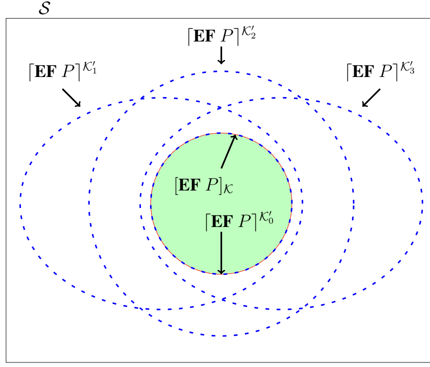


Fig. 2. Possible values for $\lceil \mathbf{EF}\ P \rceil$

sets that satisfy formulae A_1 and A_2. Any Kripke structure $\mathcal{K}'$ that is a satisfying interpretation of *symbolic*$(K) \bigcup \{A_1, A_2\}$ is equal to $\mathcal{K}$ plus it can map $\lceil \mathbf{EF}\ P \rceil$ to a set that includes $[\mathbf{EF}\ P]_{\mathcal{K}}$, but is potentially larger, i.e., $\lceil \mathbf{EF}\ P \rceil$ may be an overapproximation of $[\mathbf{EF}\ P]_{\mathcal{K}}$. This property is depicted in Figure 2, where $\lceil \mathbf{EF}\ P \rceil^{\mathcal{K}'_i}$ means the value of relational symbol $\lceil \mathbf{EF}\ P \rceil$ under the interpretation/Kripke structure $\mathcal{K}'_i$.

Since $[\mathbf{EF}\ P]_{\mathcal{K}}$ equals the smallest amongst the $\lceil \mathbf{EF}\ P \rceil^{\mathcal{K}'}s$ satisfying *symbolic*$(K) \bigcup \{A_1, A_2\}$, checking whether S_0 is a subset of $[\mathbf{EF}\ P]_{\mathcal{K}}$ is equivalent to checking whether S_0 is a subset of $\lceil \mathbf{EF}\ P \rceil^{\mathcal{K}'}$ for **every** $\mathcal{K}'$:

$$S_0 \subseteq [\mathbf{EF}\ P]_{\mathcal{K}} \iff \forall \mathcal{K}' \Vdash \textit{symbolic}(K) \bigcup \{A_1, A_2\} : \mathcal{S}_0 \subseteq \lceil \mathbf{EF}\ P \rceil^{\mathcal{K}'} \quad (2)$$

The universal quantifier in Equation 2 is over interpretations, which is not available in FOL, but it is implicitly used in the definition of validity: recall that $\Gamma \models \Phi$ iff **every** satisfying interpretation of Γ satisfies Φ; therefore:

$$\mathcal{K} \Vdash_c \mathbf{EF}\ P \iff S_0 \subseteq [\mathbf{EF}\ P]_{\mathcal{K}} \iff \textit{symbolic}(K) \bigcup \{A_1, A_2\} \models S_0 \subseteq \lceil \mathbf{EF}\ P \rceil$$

We reduce model checking of **EF** to validity checking in FOL by using the higher-order quantifier in the meta-language of FOL. What we have shown here is that even though the constraints on $\lceil \mathbf{EF}\ P \rceil$ in Equation 1 do not precisely express the set of states that satisfy **EF** P, they express a set that **includes** every state that satisfies **EF** P (and possibly more). Since in model checking, it is important to see whether the set of initial states is **included** in the set of states that satisfy **EF** P, these constraints along with the definition of validity in FOL, which implicitly uses a universal quantifier over interpretations, can be used to express the model checking problem for the CTL connective **EF**.

The key idea behind this result is that the CTL connective **EF** can be expressed as the smallest set that satisfies some FOL formulae. We can generalize this result for other CTL connectives that have the same property: **AF**, **EU**, and **AU**. We can also include the propositional connectives $\wedge$ and $\vee$ since their corresponding set operations (intersection and union) are monotonic with respect to set inclusion.

Temporal part		
φ	::=	$\pi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$
	::=	$\mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi$
	::=	$\varphi_1 \mathbf{EU} \varphi_2 \mid \varphi_1 \mathbf{AU} \varphi_2$
Propositional part		
π	::=	$P \mid \neg\pi \mid \pi_1 \vee \pi_2$
	where P is a labelling predicate.	

Fig. 3. CTL-live

Figure 3 presents the fragment of CTL for which the model checking problem can be expressed in FOL. We call this fragment *CTL-live*, since it contains the CTL connectives that are usually used to express liveness properties. CTL-live's grammar has two parts: temporal and propositional. CTL-live disallows a temporal connective to be within the scope of negation ($\neg$); e.g., the CTL formula $\neg(\mathbf{AF}\ P)$ is not part of CTL-live, but $\mathbf{AF}\ (\neg P)$ is.

To check if *symbolic*$(K) \models_c \varphi$ where φ is a CTL-live formula, we use a function called **`CTLL2FOL`**, shown in Figure 4, to create a set of FOL formulae expressing the meaning of these connectives. In Figure 4, $\lceil \varphi \rceil$ is a new relational symbol that is introduced by **`CTLL2FOL`** for the formula φ; for a labelling predicate P, $\lceil P \rceil$ is equal to P. The complexity of **`CTLL2FOL`** is linear with respect to the size of φ.

Theorem 1 presents our main contribution: model checking a symbolic representation of a Kripke structure(s) ($\models_c$) for a CTL-live formula is reducible to validity checking in FOL ($\models$). Complete proofs can be found in Vakili and Day [9].

Theorem 1: Let *symbolic*(K) be a set of FOL formulae that specifies a Kripke structure(s); we have:

$$\textit{symbolic}(K) \models_c \varphi \iff \textit{symbolic}(K) \bigcup \mathtt{CTLL2FOL}(\varphi) \models S_0 \subseteq \lceil \varphi \rceil$$

V. Related Work

Based on [10], we reduced model checking of CTL with fairness constraints for finite symbolic Kripke structures to validity checking in FOL(TC) and used Alloy for model checking [11]. Since transitive closure for an infinite system is not expressible in FOL, this encoding cannot be used with a FOL reasoner.

K-induction is a technique for unbounded model checking of safety properties [5]. This technique extends bounded model checking by proving that bounded model checking for bound K is sufficient. The number K is dominated by the diameter of a Kripke structure. The diameter is computed iteratively using a SAT solver to check the equivalence of two formulae: the equivalence holds iff no new state can be reached by taking more than K steps. In [5], termination is guaranteed due to the finiteness of the Kripke structures under study.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

CTLL2FOL(φ) **:**

case φ of

1)	P	->	$\{\}$ where P is a labelling predicate
2)	$\neg\psi$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \neg\lceil\psi\rceil(s)\ \} \bigcup$ **CTLL2FOL**(ψ)
3)	$\psi_1 \vee \psi_2$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \lceil\psi_1\rceil(s) \vee \lceil\psi_2\rceil(s)\ \} \bigcup$ **CTLL2FOL**$(\psi_1) \bigcup$ **CTLL2FOL**(ψ_2)
4)	$\psi_1 \wedge \psi_2$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \lceil\psi_1\rceil(s) \wedge \lceil\psi_2\rceil(s)\ \} \bigcup$ **CTLL2FOL**$(\psi_1) \bigcup$ **CTLL2FOL**(ψ_2)
5)	$\mathbf{EX}\psi$	->	$\{\ \forall s:\ \big(\exists s':\ N(s,s') \wedge \lceil\psi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**(ψ)
6)	$\mathbf{AX}\psi$	->	$\{\ \forall s:\ \big(\forall s':\ N(s,s') \rightarrow \lceil\psi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**(ψ)
7)	$\mathbf{EF}\psi$	->	$\{\ \lceil\psi\rceil \subseteq \lceil\varphi\rceil\ ,\ \ \forall s: \big(\exists s': N(s,s') \wedge \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**(ψ)
8)	$\mathbf{AF}\psi$	->	$\{\ \lceil\psi\rceil \subseteq \lceil\varphi\rceil\ ,\ \ \forall s: \big(\forall s': N(s,s') \rightarrow \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**(ψ)
9)	$\psi_1\mathbf{EU}\psi_2$	->	$\{\ \lceil\psi_2\rceil \subseteq \lceil\varphi\rceil\ ,\ \ \forall s: \lceil\psi_1\rceil(s) \wedge \big(\exists s': N(s,s') \wedge \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**$(\psi_1) \bigcup$ **CTLL2FOL**(ψ_2)
10)	$\psi_1\mathbf{AU}\psi_2$	->	$\{\ \lceil\psi_2\rceil \subseteq \lceil\varphi\rceil\ ,\ \ \forall s: \lceil\psi_1\rceil(s) \wedge \big(\forall s': N(s,s') \rightarrow \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ **CTLL2FOL**$(\psi_1) \bigcup$ **CTLL2FOL**(ψ_2)

Fig. 4. Definition of **CTLL2FOL**. φ is a CTL-live formula.

Bultan, Gerber, and Pugh used Presburger formulae to represent infinite sets of states symbolically [6]. Their model checking approach requires a fixpoint calculation, and termination is achieved by using conservative approximation. This approach allows false negatives.

Kesten and Pnueli presented a sound and relatively complete (oracle based) deductive system for CTL* [12] to provide proof-like evidence for a model that satisfies a property. CTL-live is less expressive than CTL* but based on the completeness of FOL, CTL-live has a sound and complete deductive system.

Beyene, Popeea, and Rybalchenko encoded CTL model checking of infinite state systems into forall-exists quantified Horn clauses (which we call ExQH) [13]. The contribution of [13] is to develop a solver for ExQH and demonstrate its use for model checking CTL properties. Their method requires the models and the model checking constraints to be expressed in ExQH and to satisfy some well-foundedness conditions, whereas our results hold for any set of FOL constraints, which may describe multiple Kripke structures. Termination of their method is not guaranteed.

VI. Conclusion

We presented a fragment of CTL, called CTL-live, whose model checking problem is reducible to validity checking in FOL. Our reduction shows that FOL deductive techniques are sufficient for model checking CTL-live formulae, without the need for iteration, abstraction, or induction. The key insight in our approach is to use the implicit higher-order quantifier in the definition of validity to require that all initial states of a Kripke structure are within all the sets of states that satisfy an overapproximation of a CTL-live temporal operator, and thereby, reducing model checking to validity in FOL. Validity checking for FOL is r.e.; as a result, this reduction ensures that a proof can be automatically generated when a CTL-live formula is satisfied by a model. We have also proved that CTL-live is maximal in the sense that it is the largest fragment of CTL such that its model checking is reducible to FOL validity [9].

Our theory provides the basis for using first-order reasoners directly for model checking CTL-live properties of abstract and infinite Kripke structures expressed symbolically in FOL. By avoiding iteration, the tool can reuse its internal deductions to increase productivity. The rapid improvements in the efficiency of SMT solvers, FOL automated theorem proving, etc. have a direct effect on the practical application of our results. We are currently studying the use of SMT solvers for model checking CTL-live formulae [14].

References

[1] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Satisfiability Modulo Theories*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 26, pp. 825–885.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, ser. LNCS. Springer, 1999, pp. 193–207.

[4] T. Schüle and K. Schneider, "Bounded model checking of infinite state systems," *Formal Methods in System Design*, pp. 51–81, 2007.

[5] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *FMCAD*, ser. LNCS. Springer, 2000, vol. 1954, pp. 127–144.

[6] T. Bultan, R. Gerber, and W. Pugh, "Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic," in *CAV*, ser. LNCS, O. Grumberg, Ed. Springer, 1997, vol. 1254, pp. 400–411.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM TOPLS*, pp. 244–263, 1986.

[8] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[9] A. Vakili and N. A. Day, "Reducing CTL-live Model Checking to Semantic Entailment in First-Order Logic (Version 1)," Cheriton School of Comp. Sci., University of Waterloo, Tech. Rep. CS-2014-05, 2014.

[10] N. Immerman and M. Vardi, "Model Checking and Transitive-Closure Logic," in *CAV*, ser. LNCS. Springer, 1997, vol. 1254, pp. 291–302.

[11] A. Vakili and N. Day, "Temporal Logic Model Checking in Alloy," in *ABZ*, ser. LNCS. Springer, 2012, vol. 7316, pp. 150–163.

[12] Y. Kesten and A. Pnueli, "A compositional approach to CTL* verification," *Theoretical Computer Science*, pp. 397 – 428, 2005.

[13] T. A. Beyene, C. Popeea, and A. Rybalchenko, "Solving existentially quantified horn clauses," ser. CAV. Springer, 2013, pp. 869–882.

[14] A. Vakili and N. A. Day, "Verifying CTL-live Properties of Infinite State Models using an SMT Solver," in *FSE'14*, Oct 2014, To appear.

Predicate Abstraction for Reactive Synthesis

Adam Walker[§] Leonid Ryzhyk[§¶]
[§] NICTA and UNSW, Sydney, Australia
[¶] University of Toronto

Abstract—**We present a predicate-based abstraction refinement algorithm for solving reactive games. We develop solutions to the key problems involved in implementing efficient predicate abstraction, which previously have not been addressed in game settings: (1) keeping abstractions concise by identifying relevant predicates only, (2) solving abstract games efficiently, and (3) computing and solving abstractions symbolically. We implemented the algorithm as part of an automatic device driver synthesis toolkit and evaluated it by synthesising drivers for several real-world I/O devices. This involved solving game instances that could not be feasibly solved without using abstraction or using simpler forms of abstraction.**

I. Introduction

Two-player games are a useful formalism for synthesis of reactive systems [17]. Many problems in electronic design automation [3], industrial automation [5], device driver development [19], etc., can be formalised as games. The resulting games often have very large state spaces and can not be efficiently solved using existing techniques.

Abstraction offers an effective approach to mitigating the state explosion. For example, in the model checking domain abstraction proved instrumental in enabling automatic verification of complex hardware and software systems [6], [7], [15]. The reactive synthesis community has also identified the key role of abstraction in tackling real-world synthesis problems; however most research in this area has so far been of theoretical nature [10], [14].

In this paper we present the first practical abstraction-refinement algorithm for solving games. Our algorithm is based on *predicate abstraction*, which proved to be particularly successful in model checking [13]. Predicate abstraction partitions the state space of the game based on a set of predicates, which capture essential properties of the system. States inside a partition are indistinguishable to the abstraction, which limits the maximal precision of solving the game achievable within the given abstraction. The abstraction is iteratively refined by introducing new predicates.

The key difficulty in applying predicate abstraction to games is to efficiently solve the abstract game arising at every iteration of the abstraction refinement loop. This requires computing the abstract *controllable predecessor* operator, which maps a set of abstract states, winning for one of the players, into the set of states from which the player can force the game into the winning set in one round of the game. This involves enumerating concrete moves available to both players in each abstract state, which can be prohibitively expensive.

We address the problem by further approximating the expensive controllable predecessor computation and refining the approximation when necessary. To this end, we introduce additional predicates that partition the set of actions available to the players into *abstract actions*. The controllable predecessor computation then consists of two steps: (1) computing abstract actions available in each abstract state, and (2) evaluating controllable predecessor over abstract states and actions.

The first step involves potentially expensive analysis of concrete transitions of the system and is therefore computed approximately. More specifically, solving the abstract game requires overapproximating moves available to one of the players, while underapproximating moves available to the other [14]. The former is achieved by allowing an abstract action in an abstract state if it is available in at least one corresponding concrete state, the latter allows an action only if it is available in all corresponding concrete states. We compute the overapproximation by initially allowing all actions in all states and gradually refining the abstraction by eliminating spurious actions. Conversely, we start with an empty underapproximation and add available actions as necessary.

We incorporated our predicate abstraction algorithm in the three-valued abstraction refinement framework of de Alfaro and Roy [10]. However, it can be readily adapted for use with other abstraction refinement methods, such as the counterexample-guided framework of Henzinger et al [14].

This paper makes three contributions:

1) We propose the first practical predicate-based abstraction refinement algorithm for two-player games.
2) We introduce a new type of refinement, which increases the precision of controllable predecessor computation without refining the abstract state space of the game. This approach avoids costly operations involved in solving the abstract game, approximating them with a sequence of light-weight operations performed on demand, leading to dramatically improved scalability.
3) We evaluate the algorithm by implementing it as part of the Termite driver synthesis toolkit [19] and using it to synthesise drivers for complex real-world devices. Our algorithm efficiently solves games with very large state spaces, which is impossible without using abstraction or using simpler forms of abstraction.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
This research is supported by a grant from Intel Corporation.

II. RELATED WORK

Predicate abstraction has been extensively explored in automatic verification [13], including hardware [6] and software [7], [15] verification. In verification, given a set of abstract error states, we would like to overapproximate the set of predecessor states, from which the system may transition to one of the error states. To this end, one constructs an overapproximation of the abstract transition relation of the system, which relates a pair of abstract states if there exists a matching concrete transition between these two states [1]. De Alfaro et al. [9] pointed out that similar approach is not applicable to solving abstract games. In game settings, given a set of abstract goal states, we would like to compute its abstract controllable predecessor, i.e., the set of abstract states from which one of the players can force the game into the goal in one round. This fundamentally cannot be encoded as a relation over pairs of abstract states as, although the player may not be able to force the game into an individual abstract state, it may be able to force it into a subset of goal states. Therefore, instead of approximating the abstract transition relation of the game, we approximate its abstract controllable predecessor operator.

The three-valued abstraction refinement technique was first proposed as a method for CTL model checking [20] and was later adapted to games [9]. It was further developed by de Alfaro and Roy [10] into a form amenable to fully symbolic implementation. They present an instantiation of their method for a particular type of abstraction—*variable abstraction*. In the present paper, we combine their method with the more flexible predicate abstraction.

Counterexample-guided abstraction refinement (CEGAR) [8] is another method of constructing abstractions automatically. Henzinger et al. [14] present an adaptation of CEGAR to games. Similar to the three-valued abstraction framework of de Alfaro and Roy, their technique can be instantiated for different forms of abstraction. Dimitrova and Finkbeiner present an instantiation based on predicate abstraction [11], [12]. They focus on partial information and timed games, as opposed to perfect-information games with large state spaces, as we do in the present work. They report solving games with up to 2000 abstract states, whereas our case studies reported in Section VII required abstractions with up to 2^{33} abstract states.

III. PRELIMINARY DEFINITIONS

A two-player *game structure* $G = \langle S, L, I, \tau_1, \tau_2, \delta\rangle$ consists of a set of states S, a set of transition labels L, a set $I \subseteq 2^S$ of initial states, a partitioning of S into player-1 states τ_1 and player-2 states τ_2 ($\tau_1 \cap \tau_2 = \emptyset$, $\tau_1 \cup \tau_2 = S$), a transition function $\delta : (S, L) \to S$ associating with a state $s \in S$ and a label $l \in L$ a successor state $\delta(s, l)$. We refer to the opponent of player i as $\bar{i}$ ($\bar{1} = 2$, $\bar{2} = 1$).

The game proceeds in an infinite sequence of rounds, starting from an initial state. In each round, in state $s \in \tau_i$, player i chooses a label l and the game transitions to state $s' = \delta(s, l)$. We do not require the game to be strictly alternating, i.e., $s' \in \tau_{\bar{i}}$ is not generally true. The infinite sequence of states visited $(s_0, s_1, \ldots) \in S^\omega$ is called a *path*.

An *objective* $\Phi \subseteq S^\omega$ is a subset of state sequences of G. In this paper we are concerned with ω-regular objectives, i.e., objectives characterised by ω-regular languages. Two special cases of ω-regular objectives are *reachability objectives* that consist of all paths $s_0, s_1, \ldots$ that visit a target set T at least once: $\exists i.s_i \in T$ and *safety objectives* that consist of paths that stay in a safe set T forever: $\forall i.s_i \in T$.

A *strategy* for player i is a function $\pi_i : S^* \times \tau_i \to L$ that, in any player i state, associates the history of the game with a label to play. The set of initial states I and a player i strategy π_i determines a set $Outcomes_i(I, \pi_i)$ of paths $s_0, s_1, s_2, ...$ such that $s_0 \in I$ and $s_{k+1} = \delta(s_k, \pi_i(s_0, ..., s_k))$ when $s_k \in \tau_i$ and $s_{k+1} = \delta(s_k, l)$ for some l when $s_k \in \tau_{\bar{i}}$. Given an objective $\Phi \in S^\omega$ we say that state $s \in S$ is winning for player i if there is a strategy π_i such that $Outcomes_i(s, \pi_i) \subseteq \Phi$.

A. Symbolic games

We deal with *symbolic games* defined over a finite set of state variables X and a finite set of label variables Y in some theory. Each state $s \in S$ represents a valuation of variables X, each label $l \in L$ represents a valuation of variables Y. For a set Z of variables, we denote by $\mathcal{F}(Z)$ the set of propositional formulas in the underlying theory constructed from the variables in Z. Sets of states and transition relations of a symbolic game are represented by their characteristic formulas. In particular I, τ_1, τ_2 are given as formulas in $\mathcal{F}(X)$. The transition relation is specified as $\delta \in \mathcal{F}(X \cup Y \cup X')$, where $X' = \{x' \mid x \in X\}$ is the set of next-state variables. We refer to sets and their characteristic formulas interchangeably.

Example. We introduce our running example, where we aim to synthesise a software driver for an artificially trivial I/O device. The device contains 32 bits of non-volatile memory, which can be accessed from software via the data register. The task of the driver is to transfer a data value from the main memory to the device memory.

We set up a game between the driver (player 1) and the device (player 2). Device and driver internal state is modelled using state variables (Figure 1a). The player who makes the next move is determined by the value of the `bsy` flag inside the device. When the flag is set to `0`, the device remains idle and the driver performs a write to the data register. The argument of the write is modelled by the `val` label variable. The write operation flips the `bsy` flag to `1`. This triggers a device transition at the next round of the game, which copies the value in the data register to memory. The objective of the game on behalf of player 1 is to reach the target set `T` = (`req` = `mem`), i.e., the device memory must store the requested value `req` (Figure 1c). We require that the game is winnable from any initial state, hence `I` = $\top$. The winning strategy for player 1 in this example is to write the value of `req` in the first transition (by setting `val` = `req`), thus forcing the device to copy this value to memory at the second transition.

Figure 1b specifies the transition relation δ of the game in the form of variable update functions $\mathtt{x}' = \mathtt{t_x}(\mathtt{X}, \mathtt{Y})$, one for

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

var	type	description
	state vars (`X`)	
`mem`	`int32`	device memory
`dat`	`int32`	data register
`bsy`	`bool`	device busy bit
`req`	`int32`	value to write to `mem`
	label vars (`Y`)	
`val`	`int32`	value to write to `dat`

(a) Game variables

$\tau_1 = (\mathtt{bsy} = \mathtt{false})$ $\tau_2 = (\mathtt{bsy} = \mathtt{true})$ $\mathtt{I} = \top$ $\mathtt{T} = (\mathtt{req} = \mathtt{mem})$

(b) Turn functions, initial and target sets

$$\mathtt{dat}' = \begin{cases} \mathtt{val}, & \text{if } \neg\mathtt{bsy} \\ \mathtt{dat}, & \text{otherwise} \end{cases}$$

$$\mathtt{bsy}' = \begin{cases} \mathtt{true}, & \text{if } \neg\mathtt{bsy} \\ \mathtt{false}, & \text{if } \mathtt{bsy} \end{cases}$$

$$\mathtt{mem}' = \begin{cases} \mathtt{dat}, & \text{if } \mathtt{bsy} \\ \mathtt{mem}, & \text{otherwise} \end{cases}$$

$$\mathtt{req}' = \mathtt{req}$$

(c) Variable update functions

a.var	predicate
	state predicates
σ_1	`req = dat`
σ_2	`req = mem`
	untracked predicates
ω_1	`bsy = false`
ω_2	`req = 5`
	label predicates
λ_1	`val = req`
λ_2	`val = 5`

(d) Abstract variables and corresponding predicates

Fig. 1: A driver synthesis problem encoded as a game

each variable $\mathtt{x} \in \mathtt{X}$. Consider the update function for `bsy` as an example. The variable switches between values `true` and `false` on each transition, thus enabling player 1 and player 2 in a round robin fashion. □

B. Controllable predecessor

Omega-regular games are often solved using the *controllable predecessor* operator. Player i controllable predecessor of set $\phi \subseteq S$ consists of all states from which i can force the game into ϕ in one transition. It is a union of player i states where there exists a winning transition to ϕ and player $\bar{i}$ states where all outgoing transitions terminate in ϕ.

$$\begin{aligned} Cpre_i(\phi) =& \tau_i \wedge \exists Y, X'.\ \delta \wedge \phi' \ \ \vee \\ & \tau_{\bar{i}} \wedge \forall Y, X'.\ \delta \rightarrow \phi' \end{aligned} \tag{1}$$

where ϕ' denotes the formula obtained from ϕ by replacing every $x \in X$ with x'.

We can compute the winning set of a reachability game by iterating the controllable predecessor operator starting from the target set T of the game. Using fix-point notation: $\textsc{Reach}(T, Cpre) = \mu X.Cpre(X) \vee T$, We pass the controllable predecessor operator as an argument to the $\textsc{Reach}$ function, so that it can be used with multiple different versions of $Cpre$ introduced below.

C. Abstraction

An abstraction of a game structure G is a tuple $\langle V, \downarrow\rangle$, where V is a finite set of abstract states and $\downarrow : V \rightarrow 2^S$ is the *concretisation function*, which takes an abstract state and returns the possibly empty set of concrete states that the abstract state corresponds to. We require that $\bigcup_{v\in V} v\!\downarrow = S$ and $v_1\!\downarrow \cap\, v_2\!\downarrow = \emptyset$ for any v_1 and v_2, $v_1 \neq v_2$. In the case when $v\!\downarrow = \emptyset$ the abstract state v is said to be *inconsistent*. We extend the $\downarrow$ operator to sets of abstract states. For $U \subseteq V$: $U\!\downarrow = \bigcup_{u\in U} u\!\downarrow$.

Algorithm 1 Three-valued abstraction refinement for games.

Input: A game structure $G = \langle S, L, I, \tau_1, \tau_2, \delta\rangle$, a set of target states $T \subseteq S$, and an initial abstraction $\alpha = \langle V, \downarrow, Cpre_1^{m+}, Cpre_1^{M-}\rangle$ that is precise for T, I, and τ_i.
Output: *Yes* if $I \subseteq \textsc{Reach}(T, Cpre_1)$, and *No* otherwise.

1: **loop**
2: $\quad W^M \leftarrow \textsc{Reach}(T\!\uparrow^M, Cpre_1^{M-})$
3: $\quad W^m \leftarrow \textsc{Reach}(T\!\uparrow^m, Cpre_1^{m+})$
4: $\quad$ **if** $I\!\uparrow^M \subseteq W^M$ **return** Yes
5: $\quad$ **else if** $I\!\uparrow^M \nsubseteq W^m$ **return** No
6: $\quad$ **else**
7: $\quad\quad refined \leftarrow \textsc{refineCpre}(W^M)$
8: $\quad\quad$ **if** $(\neg refined)$ $\textsc{refineAbstraction}(W^M)$ **endif**
9: $\quad$ **end if**
10: **end loop**

IV. Three-valued abstraction refinement

In this section we present a modified version of the three-valued abstraction refinement technique of de Alfaro and Roy [10]. To simplify the presentation, we focus on solving reachability games. De Alfaro and Roy present an extension of their method to arbitrary ω-regular games. This extension is directly applicable to the version of the algorithm presented here.

We start with defining two versions of the abstraction operator: the *may-abstraction* $\uparrow^m$ and the *must-abstraction* $\uparrow^M$. For a set of concrete states $T \subseteq S$: $T\!\uparrow^m = \{v \in V \mid v\!\downarrow \cap T \neq \emptyset\}$, $T\!\uparrow^M = \{v \in V \mid v\!\downarrow \subseteq T\}$. We say that abstraction is *precise* for a set $T \subseteq S$ if $(T\!\uparrow^m)\!\downarrow = (T\!\uparrow^M)\!\downarrow$.

Next, we define may and must versions of the abstract controllable predecessor operator:

$$Cpre_i^m(U) = Cpre_i(U\!\downarrow)\!\uparrow^m, \quad Cpre_i^M(U) = Cpre_i(U\!\downarrow)\!\uparrow^M \tag{2}$$

These operators have the property: $Cpre_i^M(U)\!\downarrow \subseteq Cpre_i(U\!\downarrow) \subseteq Cpre_i^m(U)\!\downarrow$, and hence $\textsc{Reach}(T\!\uparrow^M, Cpre_i^M)\!\downarrow \subseteq \textsc{Reach}(T, Cpre_i) \subseteq \textsc{Reach}(T\!\uparrow^m, Cpre_i^m)\!\downarrow$.

The abstract $Cpre_i^m$ and $Cpre_i^M$ operators are defined in terms of the concrete controllable predecessor $Cpre$. As these may not be possible to compute efficiently in practice, we introduce approximate versions, $Cpre_i^{m+}$ and $Cpre_i^{M-}$, such that for all $U \subseteq V$: $Cpre_i^m(U)\!\downarrow \subseteq Cpre_i^{m+}(U)\!\downarrow$ and $Cpre_i^{M-}(U)\!\downarrow \subseteq Cpre_i^M(U)\!\downarrow$. The definition of $Cpre_i^{m+}$ and $Cpre_i^{M-}$ is determined by each particular instantiation of the abstraction refinement scheme. We present our version of these operators in Section V-A.

Figure 2 illustrates the main idea of our approach, which is presented in algorithm 1. At every iteration, the algorithm computes the must-winning set W^M that underapproximates, and the may-winning set W^m that overapproximates the true winning set (lines 2–3). The algorithm terminates if the must-winning set contains the entire initial set or the may-winning set has shrunk beyond the initial set (lines 4–5). Otherwise, the algorithm refines the abstraction in a way that expands the must-winning set.

The key observation behind the refinement procedure is that candidate winning states can be found at the *may-must boundary* of the game, i.e., the set $Cpre_1^{m+}(W^M)\backslash W^M$, of all may-predecessors of the must-winning set. The boundary consists

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

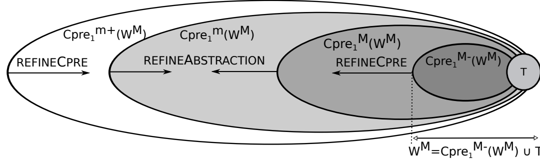


Fig. 2: Refining the may-must boundary. Arrows indicate how the two refinement functions change the boundary region.

of three regions shown in Figure 2: (1) $Cpre_1^M(W_M) \setminus W_M$, (2) $Cpre_1^m(W^M) \setminus Cpre_1^M(W^M)$, and (3) $Cpre_1^{m+}(W^M) \setminus Cpre_1^m(W^M)$. The first and the third regions can be shrunk by increasing the precision of the $Cpre^{M-}$ and $Cpre^{m+}$ operators respectively. The second region can only be shrunk by refining the abstraction itself, i.e., partitioning abstract states into smaller regions.

These two types of refinement are performed in lines 7 and 8 of the algorithm. The REFINECPRE function computes a more precise version of the controllable predecessor operators. It returns $false$ iff no such refinement is possible, i.e., $Cpre^M(W_M) = Cpre^{M-}(W_M)$ and $Cpre^{m+}(W^M) = Cpre^m(W^M)$. The REFINEABSTRACTION function refines the abstract state space in a way that expands the set $Cpre^M(W^M)$ with at least one new abstract state.

Algorithm 1 differs from [10] in that it uses an additional type of refinement which refines the controllable predecessor operators without changing the abstract state space.

V. Predicate abstraction

We instantiate the three-valued abstraction refinement scheme for predicate abstraction. Consider a symbolic game $G = \langle S, L, I, \tau_1, \tau_2, \delta\rangle$ defined over state variables X and label variables Y. Let $\Sigma \subseteq \mathcal{F}(X)$ be a finite set of boolean predicates over state variables. We refer to Σ as *state predicates*. We introduce boolean variables $\vec{\sigma} = (\sigma_1 \ldots \sigma_n)$ to represent values of predicates Σ. Given a boolean variable σ, $\|\sigma\|$ denotes its corresponding state or label predicate. $\|\vec{\sigma}\|$ denotes the vector of all state predicates in Σ.

The state space V of the abstract game is defined as $V = \mathbb{B}^n$, where each abstract boolean state vector $v \in V$ represents a truth assignment of variables $\vec{\sigma}$. The concretisation function $\downarrow$ from Section III-C can be expressed as: $v\!\downarrow = (\bigwedge_{i=1..n} \|\sigma_i\| = v_i)$, which maps an abstract state v into the set of concrete states such that each predicate in Σ evaluates to true or false depending on the value of the corresponding element of v.

Example. Consider an abstraction of the running example game induced by abstract variables σ_1, σ_2 and corresponding predicates: $\|\sigma_1\| = (\texttt{req} = \texttt{dat})$, $\|\sigma_2\| = (\texttt{req} = \texttt{mem})$. Consider an abstract state $\texttt{v} = (\texttt{true}, \texttt{false})$. We compute $\texttt{v}\!\downarrow = ((\texttt{req} = \texttt{dat}) = \texttt{true} \wedge (\texttt{req} = \texttt{mem}) = \texttt{false})$ or equivalently $\texttt{v}\!\downarrow = (\texttt{req} = \texttt{dat} \wedge \texttt{req} \neq \texttt{mem})$. Hence v represents the set of all concrete states where conditions $(\texttt{req} = \texttt{dat})$ and $(\texttt{req} \neq \texttt{mem})$ hold for concrete state variables mem, req, and dat. □

We obtain the initial abstraction by extracting atomic predicates from expressions T, I, and τ_i, which guarantees that the abstraction is precise for T, I, and τ_i. While this property is not essential for our approach, we will rely on it to simplify the presentation of the algorithm.

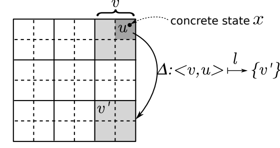


Fig. 3: Concrete state space partitioned into abstract states (solid lines) and untracked sub-states (dashed lines).

A. Abstract controllable predecessors

Following the three-valued algorithm presented in Section IV, we would like to find an efficient way to compute over- and under-approximations $Cpre^{m+}$ and $Cpre^{M-}$ of the abstract controllable predecessor operators. Recall that computing $Cpre^m$ and $Cpre^M$ precisely is expensive, as it requires applying the controllable predecessor operator to the concrete transition relation δ. We approximate this costly computation by computing the controllable predecessor over the *abstract transition relation* instead. The abstract transition relation of the game is defined over boolean predicate variables and therefore can be manipulated much more efficiently than the concrete one.

We construct the abstract transition relation via efficient syntactic analysis of the concrete transition relation δ. We present the construction assuming that δ is given in the variable update form, as in Figure 1c. A similar construction is possible for specifications written in real-world hardware and software description languages.

For each state predicate in Σ, we compute the update function by replacing concrete variables in the predicate with their corresponding update functions. We then transform the resulting formula into a boolean combination of atomic predicates over concrete state and label variables.

Example. Let us compute the update function for abstract variable σ_1 (Figure 1d). Using update functions for req and dat variables (Figure 1c), we obtain: $\sigma_1' = (\texttt{req}' = \texttt{dat}') = \big(\neg(\texttt{bsy} = \texttt{false}) \wedge (\texttt{req} = \texttt{dat}) \vee (\texttt{bsy} = \texttt{false}) \wedge (\texttt{val} = \texttt{req})\big)$. This equation contains three atomic predicates: in addition to the existing predicate $\sigma_1 \leftrightarrow (\texttt{req} = \texttt{dat})$, it introduces new predicates $(\texttt{bsy} = \texttt{false})$ and $(\texttt{val} = \texttt{req})$. □

In the general case, the syntactically computed update function for a predicate may depend on existing state predicates in Σ as well as new predicates that are not yet part of the abstraction. The new predicates are partitioned into *untracked predicates* defined over concrete state variables (e.g., $\texttt{bsy} = \texttt{false}$ in the above example) and *label predicates* that involve at least one concrete label variable (e.g., $\texttt{val} = \texttt{req}$). The term "untracked predicate" indicates that these predicates are not part of the abstract state space of the game. Untracked predicates can be seen as partitioning abstract states in V into smaller *untracked sub-states*, as illustrated in Figure 3.

By substituting untracked and label predicates with fresh

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

boolean variables, $\vec{\omega}$ and $\vec{\lambda}$ respectively, we obtain the abstract transition relation Δ in the form:

$$\vec{\sigma}' = \Delta(\vec{\sigma}, \vec{\omega}, \vec{\lambda})$$

This syntactically computed transition relation contains two sources of imprecision. First, untracked variables $\vec{\omega}$ are not part of the abstract state space Σ and are therefore treated as external inputs. Second, not all abstract labels are available in all abstract states and hence not all transitions in Δ correspond to a feasible concrete transition. For example, given the set of predicates shown in Figure 1d, the abstract label $\lambda_1 = true, \lambda_2 = true$ is only available in concrete states that satisfy the condition $req = 5$. In general, given a state-untracked-label tuple $\langle v, u, l \rangle$, the abstract label l may be available in all, some, or none of the concrete states consistent with v and u.

We formalise this by introducing *consistency relations* C^m and C^M that over- and under-approximate available abstract labels. A state-untracked-label tuple $\langle v, u, l \rangle$ is *may-consistent* if the abstract label l is available in *at least one* concrete state consistent with v and u:

$$C^m(v, u, l) = \exists X, Y. \|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l. \quad (3)$$

The tuple $\langle v, u, l \rangle$ is *must-consistent* if l is available in *any* concrete state consistent with v and u:

$$C^M(v, u, l) = \forall X. ((\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u) \rightarrow \exists Y. \|\vec{\lambda}\| = l) \quad (4)$$

Computing C^m and C^M can be prohibitively expensive. Therefore we use approximations C^{m+} and C^{M-} such that $C^m \subseteq C^{m+}$ and $C^{M-} \subseteq C^M$. Initially we assign $C^{m+} = \top$ and $C^{M-} = \bot$. Approximations are refined lazily as part of the abstraction refinement process, as explained below.

We compute over- and under-approximations of the controllable predecessor operator by resolving the two sources of imprecision in favour of one of the players. In particular, we compute $Cpre_i^{m+}$ by (1) allowing player i to pick assignments to untracked predicates, (2) over-approximating consistent labels available to i, and (3) under-approximating consistent labels available to the opponent player $\bar{i}$:

$$\begin{aligned} Cpre_i^{m+}(\phi) = \exists \vec{\omega}.\ & \tau_i{\uparrow}^M \wedge \exists \vec{\lambda}, \vec{\sigma}'.((C^{m+} \wedge \Delta) \wedge \phi') \ \vee \\ & \tau_{\bar{i}}{\uparrow}^M \wedge \forall \vec{\lambda}, \vec{\sigma}'.((C^{M-} \wedge \Delta) \rightarrow \phi') \end{aligned} \quad (5)$$

This formula has a similar structure to the definition of the concrete controllable predecessor operator (1). It replaces the concrete transition relation δ with the abstract transition relation Δ restricted with consistency relations (C^{m+} and C^{M-}). In addition, it existentially quantifies untracked variables $\vec{\omega}$, i.e., an abstract state v is a may-predecessor of ϕ if at least one of its untracked sub-states is a may-predecessor of ϕ.

Dually, we compute $Cpre_i^{M-}$ by (1) allowing the opponent player $\bar{i}$ to pick values of untracked predicates, (2) under-approximating labels available to i and (3) over-approximating labels available to $\bar{i}$:

$$\begin{aligned} Cpre_i^{M-}(\phi) = \forall \vec{\omega}.\ & \tau_i{\uparrow}^M \wedge \exists \vec{\lambda}, \vec{\sigma}'.((C^{M-} \wedge \Delta) \wedge \phi') \ \vee \\ & \tau_{\bar{i}}{\uparrow}^M \wedge \forall \vec{\lambda}, \vec{\sigma}'.((C^{m+} \wedge \Delta) \rightarrow \phi') \end{aligned} \quad (6)$$

Equations (5) and (6) suggest two possible abstraction refinement tactics, which correspond to the two types of refinement used in Algorithm 1. First, we can refine C^{m+} and C^{M-} by removing spurious transitions from C^{m+} or adding new consistent transitions to C^{M-}. Such a refinement increases the precision of controllable predecessor computation without introducing new state predicates, which corresponds to the REFINECPRE operation in the algorithm. Second, we can add some of the untracked predicates to the set of state predicates Σ, thus reducing the imprecision introduced by treating them as external inputs. This refinement increases the precision of the abstraction, which corresponds to the REFINEABSTRACTION function in the algorithm.

In summary, we solve the abstract game by decomposing potentially expensive computations into three types of lightweight operations performed on demand, as required to improve the precision of the abstraction:

- Computing the abstract transition relation Δ via lightweight syntactic analysis of the concrete game
- Computing consistency relations C^{m+} and C^{M-} by iteratively identifying spurious and consistent transitions
- Solving the abstract game using abstract controllable predecessor operators (5) and (6)

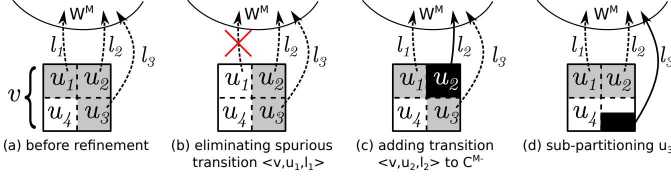
The computational bottleneck in this method can arise either from having to perform an excessive number of refinements or if abstractions generated by the algorithm are too complex. Our refinement procedures, described below, are designed to avoid such situations by heuristically picking refinements that are likely to speed up the convergence of the algorithm.

B. REFINECPRE

Figure 4 illustrates the main idea of the consistency refinement algorithm. It shows an abstract state v (Figure 4a) at the may-must boundary whose untracked substates u_1, u_2, and u_3 have C^{m+}-consistent transitions to the must-winning set W^M, but none of these transitions is consistent with C^{M-}. The REFINECPRE algorithm attempts to precisely categorise these substates as must-winning or must-losing. In Figure 4b, the algorithm identifies the abstract transition $\langle v, u_1, l_1 \rangle$ as spurious and eliminates it from C^{m+}, thus making the u_1 sub-state must-losing. Alternatively, it may detect that abstract transition $\langle v, u_2, l_2 \rangle$ is available in all concrete states in u_2 and thus add this transition to C^{M-}, making the u_2 sub-state must-winning (Figure 4c). Finally, it may determine that abstract transition $\langle v, u_3, l_3 \rangle$ is available in some, but not all, concrete states in u_3, i.e., $\langle v, u_3, l_3 \rangle \in C^m \setminus C^M$. It then partitions u_3 into two or more subsets, exactly one of which has a C^{M-}-consistent transition to W^M, by introducing new untracked predicates (Figure 4d).

Algorithm 2 shows the pseudocode of REFINECPRE. Lines 3–6 compute the set of candidate tuples $\langle v, u, l \rangle \in C^m \setminus C^M$. Note that for player i states we consider may-consistent transition to W^M, whereas for player $\bar{i}$ states we consider spoiling transitions to $V \setminus W^M$. Line 9 picks a single refinement candidate $\langle v, u, l \rangle$ from the set. By construction we

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.



(a) before refinement (b) eliminating spurious transition <v,u$_1$,l$_1$> (c) adding transition <v,u$_2$,l$_2$> to C^{M-} (d) sub-partitioning u$_3$

Fig. 4: Different types of consistency refinements. White, grey, and black background is used to mark respectively must-losing, may-winning, and must-winning untracked substates. Dashed and solid arrows show C^{m+} and C^{M-}-consistent abstract transitions.

Algorithm 2 Pseudocode of the REFINECPRE function

1: **function** REFINECPRE(W^M)
2: $\quad\triangleright$ player i may-winning transitions
3: $\quad T_i \leftarrow \tau_i\!\uparrow^M \wedge\, C^{m+} \wedge \overline{C^{M-}} \wedge \forall \vec{\sigma}'.(\Delta \rightarrow (W^M)')$
4: $\quad\triangleright$ player $\bar{i}$ may-spoiling transitions
5: $\quad T_{\bar{i}} \leftarrow \tau_{\bar{i}}\!\uparrow^M \wedge\, C^{m+} \wedge \overline{C^{M-}} \wedge \exists \vec{\sigma}'.(\Delta \wedge \overline{(W^M)'})$
6: $\quad T \leftarrow T_i \vee T_{\bar{i}}$
7: $\quad$**if** $T = \bot$ **then return** $false$ $\triangleright$ no refinement is possible
8: $\quad$**else**
9: $\quad\quad$choose $\langle v, u, l\rangle \in T$
10: $\quad\quad F \leftarrow (\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l)$
11: $\quad\quad$**if** SATISFIABLE(F) **then**
12: $\quad\quad\quad A \leftarrow$ ELIMINATEQUANTIFIERS($\exists Y.\|\vec{\lambda}\| = l$)
13: $\quad\quad\quad \hat{A} \leftarrow$ replace atomic predicates in A with boolean vars, introducing fresh vars when necessary
14: $\quad\quad\quad C^{M-} \leftarrow C^{M-} \vee (\hat{A} \wedge \vec{\lambda} = l)$
15: $\quad\quad$**else**
16: $\quad\quad\quad C^{m+} \leftarrow C^{m+} \wedge \overline{\text{UNSATCORE}(F)}$
17: $\quad\quad$**end if**
18: $\quad\quad$**return** $true$
19: $\quad$**end if**
20: **end function**

know that $\langle v, u, l\rangle \in C^{m+}$. Since C^{m+} is an overapproximation of C^m, we check whether $\langle v, u, l\rangle \in C^m$, i.e., whether v, u, and l satisfy equation (3). To this end, in line 11 we invoke a decision procedure for the underlying theory to check satisfiability of the formula: $(\|\vec{\sigma}\| = v \wedge \|\vec{\omega}\| = u \wedge \|\vec{\lambda}\| = l)$. If the formula is unsatisfiable, then $\langle v, u, l\rangle$ is a spurious transition that must be eliminated from C^{m+}. Furthermore, by extracting an unsatisfiable core of the formula, we obtain an inconsistent subset of its conjuncts $(\bigwedge \|\alpha_i\| = c_i)$, $\alpha_i \in \vec{\sigma} \cup \vec{\omega} \cup \vec{\lambda}$, which represents a potentially large set of similar spurious transitions. We eliminate all of these transitions from C^{m+} in line 16.

If, on the other hand, the formula is satisfiable, then there exists a concrete state-label pair consistent with $\langle v, u, l\rangle$. In this case we want to precisely characterise the set of states where label l is available, so that we can either add $\langle v, u, l\rangle$ to C^{M-} (as in Figure 4c) or refine it with additional untracked predicates (as in Figure 4d).

Line 12 computes the set of concrete states where abstract label l is available by performing quantifier elimination from formula $(\exists Y.\|\vec{\lambda}\| = l)$, resulting in a quantifier-free formula A over concrete state variables X. We assume that the underlying theory supports quantifier elimination, which is the case for many practically relevant theories, including the theory of fixed-size bit vectors supported by our tool. In line 13, the resulting formula A is decomposed into atomic predicates possibly introducing new untracked and label predicates. By replacing all atomic predicates in A with corresponding boolean variables, we obtain a formula $\hat{A}$ that describes the set

Algorithm 3 Pseudocode of REFINEABSTRACTION

1: **function** REFINEABSTRACTION(W^M)
2: $\quad U^M \leftarrow CpreU_1^{M-}(W^M) \wedge \overline{W^M}$
3: $\quad toPromote \leftarrow \vec{\omega} \cap$ SUPPORT(SHORTPRIME(U^M))
4: $\quad$PROMOTE($toPromote$)
5: **end function**

of all state-untracked pairs must-consistent with the abstract label l. Line 14 refines C^{M-} with the set of newly discovered must-consistent transitions.

Example. Assume that in line 9 the algorithm picks a tuple $\langle \mathtt{v}, \mathtt{u}, \mathtt{l}\rangle$ where $\mathtt{l} = (\mathtt{true}, \mathtt{true})$. Line 12 performs quantifier elimination from the formula $\exists \mathtt{val}.(\|\lambda_1\| = \mathtt{true} \wedge \|\lambda_2\| = \mathtt{true}) = \exists \mathtt{val}.(\mathtt{val} = \mathtt{req} \wedge \mathtt{val} = 5) = (\mathtt{req} = 5)$. We have discovered a new predicate $\mathtt{req} = 5$ that must hold in states where abstract label $\mathtt{l}$ is available. We introduce a new untracked variable ω_2, $\|\omega_2\| = (\mathtt{req} = 5)$ and refine $\mathtt{C}^{\mathtt{M}-}$ with a new consistent transition: $\mathtt{C}^{\mathtt{M}-} \leftarrow \mathtt{C}^{\mathtt{M}-} \vee (\omega_2 \wedge \lambda_1 \wedge \lambda_2)$. $\square$

The accompanying technical report presents an important optimisation of the REFINECPRE function [22, Appendix].

C. REFINEABSTRACTION

The REFINEABSTRACTION function is invoked by the abstraction refinement algorithm when no further consistency refinements are possible. At this point, every untracked substate of the boundary region is either must-winning or must-losing, i.e., can be coloured white or black using notation of Figure 4. REFINEABSTRACTION promotes a subset of untracked predicates making sure that the winning region W^M expands after re-solving the game in line 2 of Algorithm 1.

Algorithm 3 shows the pseudocode of REFINEABSTRACTION. Line 2 computes all untracked boundary substates that are must-predecessors of W^M. Here, $CpreU^{M-}$ is the same as $Cpre^{M-}$ (Equation (6)), but without untracked variable quantification:

$$CpreU_i^{M-}(\phi) = \tau_i\!\uparrow^M \wedge\, \exists \vec{\lambda}, \vec{\sigma}'.((C^{M-} \wedge \Delta) \wedge \phi') \;\vee\; \tau_{\bar{i}}\!\uparrow^M \wedge\, \forall \vec{\lambda}, \vec{\sigma}'.((C^{m+} \wedge \Delta) \rightarrow \phi')$$

We aim to grow W^M by promoting as few untracked predicates as possible. To this end, we extract a short prime implicant from U^M and promote the untracked variables in the support of the prime implicant (line 3). This has the effect of adding a large cube over state and untracked predicates to W^M. The PROMOTE function invoked in line 4 moves the selected untracked predicates to the set of state predicates Σ and recomputes the abstraction transition relation Δ for the new state predicates. This can lead to the introduction of new untracked and label predicates, which can serve as refinement candidates in the future.

D. Correctness

The correctness and termination theorems of [10] hold for Algorithm 1 with REFINECPRE and REFINEABSTRACTION functions defined above.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.

Theorem 1. *If Algorithm 1 terminates, it returns the correct answer.*

Proof. By construction, $Cpre_i^{m+}$ and $Cpre_i^{M-}$ over- and under-approximate abstract controllable predecessor operators, i.e., $Cpre_i^{m}(\phi)\downarrow \subseteq Cpre_i^{m+}(\phi)\downarrow$ and $Cpre_i^{M-}(\phi)\downarrow \subseteq Cpre_i^{M}(\phi)\downarrow$, for any set ϕ. Hence, winning sets $W^m = \text{REACH}(T{\uparrow}^m, Cpre_1^{m+})$ and $W^M = \text{REACH}(T{\uparrow}^M, Cpre_1^{M-})$ computed using these operators over- and under-approximate the winning set W of the concrete game: $W^M\downarrow \subseteq W \subseteq W^m\downarrow$.

If the algorithm returns *Yes* then the initial set of the game is a subset of the must-winning region ($I \subseteq W^M\downarrow$) and hence $I \subseteq W$. Likewise, if the algorithm returns *No* then $I \not\subseteq W^m\downarrow$ and hence $I \not\subseteq W$. In both cases the answer produced by the algorithm is correct. □

Theorem 2. *If there exists a finite region algebra $\mathcal{A}$ such that all abstractions $\langle V, \downarrow\rangle$ produced by Algorithm 1 are contained in $\mathcal{A}$ then the algorithm terminates.*

Proof outline. Let W^M and $\hat{W}^M$ be must-winning sets computed at two subsequent iterations of Algorithm 1.

We first show that refinement procedures REFINECPRE and REFINEABSTRACTION guarantee that the must-winning set computed at every iteration of the refinement loop grows monotonically, i.e., $W^M\downarrow \subseteq \hat{W}^M\downarrow$. This follows from the soundness of the refinement procedures, which improve the precision of $Cpre_i^{M-}$ at every iteration.

Next we show that the algorithm is guaranteed to make forward progress, i.e., after a finite number of refinements it either terminates or discovers new must-winning states ($W^M\downarrow \subset \hat{W}^M\downarrow$). Consider the consistency refinement procedure REFINECPRE first. Every invocation of this procedure classifies some of the untracked substates at the may/must boundary as either must-winning or must-losing (see Figure 4). Eventually, it will either classify all boundary states as must-losing, in which case $W^m\downarrow = W = W^M\downarrow$, and the algorithm terminates, or find at least one must-winning sub-state (as in Figures 4c and 4d). In the latter case, a subsequent invocation of the abstraction refinement procedure REFINEABSTRACTION is guaranteed to partition one of the boundary states so that one of the resulting abstract states is must-winning. This state will be discovered at the next run of the reachability algorithm, thus expanding the must-winning set.

Since, by the assumption of the theorem, all must-winning sets W^M generated by the algorithm belong to a finite region algebra, the algorithm is guaranteed to terminate after a finite number of iterations. □

The theory of fixed-size bit vectors supported by our current implementation satisfies the premise of Theorem 2, which guarantees the termination of the algorithm.

VI. IMPLEMENTATION

We implemented our abstraction refinement algorithm in the Termite [19] driver synthesis toolkit. Termite takes a model of an I/O device and a specification of the service that the driver must provide to the operating system, and synthesises a driver implementation in C. Termite provides powerful debugging facilities such as tools for analysis of synthesis failures based on counterexample strategies, interactive exploration of synthesised strategies and user-guided interactive code generation.

Our current implementation handles games with Generalised Reactivity-1 (GR(1)) [16] objectives. GR(1) games are sufficiently expressive to formalise many real-world problems, including the driver synthesis problem. They strike a balance between expressiveness and computational difficulty. We extended our abstraction refinement algorithm to handle GR(1) games as outlined by de Alfaro and Roy [10].

Termite currently supports input specifications over the concrete domain of fixed-size bit vectors and arrays. We use the Z3 SMT solver to check satisfiability and retrieve unsatisfiable cores of formulas over concrete variables (lines 11 and 16 of Algorithm 2). Quantifier elimination (line 12) over bit vector formulas is performed using our custom implementation of the decision procedure for bit vectors by Barrett et al. [2]. Termite interacts with the theory solver through a well-defined interface and hence can be readily extended with additional theories. All computations over the abstract domain are performed symbolically using the CUDD BDD package.

In addition to the techniques described in the paper we implemented a number of performance optimisations. First, we relax the requirement of Algorithm 1 that the initial abstraction must be precise for initial set I and instead overapproximate it and refine the approximation lazily whenever the algorithm discovers a spurious losing initial state. Second, we take advantage of the natural conjunctive partitioning of the transition relation and perform early quantification [4] when computing the controllable predecessor. Third, we avoid re-solving the game from scratch by reusing results of previous computations. For example, when computing the must-winning set W^M in Algorithm 1, we use the must-winning set W^M from the previous abstraction-refinement iteration as the starting value of the fixed point computation. Finally, we use BDD-specific optimisations supported by CUDD, including dynamic variable reordering [18] and variable grouping.

In addition to the predicate-based abstraction refinement algorithm, we implemented the original algorithm by de Alfaro and Roy, based on variable abstraction, which enables direct comparison of the two techniques.

Termite consists of 30,000 lines of Haskell code, with the core abstraction refinement algorithm accounting for 1,800 lines, and took approximately 10 person-years to develop.

VII. EVALUATION

We evaluate Termite by synthesising drivers for several real-world I/O devices, including an IDE hard disk, a real-time clock, two versions of UART serial controller, two versions of I2C bus controller, an SPI bus controller, and a UVC webcam. We developed corresponding device and OS models using the Termite Specification Language (TSL) by following the common methodology used by hardware developers in building high-level device models. We refer the reader to

	Statistic	Case study									
		IDE	RTC	UART-1	UART-2	I2C-1	I2C-2	SPI	UVC	simple SPI	simple I2C
1	concrete state vars (bits)	83 (952)	64 (624)	61 (335)	65(896)	64 (458)	50(222)	66(644)	95 (75908)	7 (46)	11 (64)
2	concrete label vars (bits)	27 (389)	24 (199)	20 (86)	15(289)	25 (199)	15(81)	24(384)	33 (49657)	9 (58)	14 (42)
3	consistency refinements	11	9	42	4	12	4	6	22	0	23
4	state refinements	18	16	18	50	15	17	26	25	11	9
5	state predicates	31	25	33	58	24	24	31	30	14	17
6	label predicates	57	41	40	53	36	32	28	130	19	36
7	untracked predicates	7	4	35	2	5	1	6	32	0	0
8	run time (s)	71	74	309	603	39	43	14	190	1	10
9	peak BDD size	864612	515088	907536	1142596	440482	688828	324996	785918	87892	242214
	Performance of the de Alfaro and Roy algorithm [10]										
10	run time (s)	∞	∞	∞	∞	∞	∞	∞	∞	865	1151
11	peak BDD size	-	-	-	-	-	-	-	-	400624	4088000

TABLE I: Summary of experimental case studies.

an accompanying paper for a more detailed description of the TSL language and the modelling methodology [19]. The source code of the case studies is available as part of the Termite distribution [21].

Table I summarises our experiments. The first two rows characterise the complexity of the input models in terms of the number of variables and the total number of bits used in the concrete specification of the game. Concrete state variables model internal device state, as well as the state of the driver-OS interface; label variables model commands and responses exchanged by the driver, the device, and the OS.

Rows 3 and 4 show the number of iterations of the abstraction refinement loop required to solve the game. Rows 5 through 7 show the size of the abstract game at the final iteration, when a winning strategy for the driver was obtained, in terms of the number of state, label, and untracked predicates. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The resulting abstract games are still too complex to solve using explicit state enumeration, hence the use of symbolic techniques is essential. In all case studies, Termite was able to find the winning strategy within 11 minutes running on a 2.9GHz Intel Core i7 laptop (row 8), with peak BDD size under one million nodes (row 9).

The two final rows show the performance of the original three-valued abstraction refinement algorithm of de Alfaro and Roy on our benchmarks. As expected, the algorithm does not terminate on any of the real-world driver benchmarks within a two-hour time limit. We therefore developed simplified versions of two of the benchmarks (SPI and I2C-2) with significantly reduced state spaces. As shown in the last two columns of the table, the de Alfaro and Roy algorithm terminates on these benchmarks; however it takes several orders of magnitude longer than our new algorithm, which uses predicate abstraction. These results show that predicate abstraction is essential to solving complex real-world games.

VIII. Conclusion

We presented and evaluated a practical predicate-based abstraction refinement algorithm for solving games. To the best of our knowledge, this is the first such algorithm described in the literature. We addressed key performance bottlenecks involved in applying predicate abstraction in game settings and demonstrated that our algorithm performs well on real-world reactive synthesis benchmarks.

References

[1] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, pages 388–403, Barcelona, Spain, Mar. 2004.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, San Francisco, California, USA, June 1998.

[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *ENTCS*, 190(4):3–16, Nov. 2007.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland, 1991.

[5] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier. Automatic synthesis of robust and optimal controllers - an industrial case study. In *HSCC*, pages 90–104, San Francisco, CA, USA, Apr. 2009.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, Chicago, IL, USA, July 2000.

[7] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[8] S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS*, pages 51–58, Boston, MA, USA, June 2001.

[9] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: uncertainty, but with precision. In *LICS*, pages 170–179, Turku, Finland, July 2004.

[10] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *CONCUR*, pages 74–89, Lisboa, Portugal, Sept. 2007.

[11] R. Dimitrova and B. Finkbeiner. Abstraction refinement for games with incomplete information. In *FSTTCS*, Bangalore, India, Dec. 2008.

[12] R. Dimitrova and B. Finkbeiner. Counterexample-guided synthesis of observation predicates. In *FORMATS*, pages 107–122, London, UK, Sept. 2012.

[13] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, Haifa, Israel, June 1997.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP*, pages 886–902, Eindhoven, The Netherlands, July 2003.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, Portland, Oregon, Jan. 2002.

[16] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) designs. pages 364–380, Charleston, SC, USA, Jan. 2006.

[17] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, Austin, Texas, USA, Jan. 1989.

[18] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47, Santa Clara, CA, USA, 1993.

[19] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *OSDI*, Broomfield, CO, USA, Oct. 2014.

[20] S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In *CAV*, pages 275–287, Boulder, Colorado, USA, July 2003.

[21] Termite 2 driver synthesis tool. http://www.termite2.org.

[22] A. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. Technical Report NRL-8281, NICTA, Aug. 2014.

Author Index

Abraham, Erika 59
Außerlechner, Simon 35

Barrett, Clark 7, 139
Becker, Bernd 203
Belov, Anton 115
Biere, Armin 3, 107, 179
Bingham, Brad 15
Bittner, Benjamin 23
Bloem, Roderick 31, 35
Bozzano, Marco 23

Cabodi, Gianpiero 43
Chaki, Sagar 51
Chen, Xin 59
Cimatti, Alessandro 23
Cook, Byron 67, 75

Davi, Francesco 207
Day, Nancy A. 215
De Moura, Leonardo 195
Deters, Morgan 7
Dierkes, Thomas 207
Dutertre, Bruno 83

Egli, Marcel 207
Egly, Uwe 31
Ehrig, Rainald 207

Fuhs, Carsten 67

Gario, Marco 23
Gascon, Adria 83
Ghosh, Soumava 91
Goel, Shilpi 91
Greenstreet, Mark 15
Griggio, Alberto 23
Gurfinkel, Arie 51, 99, 115

Hanna, Ziyad 1
Henzinger, Thomas A. 11

Heule, Marijn 107
Hofferek, Georg 35
Hunt, Warren 91

Ille, Fabian 207
Ivrii, Alexander 115

Jancik, Pavel 123
Jovanovic, Dejan 83

Kaiss, Daher 131
Kalechstain, Jonathan 131
Kaufmann, Matt 91
Khlaaf, Heidy 75
Kinder, Johannes 5
King, Tim 7, 139
Klampfl, Patrick 31
Koenighofer, Robert 31
Kofron, Jan 123
Kruger, Tillmann 207
Könighofer, Bettina 35
Könighofer, Robert 35

Lal, Akash 147
Leeners, Brigitte 207
Leroy, Xavier 9
Liu, Peizun 155
Lonsing, Florian 31

Majumdar, Rupak 163
Malik, Sharad 83
Mancini, Toni 207
Manolios, Panagiotis 171
Mari, Federico 207
Massini, Annalisa 207
Melatti, Igor 207

Niemetz, Aina 179
Nimkar, Kaustubh 67

O'Hearn, Peter 67

Palena, Marco 43
Papavasileiou, Vasilis 171
Pasini, Paolo 43
Piskac, Ruzica 13

Piterman, Nir	75
Popeea, Corneliu	187
Preiner, Mathias	179
Qadeer, Shaz	147
Reynolds, Andrew	7, 195
Riedewald, Mirek	171
Rollini, Simone Fulvio	125
Rybalchenko, Andrey	187
Ryzhyk, Leonid	219
Röblitz, Susanna	207
Salvo, Ivano	207
Sankaranarayanan, Sriram	59
Scheibler, Karsten	203
Seidl, Martina	107
Sharygina, Natasha	125
Sinha, Nishant	51
Sinisi, Stefano	207
Spörk, Raphael	35
Subramanyan, Pramod	83
Tetali, Sai Deep	163
Tinelli, Cesare	7, 139, 195
Tiwari, Ashish	83
Tronci, Enrico	207
Vakili, Amirhossein	215
Vizel, Yakir	99
Wahl, Thomas	155
Walker, Adam	219
Wang, Zilong	163
Wilhelm, Andreas	187

FMCAD 2014 SPONSORS

ARM

JASPER design automation

SPYGLASS FROM ATRENTA

Mentor Graphics

cadence

OneSpin SOLUTIONS

CenTaur Technology

Oski TECHNOLOGY

IBM

REAL INTENT

intel

SYNOPSYS