Proceedings of the 18th Conference on
Formal Methods in Computer-Aided Design (FMCAD 2018)
Austin, Texas, USA, October 30 – November 2, 2018

Edited by
Nikolaj Bjørner and Arie Gurfinkel

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

Technical co-sponsorship of IEEE

Proceedings of the 18th Conference on

Formal Methods in Computer-Aided Design

FMCAD 2018

October 30 – November 2, 2018

Austin, Texas, USA

Edited by Nikolaj Bjørner and Arie Gurfinkel

ISBN: 978-0-9835678-8-2

Preface

The International Conference on Formal Methods in Computer Aided Design (FMCAD), held at Austin, Texas, from October 30-November 2 in 2018, is the eighteenth in a series of meetings on the theory and applications of rigorous formal techniques for the automated design of systems. The FMCAD conference covers formal aspects of specification, verification, synthesis, testing, and security, and is a leading forum for researchers and practitioners in academia and industry alike.

The program of FMCAD 2018 comprises a tutorial day with three tutorials on deep neural networks, certified SAT solving and distributed protocol verification; two keynotes on formal methods applied to block chains and financial algorithms, a forum for doctoral students. Finally, the main program contains the presentations of the accepted papers.

The tutorial day features three presentations

- "Formal Verification of Deep Neural Networks", by Nina Narodytska, VMWare Research.
- "Formal Verification of Unsatisfiability Results", by Marijn Heule, UT Austin.
- "Deductive Verification of Distributed Protocols in First-Order Logic", by Oded Padon, Stanford University.

The keynotes focus on the application of formal verification in industry, and on the verification of cloud computing platforms and dependable systems in particular:

- "Formal Verification of Financial Algorithms with Imandra" by Grant Passmore, Aesthetic Integration.
- "Formal Design, Implementation and Verification of Blockchain Languages" by Grigore Rosu, University of Illinois Urbana-Champaign.

FMCAD also hosts the sixth edition of the Student Forum, which has been held annually since 2013 and provides a platform for graduate students at any career stage to introduce their research to the FMCAD community. The FMCAD Student Forum 2018 was organized by Dejan Jovanović and Andrew Reynolds and features posters and short presentations of fourteen accepted contributions. A detailed description of the Student Forum, listing all accepted contributions, is provided in the conference proceedings.

FMCAD 2018 received 73 submissions. The committee decided to accept 26 papers. Each submission received at least four reviews. The topics of the accepted papers include hardware and software verification, SAT, SMT, and Horn clause solving, temporal logics, concurrency, learning, synthesis, and certification.

Organizing this event would not have been possible without the support of a large number of people and our sponsors. The program committee members and additional reviewers, listed on the following pages, did an excellent job providing detailed and insightful reviews, which helped the authors to improve their submissions and guided the selection of the papers accepted for publication. We thank each and every one of them for dedicating their time and providing their expertise. Moreover, we'd like to give special thanks to the sub-committee which agreed to select the recipients of this year's Best Paper Award. We thank Jade Alglave (ARM and UCL) for agreeing to be Publication Chair, and Dejan Jovanović and Andrew Reynolds for organizing this year's FMCAD Student Forum. Our webmaster, Tom vaj Dijk, has our gratitude for maintaining and regularly updating the FMCAD website. We thank all students who volunteered to help running the event. As always, the help and expertise of the FMCAD steering committee made the organization of FMCAD much easier. We thank Armin Biere (Johannes Kepler University in Linz, Austria), Alan Hu (University of British Columbia, Canada), and especially Warren A. Hunt,. Jr. (University of Texas at Austin) and Vigyan Singhal (Oski Tech) and Georg Weissenbacher (TU Wien) for supporting and encouraging us, and guiding us through the organization process.

Holding a conference like FMCAD would not be feasible without the financial support of our sponsors. We would like to express our gratitude to our sponsors Amazon, Centaur Technology Inc., Galois Inc., IBM, Mentor Graphics, Microsoft, and Synopsis.

FMCAD 2018 is in-cooperation with the ACM and its Special Interest Groups on Programming Languages (SIGPLAN) and on Software Engineering (SIGSOFT). The FMCAD conference also received technical sponsorship from the IEEE Council on Electronic Design Automation. The conference proceedings will be available through the ACM Digital Library, the IEEE Xplore Digital Library, and are also freely accessible on the FMCAD Website.

Last but not least, we thank all authors who submitted their papers to FMCAD 2018 (accepted or not), and whose contributions and presentations form the core of the conference. We are grateful to everyone who presented their paper, gave a keynote or a tutorial, devoting a significant amount of their time to the FMCAD conference. We thank all attendees of FMCAD for supporting the conference and making FMCAD a stimulating and enjoyable event.

Nikolaj Bjørner and Arie Gurfinkel
FMCAD 2018 Program Chairs
Austin, Texas, USA, October 2018

Organization Committee

Program Co-Chairs

Nikolaj Bjørner	Microsoft, USA
Arie Gurfinkel	University of Waterloo, Canada

Webmaster

Tom van Dijk	Johannes Kepler University Linz

Publicity Chair

Yakir Vizel	The Technion

Publication Chairs

Jade Alglave	UCL
Arie Gurfinkel	University of Waterloo

Student Forum Chairs

Dejan Jovanović	SRI International
Andrew Reynolds	The University of Iowa

Steering Committee

Armin Biere	Johannes Kepler University in Linz, Austria
Alan Hu	University of British Columbia, Canada
Warren Hunt	University of Texas at Austin, USA
Vigyan Singhal	Oski Tech
Georg Weissenbacher	TU Wien, Austria

Program Committee

Jade Alglave	University College London
June Andronick	CSIRO|Data61 and UNSW
Armin Biere	Johannes Kepler University Linz
Per Bjesse	Synopsys Inc.
Nikolaj Bjørner	Microsoft
Roderick Bloem	Graz University of Technology
Gianpiero Cabodi	Politecnico di Torino
Supratik Chakraborty	IIT Bombay
Sylvain Conchon	Universite Paris-Sud
Bruno Dutertre	SRI international
Alberto Griggio	Fondazione Bruno Kessler
Arie Gurfinkel	University of Waterloo
Liana Hadarean	Synopsys
Fei He	Tsinghua University
Joe Hendrix	University of Illinois at Urbana-Champaign
Warren Hunt	The University of Texas at Austin
Alexander Ivrii	IBM
Dejan Jovanović	SRI International
Temesghen Kahsai	Groq
George Karpenkov	Apple
Tim King	Google
Igor Konnov	INRIA Nancy (LORIA)
Ken McMillan	Microsoft
Alexander Nadel	Intel
Giles Reger	The University of Manchester
Andrew Reynolds	The University of Iowa
Leonid Ryzhyk	VMware Research
Martina Seidl	Johannes Kepler University Linz
Natasha Sharygina	Università della Svizzera italiana (USI Lugano, Switzerland)
Sharon Shoham	Tel Aviv University
Anna Slobodova	Centaur Technology
Mathias Soeken	Ecole Polytechnique Fédérale de Lausanne
Daryl Stewart	ARM
Christoph Sticksel	The MathWorks
Niklas Sörensson	Mentor Graphics
Murali Talupur	FormalSim
Jaco van de Pol	University of Twente
Tom van Dijk	Johannes Kepler University Linz
Yakir Vizel	The Technion
Georg Weissenbacher	Vienna University of Technology

Additional Reviewers

Arbel, Eli
Asadi, Sepideh

Bakhirkin, Alexey
Bhayat, Ahmed
Blicha, Martin
Brecknell, Matthew
Bu, Lei
Burlyaev, Dmitry

Camurati, Paolo
Casburn, Ledah

Ebrahimi, Masoud
Eisner, Cindy

Fedyukovich, Grigory
Finocchiaro, Fabrizio

Hartmanns, Arnd
Hassan, Zyad
Hyvärinen, Antti

Iusupov, Rinat

Jain, Himanshu
Jain, Mitesh
Jansen, Nils
Joosten, Sebastiaan

Karl, Anja
Koelbl, Alfred
Koenighofer, Bettina
Kukovec, Jure
Kundu, Sudipta

Lazić, Marijana

Manevich, Roman
Marescotti, Matteo
Melquiond, Guillaume
Morgan, Carroll
Möhle, Sibylle

Nadezhin, Dmitry
Namjoshi, Kedar
Narodytska, Nina
Nevo, Ziv

O'Leary, John

Padon, Oded
Palena, Marco
Pasini, Paolo
Pu, Geguang

Riener, Heinz
Ritirc, Daniela
Ryvchin, Vadim

Sethi, Divjyot
Sewell, Thomas
Solovyev, Alexey
Sumners, Rob

Tian, Chun

Table of Contents

Invited Papers

Hardware

Quantifiers and SAT

Liveness

Concurrency

Verification

Learning and Synthesis

SMT and CHC

Rails

Certificates

Formal Verification of Deep Neural Networks

(*Invited Tutorial*)

Nina Narodytska
VMware Research, Palo Alto, California
Email: `nnarodytska@vmware.com`

***Abstract*—Deep neural networks are among the most successful artificial intelligence technologies making impact in a variety of practical applications. However, many concerns were raised about the 'magical' power of these networks. It is disturbing that we are really lacking of understanding of the decision making process behind this technology. Therefore, a natural question is whether we can trust decisions that neural networks make. One way to address this issue is to define properties that we want a neural network to satisfy. Verifying whether a neural network fulfills these properties sheds light on the properties of the function that it represents. In this tutorial, we overview several approaches to verifying neural networks properties. The first set of methods encode neural networks into Integer Linear Programs or Satisfiability Modulo Theory formulas. They come up with domain-specific algorithms to solve verification problems. The second approach is to treat the neural network as a non-linear function and to use global optimization techniques for verification. The third line of work uses abstract interpretation to certify neural networks. Finally, we consider a special class of neural networks – Binarized Neural Networks – that can be represented and analyzed using Boolean Satisfiability. We discuss how we can take advantage of the structure of neural networks in the search procedure.**

I. INTRODUCTION

Deep neural networks have become ubiquitous in machine learning with applications ranging from computer vision to speech recognition and natural language processing. Neural networks demonstrate excellent performance on many practical problems, often beating specialized algorithms for these problems, which led to their rapid adoption in industrial applications. With such a wide adoption, important questions arise regarding our understanding of the decision making process of these neural networks: Is there a way to analyze deep neural networks? How robust are these networks to perturbations of inputs? Recently, a new line of research on understanding neural networks has emerged that looks into a wide range of such questions, from interpretability of neural networks to verifying their properties [1], [2], [3], [4], [5], [6], [7], [8].

One emerging technique to analyze a neural network is based on formal verification. The idea is to encode the network and the property we aim to verify as a formal statement, using ILP, SMT or SAT, for example. If the encoding provides an exact representation of the network then we can study any property related to this network, e.g. how sensitive the network is to perturbations of the input.

In this tutorial, we look at main trends in verification of deep learning networks.

- We recap basic neural networks concepts and discuss a set of interesting properties of neural network, including properties that relate inputs and outputs of the network, e.g. robustness and invertibility, and properties that relate two networks, like network equivalence.
- We discuss common encodings of deep neural networks as Boolean, SMT or ILP formulas. We will consider how various NN properties that can be represented in these formalisms.
- We survey the main methods developed in neural networks verification. We start with a group of methods that use SMT or ILP solvers to encode verification problems. These methods range from methods that use only one technology to solve the problem to methods that combine SMT and ILP techniques during the search process. Then we will look into methods that treat neural networks as non-linear functions and use global optimization techniques to perform verification. Finally, we consider the line of work that uses abstract interpretation to certify neural networks.
- We consider a special class of neural networks – Binarized Neural Networks. These networks have a number of important features that are useful in resource constrained environments, like embedded devices. We discuss how binarized neural networks can be represented as Boolean formulas. We show that structural properties of binarized neural networks can be exploited to reason about this class of networks.

REFERENCES

[1] D. Bau, B. Zhou, A. Khosla, A. Oliva, and A. Torralba, "Network dissection: Quantifying interpretability of deep visual representations," *CoRR*, vol. abs/1704.05796, 2017.

[2] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR*, 2014.

[3] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *CAV*, 2010, pp. 243–257.

[4] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," in *CAV'17*, ser. Lecture Notes in Computer Science. Springer, 2017, pp. 3–29.

[5] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *CAV'17*, 2017, pp. 97–117.

[6] C. Cheng, G. Nührenberg, and H. Ruess, "Verification of binarized neural networks," *CoRR*, vol. abs/1710.03107, 2017.

[7] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," *CoRR*, vol. abs/1709.06662, 2017.

[8] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," *CoRR*, vol. abs/1805.09938, 2018.

ISBN: 978-0-9835678-8-2.

Formal Verification of Unsatisfiability Results

(Invited Tutorial)

Marijn J.H. Heule
The University of Texas at Austin
marijn@cs.utexas.edu

Satisfiability (SAT) solvers are used for determining the correctness of hardware and software systems. It is therefore crucial that these solvers justify their claims by providing proofs that can be independently verified. This holds also for various other applications that use SAT solvers. Just recently, long-standing mathematical problems were solved using SAT, including the Erdos Discrepancy Problem, the Pythagorean Triples Problem, and Schur Number Five. Especially in such cases, proofs are at the center of attention, and without them, the result of a solver is almost worthless.

What the mathematical problems and the industrial applications have in common, is that proofs are often of considerable size—in the case of the Schur Number Five about 2 petabytes in a highly compressed format. To demonstrate how to increase trust in the correctness of multi-CPU-year computations, we validated the poof of the Schur Number Five problem. We certified the proof using the ACL2 theorem proving system. Given the enormous size of the proof, we argue that any result produced by SAT solvers can now be validated using highly trustworthy systems with reasonable overhead.

The tutorial also covers how to use tools that validate proofs of unsatisfiability. Apart from verifying SAT-solving results, these tools support producing unsatisfiable cores and optimized proofs. Unsatisfiable cores can be useful in various debugging settings, while optimized proofs allow for fast validation by a formally-verified tool and an independent party.

ISBN: 978-0-9835678-8-2.

Deductive Verification of Distributed Protocols in First-Order Logic

(Invited Tutorial)

Oded Padon
Stanford University, USA

Formal verification of infinite-state systems, and distributed systems in particular, is a long standing research goal. In the deductive verification approach, the programmer provides inductive invariants and pre/post specifications of procedures, reducing the verification problem to checking validity of logical verification conditions. This check is often performed by automated theorem provers and SMT solvers, substantially increasing productivity in the verification of complex systems. However, the unpredictability of automated provers presents a major hurdle to usability of these tools. This problem is particularly acute in case of provers that handle undecidable logics, for example, first-order logic with quantifiers and theories such as arithmetic. The resulting extreme sensitivity to minor changes has a strong negative impact on the convergence of the overall proof effort.

On the other hand, there is a long history of work on decidable logics or fragments of logics. Generally speaking, decision procedures for these logics perform more predictably and fail more transparently than provers for undecidable logics. In particular, in the case of a false proof goal, they usually can provide a concrete countermodel to help diagnose the problem. However, decidable logics pose severe limitations on expressiveness, and it is not immediately clear that such logics can be applied to proving complex protocols or systems.

In this tutorial, we will explore a practical approach to using first order-logic, and a decidable fragment thereof, to prove complex distributed protocols and systems. The approach, implemented in the Ivy verification tool, applies abstraction and modular reasoning techniques to mitigate the expressiveness limitations of decidable fragments. The high-level strategy involves the following ideas:

- Abstracting infinite-state systems using first-order logic.
- Carefully controlling quantifier-alternations to ensure decidability.
- Using modular reasoning principles to decompose a proof into decidable lemmas.

Experience to date indicates that the approach, based on first-order logic, is surprisingly powerful, and it is possible to prove safety and liveness properties of complex protocols (e.g., Paxos variants), and also to produce verified low-level implementations, using decidable logics. Moreover, the effort required to structure the proof in this way is more than repaid by greater reliability of proof automation, which significantly reduces the overall verification effort. Better matching human reasoning capabilities to the capabilities of automated provers results in a more stable and predictable formal development process.

This tutorial is based on joint works [1], [2], [3], [4], [5], [6], [7], [8] with Jochen Hoenicke, Neil Immerman, Aleksandr Karbyshev, Giuliano Losa, Kenneth L. McMillan, Aurojit Panda, Andreas Podelski, Mooly Sagiv, Sharon Shoham, Marcelo Taube, James R. Wilcox, and Doug Woos.

References

[1] K. L. McMillan, "Modular specification and verification of a cache-coherent interface," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 109–116. [Online]. Available: http://dx.doi.org/10.1109/FMCAD.2016.7886668

[2] K. L. McMillan and O. Padon, "Deductive verification in decidable fragments with ivy," in *Static Analysis - 25th International Symposium, SAS 2018, Freiburg im Breisgau, Germany, August 29-31, 2018, Proceedings*, 2018.

[3] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," *PACMPL*, vol. 2, no. POPL, pp. 26:1–26:33, 2018. [Online]. Available: http://doi.acm.org/10.1145/3158114

[4] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, "Temporal prophecy for proving temporal properties of infinite-state systems," in *2018 Formal Methods in Computer-Aided Design, FMCAD 2018, Austin, Texas, USA, October 30 - November 2, 2018*, 2018.

[5] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv, "Decidability of inferring inductive invariants," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 217–231. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837640

[6] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made epr: Decidable reasoning about distributed protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 108:1–108:31, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140568

[7] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 614–630.

[8] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, "Modularity for decidability of deductive verification with applications to distributed systems," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 662–677. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192414

ISBN: 978-0-9835678-8-2.

Formal Verification of Financial Algorithms with Imandra

(Invited Keynote)

Grant Olney Passmore
Aesthetic Integration and Clare Hall, Cambridge
`grant.passmore@cl.cam.ac.uk`
https://www.cl.cam.ac.uk/~gp351/
https://www.imandra.ai/

Index Terms

formal verification, financial algorithms, Imandra, dark pools, market microstructure

Many deep issues plaguing today's financial markets are symptoms of a fundamental problem: The complexity of algorithms underlying modern finance has significantly outpaced the power of traditional tools used to design and regulate them. At Aesthetic Integration, we've pioneered the use of formal verification for analysing the safety and fairness of financial algorithms. With a focus on financial infrastructure (e.g., the matching logics of exchanges and dark pools), we'll describe the landscape, and illustrate our Imandra formal verification system on a number of real-world examples. We'll sketch many open problems and future directions along the way.

ISBN: 978-0-9835678-8-2.

Formal Design, Implementation and Verification of Blockchain Languages

(Invited Keynote)

Grigore Rosu
University of Illinois at Urbana-Champaign, USA
`grosu@illinois.edu`
`http://fsl.cs.illinois.edu/grosu`
and
Runtime Verification, Inc., USA
`grigore.rosu@runtimeverification.com`

Index Terms

formal verification, semantics, blockchain

Many of the recent cryptocurrency bugs and exploits are due to flaws or weaknesses of the underlying blockchain programming languages or virtual machines. The usual post-mortem approach to formal language semantics and verification, where the language is firstly implemented and used in production for many years before a need for formal semantics and verification tools naturally arises, simply does not work anymore. New blockchain languages or virtual machines are proposed at an alarming rate, followed by new versions of them every few weeks, together with programs (or smart contracts) in these languages that are responsible for financial transactions of potentially significant value. Formal analysis and verification tools are therefore needed immediately for such languages and virtual machines. We present recent academic and commercial results in developing blockchain languages and virtual machines that come directly equipped with formal analysis and verification tools. The main idea is to generate all these automatically, correct-by-construction from a formal specification. We demonstrate the feasibility of the proposed approach by applying it to two blockchains, Ethereum and Cardano.

LINKS

Runtime Verification, Inc:
- `http://runtimeverification.com`

Smart contract verification approach and verified contracts:
- `https://runtimeverification.com/smartcontract/`
- `https://github.com/runtimeverification/verified-smart-contracts`

Formally specified, automatically generated virtual machines for the blockchain:
- EVM: `https://github.com/runtimeverification/evm-semantics`
- IELE: `https://github.com/runtimeverification/iele-semantics`

Supported in part by NSF grant CCF-1421575, NSF grant CNS-1619275, and an IOHK (`http://iohk.io`) gift.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

The FMCAD 2018 Graduate Student Forum

Dejan Jovanović
SRI International

Andrew Reynolds
The University of Iowa

Abstract—**The FMCAD Student Forum provides a platform for graduate students at any career stage to introduce their research to the wider Formal Methods community, and solicit feedback. In 2018, the event took place in Austin, Texas, as integral part of the FMCAD conference. Fourteen students were invited to give a short talk and present a poster illustrating their work. The presentations covered a broad range of topics in the field of verification, such as from SAT/SMT solving and theorem proving, analysis and verification of hardware, software, and cyber-physical systems.**

Since 2013, the FMCAD conference features a Student Forum, providing a platform for graduate students at any career stage to introduce their research to the wider Formal Methods community. The FMCAD 2018 Graduate Student Forum follows the tradition of its predecessors, which took place in

1) Portland, Oregon, USA in 2013 [4],
2) Lausanne, Switzerland in 2014 [3],
3) Austin, Texas, USA in 2015 [5],
4) Mountain View, CA, USA in 2016 [2], and
5) Vienna, Austria in 2017 [1].

Graduate students were invited to submit short reports describing their ongoing research in the scope of the FMCAD conference. Based on the reviews provided by the organizing committee, 14 high-quality submissions were accepted and presented at the forum. The reviews focused on the novelty of the work, the technical maturity of the submission, and the quality and soundness of the presentation. The presentations covered a broad spectrum of topics relevant to the FMCAD community, from SAT/SMT solving and theorem proving, to analysis and verification of hardware, software, and cyber-physical systems. The following contributions have been accepted:

- *Thomas Pani*, Georg Weissenbacher and Florian Zuleger. Rely-Guarantee Reasoning for Automated Bound Analysis of Concurrent, Shared-Memory Programs.
- *Bjørnar Lutebergen*. On Synthesis and Optimization of Railway Signalling and Interlocking Designs.
- *David Narváez*. A Formally Verified Symmetry Breaking Tool for SAT.
- *Yi Chou*. Run-time Assurance for Unmanned Aerial Vehicles using Stochastic Modeling and Reachability Analysis.
- *Souradeep Dutta*. Verification of Deep Neural Networks.
- *Makai Mann* and Clark Barrett.Finding Critical Clauses in SMT-based Hardware Verification
- *Hari Govind Vediramana Krishnan*. Prioritizing Lemmas While Pushing.
- *Li Huang* and Eun-Young Kang. SMT-based Probabilistic Analysis of Timing Constraints in Cyber-Physical Systems
- *Nikita Zyuzin*, Heiko Becker, Eva Darulova and Magnus Myreen. Formalisation of Affine Arithmetic in Coq.
- *Jakub Kuderski*, Arie Gurfinkel and Jorge Navas. Type-aware DSA-Style Points-To Analysis for Low Level Code.
- *Adrian Rebola Pardo*. A Theory of Satisfiability-Preserving Proofs in SAT Solving.
- *Pavel Čadek*. Upper and Lower Loop Bound Estimation by Symbolic Execution and Loop Acceleration.
- *Anton Xue*, Ross Mawhorter, Gian Pietro Farina and Stephen Chong. Towards the Formalization and Analysis of R.
- *Maxwell Shinn*, Clarence Lehman and Ruzica Piskac. Runtime Verification of Scientific Software.

The 2018 student forum also featured a Best Contribution Award (based on the quality of the submission, the poster, and the presentation), announced during the conference and publicized on the FMCAD website.[1]

The Student Forum would not have been possible without the excellent contributions of the student authors. The help and advice of Georg Weissenbacher, who organized the earlier FMCAD 2015 student forum was invaluable. We would also like to express our gratitude to all the reviewers of the FMCAD Student Forum for their work.

REFERENCES

[1] K. Heljanko. The FMCAD 2017 graduate student forum. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 10–10. FMCAD Inc, 2017.
[2] H. Hojjat. The FMCAD 2016 graduate student forum. In *Formal Methods in Computer-Aided Design (FMCAD), 2016*, pages 8–8. IEEE, 2016.
[3] R. Piskac. The FMCAD 2014 graduate student forum. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 13–13. IEEE, 2014.
[4] T. Wahl. The FMCAD graduate student forum. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 16–17. IEEE, 2013.
[5] G. Weissenbacher. The FMCAD 2015 graduate student forum. In *Formal Methods in Computer-Aided Design (FMCAD), 2015*, pages 8–8. IEEE, 2015.

---

[1] https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD18/student-forum/

ISBN: 978-0-9835678-8-2.

CoSA: Integrated Verification for Agile Hardware Design

Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan
Stanford University
Stanford, California (USA)
{mattarei, makaim, clarkbarrett, ross.daly, dhuff, pmh}@stanford.edu

***Abstract*—Symbolic model-checking is a well-established technique used in hardware design to assess, and formally verify, functional correctness. However, most modern model-checkers encode the problem into propositional satisfiability (SAT) and do not leverage any additional information beyond the input design, which is typically provided in a hardware description language such as Verilog.**

In this paper, we present CoSA (CoreIR Symbolic Analyzer), a model-checking tool for CoreIR designs. CoreIR is a new intermediate representation for hardware. CoSA encodes model-checking queries into first-order formulas that can be solved by Satisfiability Modulo Theories (SMT) solvers. In particular, it natively supports encodings using the theories of bitvectors and arrays. CoSA is closely integrated with CoreIR and can thus leverage CoreIR-generated metadata in addition to user-provided lemmas to assist with formal verification. CoSA supports multiple input formats and provides a broad set of analyses including equivalence checking and safety and liveness verification. CoSA is open-source and written in Python, making it easily extendable.

I. INTRODUCTION

Formal verification has become an important part of the design process, particularly in the hardware domain. As hardware and software systems become increasingly complex, more time than ever before is spent on verification to avoid costly and potentially dangerous bugs.

For many years, hardware model-checking experts focused on general techniques applicable to any design provided in a standard format such as a hardware description language (HDL) or AIGER [6], without any extra information from the designers. While there has been impressive progress, these techniques still often fail to scale on industrial-sized systems. This requires verification engineers to either shrink the parameter sizes if possible, or manually add additional lemmas. Frequently, these additional lemmas are simple invariants which are known by the designer or design tool, but are not easily inferred by the formal system.

This paper introduces the CoreIR Symbolic Analyzer (CoSA), a model-checking tool for the hardware intermediate representation CoreIR [11]. CoSA can leverage additional knowledge provided by CoreIR to improve performance on many classes of proofs.

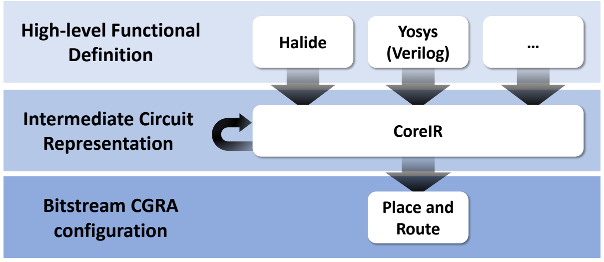


Fig. 1. AHA Flow

CoSA was developed as a tool for verifying correctness at various stages of the toolflow in the Agile Hardware (AHA) Project at Stanford University [18]. This project aims to improve performance and design productivity by incorporating ideas from agile software development to speed up the development cycle.

Compared to the software community, there are very few open-source tools for hardware design and verification. As seen in the software domain, open-source tools can help encourage innovation and distribute effort, the latter of which is particularly lacking in the hardware community. Furthermore, in the last decade, open-source SMT solvers have become powerful tools for verification, and the community no longer needs to rely exclusively on commercial tools. In support of these goals, the Agile Hardware Project is developing an end-to-end open-source toolchain.

The rest of the paper is organized as follows: Section II provides background on CoreIR and the Agile Hardware Project; Section III describes CoSA's supported formal analyses, architecture, and integration with design; Section IV describes a set of applications of the tool; Section V covers related work on hardware verification tools; and Section VI provides concluding remarks.

II. COREIR

CoreIR is an intermediate representation and compilation framework for digital designs [11]. It is front-end agnostic and thus can be a compiler target for any language representing hardware designs. Primitives in the IR have the same semantics as the SMT theory of bitvectors [3], allowing for easy formal verification integration. CoreIR can be transformed into custom back-ends using a flexible pass framework, and serialized into different hardware and SMT-based formats.

This research was supported in part by the Defense Advanced Research Projects Agency (contract FA8650-18-2-7854) and by gifts from Intel Corporation (through the Stanford Agile Hardware Project) and Cisco Systems.

ISBN: 978-0-9835678-8-2.

In the AHA toolflow [18], depicted in Figure 1, a user first writes an application in a high-level language, such as the image processing domain-specific language, Halide [19]. This compiles to CoreIR and then goes through several optimization passes before being mapped to a back-end. One of the main targets of the AHA tool flow is a custom Course-Grained Reconfigurable Array (CGRA). The CGRA is designed to have the flexibility of an FPGA while improving performance on certain kinds of applications (e.g. image processing) [23]. This performance is gained by configuring at the word level and by composing specialized heterogeneous tiles containing memories and dedicated processing elements (essentially ALUs). A set of place and route tools produce a bitstream which configures the CGRA to implement the application.

As shown in Figure 1, other high-level hardware description languages can integrate with CoreIR in addition to Halide. In fact, the CGRA is written in Verilog, which is compiled into CoreIR using the VerilogToCoreIR [13] Yosys [25] pass. Another example is the hardware design language Magma [21].

The verification goals in the AHA project include assessing functional correctness of the CGRA, as well as verifying that the firmware produces the correct configuration for the high-level, behavioral definition from Halide. Given these requirements, we integrated the formal verification at the CoreIR level, thus allowing us to support the required analyses.

III. CoSA: CoreIR Symbolic Analyzer

CoSA integrates with CoreIR to provide formal analyses. In this section we explain the analyses supported by the tool and describe its architecture.

A. Formal Analyses

CoSA reduces all analyses to symbolic model-checking problems [10]. The underlying theoretic model is a Symbolic Transition System (STS), as expressed in Def. 1.

Def. 1 (Symbolic Transition System)**.** A *Symbolic Transition System* is a tuple $S = \langle V, I, T \rangle$ where V is a set of (input V_I, state V_S, and output V_O) variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula representing the transitions. A *state* of S is an assignment to the variables V_S.

The core analyses of CoSA are primarily based on *safety* and *liveness* checking. A *safety* property is a formula φ which should hold in every state of an STS M (denoted in Linear Temporal Logic [22] as $M \models G\varphi$). This is essentially invariant verification, meaning that if the property holds then φ is an invariant of the system. If the property does not hold, an execution of the system that leads to $\neg\varphi$ is typically provided as a counterexample.

Alternatively, a *liveness* property is a formula φ which should hold infinitely often in every execution of an STS M (denoted $M \models GF\varphi$), A practical example of this analysis is to verify that a processor is always going to be ready to receive a new command. In liveness verification, a counterexample is an execution where, at some point, φ no longer holds along an infinite execution path. A typical representation of such a trace is a "lasso-shaped" execution, in which the last state of the trace is equal to one of the previous states.

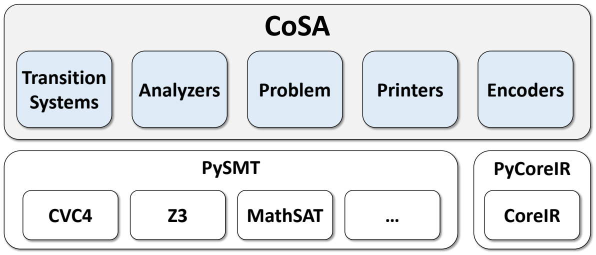


Fig. 2. CoSA Architecture

When analyzing circuit designs, it is often necessary to perform equivalence checking between two systems. The checking is usually based on standard safety verification on a synchronous combination of the systems under analysis, as expressed in Definition 2.

Def. 2 (Synchronous Product of STS)**.** Given two Symbolic Transition Systems $S_1 := \langle V_1, I_1, T_1 \rangle$ and $S_2 := \langle V_2, I_2, T_2 \rangle$ where $V_1 \cap V_2 = \emptyset$, the synchronous product S of S_1 and S_2, namely $S_1 \times S_2$, is defined as $S := \langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$.

B. Verification Engines

CoSA analyzes model-checking problems with Bounded Model-Checking (BMC) [5] techniques, and encodes them using SMT formulas. For each analysis, CoSA provides techniques able to prove or disprove the property. More specifically, for the counterexample generation of safety and liveness verifications the tool relies on BMC [5], while K-Induction [20]/Interpolation [15] and K-Liveness [9] are used to prove safety and liveness properties, respectively.

C. Framework

CoSA [14] is written in Python and its usage is regulated by the modified BSD license. As represented in Figure 2, CoSA builds on top of PySMT [12], which provides a solver-agnostic Python library to interface with SMT solvers. The internal architecture of CoSA is divided into the following parts:

- Transition Systems: defines the internal representation of the model, which is based on a hierarchical set of Transition Systems;
- Analyzers: implements the logic responsible for solving a verification problem. This includes BMC engines and liveness checking;
- Problems: used to define and manage the status of a verification problem;
- Printers: provides support for trace printing (i.e., textual or VCD format), and model translation such as the generation of an SMV file [8];
- Encoders: responsible for encoding different model descriptions into the internal representation. This includes interpreting CoreIR models, and extracting additional information used to optimize the verification process.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Case Study	# State Vars	Total # Bits
A	44	14,771
B	110	27,307
C	1,029 (5 Arrays)	414,847

TABLE I
SIZES OF THE CASE STUDIES - REPORTED FOR COMPOSED SYSTEMS.

For added flexibility, CoSA supports multiple input formats, all of which get translated internally into STS's. In fact, the model under analysis is defined using a list of files whose STS's are synchronously combined (see Def. 2) to produce a single STS. The supported input formats are CoreIR, Explicit-state Transition System (ETS), Symbolic Transition System (STS), and BTOR2 [16]. More information on the input formats is provided in [14]. This approach allows the user to describe complex analyses without modifying the original CoreIR model. For instance, the analysis of programmable hardware often requires a configuration sequence before checking its behavior. This sequence typically includes a reset procedure, for both pos-edge and neg-edge registers, as well as a configuration phase which sequentially loads a bitstream through the configuration port. CoSA facilitates a clear separation between hardware definition, e.g., CoreIR design, and configuration sequence, e.g., ETS. CoSA can generate SMT-LIB files for each of the analyses. Moreover, the ability to translate to SMV format makes it possible to use additional model-checkers such as nuXmv [8].

IV. CASE STUDIES

Below we include several case studies illustrating the utility of CoSA. All of these examples come from the Agile Hardware Project, and cover various stages in the Agile Hardware flow including hardware design, optimization passes, and mapping image processing applications to reconfigurable hardware. All models were translated to CoreIR from (System) Verilog or Halide in order to be analyzed with CoSA. Table I reports the number of variables in the models, including the total size in Bits. All experiments were run on a 2.6GHz Intel Core i7 with 16GB of RAM, and we compared with Yosys, as a reference for open-source word-level model checking.

A. Hardware: Global Controller

The global controller is responsible for configuring the CGRA, managing clock domains, and reading register values for debugging. This module interfaces the JTAG controller, which handles serial communications to and from the chip, with the main CGRA fabric. In this case study, we focused on verifying the global controller in isolation.

The global controller has a register named **state** which records the current state. Certain operations might take multiple cycles to complete, so it uses a counter to keep track of the number of cycles. At the beginning of an operation, the counter is set to the expected delay, and the controller returns to the **ready** state when the counter reaches zero.

Table II lists a selection of properties we attempted to verify using CoSA and the result of each. For the third property, CoSA exposed a bug in the design that could cause the global controller to be stuck in the current state for 2^{32} cycles. The global controller allows the user to configure the operation delay, and because of subtle timing issues, the counter is assigned to the user-specified delay minus one. Thus, if the user asks for a delay of zero, the counter underflows. In this case, the counter would count down starting at the maximum value of a 32-bit unsigned integer and the only way to recover would be to reset the controller. This issue was fixed by special-casing zero-delay requests.

Property	Result
Always return to **ready** state, assuming counter delay < 10	**T**
When not in **ready** state, the counter always decreases	**T**
No underflow in counter	**F**
Read signal is high implies the controller is in the **read** state	**T**
Write signal is high implies the controller is in the **write** state	**F**

TABLE II
PROPERTIES FOR THE GLOBAL CONTROLLER

CoSA also found a counterexample trace in which the write signal could be corrupted. This is accomplished by asking the global controller to switch clock domains, then immediately requesting a write operation. The clock domain switch disables all other operations until the switch is completed, but there is a delay of one clock cycle. Thus, if the write signal is enabled within that delay, it is kept high throughout the clock domain switch, but the controller is not in the **write** state. While interesting, this could not happen in the full system, because it always takes multiple cycles to produce each operation through the JTAG controller.

We also compared the performance of CoSA against the Yosys verification engine, only considering safety properties since Yosys does not natively support liveness checking. We ran the SMT solver CVC4 [1] on the SMT-LIB generated by CoSA and by Yosys (configured with Verific [24] bindings for parsing temporal SystemVerilog Assertions). It takes 4.684s to check all the properties generated by CoSA and 5.395s to check the properties generated by Yosys. The runtimes are comparable, with CoSA running slightly faster.

B. Software: Fold-Constants Pass

CoreIR has an extensible infrastructure for optimization and analysis passes on hardware designs. In the context of the Agile Hardware Project, the design goes through multiple passes before being placed and routed on the fabric. To catch bugs as close to the source as possible, it is desirable to check that these passes produce functionally equivalent designs.

CoSA supports equivalence checking on CoreIR design files and, when necessary, incorporates extra information provided by the CoreIR pass to assist in the proof.

The fold-constants pass is interesting because it can change the number of state variables in the system, which traditionally makes equivalence checking far more difficult. The pass takes any subgraph of the design which is always constant and replaces it with a constant module. The replaced subgraph could be combinational logic operating on constants, or it could be a register which never changes value.

1) Equivalence Checking: Although this pass modifies the design, the functional behavior of the system should not

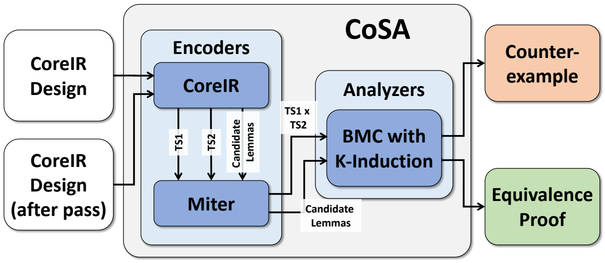


Fig. 3. CoSA automatic proof decomposition strategy for CoreIR passes

change. Given two STS's S_1 and S_2, we need to check that $S_1 \times S_2 \models G(V_{I_1} = V_{I_2}) \implies G(V_{O_1} = V_{O_2})$.

A pure SMT-based K-Induction technique could solve this problem; however, it does not scale well even for moderately sized systems. Alternatively, a verification expert could manually add additional lemmas, but this is time-consuming and procedural. Instead, our approach is to generate lemmas from CoreIR, as depicted in Figure 3. In this specific case, these lemmas express the part of the circuit that has been replaced with a constant by CoreIR, and CoSA adds them as assumptions for the equivalence proof only if they are invariants in the model.

With this proof decomposition, CoSA can check 52 lemmas and prove equivalence between pre-pass and post-pass CoreIR of a CGRA processing element tile configured to do a multiplication in 50 seconds, whereas K-Induction without the additional lemmas does not complete in 2 hours. To compare with Yosys, we produced Verilog from CoreIR for the pre-pass and post-pass designs. These were instantiated together in a top module, similar to the synchronous product encoding in CoSA. K-Induction in Yosys was also unable to prove equivalence in 2 hours.

C. Firmware: Sequential Equivalence of Design and Configured Hardware

We have shown above that CoSA can prove properties of Verilog designs, as well as functional equivalence between CoreIR designs transformed by optimization passes. It is also useful to verify that the configured CGRA faithfully implements the application described by a CoreIR file.

As a simple example, we generated CoreIR that implements a 2x1 convolution, henceforth referred to as the application. This was mapped to CGRA primitives, and then the place and route tools were used to produce a bitstream for a 4x4 CGRA. From the bitstream, we generated an ETS, S_{ETS}, which toggles configuration signals and passes the bitstream to the CGRA inputs. We simulated the CGRA synchronized with S_{ETS} in CoSA to configure the CGRA.

For performance reasons, it helps to simulate without unrolling. In this case, the transition relation was only unrolled one step. The SMT solver was called repeatedly to generate the next step, and the initial state was reassigned each time. A separate check can verify that the configuration phase is deterministic and correct. For space reasons this is not covered here. Once the CGRA was configured, the reset and configuration signals were disabled, and the initial state was assigned to the configured state.

A 2x1 convolution slides a 2-dimensional kernel over an input image. In hardware, this is implemented serially using a linebuffer to delay input pixels. In this case, it was configured for 10x10 input images, and thus the linebuffer has depth 10.

The application implements the linebuffer using a memory with a 5-bit address and a counter. The CGRA implements the linebuffer with nontrivial use of two memories with 9-bit addresses. Convolution depends on the correct linebuffer behavior; thus, these memories could not be soundly blackboxed in a SAT-based model checker. CoSA encodes memories from both the application file and the translated CGRA using the SMT theory of arrays.

We were unable to prove full equivalence because, due to the linebuffers, the equivalence property is not inductive. Unfortunately, we also cannot strengthen the property with array extensionality because of the different use and address widths of memories in the two linebuffer implementations: the memory abstractions are incomparable via standard array equivalence. However, in 2 minutes CoSA was able to prove that if reset is held low, the configuration of the CGRA does not change. Furthermore, CoSA showed in just over 80 minutes that, under basic assumptions of correct usage, the configured CGRA matches the behavior of the CoreIR 2x1 convolution for all executions up to 20 cycles (10 cycles of valid pixel output). For the first ten cycles, inputs are invalid. Thus, CoSA begins sequential equivalence checking once the linebuffer is full and output pixels are valid. Full verification with larger designs is the aim of ongoing work.

V. RELATED WORK

BtorMC [17] is a word-level model checker that relies on the SMT-solver Boolector 3.0 [17] to solve (invariant) model checking problems using bounded techniques [4]. Differently from CoSA, BtorMC is tightly integrated with Boolector, and it does not allow for a simple integration with different solvers.

Yosys [25] is an open source Verilog synthesis suite that provides SMT-based invariant model checking. It interfaces with SMT solvers via SMT-LIB [2] files. Yosys can also rely on ABC [7] for other analyses such as liveness checking. However, ABC engines are based on an encoding into SAT.

VI. CONCLUSION

In this paper we introduced the CoreIR Symbolic Analyzer (CoSA), an open-source formal verification tool for CoreIR. CoSA provides a broad set of SMT-based formal analyses including model checking and equivalence checking. Moreover, CoSA is able to automatically extract additional information, such as lemmas, from CoreIR to speed up verification tasks.

A series of case studies from the Agile Hardware (AHA) Project at Stanford University [18] were described in order to show that CoSA is capable of handling real hardware verification problems.

For future work, we intend to extend the functionality of CoSA to include full support of Linear Temporal Logic (LTL) and additional input formats such as SMV.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

REFERENCES

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. Cvc4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, jul 2011. Snowbird, Utah.

[2] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[6] A. Biere, K. Heljanko, and S. Wieringa. Aiger 1.9 and beyond. *Available at fmv. jku. at/hwmcc11/beyond1. pdf*, 2011.

[7] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[8] 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 *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[9] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 52–59. IEEE, 2012.

[10] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *International Conference on Computer Aided Verification*, pages 419–422. Springer, 1996.

[11] R. Daly. CoreIR: A simple LLVM-style hardware compiler. https://github.com/rdaly525/coreir, 2017.

[12] M. Gario and A. Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 373–384, 2015.

[13] D. Huff. Verilog to CoreIR translator. https://github.com/dillonhuff/VerilogToCoreIR, 2018.

[14] C. Mattarei. CoSA: CoreIR Symbolic Analyzer. https://github.com/cristian-mattarei/CoSA, 2018.

[15] K. L. McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

[16] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , BtorMC and Boolector 3.0. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[17] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. BTOR2, BtorMC and Boolector 3.0. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17*, Lecture Notes in Computer Science. Springer, 2018.

[18] J. Parkhurst, M. Horowitz, P. Hanrahan, and C. Barrett. AHA Agile Hardware Project. https://aha.stanford.edu/, 2018.

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[20] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[21] S. University. Magma: a Hardware Design Language Embedded in Python. https://github.com/phanrahan/magma, 2017.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[23] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz. Evaluating programmable architectures for imaging and vision applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[24] Verific Design Automation. Verific. http://www.verific.com/.

[25] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

ISBN: 978-0-9835678-8-2.

ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification

Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, Sharad Malik
Princeton University, USA

***Abstract*—Modern Systems-on-Chip (SoCs) integrate heterogeneous compute elements ranging from non-programmable specialized accelerators to programmable CPUs and GPUs. To ensure correct system behavior, SoC verification techniques must account for inter-component interactions through shared memory, which necessitates reasoning about *memory consistency models* (MCMs) This paper presents ILA-MCM, a symbolic reasoning framework for automated SoC verification, where MCMs are integrated with *Instruction-Level Abstractions (ILAs)* that have been recently proposed to model architecture-level program-visible states and state updates in heterogeneous SoC components.**

ILA-MCM enables reasoning about system-wide properties that depend on functional state updates as well as ordering relations between them. Central to our approach is a novel *facet* abstraction, where a single program-visible variable is associated with potentially multiple facets that act as auxiliary state variables. Facets are updated by ILA "instructions," and the required orderings between these updates are captured by MCM axioms. Thus, facets provide a symbolic constraint-based integration between operational ILA models and axiomatic MCM specifications. We have implemented a prototype ILA-MCM framework and use it to demonstrate two verification applications in this paper: (a) finding a known bug in an accelerator-based SoC, plus a new potential bug under a weaker MCM, and (b) checking that a recently proposed low-level GPU hardware implementation is correct with respect to a high-level ILA-MCM specification.

I. INTRODUCTION

Systems-on-Chip (SoCs) integrate specialized hardware to meet the power-performance requirements posed by emerging applications. Specialized hardware can be programmable (e.g., Graphics Processing Units or GPUs) or non-programmable (e.g., an AES cryptographic accelerator). They outperform general purpose processors in specific domains like machine learning [1], scientific computation [2], and cryptographic operations [3]. The multiple processing units in an SoC typically run concurrently. This concurrency can be difficult to reason about, leading to design and implementation bugs in functional correctness as well as security. Furthermore, when SoC components interact via shared memory or memory-mapped input and output (MMIO), one also needs to reason about *memory consistency models* (MCMs). Although programmers generally find it easier to think about concurrent code with sequentially consistent (SC) ordering semantics, modern instruction set architectures (ISAs) have weaker MCMs in an effort to achieve better performance and scalability.

Previous MCM verification efforts have focused on modeling and analyzing MCMs at different levels of the software/hardware stack in parallel systems [4–11]. These approaches typically use small parallel programs, called *litmus tests*, for reasoning about the MCMs themselves. They focus on *ordering relations* between simple instructions, rather than on symbolic reasoning of complex control and data flow in programs, which is often needed in SoC verification. Moreover, none of these efforts consider non-programmable hardware accelerators, which may not have an ISA.

Recently, an instruction-centric operational model for heterogeneous SoC components has been proposed, called an Instruction-Level Abstraction (ILA) [12]. Analogous to a processor ISA, an ILA models a hardware component's program-visible states and their updates in the form of instructions. This provides a well-defined interface between sequential software and the underlying hardware component. For an accelerator, its ILA instructions correspond to commands at its interface. ILAs have been successfully generated (using semi-automated synthesis-based techniques) for many accelerators in practice [12–14]. In the rest of this paper, we use "instructions" to denote ILA instructions, which correspond to instructions in a processor ISA or to derived instructions for an accelerator.

An ILA can uniformly model *rich* instruction semantics (i.e., including control and data flow) of a *single* processing unit, e.g., a processor or an accelerator. Although existing MCM specifications and verifiers are well-suited for representing orderings between memory operations of multiple processing units, they lack such rich instruction models. We show that for general SoC verification, it is essential to reason about *both* rich instructions in heterogeneous components and memory orderings between them.

In this paper, we address this central challenge by proposing a general symbolic framework called ILA-MCM, shown in Figure 1. In this framework, each processing unit in an SoC, such as a programmable processor or an accelerator, is uniformly represented by an ILA. The MCM is described using axioms, as in previous efforts [4–11], but is *integrated with the ILA operational models*. This enables our ILA-MCM framework to reason about functional state updates in instructions as well as the effects of MCMs, thereby supporting expressive properties involving both states and orderings for SoC verification.

A novel feature of our ILA-MCM framework is the *facet* abstraction, where a single program variable in an instruction can be associated with multiple auxiliary state variables called facets in the verification model. Facets are useful for modeling

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

This work was supported in part by the National Science Foundation, XPS Program, Grant No. 1628926.

ISBN: 978-0-9835678-8-2.

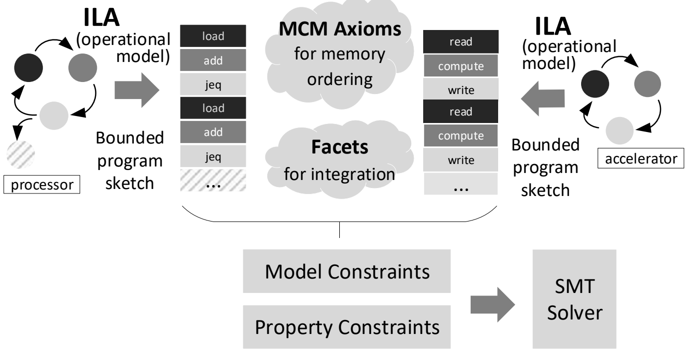


Fig. 1. The ILA-MCM Framework for heterogeneous SoC verification

memory subsystems and consistency effects, where different observers in an SoC may see logically distinct values of the same program-visible variable. The allowed values of facets are constrained by the operational semantics of the instructions as well as the memory consistency axioms. Thus, facets form a critical link between operational ILA models and axiomatic MCM specifications.

Another feature is that our verification procedure supports both operational and axiomatic models in general. (For example, our second application uses a low-level operational model for memory consistency.) The executions of operational models (e.g., ILAs) are based on a *program sketch* [15][1], which depends on the property to be verified. This creates *symbolic trace events* (events, in short). Each event is guarded by a condition and updates the state in an ILA or a facet. The axioms are then instantiated, which may create additional events or impose *happens-before* [16] ordering relations between events. We refer to these sets of constraints as the *model constraints*. Finally, we add *property constraints* that refer to states and ordering requirements for verification.

We use standard theories in first order logic to capture all constraints, including the semantics of instructions in a program and happens-before ordering relations between events. The formula comprising all constraints is checked by a Satisfiability Modulo Theory (SMT) solver [17]. Our framework supports diverse verification tasks formulated as SMT queries, including finding bugs (via falsification) or proving correctness (via verification condition generation). We have implemented a prototype ILA-MCM framework and demonstrate its use in two challenging SoC verification applications in this paper.

To summarize, this paper makes the following contributions:

- **ILA-MCM framework:** We propose a framework that combines operational models for processing cores (including accelerators) with axiomatic memory consistency models to enable SMT-based reasoning of complex interactions between hardware, software, and memory subsystems in heterogeneous SoCs.
- ***Facet* abstraction:** We propose the facet abstraction, where a single program-visible state variable can be associated with multiple logically-distinct variables, to represent updates on program-visible states with memory consistency effects. The facets provide the basis for a constraint-based integration of ILAs with MCMs.
- **Evaluation on real-world SoCs designs:** First, we show an application of the ILA-MCM framework for finding security bugs in SoC firmware [18], where our support for expressive properties enables finding a malicious exploit from a program sketch. Second, we show an application for checking correctness of a low-level GPU hardware implementation [19] against a high-level ILA-MCM specification, where our instruction-centric approach enables its decomposition into simpler verification tasks.

An overview of various components in the ILA-MCM framework is shown in Figure 2, annotated by the section numbers that describe these components. We start by introducing the relevant background on ILAs and MCMs.

II. BACKGROUND

A. Instruction-Level Abstraction (ILA)

An ILA is a uniform abstraction for hardware accelerators as well as general-purpose/specialized programmable processors [12]. It is an operational model that captures updates by hardware to *program-visible states* (i.e., the states that are accessible or observable via a user-facing program instruction). It can be viewed as a generalization of the processor ISA in the heterogeneous context, where the instructions for accelerators are defined as the commands on their interface that update program-visible states. In an ILA, each instruction has a decode condition, and the instruction executes only when this condition is true. An ILA also supports hierarchy, where an instruction at a high level can be represented as a *sequence* of *child instructions* at a lower level, as shown in Figure 2 for `Instr A` of `ILA1` (under the "ILAs" column). Thus, the granularity of ILA instructions can vary, ranging from processor instructions to software functions. Furthermore, an ILA is used for modeling a sequential thread of control, while parallelism is modeled using multiple such threads.

B. Memory Consistency Model (MCM)

An MCM provides a specification to a programmer of the order in which memory operations appear to execute [20]. Sequential consistency (SC), defined by Lamport [21], specifies that: (1) memory accesses preserve the order within each thread of a program, and (2) across threads, there is an order of accesses that every observer agrees upon. Despite the intuition of SC, nearly all modern ISAs adopt MCMs weaker than SC. A weak MCM allows certain memory accesses to be reordered within a program, and supplies fences or other synchronization mechanisms to enforce required orders when necessary. For example, the Total Store Order (TSO) model allows a load to be reordered with earlier stores that access a different address to allow the store-buffer optimization [22].

Figure 3 illustrates the effects of MCMs on a small multi-threaded program with a proposed outcome, called a litmus test. In this litmus test, each thread executes a store (`st`)

---

[1] Similar to automated program synthesis, the "holes" in our program sketch are filled in by a solver.

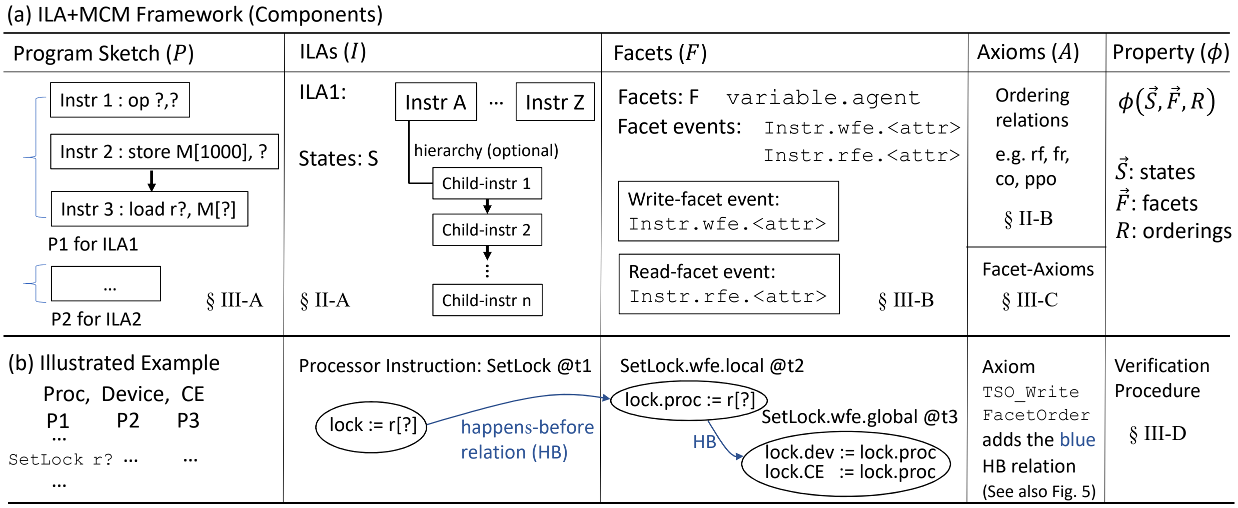


Fig. 2. Components of the ILA-MCM framework, with example fragments.

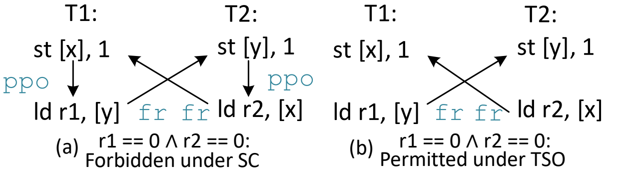


Fig. 3. A forbidden outcome in SC can become permitted under a weaker MCM. Arrows show the ordering relations (in blue) between instructions.

and then a load (`ld`) instruction, where all memory locations and registers are initially 0. Figure 3(a) assumes the SC MCM, and thus forbids a program outcome where both load instructions return 0. This is evident in a cycle of edges that comprise the preserved program order between the store and load instructions (shown as *ppo* edges) and the order between the read in one thread and the write in the other (shown by from-read (*fr*) edges). In contrast, under TSO (Figure 3(b)), the *ppo* edges are removed (since a read can be reordered with an earlier write), so the proposed outcome is permitted since there is no cycle. In general, MCMs also consider the *co* edge (coherence order between writes to the same address) and the *rf* edge (reads-from order from a write to a load which reads from that value).

C. Gaps in Prior Work

Despite a rich history of prior work in MCM verification, they lack some key capabilities described below.

Symbolic Reasoning with Conditional Orderings. Our main goal is to support general verification of SoC software and hardware. However, most prior efforts in MCM verification rely upon an explicit enumeration over addresses, data, and conditional predicates that may affect orderings between memory operations. Specifically, we consider the following two types of conditional orderings: ❶ relations involving predicated instructions or instructions after branches, and ❷ relations involving address/data-dependent values.

For example, Figure 4(a) shows ❶, with a predicate `p1` on the last load instruction in thread `T2`. Note that the existence of the load event and the related *fr* edge (shown as a dashed

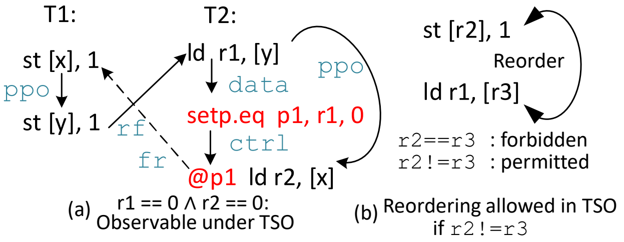


Fig. 4. Examples of conditional orderings. (a) `@p1 ld` executes iff `p1` is `true`, i.e., iff `r1==0` (setting/using predicate `p1` is marked in red). (b) In TSO, store-to-load reordering is allowed if the addresses are different.

arrow) are *control-dependent*. If this control-dependency is ignored, the analysis will *incorrectly* deduce that the graph is cyclic, i.e., the outcome is unobservable. Figure 4(b) shows an example for case ❷, where reordering is allowed only when the addresses in registers `r2` and `r3` are different.

In prior MCM efforts based on relational models, e.g., using Alloy [5] or Check tools [7–11], the addresses and data are modeled by relational predicates, e.g., whether two addresses are the same. However, such relations have to be pre-specified and are not explored symbolically in the solver. Similarly, Herd uses enumeration over all possible values of relevant addresses/data. In contrast, ILA-MCM uses symbolic reasoning to represent ordering relations dependent on complex contro/data flow and avoids explicit enumeration.

Rich Instruction-Centric Models. Most previous efforts in MCM verification focus on ordering relations between instructions, rather than on updates of program-visible states. For example, arithmetic instructions are abstracted away in relational models [5]. In Herd [4], the instructions are hard-coded and do not model bit-precise hardware (e.g., there is no register overflow behavior). Our goal is to support SoC verification by modeling rich instruction semantics for processors as well as non-programmable accelerators, which is required for reasoning about general (not just litmus) programs.

Expressive Properties. MCM verification has typically focused on specifying orderings and litmus tests, while program/processor verification has focused on state-based veri-

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

fication or control-oriented properties. We aim to support SoC verification using a wide range of expressive properties that can refer to *both* states and orderings.

III. ILA-MCM FRAMEWORK

We now provide details of the main components of our ILA-MCM framework (shown in Figure 2): program sketches, facets, axioms, and verification procedure.

A. Program Sketch

We leverage existing work on programming by sketches [23, 24] to synthesize a program that would exercise a bug or abstractly capture unbounded executions. Our program sketch comprises: (1) a set of partially-specified state updates in instructions (and any child instructions), and (2) a partial order on them. Holes (shown as question marks in Figure 2) are allowed in the sketch. These are filled in by the SMT solver during verification. Examples of holes include symbolic values (e.g., content of a memory location) or fields in an instruction encoding (e.g., address/data field of the `store` and `load` in Figure 2).

The program sketch, which needs to be provided by the user, typically depends on the correctness property. Although a program sketch has a bounded number of instructions, one can use an outer procedure to iteratively increase the bound, to perform a deeper search for bugs or for a proof by induction using given invariants. In the first column in Figure 2(b), the example considers an SoC with a processor, a device, and a cryptographic engine (CE). Thus, there are three program sketches (P1, P2, P3) and a `SetLock` instruction is illustrated in the program sketch (P1) for the processor. The second column (under ILA) shows the related event, which updates the lock variable by the value of some register (left as a hole `r?`) and associates a symbolic timestamp `t1` with the event.

B. The Facet Abstraction

To reason about the interactions between SoC components via shared memory, we need to establish a relation between program variables in instructions of different ILA models via axioms in MCMs. We model this using a novel abstraction described below.

1) State Variables for Facets: Facets are auxiliary variables associated with a *shared* program-visible state variable that can be observed by an "agent," which may be a thread, a physical structure or a processing core/accelerator, depending on the ILA modeling granularity. Facets reflect the fact that different agents may observe distinct values of the same shared variable in different orders. For example, the store-to-load reordering in TSO can result in the load seeing the new value from the store earlier than instructions on another thread. In general, each agent can potentially have its own facet for a shared variable. In our experience, this per-agent-facet is general enough to model weak consistency behaviors. (More facets can be added if one wishes to model memory consistency at the microarchitecture level, e.g., with store-buffers or caches, etc.)

We use the notation *variable.agent* for the facet that corresponds to a specific agent's view of a given program variable. For the example considered in Figure 2(b), suppose there is an on-chip interconnect between the three components, and that there is a register in the device denoting a lock. The device observes its value by directly reading the register, which is regarded as the facet of the device (denoted *lock.dev*). The device provides a memory-mapped interface, where other agents can access the lock register as if accessing a memory location. We model the lock register seen by the other agents as facets, denoted *lock.proc* and *lock.CE*, respectively.

2) State Updates for Facets: Continuing with our example, the ILA instruction `SetLock` on the processor can update the lock by writing to the memory-mapped address of the lock register in the device. The new value may first appear in the processor's local buffer, then go into a cache, and through the interconnect, propagate to the device and finally update the device's register. This could result in different agents seeing different values in different orders. We model this by creating new events: write-facet events to update facets, and read-facet events to read facets.

For example, TSO can be modeled such that each agent has a facet for a shared program variable. A store instruction creates two write-facet events, one to its own facet (local write-facet event) and the other to all other facets (global write-facet event). A load instruction corresponds to one read-facet event, since it only needs to read from its own facet. In general, any instructions or child-instructions accessing shared variables can have associated facet events. The values that facet read/write events use for updates are derived from the ILA instruction semantics, while the orderings of facet read/write events are specified by the facet-axioms in the MCM. We use the notation *instr.wfe/rfe.<attr>* to refer to the write-facet events (wfe) or read-facet events (rfe), related to a given instruction (*instr*), with a given attribute *<attr>*. In the TSO model, *<attr>* can be *local* or *global*. The example in Figure 2(b) shows two write-facet events (under Facets) related to the `SetLock` instruction under the TSO model.

C. Facet-Axioms for Integrating ILAs and MCMs

So far, we have described facets as state variables, and new facet events associated with ILA instructions that update or read them. The orderings between these events are specified by MCM axioms. For SC and TSO, the complete set of facet-axioms can be found in the Appendix. We highlight some fragments of these in Figure 5. Note that we uniformly use happens-before relations (denoted as `HB`) to specify orderings between events. In the SC model (top part), all facet read/write events are synchronous (i.e., these events occur at the same time) with the instructions (lines 1-2). In the TSO model (lower part), the two write-facet events (local or global) of a store instruction happen after the instruction and follow the program order (lines 3-9). These axioms are similar to those used in prior work, e.g., in the μspec TSO model [7], except that facet-axioms relate instructions with facet read/write events, while μspec axioms relate instructions

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

```
 1: Axiom SC_WriteFacetOrder
 2: forall w:WRITE | Sync[ w , w.wfe.global ]
    …
-------------------------------------------------------------------------
 3: Axiom TSO_WriteFacetOrder
 4: forall w:WRITE | HB[ w , w.wfe.local ]  /\
 5:   HB[ w.wfe.local , w.wfe.global ]
 6: Axiom TSO_Store
 7: forall w1:WRITE | forall w2: WRITE (not w1) |
 8:   PO[ w1, w2 ] => HB[ w1.wfe.local, w2.wfe.local]
 9:      /\ HB[ w1.wfe.global, w2.wfe.global]
    …
10: Axiom RF_CO_FR
11: forall r:READ | exists w:WRITE |
12:   SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\RF[w,r]/\(
13:      forall w2:WRITE (not w) | ( SameAddress[w,w2] /\
14:         w2.decode )=> CO[w2,w] \/ FR[r,w2] )
15: Define RF[ w, r] := …
16: Define CO[w1,w2] := …
17: Define FR[ r, w] := …
```

Fig. 5. SC and TSO axioms (fragments)

with microarchitectural structures like pipeline stages and caches. Further, axioms for other MCMs can be similarly defined. We have designed these axioms by hand (similar to prior MCM work); addressing their correctness is beyond the scope of this work.

The main highlight of the facet-axioms is that the relations over facet events in the MCM are linked with control/data flow in the ILA instructions via predicates interpreted over ILA state variables and facets. Consider the `RF_CO_FR` axiom (lines 10-14), which states that: (a) all read events should read from some executed write event with the same address, and the data values of read and write should match, (b) if a read r reads from a write w, any other executed write w_2 should not interfere. Here, the predicates `SameAddress` and `SameData` are interpreted over ILA state variables and facets. Similarly, the symbolic decode condition of an instruction (denoted *instruction.decode*) is a predicate over ILA state variables. Note also that the definitions of *rf*, *fr*, and *co* edges are based on the happens-before relation over facet-events.

D. ILA-MCM Verification Procedure

Our verification procedure is shown in Algorithm 1. Among its inputs, the first is a program sketch $P(T, R)$, where T is a set of instances[2] of partially-specified (child-) instructions, and R is a partial order. Other inputs are a set of ILAs I, the axioms A, and a property ϕ. For each possible instruction instance, the algorithm creates a trace step (simply called step) using the instruction semantics[3] (line 5). We also associate a symbolic timestamp with the step, encoded as an integer (t_a for step a). Values of timestamps only reflect relative orderings. Recall that the instructions/child-instructions may lead to facet read/write events, and steps are also created for these events (lines 6-8). Next, any happens-before orderings in the program sketch are interpreted as a less-than relation on the associated timestamps (line 10). Then, we instantiate the quantifiers and interpret the predicates in the axioms over the set of steps (line 11), and add the negation of the property (line 12). Finally, the set of constraints is checked by an SMT solver. (Our prototype uses Z3 [27].) If the constraints are satisfiable, we get a counterexample in the form of an event trace; otherwise, the property is valid within the space allowed by the program sketch. To verify unbounded correctness, we can check whether given invariants are inductive and use abstractions to model nondeterministic environments, as discussed later in Section IV-B.

Algorithm 1 ILA-MCM Verification Procedure

```
 1: procedure VERIFY($P(T, R), I, A, \phi$)
 2:     ▷ $P(T, R)$: program sketch $P$, where $T$ is a set of instances
          of (child-) instructions and $R$ is a partial order, $I$: set of
          ILAs, $A$: axioms, $\phi$: property
 3:     $C \leftarrow \top$                                   ▷ C is set of constraints
 4:     for each $ts \in T$ do
 5:         $C \leftarrow C \wedge$ CreateStep($ts, I$)
 6:         $T' \leftarrow$ AssocFacetEvent($ts, A$)          ▷ Get facet-events
 7:         for each $ts' \in T'$ do
 8:             $C \leftarrow C \wedge$ CreateStep($ts', I$)
 9:     for each $a \rightarrow b \in R$ do
10:         $C \leftarrow C \wedge t_a < t_b$                ▷ Orders are on timestamps
11:     $C \leftarrow C \wedge$ InstantiateAxioms($A$)
12:     $C \leftarrow C \wedge \neg \phi$
13:     if SMTCheck($C$) = SAT then
14:         return INVALID, GetModel($C$)
15:     else return VALID
```

IV. VERIFICATION APPLICATIONS

A. Security Bug in a Firmware Load Protocol

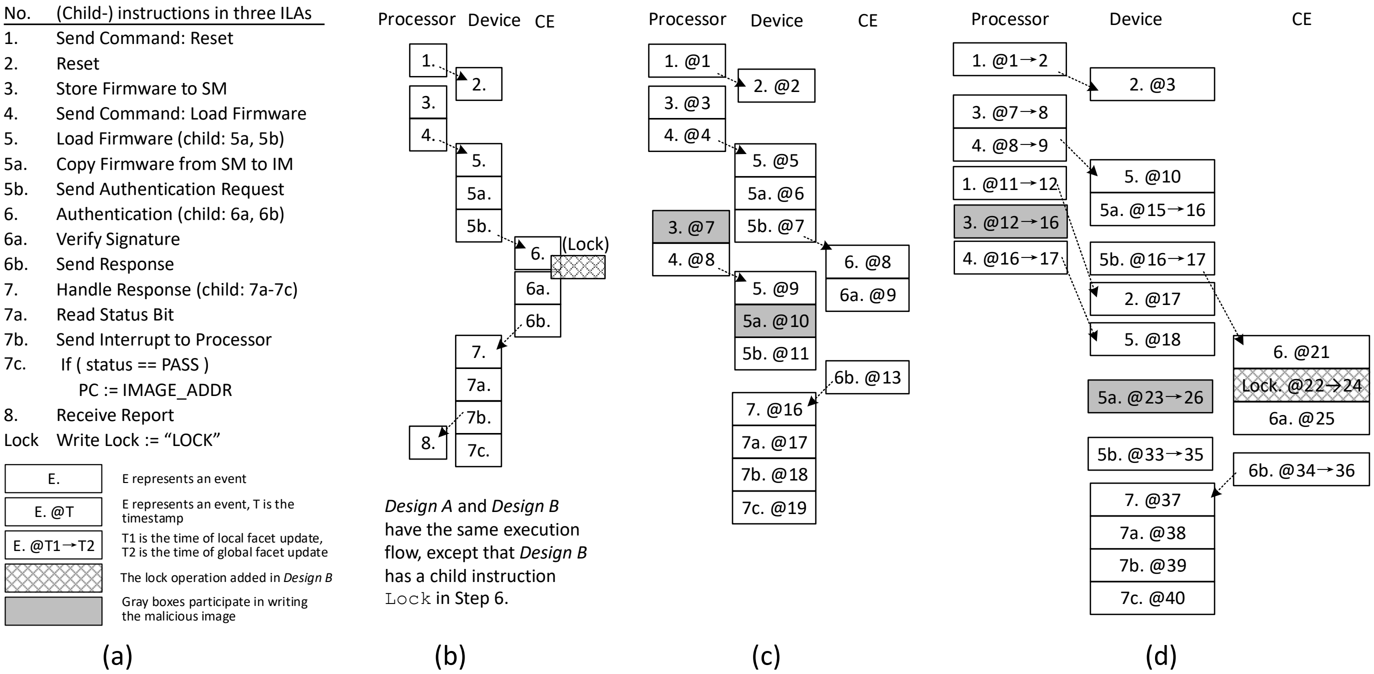
1) System Overview: The SoC [18] used in this application consists of a processor, a device, and a cryptographic accelerator engine (CE). The processor runs a driver that loads a firmware image onto the device. The CE is responsible for authenticating the image before it can be used by the device. The SoC has a system memory (SM) that all three agents can access, and an isolated memory (IM) that can only be written by the device but is readable by both the device and the CE. The threat model assumes that the driver on the processor can be compromised. The attacker's goal is to fool the device into running a malicious firmware image that does not carry a correct signature.

2) ILAs and Instructions: The first step is to construct an ILA for each of the agents: the processor, the device, and the CE. The set of instructions and child instructions are shown in Figure 6(a) (along with a legend). The processor uses store operations to send commands to the memory-mapped device or the accelerator interface, and can query the status via reading through this interface. The ILA instructions in the processor (device driver) are `Send_Command_Reset`, `Store_Firmware`, or `Send_Command_Load`. The processor also has a `Receive_Report` instruction that, when triggered by an interrupt, reads from the device's status register to learn the result of firmware image authentication. The device ILA has three instructions: `Reset`, `Load` and `Handle_CE_Response`. The CE ILA has only one instruction (`Authentication`), which handles the authentication request.

---

[2] Multiple occurrences of the same (child-) instruction are regarded as separate instances in a trace.

[3] Although not shown here, we use a concurrent static single assignment (CSSA) encoding [25, 26], where uses of shared state variables are encoded as π-variables and updates to them are encoded as new definitions.



No.	(Child-) instructions in three ILAs
1.	Send Command: Reset
2.	Reset
3.	Store Firmware to SM
4.	Send Command: Load Firmware
5.	Load Firmware (child: 5a, 5b)
5a.	Copy Firmware from SM to IM
5b.	Send Authentication Request
6.	Authentication (child: 6a, 6b)
6a.	Verify Signature
6b.	Send Response
7.	Handle Response (child: 7a-7c)
7a.	Read Status Bit
7b.	Send Interrupt to Processor
7c.	If (status == PASS) PC := IMAGE_ADDR
8.	Receive Report
Lock	Write Lock := "LOCK"

E.	E represents an event
E. @T	E represents an event, T is the timestamp
E. @T1→T2	T1 is the time of local facet update, T2 is the time of global facet update
▨	The lock operation added in *Design B*
■	Gray boxes participate in writing the malicious image

Design A and *Design B* have the same execution flow, except that *Design B* has a child instruction `Lock` in Step 6.

Fig. 6. (a) Instructions/child-instructions in the ILAs, plus legend. (b) Intended execution flow for *Designs A* and *B*, where a dashed arrow indicates an agent triggering an instruction in another agent via instruction-decode conditions. (c) Malicious exploit for *Design A* under SC, with event timestamps (`@T`) generated by the SMT solver. (d) Malicious exploit for *Design B* under TSO, with timestamps for local/global facet updates also generated by the SMT solver.

The intended execution flow of these instructions is shown in Figure 6(b). First, the driver sends a `Reset` command to the device by writing into the command register and the device performs reset (Step 1 and 2). The driver stores the firmware image in a dedicated region in SM (3) and invokes the device (4). Upon receiving the `Load_Firmware` command (5), the device copies the firmware image into its IM (child-instruction 5a) and sends an authentication request to the CE (5b). The CE checks the signature of the image in IM (6a), stores the result into its register and signals the device of its completion (6b). The device will read the verification result from the CE's address space (7a) and report the result to the driver (7b). If the result indicates that the image is authenticated, the device sets its own program counter to point to the firmware location in IM and starts its execution from there (7c). Finally, the processor handles the interrupt and knows that the firmware image has been loaded (8).

We refer to the above implementation as *Design A*, which is known to have a time-of-check to time-of-use (TOCTOU) vulnerability. Prior work originally identified and presented a solution to this vulnerability, namely *Design B* [18], where the device protects IM contents with a lock that is accessible only by the device and the CE. Once locked, the image stored in IM cannot be changed. Our ILA model for *Design B* is similar to *Design A*, except that the CE has an extra child-instruction `Lock` in ILA instruction 6 which enables the lock.

3) Program Sketch: We created a program sketch based on the instructions shown in Figure 6(a), where the solver explores which instructions to include in the malicious exploit by creating a hole for the decode condition of each instruction. Further, the values and addresses of the stores by the driver are left as holes in the program sketch.

4) Facets and Axioms: In this application, we consider two possible MCMs: SC and TSO. We use facets and axioms (shown in the Appendix) to model the MCMs.

5) The Property: The SoC should ensure the following safety property ϕ: $(DevPC = FwAddr) \rightarrow Check(IM[FwAddr]) \neq FAIL$. It says that when the device's program counter points to the region holding the firmware image, the image should not be malicious. Our verification procedure aims to synthesize an exploit that violates this property.

6) Results: Under the SC model, our verification procedure successfully reproduced the known malicious exploit [18] for *Design A* in 3.5 seconds, with a bound of 30 ILA instructions. The malicious exploit is shown in Figure 6(c), where the timestamps (`@T`) found by the SMT solver are shown for each event. Note that the correct image is authenticated, but the firmware overwrites it with a malicious image, which is then executed. This is a TOCTOU vulnerability.

Design B is intended to fix the above issue and works correctly under the SC model. However, under the TSO model, our verification procedure found a malicious exploit in 6.5 seconds, with a 32-instruction bound. To the best of our knowledge, this TSO-based vulnerability was not known before. The resulting trace is shown in Figure 6(d), where the essential problem is around timestamp 22 to 24. Although the CE updates the device's lock register at time 22, the device does not see this update until later. As shown, at time 23, the device overwrites the firmware with a malicious image. This bug can be fixed by adding a fence on the CE to ensure that the device sees the lock before the CE proceeds to authenticate.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

B. Verifying Correctness of a GPU Implementation

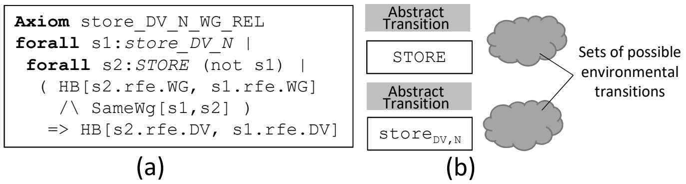
Graphics Processing Units (GPUs) often use very weak consistency models that allow for a large amount of buffering and reordering of memory requests, to provide mitigation of high memory latency. An operational model of a GPU implementation is discussed by Wickerson et al. [19]. The implementation is intended to be compliant with OpenCL [28] (a variant of the heterogeneous-race-free (HRF) MCM [29]), with an extension called *remote scope promotion (RSP)* proposed by AMD. Under OpenCL, all programs must be free of data races (i.e., two unsynchronized accesses to the same address with at least one write); the behavior is undefined otherwise. Synchronization can be achieved by an acquire-load reading from a release-store with or promoted to a matched scope.

We aim to verify that the given hardware implementation is correct with respect to a high-level specification model that we build in ILA-MCM. We should mention that our specification is actually *more conservative* than the language-level OpenCL+RSP model described by Wickerson et al. – developing an equivalent ILA-MCM model for the latter is left to future work.

1) ILA-MCM Specification Model: This model comprises the functions of store, load, and atomic increment operations, plus the ordering relations they enforce. Each operation may have additional attributes that affect the ordering relations: (a) whether it is a release (for a store), an acquire (for a load), neither, or both, (b) the scope of the synchronization, and (c) whether it promotes the scope of a remote synchronization. We model these operations using ILA instructions, where different attributes lead to different instructions, e.g., store-relaxed and store-release are modeled as two distinct instructions. They have the same state updates, but the difference in their orderings is captured by the associated MCM axioms.

The system has a hierarchical structure comprising M devices, each device with N workgroups, with a workgroup having L threads. For a shared program variable, each thread possesses a facet, and additionally each workgroup (and each device) also has a facet. A store instruction will first update the facet of its own thread (TH-facet update), then the facet of its workgroup (WG-facet update) and the device facet (DV-facet update). A load instruction will have a TH-facet-read event (and potentially WG-facet-read and DV-facet-read events).

For each instruction, we use facet-axioms to model the enforced ordering requirements. For example, for the store-release (device scope, no remote promotion) instruction $\texttt{store}_{DV,N}$, one of its axioms is shown in Figure 7(a). It says that for a $\texttt{store}_{DV,N}$ instruction s_1, for all the other store instructions s_2 different from s_1, if they are on the same workgroup and there is a happens-before relation on their WG-facet updates, then their DV-facet update events also follow a happens-before relation. For each instruction, there can be multiple axioms specifying its ordering relations with different types of instructions under different conditions.



```
Axiom store_DV_N_WG_REL
forall s1:store_DV_N |
 forall s2:STORE (not s1) |
  ( HB[s2.rfe.WG, s1.rfe.WG]
    /\ SameWg[s1,s2] )
   => HB[s2.rfe.DV, s1.rfe.DV]
```

Fig. 7. (a) An axiom for instruction $\texttt{store}_{DV,N}$ (b) related program sketch

2) SoC Implementation: The implementation model, from Wickerson et al. [19], is fully operational (does not require facets or axioms). It contains a number of GPUs, where each GPU performs a series of operations to achieve the effect of an instruction in the high-level specification. These operations are modeled as child instructions, which make use of the physical locks, FIFOs, and caches to guarantee correct data transfers and orderings.

We model 13 child instructions. Some examples are `LD` (load from L1 cache to register), `ST` (store from register to L1 cache), $\texttt{FLU}_{L1}\texttt{WG}$ (flush the L1 cache in its workgroup), $\texttt{INV}_{L1}\texttt{WG}$ (invalidate L1 cache of its workgroup). Inside a GPU, there are also other environmental transitions, e.g., a store may later trigger a cacheline flush. We model these state changes by child instructions as well.

3) Verification: We verify correctness of the implementation by checking that: (1) the program variables are updated to the same values as in the specification, and (2) the ordering of the updates is correct. The first check corresponds to functional equivalence checking between child-instructions on the GPU and the instructions in an ILA-MCM model, which can be handled using prior techniques [12]. Therefore, we focus here on the second check, where we use our facet-axioms as properties, and check if it is possible to synthesize a sequence of child instructions whose execution can violate the property. To ensure correctness using bounded traces, we need to further use invariants and abstractions.

We perform verification as follows. First we choose an instruction from the ILA-MCM specification model, collect axioms that refer to this instruction, and verify these axioms one by one. Since our facets and axioms are all instruction-centric, this instruction-based decomposition of the overall verification problem is directly enabled by our ILA-MCM framework, thereby providing a potential scalability benefit in comparison to handling all axioms monolithically.

An axiom may refer to other related instructions. For example, in the axiom in Figure 7(a) for the $\texttt{store}_{DV,N}$ instruction, there is a reference to another store instruction (of any type). We build a program sketch accordingly, as shown in Figure 7(b) for this example. Here, each of the two white boxes ($\texttt{store}_{DV,N}$ and `STORE`) denotes the sequence of child instructions that implement the high-level specification instruction, respectively. Since GPU operations may trigger environment transitions, we also add them in our program sketch. Finally, we add abstract transitions before and between the two sequences of child instructions. An abstract transition is allowed to update the state to any value (i.e., it is a $havoc$ operation), which is constrained subsequently by given invariants. The given invariants are checked separately on all child instructions (some require checking on all pairs). An example invariant is that the tail of a FIFO never passes the

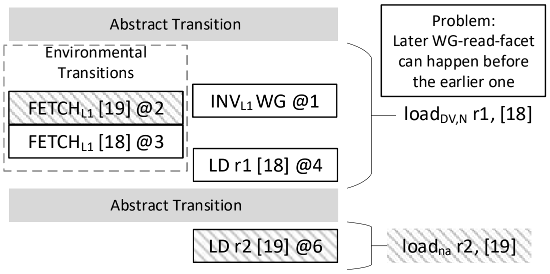


Fig. 8. The counterexample found for $\texttt{load}_{DV,N}$, where the addresses, timestamps and specific environmental transitions are generated by the solver

head, i.e., the FIFO does not underflow. In the future, we aim to maintain a library of invariants and abstract transitions for reuse. Further, the ILA-MCM verification procedure could be integrated with a general-purpose theorem prover to formally ensure their soundness and aid bookkeeping.

Although our ILA-MCM specifications are parametric, we do not perform parametric verification here, since the GPU implementation is fixed by a concrete system configuration (M, N, L). We currently performed verification for $M, N, L = 2$ and $M, N, L = 3$.

4) Results: For the original GPU implementation verified by Wickerson et al. [19], our verification failed with counterexamples for the following 5 instructions: $\texttt{load}_{DV,N}$, $\texttt{load}_{DV,R}$, $\texttt{store}_{DV,R}$, $\texttt{fetch\_inc}_{DV,N}$, and $\texttt{fetch\_inc}_{DV,R}$. Among them, $\texttt{load}_{DV,N}$, $\texttt{fetch\_inc}_{DV,N}$, $\texttt{store}_{DV,R}$ match with the buggy scenarios discussed in the previous work [19]. Specifically, Figure 8 shows a buggy trace that we found for instruction $\texttt{load}_{DV,N}$, where the facet-read event of the later non-atomic load instruction comes earlier than the facet-read event of the load-acquire instruction. This violates the load-acquire semantics. On the other hand, the counterexamples for $\texttt{fetch\_inc}_{DV,R}$ and $\texttt{load}_{DV,R}$ are false positives, since these traces cannot be extended to litmus tests with a property violation without having data races (prohibited by OpenCL). Interestingly, the proposed changes by Wickerson et al. to the compiler mappings of OpenCL+RSP operations strengthened the ordering guarantees of the hardware operations to match our ILA-MCM model. Under their new compiler mappings, we successfully validated that the hardware implementation is compliant with our ILA-MCM model. This validation was completed in 14 minutes 9 seconds (for $M, N, L = 3$) on a laptop with a 2.8GHz Core-i5 processor and 16GB memory.

V. RELATED WORK

A. Hardware Specification and Verification

A number of formal hardware abstractions have been developed that enable verification. Kami [30, 31] is a Coq-based framework that supports hardware design and verification in Bluespec. In comparison to Kami, ILA-MCM is an ISA-level abstraction that provides the interface between hardware and software. In addition to verifying hardware, it can also be used for verifying correctness/security of software interacting with accelerators, as demonstrated in our paper. Furthermore, it can reason about a wide range of memory consistency behaviors, including SC, TSO, and HRF. In contrast, currently Kami has only been applied for SC. Finally, the ILA-MCM framework targets automated reasoning using SMT solvers, in contrast to interactive theorem-proving in Kami.

ISA-Formal [32, 33] has been developed to formally model and verify ARM processors. As its name suggests, it is an ISA-level model. However, it has not been applied to accelerators or other heterogeneous SoC components. Further, as far as we know, it has not been integrated with MCMs to reason about multicore memory consistency.

B. MCM and Program Verification

We have already discussed prior MCM verification tools and techniques. For reasoning about general concurrent programs, there are many related efforts in weak consistency models [34–36], logics [37, 38], and verification tools [39–41]. Here we discuss details of specific related ideas.

1) Facets vs. ViCLs: In the Check tools [7–11], the Value in Cache Lifetime (ViCL) abstraction has been proposed to capture cache occupancy. Although both facets and ViCLs can model multiple “live” data for the same memory location, they are inherently different. First, facets are state variables that are updated according to instructions in ILAs and MCM axioms; they are not created or destroyed. In contrast, ViCLs have creation and expiration events in happens-before graphs. Second, facets are more general than ViCLs and are not necessarily tied to caches or other microarchitectural structures. Third, facets enable integration of axiomatic MCMs with operational instruction semantics, while the latter are ignored by ViCLs.

2) Facets vs. Views: In recent work [34, 40], a *view* abstraction was proposed to model the C11 MCM. Our facets are different from views as follows: (i) a view is a map from locations to timestamps, whereas facets are auxiliary state variables, (ii) the views assign explicit timestamps to events, whereas facet-axioms associate events with symbolic timestamps, whose values are not fixed but explored implicitly during verification, (iii) unlike views, facets have been applied in automated SMT-based reasoning.

VI. CONCLUSIONS

In this paper, we have presented the ILA-MCM framework, which combines the benefits of operational ILA models with axiomatic MCMs for reasoning about concurrent interactions between heterogeneous components in an SoC. We have introduced a novel facet abstraction that models consistency effects on program-visible states, and use facet-axioms to specify consistency ordering requirements. This provides a constraint-based integration between operational ILA models and axiomatic MCM specifications. Our SMT-based verification procedure supports symbolic reasoning for expressive properties involving both rich instruction semantics and orderings. We have demonstrated two verification applications of our prototype ILA-MCM framework, where we reasoned about an SoC firmware program and a GPU hardware implementation, respectively. Our support for expressive properties allowed synthesizing a malicious exploit in the first case, and our instruction-centric approach enabled compositional verification in the second.

REFERENCES

[1] C. Pilato, Q. Xu, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, “On the Design of Scalable and Reusable Accelerators for Big Data Applications,” in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 406–411.

[2] I. Ohmura, G. Morimoto, Y. Ohno, A. Hasegawa, and M. Taiji, “MDGRAPE-4: a Special-Purpose Computer System for Molecular Dynamics Simulations,” *Philosophical Transactions, Series A*, vol. 372, no. 2021, 2014.

[3] J. Rott, “Intel Advanced Encryption Standard Instructions (AES-NI),” *Technical Report, Intel*, 2012.

[4] J. Alglave and M. Tautschnig, “Herding Cats: Modelling, Simulation, Testing, and Data-Mining for Weak Memory,” *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 2, pp. 1–7, 2014.

[5] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, “Automatically Comparing Memory Consistency Models,” in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2016.

[6] J. Bornholt and E. Torlak, “Synthesizing Memory Models from Framework Sketches and Litmus Tests,” in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 467–481.

[7] D. Lustig, M. Pellauer, and M. Martonosi, “PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2015, pp. 635–646.

[8] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, “CCICheck: Using hb Graphs to Verify the Coherence-Consistency Interface,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2015, pp. 26–37.

[9] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, “COATCheck : Verifying Memory Ordering at the Hardware-OS Interface,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, no. 212, 2016, pp. 233–247.

[10] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, “TriCheck : Memory Model Verification at the Trisection of Software, Hardware, and ISA,” in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 119–133.

[11] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, “RTLCheck : Verifying the Memory Consistency of RTL Designs,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2017, pp. 463–476.

[12] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, “Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification,” 2018. [Online]. Available: http://arxiv.org/abs/1801.01114

[13] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, “Template-based Synthesis of Instruction-Level Abstractions for SoC Verification,” in *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017, pp. 160–167.

[14] P. Subramanyan, B.-Y. Huang, Y. Vizel, A. Gupta, and S. Malik, “Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. 99, 2017.

[15] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, “Syntax-guided synthesis,” in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 1–8.

[16] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability Modulo Theories,” *Handbook of Satisfiability*, vol. 185, pp. 825–885, 2009.

[18] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, “Security of SoC Firmware Load Protocols,” in *Proceedings of the International Symposium on Hardware-Oriented Security and Trust*, 2014, pp. 70–75.

[19] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Remote-Scope Promotion : Clarified , Rectified , and Verified,” in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[20] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[21] L. Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Progranm,” *IEEE Transactions on Computers*, no. 9, pp. 690–691, 1979.

[22] Intel, “Intel® 64 and IA-32 Architectures Software Developers Manual,” *Volume 3A: System Programming Guide, Chapter 8: Multiple-Processor Management*, no. 253665-067US, 2018.

[23] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, “Programming by Sketching for Bit-Streaming Programs,” in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 281–294.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, “Combinatorial Sketching for Finite Programs,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 404–415.

[25] J. Lee, D. A. Padua, and S. P. Midkiff, “Basic Compiler Algorithms for Parallel Programs,” in *Symposium on Principles and Practice of Parallel Programming*, 1999, pp. 1–12.

[26] C. Wang, S. Kundu, R. Limaye, M. Ganai, and A. Gupta, “Symbolic Predictive Analysis for Concurrent Programs,” *Formal Aspects of Computing*, vol. 23, no. 6, pp. 781–805, 2011.

[27] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[28] L. Howes and A. Munsh, “The OpenCL Specification,” 2015. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf

[29] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, “Heterogeneous-Race-Free Memory Models,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 427–440.

[30] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: a platform for high-level parametric hardware specification and its modular verification,” *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.

[31] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, “Modular Deductive Verification of Multiprocessor Hardware Designs,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2015, pp. 109–127.

[32] A. Reid, “Trustworthy Specifications of ARM® v8-A and v8-M System Level Architecture,” in *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017, pp. 161–168.

[33] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, “End-to-End Verification of ARM® Processors with ISA-

ISBN: 978-0-9835678-8-2.

Formal," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2016, pp. 42–58.

[34] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A Promising Semantics for Relaxed-Memory Concurrency," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2017, pp. 175–189.

[35] O. Lahav and D. Dreyer, "Repairing Sequential Consistency in C/C++11," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[36] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming Release-Acquire Consistency," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 649–662.

[37] V. Vafeiadis and C. Narayan, "Relaxed Separation Logic: A Program Logic for C11 Concurrency," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013, pp. 867–884.

[38] A. Turon, V. Vafeiadis, and D. Dreyer, "GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014, pp. 691–707.

[39] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2015, pp. 637–650.

[40] J.-O. Kaiser, H.-H. Dang, D. Dreyer, and O. Lahav, "Strong Logic for Weak Memory : Reasoning About Release-Acquire Consistency in Iris," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2017, pp. 1–17.

[41] A. J. Summers and P. Müller, "Automating Deductive Verification for Weak-Memory Programs," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018, pp. 190–209.

APPENDIX
FACET-AXIOMS FOR SC AND TSO

The facet-axioms for SC and TSO are shown in Figure 9 and Figure 10, respectively. In the SC model, all facet read/write events are synchronous with the instructions (lines 7-10 in Figure 9), while in TSO model, the two write-facet events of a store instruction follow the program order of the stores (lines 7-13 in Figure 10). Read-facet events are still synchronous (lines 14-15). Fences ensure that previous writes are globally visible at that point, and read-modify-write (RMW) is atomic in the sense that its read and write facets are not breakable (lines 17-21). We define additional functions to specify the corresponding read-from, from-read, and coherence-order relations based on happens-before (HB) relations over facet events, e.g., lines 13-15 in Figure 9 and lines 23-31 in Figure 10. These functions are defined for use in the first axiom in both models.

```
1  Axiom RF_CO_FR
2  forall r:READ | exists w:WRITE |
3     SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\
4        RF[w,r] /\( forall w2:WRITE (not w) |
5           ( SameAddress[w,w2] /\ w2.decode )=>
6              CO[w2,w] \/ FR[r,w2] )
7  Axiom SC_WriteFacetOrder
8  forall w:WRITE | Sync[ w , w.wfe.global ]
9  Axiom SC_ReadFacetOrder
10 forall r: READ | Sync[ r , r.rfe.global ]
11
12 Define RF[w,r] := HB[ w.wfe.global , r.rfe.global ]
13 Define FR[r,w] := HB[ r.rfe.global , w.wfe.global ]
14 Define CO[w1,w2] := HB[ w1.wfe.global , w2.wfe.global ]
```

Fig. 9. Facet-Axioms for SC

```
1  Axiom RF_CO_FR
2  forall r:READ | exists w:WRITE |
3     SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\
4        RF[w,r] /\( forall w2:WRITE (not w) |
5           ( SameAddress[w,w2] /\ w2.decode )=>
6              CO[w2,w] \/ FR[r,w2] )
7  Axiom TSO_WriteFacetOrder
8  forall w:WRITE | HB[ w , w.wfe.local ]  /\
9     HB[ w.wfe.local , w.wfe.global ]
10 Axiom TSO_Store
11 forall w1:WRITE | forall w2: WRITE (not w1) |
12    PO[ w1, w2 ] => HB[ w1.wfe.local, w2.wfe.local]
13       /\ HB[ w1.wfe.global, w2.wfe.global]
14 Axiom TSO_ReadFacetOrder
15 forall r:READ | Sync[ r , r.rfe.local ]
16
17 Axiom TSO_Fence
18 forall f:FENCE | forall w: WRITE | PO[w,f] =>
19    HB[ w.wfe.global, f ]
20 Axiom TSO_RMW
21 forall i:RMW |
22    Sync[i.rfe.local, i.wfe.local, i.wfe.global]
23 Define RF[w,r] :=
24    SameCore[w,r] => HB[w.wfe.local , r.rfe.local ] /\
25   ~SameCore[w,r] => HB[w.wfe.global, r.rfe.local ]
26 Define FR[r,w] :=
27    SameCore[w,r] => HB[r.rfe.local , w.wfe.local ] /\
28   ~SameCore[w,r] => HB[r.rfe.local, w.wfe.global ]
29 Define CO[w1,w2] :=
30    SameCore[w1,w2] => HB[w1.wfe.local, w2.wfe.local] /\
31   ~SameCore[w1,w2] => HB[w1.wfe.global, w2.wfe.global]
```

Fig. 10. Facet-Axioms for TSO

BMC with Memory Models as Modules

Hernán Ponce-de-León
fortiss GmbH
Germany
ponce@fortiss.org

Florian Furbach
TU Braunschweig
Germany
f.furbach@cs.tu-bs.de

Keijo Heljanko
University of Helsinki, Aalto University, and HIIT
Finland
keijo.heljanko@iki.fi

Roland Meyer
TU Braunschweig
Germany
roland.meyer@tu-bs.de

***Abstract*—This paper reports progress in verification tool engineering for weak memory models. We present two bounded model checking tools for concurrent programs. Their distinguishing feature is modularity: Besides a program, they expect as input a module describing the hardware architecture for which the program should be verified. Dartagnan verifies state reachability under the given memory model using a novel SMT encoding. Porthos checks state equivalence under two given memory models using a guided search strategy. We have performed experiments to compare our tools against other memory model-aware verifiers and find them very competitive, despite the modularity offered by our approach.**

Keywords: Memory models, CAT, concurrent programs, bounded model checking, SMT encodings.

I. Introduction

The semantics of concurrent programs depends on the memory model of the underlying hardware architecture. This has recently seen considerable interest [2], [6], [11], [15], [16], [21], [23], [27], [28], [46], [48]. A key insight is that, for verification purposes, the semantics is best formulated in an axiomatic style. The memory model is given in terms of assertions that constrain a set of candidate executions. A considerable achievement in this line of research is a specification language, CAT [7], [9], [15], in which basically all memory models of interest can be expressed. CAT is made for rapid prototyping. New models are easy to write so that the developer is able to quickly, yet precisely, assess the behavior of the program of interest on the corresponding hardware.

While CAT is successful as a modeling language, the tool support is lagging behind. Memory model-aware verification tools are still being developed for specific memory models. Nidhugg [2], [6] implements stateless model checking for TSO, Power, and a version of ARM. CBMC [11] is a bounded model checker for TSO. The RCMC tool [32] targets the C11 programming language. Other verification problems (e.g. fence insertion to restore sequential consistency) are tackled by Memorax [3], [4], [5], offence [13], Fender [33], and DFence [35]. These tools support TSO and similar models, such as PSO or RMO, but cannot handle Power or ARM.

What is missing are verification tools that are *modular* in the following sense: Besides the program, they should take a memory model as an input and then perform the analysis relative to that model. The Herd tool [15] accompanying CAT satisfies this requirement. Unfortunately, it is designed for litmus tests and limited to small programs.

thread t_0
`x.store(rx, 1)`

thread t_1
`y.store(rx, 1)`

thread t_2
$r_1 =$ `x.load(rx);`
$r_2 =$ `y.load(rx)`

thread t_3
$r_3 =$ `y.load(rx);`
$r_4 =$ `x.load(rx)`

Fig. 1: Program **IRIW**.

We set out to address the need for modular verification and developed two tools. Dartagnan is a safety verification engine that checks *reachability* of a (bad) state. It is modular and can handle memory models written in the core subset of the CAT language (see Fig. 4). Porthos employs this engine as a back-end and checks *equivalence* of the reachable states under two given memory models.

The following example illustrates how the hardware architecture influences the semantics of a concurrent program in subtle ways and motivates the verification problems. Consider the program **IRIW** given in Fig. 1 which is written in C11. Variables are initially set to 0. The memory order tag `rx` (for relaxed) indicates that an operation provides minimal guarantees w.r.t. the ordering of memory accesses. On x86-TSO [42], each thread has a store buffer of pending stores. When a store is propagated from a buffer to the memory, it becomes visible to all threads simultaneously. Power, on the other hand, does not guarantee that stores become visible to all threads at the same point in time. With these architectures in mind, consider the following execution: Thread t_2 reads $x = 1, y = 0$ and t_3 reads $x = 0$, $y = 1$. Since under TSO every execution has a unique global view of all store operations, this execution is impossible and a state with $r_1 = 1, r_2 = 0$ and $r_3 = 1, r_4 = 0$ is not reachable. Under Power, this is possible. The program thus behaves differently under the two memory models.

Dartagnan helps programmers find bugs due to unexpected executions. It checks whether a specified (undesirable) state can be reached in the program — relative to a given memory model. Reachability is analyzed with an efficient SMT-based bounded model checking algorithm [17], [24]. The tool computes an acyclic unwinding of the program and translates it, together with the module of the memory model and the specification of the state, into an SMT query. If the query is

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

satisfiable, the state is reachable. Otherwise it is not.

The challenge is to deal with modularity. It requires us to give an efficient encoding of all operations defined by CAT. Notably, we have to compute — in SMT — least fixpoints. They are used in prominent memory models like POWER and ARM [15]. A naive approach would implement the Kleene iteration in SAT by introducing copies of the variables for each iteration step. In [40], we showed that such an explicit iteration can be avoided by moving to an encoding based on SAT + integer difference logic.

In this paper, we present another improvement to the fixpoint encoding. For reachability, we show it is sound to encode any fixpoint, not necessarily the least one. This is the first technical contribution and implies the encoding from [40] can be simplified. DARTAGNAN implements the idea.

PORTHOS supports programmers in porting code from one architecture (for which it has been thoroughly validated) to another. The portability problem asks whether no new (potentially unsafe) states are introduced and whether all reachable states can still be reached (no functionality has been lost). PORTHOS checks this equivalence for two memory models that are given as modules. If equivalence does not hold, it reports a counterexample execution leading to a reachable state allowed by only one architecture. Equivalence checking is useful when programming performance-critical code for different architectures. Operating System kernel developers and library designers can use equivalence checks to understand whether a programming idiom, an algorithm, or a data structure that is known to work under one memory model can also be used under another.

Note that the assembly versions of the program will be different for the two architectures of interest. We address this by incorporating compiler mappings into our analysis. We return to this when we have our assembly language at hand.

State equivalence is checked in the form of inclusions in both directions. Due to the alternation of quantifiers, inclusion is notoriously difficult to check [49]: For every state reachable in one architecture we have to find an execution in the other that leads to the same state. In [40], we solved the trace inclusion problem and showed that it is easier to solve (in terms of complexity) than state inclusion. Despite that theoretical result, this paper shows that state inclusion can be solved practically using a guided search strategy.

The idea is to be pessimistic and try to disprove the inclusion. The analysis looks for a state that is reachable in one but not in the other model (like the one in the **IRIW** example above). To find states that may disprove the inclusion, PORTHOS invokes an oracle function. This oracle proposes a series of candidate states for which it gives the following guarantees.

(Progress) The series does not contain the same state twice.
(Soundness) If the oracle has no more states to propose, then the inclusion indeed holds.

Progress is certainly desirable and soundness is indispensable for verification. The interesting thing to note is that soundness leaves it to the oracle to terminate early if it finds out, by whatever reasoning, that the inclusion holds.

Our second technical contribution is the implementation of an oracle in SMT which makes progress, is sound, and may terminate early. The idea is to look for so-called *delta executions*: Executions that are inconsistent with one memory model but consistent with the other. Finding a delta execution corresponds to solving the trace inclusion problem. As we showed in [40], this does not require a quantifier alternation and can be done by suitably extending the reachability procedure of DARTAGNAN. A state resulting from a delta execution is clearly a candidate to violate the inclusion. Moreover, if there are no more states resulting from delta executions, the oracle can conclude that the inclusion holds — even if not all reachable states have been considered.

We evaluated the performance of both DARTAGNAN and PORTHOS on a benchmark suite of mutual exclusion algorithms and compared it against several other memory model-aware verification tools. Experiments show that our tools scale significantly better for larger programs.

Contributions: We report progress in memory modular verification in the form of new encoding techniques and oracle heuristics with SMT queries. In particular:

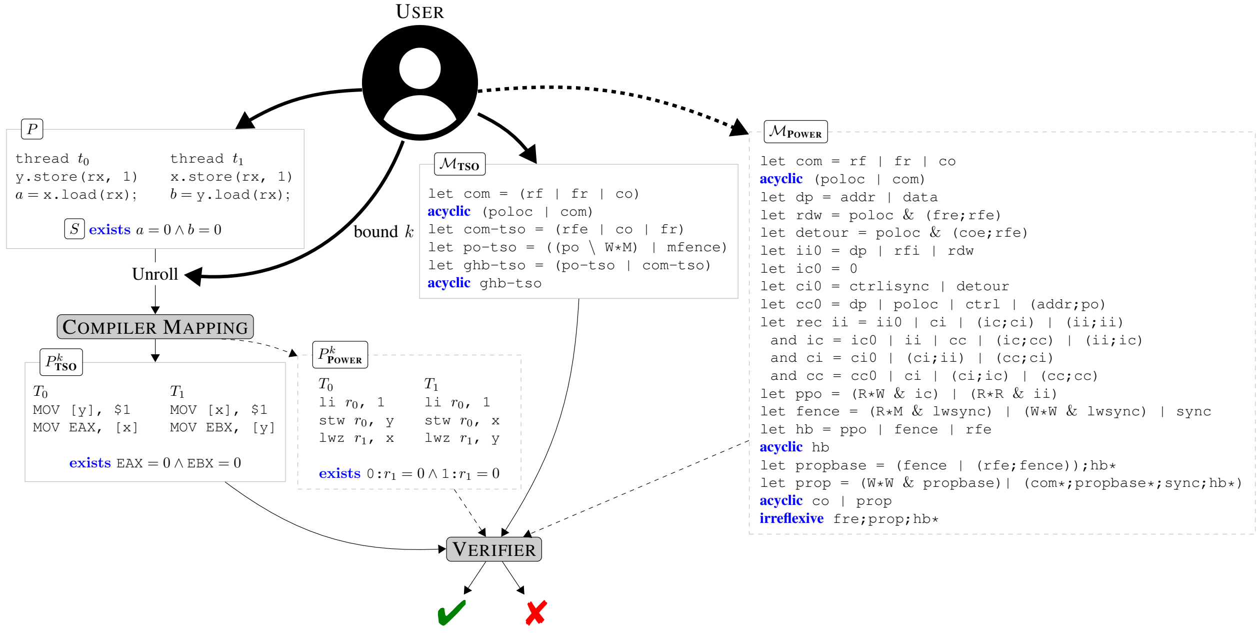
- We present two bounded model checkers for concurrent programs. Both tools are modular: They expect memory models as inputs rather than implementing the analysis for a fixed memory model.
- DARTAGNAN is a reachability checker. It simplifies our previous encoding by admitting arbitrary fixpoints. Its current implementation is an order of magnitude faster than the earlier prototype from [40]. It can be used as a back-end engine for other memory model-aware tools.
- PORTHOS is a portability checker. It implements a new method for checking state inclusion. The algorithm is an oracle-guided search that employs DARTAGNAN as a back-end. The oracle is driven by delta executions. In our experiments it requires only few iterations.
- We perform an exhaustive evaluation of DARTAGNAN and PORTHOS w.r.t. other memory model-aware tools, often observing significant speed ups. This shows the benefits of an SMT-based approach.

Outline: The remainder of the paper is structured as follows. In Section II we describe the user interface of the tools. Section III discusses the BMC for reachability. The guided search for inclusion is described in Section IV. Section V gives the experimental results. The related work is discussed in Section VI.

II. USER INTERFACE

We present our tools from a user's perspective. We examine the verification problems they solve together with the required inputs and their formats. Two verification tasks are supported: Reachability and state equivalence. The solid lines in Fig. 2

ISBN: 978-0-9835678-8-2.



```
thread t0             thread t1
y.store(rx, 1)        x.store(rx, 1)
a = x.load(rx);       b = y.load(rx);

    S exists a = 0 ∧ b = 0
```

```
let com = (rf | fr | co)
acyclic (poloc | com)
let com-tso = (rfe | co | fr)
let po-tso = ((po \ W*M) | mfence)
let ghb-tso = (po-tso | com-tso)
acyclic ghb-tso
```

```
T0                 T1
MOV [y], $1        MOV [x], $1
MOV EAX, [x]       MOV EBX, [y]

     exists EAX = 0 ∧ EBX = 0
```

```
T0              T1
li r0, 1        li r0, 1
stw r0, y       stw r0, x
lwz r1, x       lwz r1, y

exists 0:r1 = 0 ∧ 1:r1 = 0
```

```
let com = rf | fr | co
acyclic (poloc | com)
let dp = addr | data
let rdw = poloc & (fre;rfe)
let detour = poloc & (coe;rfe)
let ii0 = dp | rfi | rdw
let ic0 = 0
let ci0 = ctrlisync | detour
let cc0 = dp | poloc | ctrl | (addr;po)
let rec ii = ii0 | ci | (ic;ci) | (ii;ii)
 and ic = ic0 | ii | cc | (ic;cc) | (ii;ic)
 and ci = ci0 | (ci;ii) | (cc;ci)
 and cc = cc0 | ci | (ci;ic) | (cc;cc)
let ppo = (R*W & ic) | (R*R & ii)
let fence = (R*M & lwsync) | (W*W & lwsync) | sync
let hb = ppo | fence | rfe
acyclic hb
let propbase = (fence | (rfe;fence));hb*
let prop = (W*W & propbase)| (com*;propbase*;sync;hb*)
acyclic co | prop
irreflexive fre;prop;hb*
```

Fig. 2: DARTAGNAN (full arrows) and PORTHOS (full and dotted arrows) from the user's perspective.

illustrate the artifacts that are required for or produced by DARTAGNAN for checking reachability. The complete figure refers to testing for state equivalence with PORTHOS.

Verification Tasks: DARTAGNAN expects a program P annotated with a reachability condition S, a memory model $\mathcal{M}$ of the target architecture, and an unrolling bound k for the bounded model checking. It recursively unwinds all loops in P up to the bound k. The unwound program and the reachability condition are then mapped to the assembly dialect of the target architecture (we elaborate on compiler mappings below). The resulting acyclic and annotated assembly program is handed over to the analysis. In Fig. 2, program P is a simplified mutex algorithm which is mapped to x86 (P^k_{TSO}) using the compiler mapping in Table I. DARTAGNAN then verifies whether $\texttt{EAX} = 0 \wedge \texttt{EBX} = 0$ is reachable when running P^k_{TSO} under TSO. The definition of reachability will be given when we define memory models. In Fig. 2, we verify the mutex algorithm by checking whether both threads can read value 0 and thus enter their critical sections. Under TSO, this is possible.

For checking *equivalence*, PORTHOS expects as input the program P, two memory models $\mathcal{M}_S$ and $\mathcal{M}_T$, and an unrolling bound k. The tool checks whether the reachable states under $\mathcal{M}_T$ are the same as under $\mathcal{M}_S$. This analysis is performed on the unrolled and mapped programs. In Fig. 2, we check if the states reachable by P^k_{POWER} under POWER are the same as the ones reachable by P^k_{TSO} under TSO (which is the case). We process state equivalence queries with two inclusion checks. These queries compare the reachable states of two assembly versions of the same program running under different memory models.

Programs: Both DARTAGNAN and PORTHOS take as input programs written in a C11-like language with support for C11-atomics. Its grammar is given in Fig. 3. Programs consist of a finite number of threads. Each thread contains a sequence of operations such as while and if statements, computations on local variables, and accesses to the shared memory. We currently support Boolean and integer variables in the guards and expressions.

$$
\begin{aligned}
\langle \mathit{prog} \rangle &::= \texttt{program}\ \langle \mathit{thrd} \rangle^{*} \\
\langle \mathit{thrd} \rangle &::= \texttt{thread}\ \langle \mathit{tid} \rangle\ \langle \mathit{inst} \rangle^{+} \\
\langle \mathit{inst} \rangle &::= \langle \mathit{var} \rangle \leftarrow \langle \mathit{exp} \rangle \mid \langle \mathit{inst} \rangle ; \langle \mathit{inst} \rangle \\
&\quad \mid \langle \mathit{var} \rangle = \texttt{load}(\langle \mathit{mem} \rangle, \langle \mathit{atom} \rangle) \\
&\quad \mid \langle \mathit{mem} \rangle = \texttt{store}(\langle \mathit{var} \rangle, \langle \mathit{atom} \rangle) \\
&\quad \mid \texttt{while}\ \langle \mathit{pred} \rangle\ \langle \mathit{inst} \rangle \\
&\quad \mid \texttt{if}\ \langle \mathit{pred} \rangle\ \texttt{then}\ \langle \mathit{inst} \rangle\ \texttt{else}\ \langle \mathit{inst} \rangle \\
\langle \mathit{atom} \rangle &::= \texttt{sc} \mid \texttt{rel} \mid \texttt{acq} \mid \texttt{con} \mid \texttt{rx}
\end{aligned}
$$

Fig. 3: Programming language.

Load and store operations are annotated by memory order tags that define their ordering guarantees. The `sc` tag guarantees a sequentially consistent semantics for the access; `rel`/`acq` and `rel`/`con` implement the message-passing idiom; the `rx` (relaxed) tag maps directly to hardware accesses giving minimal guarantees on how those accesses are performed. Weaker guarantees yield higher performance but they usually allow additional program behavior that is hard to predict.

Although the input program is written in a C11-like language, the analysis is performed at the assembly level. The

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

C11	X86	POWER	ARMv7
Load `rx`	`MOV`	lwz	ldr
Load `con`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `acq`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `sc`	`MOV`	sync; lwz; lwsync	ldr; dmb ish
Store `rx`	`MOV`	stw	str
Store `rel`	`MOV`	lwsync; stw	dmb ish; str
Store `sc`	`MOV; mfence`	sync; stw	dmb ish; str; dmb ish

TABLE I. Compiler mappings for X86, POWER and ARMv7.

program is converted to hardware specific assembly code according to a given compiler mapping. The compiler mapping replaces load and store operations with their corresponding assembly memory accesses and adds fences to enforce the ordering guarantees provided by the memory model tag. Each compiler uses its own mapping. Our tools currently implement the mappings given in Table I, which are the ones used by the LLVM 4.0 compiler [38]. Other mappings, like the one from [1], can be easily added. For the method presented in Section IV to work, the only requirement is that the mapping of each atomic operation contains a single memory access.

It is worth noting that we assume the compiler does not perform any optimization; the program to be verified has already been optimized. Compiler optimizations under weak memory models are an active topic of research [34], [37], [47], [49], but they are out of the scope of this paper.

Memory Models: Informally, a memory model defines when store operations executed by one thread become visible to other threads. This means a memory model determines the semantics of a program on a hardware architecture. The semantics is defined in terms of so-called executions. It contains those executions that are (in a precise sense) consistent with the memory model [7], [36]. We elaborate on the notion of executions and how they define reachability. Afterwards we introduce memory models and consistency.

An execution $(\mathbb{X}, \mathit{rf}, \mathit{co})$ consists of memory events executed by the program of interest and relations between these events [7], [49]. Set $\mathbb{X}$ states which events have been executed in each thread. This forms the control flow of the program. The reads-from relation rf specifies from which store each load gets its value. The coherence order co is the order in which stores to a variable take effect.

A *state* consists of the values of local and global variables. A state *reached* by a given execution is defined as follows. The value of a global variable is given by the last store operation according to the co relation. The value of a local variable depends on the last executed event (according to the control flow) loading to the local variable.

$$
\begin{aligned}
\langle \mathit{MCM} \rangle &::= \langle \mathit{assert} \rangle \mid \langle \mathit{rel} \rangle \mid \langle \mathit{MCM} \rangle \wedge \langle \mathit{MCM} \rangle \\
\langle \mathit{assert} \rangle &::= \mathit{acyclic}(\langle r \rangle) \mid \mathit{irreflexive}(\langle r \rangle) \mid \mathit{empty}(\langle r \rangle) \\
\langle r \rangle &::= \langle b \rangle \mid \langle r \rangle \cup \langle r \rangle \mid \langle r \rangle \cap \langle r \rangle \mid \langle r \rangle \setminus \langle r \rangle \\
&\quad \mid \langle r \rangle^{-1} \mid \langle r \rangle^{+} \mid \langle r \rangle^{*} \mid \langle r \rangle ; \langle r \rangle \\
\langle b \rangle &::= \mathit{po} \mid \mathit{rf} \mid \mathit{co} \mid \mathit{ad} \mid \mathit{dd} \mid \mathit{cd} \mid \mathit{sthd} \mid \mathit{sloc} \\
&\quad \mid \mathit{mfence} \mid \mathit{sync} \mid \mathit{lwsync} \mid \mathit{isync} \mid \mathit{isb} \mid \mathit{ish} \\
&\quad \mid \mathit{id}(\langle \mathit{set} \rangle) \mid \langle \mathit{set} \rangle \times \langle \mathit{set} \rangle \mid \langle \mathit{name} \rangle \\
\langle \mathit{set} \rangle &::= \mathbb{E} \mid \mathbb{W} \mid \mathbb{R} \\
\langle \mathit{rel} \rangle &::= \langle \mathit{name} \rangle := \langle r \rangle
\end{aligned}
$$

Fig. 4: The CAT language [9].

Memory models define a consistency predicate on executions. The semantics of a program on that memory model is then given by the executions of the program that satisfy the predicate [7], [11], [36]. We use the language CAT [9] to define memory models, the core of which is shown in Fig. 4. There are functional programming features in CAT that we do not support since they are not needed to define the hardware architectures of interest. In CAT, memory models define relations over the events in executions. The program order po and relations rf and co from above are common to all memory models, and typically referred to as base relations. Base relations also include, e.g., address, data and control dependences. Further so-called derived relations are defined using operations on relations such as transitive closure, union, intersection, and composition.

Importantly, CAT allows to define derived relations as least solutions to a system of equations. The semantics of such recursive definitions is well defined only if they behave monotonously [9]. Almost all of CAT is already monotonous, the only non-monotonous construct is the right hand side of the "\"-operator. We disallow recursive definitions in the right side of it to ensure well defined semantics in a syntactic manner.

To define the notion of consistency for executions, a memory model requires a number of assertions to hold over its relations. These assertions are acyclicity, irreflexivity and emptiness guarantees. An execution is defined to be consistent with the memory model if it satisfies all assertions.

III. CHECKING REACHABILITY

DARTAGNAN encodes the reachability problem into an SMT formula which is constructed as follows. Formulas ϕ_{CF} and ϕ_{DF} encode the control flow and data flow of the program. The memory model dependent condition $\phi_{\mathcal{M}}$ ensures that the executions are *consistent* with the given model. Finally, ϕ_S is satisfied only if the final state reached by the program satisfies the predicate S. The overall BMC encoding is:

$$\phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}} \wedge \phi_S.$$

Each loop in the program is unrolled up to a user defined depth k. The program is compiled using a given mapping and then converted into its single static assignment (SSA) form. This results in a directed acyclic graph presenting all possible control flows of the program up to the unrolling depth. As the program is now acyclic and in the SSA form, each statement and variable assignment can be executed at most once.

The main idea of the BMC encoding is to guess an execution, which consists of *executed events* and the rf and *co* relations. Guessing the executed events fully specifies the control flow of the candidate execution, while guessing *rf* and *co* specifies the data-flow of the candidate execution. It is easy to see that this is basically the encoding of the weakest possible memory model expressible in CAT. All widely used models are additional restrictions of this.

The part of the encoding that is not dependent on the memory model is very similar to established BMC encodings of concurrent programs [25]. We recently introduced in [40] the encodings for the memory model specific parts, especially the ones for recursively defined relations with least fixpoint semantics (needed for POWER and ARM).

Encoding Control and Data Flow: Recall that the basic idea for the control flow is to guess the set of executed events. We encode this with a Boolean variable for each event, which is satisfied if the event is executed. We ensure that every load gets its value from one store on the same variable and that the stores to a variable form a total order in *co*. Relations are encoded as follows. For any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we use a Boolean variable $\mathtt{r}(e_1, e_2)$ representing the fact that $e_1 \xrightarrow{r} e_2$ holds.

The rest of the encoding ensures that the guessed executed events are a valid control flow path through each one of the threads, and that data-flow follows the reads-from and coherence order relations in the shared variables. The encoding also checks that all executed guards are satisfied, and that all executed data manipulation statements are correctly evaluated. The data flow encoding additionally relates the final state of the unrolled compiled program to the original program, allowing the state predicate formula ϕ_S to be expressed in terms of the variables of the original unrolled program before the SSA conversion. Thus, we ensure candidate executions that obey both the control flow and the data flow of the programs. The details of the encodings can be found in [39].

Encoding Memory Models: A memory model defined in the CAT language (see Fig. 4) is a constraint system over so-called derived relations together with some assertions. The language defines a number of base relations. Their encodings can be obtained directly from the source code of the program (e.g., the program order *po*), from statements corresponding to the synchronization primitives of the used architecture (e.g., memory fences *mfence* on TSO) or they are part of the execution (the *rf* and *co* relations). Derived relations are built from relations using operators such as union, intersection, difference, composition, transitive closure, etc. We similarly use new Boolean variables to represent the derived relations. Most of the operators can be encoded in SMT in a fairly straightforward manner.

An execution is consistent with a memory model if all its assertions are satisfied. We encode acyclicity of a relation in a compact way using IDL by ensuring that a relation implies a partial ordering. We assign each event a numerical variable and require that if an event e is related to e' then the numerical value assigned to e is less than the value assigned to e'.

Encoding Recursive Relations: CAT additionally supports recursive definitions. The semantics of such recursively defined relations are the least fixpoint solution to this system of monotone equations on relations. We argue that for reachability, it is sufficient to encode any fixpoint, not necessarily the least one. The assertions of the memory model (acyclicity, irreflexivity and emptiness) are monotone in the following sense: If a relation fulfills an assertion, all of its subsets will also fulfill the assertion. The CAT operators on relations are also monotone (except set difference which is not applied to recursive relations): Consider $r := (r;r) \cup r_0$, where the operator ";" represents relation composition. If relation r_0 is enlarged or reduced, then so is r.

These observations allow us to apply the Knaster-Tarski Theorem [44]. This is a key contribution of the paper; we use it to simplify the SMT encoding of CAT models. We can freely pick any fixpoint that satisfies all the assertions, as it always contains the least fixpoint, which also satisfies all the assertions. It removes the need to encode the least fixpoints of the CAT language exactly. We call this the relaxed encoding. The encoding of r is simply:

$$\mathtt{r}(e_1, e_2) \Leftrightarrow \mathtt{r;r}(e_1, e_2) \vee \mathtt{r}_0(e_1, e_2).$$

We argue that for reachability queries, this encoding is still correct. Assume a least fixpoint encoding of a reachability query has a satisfying assignment. Naturally, the least fixpoint also satisfies the relaxed encoding as it is a fixpoint. If the least fixpoint encoding is unsatisfiable, every execution violates some assertion. Any violated acyclicity assertion implies a cycle. Since larger fixpoints only add dependencies to relations, the cycle remains for all larger fixpoints. The assertion remains violated with the relaxed encoding. Hence, the relaxed encoding is also unsatisfiable. Similar reasoning also holds for irreflexivity and emptiness violations.

IV. CHECKING INCLUSION

We show how to efficiently check state inclusion. The inclusion requires that for all states reachable in the target memory model $\mathcal{M}_T$ there has to be an execution in the source memory model $\mathcal{M}_S$ reaching the same state. Such a $\forall\exists$-alternation of quantifiers is notoriously difficult to handle for verification tools [49]. A naive approach would iterate over all reachable states. We propose to use an oracle guiding the search by providing relevant candidate states. We present an implementation of the oracle that iterates over far fewer states but preserves completeness. The key observation is that new states always correspond to new executions. Therefore we only need to consider states coming from executions consistent with the target but inconsistent with the source memory model.

The main procedure is described by Algorithm 1. It takes as input a program, two memory models $\mathcal{M}_S, \mathcal{M}_T$[1], and a bound k. The program is first unrolled up to the bound k and converted to to the acyclic assembly programs P_S^k and P_T^k

---

[1] The latter is needed to implement a concrete oracle. However in Algorithm 1 we consider the oracle a black box object.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Algorithm 1 Incremental SMT Solving for State Inclusion

```
1: procedure PORTHOS(Program $P$, MCM $\mathcal{M}_S, \mathcal{M}_T$, Int $k$)
2:     $\phi_{\text{RCH}} \leftarrow \phi_{CF}(P_S^k) \wedge \phi_{DF}(P_S^k) \wedge \phi_{\mathcal{M}_S}(P_S^k)$
3:     while $\text{ORACLE}().hasState()$ do
4:         $s \leftarrow \text{ORACLE}().getState()$
5:         if $\phi_{\text{RCH}} \wedge \phi_s$ is UNSAT then
6:             return false
7:     return true
```

using the mappings from Table I. The procedure might perform several reachability queries for $\mathcal{M}_S$. Therefore, we construct a formula defining its consistent executions in Line 2. The formulas ϕ_{CF}, ϕ_{DF} and $\phi_{\mathcal{M}_S}$ are the ones from Section III.

The algorithm then enters a loop iterating over a sequence of states which can be thought of as candidates for violating inclusion. These candidate states are provided by an oracle, a black box providing two functions. Function *hasState()* returns a Boolean judging whether there is still a candidate state to consider. If so, function *getState()* provides the candidate. The oracle has to meet the following specification.

(O1) If *hasState()* returns false, then state inclusion holds.
(O2) If *hasState()* returns true, an invocation of *getState()* returns a state.
(O3) Function *getState()* never returns the same state twice.
(O4) Every state returned by *getState()* is reachable in $\mathcal{M}_T$.

When the oracle provides a new candidate, the algorithm checks whether it is reachable in $\mathcal{M}_S$. If the state is not reachable, state inclusion does not hold and the procedure returns false at Line 6. If it is reachable, the check is repeated with a different state. If every state provided by the oracle is reachable under $\mathcal{M}_S$, state inclusion holds by (O1) and the procedure returns true at Line 7.

A correct but naive implementation of an oracle would list all states reachable under $\mathcal{M}_T$. A more efficient exploration is guaranteed by the following idea.

An Oracle for Efficient Exploration: We present an oracle that lists good candidates likely to violate state inclusion. Moreover, the oracle may be able to guarantee state inclusion early. Finally, the computation of candidate states itself is based on SMT-solving and quite efficient. The idea is to find all executions consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$, and extract their reachable states. This guarantees (O1) and (O4): When *hasState()* returns false, all states that may violate inclusion have been considered and thus state inclusion holds. Our implementation encodes the oracle as follows:

$$\begin{aligned}\phi_{\text{ORA}} = \phi_{\text{EQ}}(P_S^k, P_T^k) \wedge \phi_{CF}(P_T^k) \wedge \phi_{DF}(P_T^k) \wedge \phi_{\mathcal{M}_T}(P_T^k)\\ \wedge\, \phi_{CF}(P_S^k) \wedge \phi_{DF}(P_S^k) \wedge \phi_{\neg\mathcal{M}_S}(P_S^k).\end{aligned}$$

Function *hasState()* denotes whether the formula ϕ_{ORA} is satisfiable. In this case, *getState()* extracts a state s from a satisfying assignment and returns it. This guarantees (O2). To ensure (O3), the same state is not returned twice, the formula is iteratively updated to $\phi_{\text{ORA}} := \phi_{\text{ORA}} \wedge \neg\phi_s$.

The formula ϕ_{EQ} relates the executions of both assembly programs by ensuring that they represent the same execution of P^k. This formula will be explained below. The next three formulas encode consistent executions in $\mathcal{M}_T$ as defined in Section III. The remaining formulas encode executions inconsistent with $\mathcal{M}_S$.

We encode acyclicity violations by guessing a cycle. For every event e, a Boolean variable $\mathtt{C}_{\mathtt{r}}(e)$ represents its presence in the cycle. We ensure that every event in the cycle has an incoming and an outgoing edge in the cycle. A more detailed description of the cycle encoding is given in [40].

Encoding Least Fixpoints: When using the relaxed encoding in the oracle, a larger fixpoint could be chosen with more dependencies between events and thus new cycles could be created. This implies that the oracle could propose additional candidate states and more iterations might be required. For this reason, we encode exact least fixpoints for PORTHOS.

Least fixpoints of recursively defined relations can be computed with the standard Kleene iteration [43], which starts from the empty relation and iterates until the least fixpoint is reached. A naive encoding approach would implement the Kleene iteration in SAT by introducing a Boolean variable for each pair of events and each iteration step. This naive encoding is too inefficient, as the number of iterations needed is basically the joint size of the involved relations.

We recently proposed in [40] a much more efficient SMT-encoding that uses Integer Difference Logic [26]. Instead of having a Boolean variable for each iteration step, it only uses one Boolean variable $\mathtt{r}(e_1, e_2)$ (representing if the relation holds) and one numerical variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the iteration in which the pair was added to the relation. Given a relation $r := (r;r) \cup r_0$, for events e_1, e_2 we construct the formula:

$$\begin{aligned}\mathtt{r}(e_1, e_2) \quad\Leftrightarrow\quad & (\mathtt{r;r}(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r;r}}_{e_1,e_2}))\\ & \vee (\mathtt{r}_0(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_0}_{e_1,e_2})).\end{aligned}$$

The first part of the disjunction specifies that (e_1, e_2) can be added to r if the pair belongs to $r;r$ (i.e. variable $\mathtt{r;r}(e_1, e_2)$ is true) and it was added to $r;r$ at some previous iteration step (i.e. $\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r;r}}_{e_1,e_2}$). The second part is analogous.

Note that this only encodes *at most* the least fixpoint: A satisfying assignment could also set a value for $\Phi^{\mathtt{r}}_{e_1,e_2}$ that is too small and thus not add the pair. We combine the formula above with the relaxed encoding to get exactly the least fixpoint.

Encoding Common Executions: We look for an execution consistent with $\mathcal{M}_T$ and inconsistent with $\mathcal{M}_S$. However, we execute two different assembly programs P_S^k and P_T^k. This means we need a way to compare their executions. Intuitively, two executions are equivalent if they represent the same execution of the program P^k. Since the compilation scheme of Table I implements each atomic memory operation using a single low-level memory access, a one-to-one mapping $\pi : \mathbb{E}_T \rightarrow \mathbb{E}_S$ between the events of P_S^k and P_T^k can be

Benchmark	#Executions	TSO				C11	#Executions	Power			ARM		
	TSO/C11	Herd	Nidhugg	CBMC	Dartagnan	RCMC	Power/ARM	Herd	Nidhugg	Dartagnan	Herd	Nidhugg	Dartagnan
Parker	11	0.08	**0.01**	0.29	0.76	**0.08**	14	0.07	**0.01**	1.32	0.08	**0.02**	1.29
Dekker	24	T/O	**0.02**	0.48	4.29	**0.05**	24	T/O	**0.05**	34.86	T/O	**0.04**	36.88
Peterson	24	4.98	**0.03**	0.32	0.94	**0.07**	24	4.89	**0.04**	4.29	4.85	**0.03**	4.13
Burns	47	284.90	**0.02**	0.29	1.10	**0.04**	47	316.33	**0.03**	4.10	289.66	**0.04**	4.05
Bakery	12492	T/O	2.60	**0.41**	4.64	**0.07**	84760	T/O	141.56	**40.06**	T/O	140.25	**41.83**
Lamport	-	T/O	T/O	**0.38**	4.56	T/O	-	T/O	T/O	**72.03**	T/O	T/O	**70.64**
Szymanski	4227148	T/O	966.71	**0.84**	18.98	**409.79**	-	T/O	T/O	**259.56**	T/O	T/O	**241.34**

TABLE II. Reachability of mutual exclusion algorithm under TSO, C11, Power, and ARM.

defined. Given two events e_S and e_T representing instructions accessing memory in the assembly programs, $\pi(e_T) = e_S$ holds if they both represent the same high-level instruction. Note that such a mapping π can always be defined as long as the compiler implements atomic memory operations with a single memory access. The following encoding relates the executions of both assembly programs:

$$\begin{aligned}\phi_{\text{EQ}} = \quad & \bigwedge_{e \in \mathbb{E}_T} \quad e \in \mathbb{X}_T \Leftrightarrow \pi(e) \in \mathbb{X}_S \\ \wedge & \bigwedge_{e_1, e_2 \in \mathbb{E}_T} \mathtt{rf}(e_1, e_2) \Leftrightarrow \mathtt{rf}(\pi(e_1), \pi(e_2)) \\ \wedge & \bigwedge_{e_1, e_2 \in \mathbb{E}_T} \mathtt{co}(e_1, e_2) \Leftrightarrow \mathtt{co}(\pi(e_1), \pi(e_2)).\end{aligned}$$

V. Experimental Evaluation

We implemented the algorithms from Sections III and IV in the Dartagnan and Porthos tools which use Z3 [29] as the backend SMT solver. Both tools are available from:

https://github.com/hernanponcedeleon/Dat3M.

The tools include the following memory models: SC, TSO, PSO, RMO, Alpha, Power, and ARM (v7). Others can be defined in the CAT language.

We compare their performance against several memory model-aware tools. Herd [12] is a tool designed for litmus tests (small programs). It takes CAT files as an input (and thus supports all memory models used in this section). It enumerates all candidate executions and then filters those accepted by the memory model. Nidhugg [2], [6] performs stateless model checking. It supports TSO, Power and a simplified version of ARM. CBMC [11] is a Bounded Model Checker with an encoding similar to ours, but it cannot handle recursive definitions efficiently and only supports TSO. For the sake of completeness, we also report results on reachability for C11 using the RCMC tool [32]. This is the memory model of a programming language instead of a hardware architecture and introduces new types of events. Therefore we cannot directly apply our approach to C11. However, the number of executions on C11 coincides with TSO for all programs and we expect our encoding to perform similar to the TSO case.

The tools listed above are designed to test reachability. They allow to reason about one memory model at a time and therefore cannot directly be used to test state inclusion. However, Herd returns information about all final states. We check state inclusion with Herd by computing the reachable states separately for both models (i.e. we run the tool twice) and comparing them afterwards.

Our benchmark suite consists of mutual exclusion algorithms. We unrolled loops twice ($k = 2$) which is sufficient to show that our approach scales better than the other tools for programs with several executions. Programs contains either two or three threads. However their size is reported in terms of the number of consistent executions since the performance of the tools strongly depends on this. The execution times are given in seconds. We set a timeout of 1800 secs for each call to the tools (3600 secs for Herd in the case of inclusion since the tool is run twice). For entries marked as T/O, the timeout was reached.

We performed two sets of experiments: *(i)* Reachability under TSO, C11, Power and ARM; and *(ii)* the inclusions TSO $\subseteq$ SC, Power $\subseteq$ TSO, and ARM $\subseteq$ TSO. Inclusion in the other direction (necessary for equivalence) holds by the definition of the memory models. E.g., every state reachable under TSO is also reachable under the weaker models Power and ARM.

The results on reachability are given in Table II. We present the analysis for unreachable states since it forces all tools to perform a complete exploration and provides the worst case scenario. For TSO, the best results are obtained by Nidhugg in benchmarks with small number of executions and by CBMC as soon as this number grows. Even though CBMC outperforms Dartagnan for TSO, our tool can be at least two orders of magnitude faster than stateless model checking techniques when the number of executions is in the order of millions. See, e.g., Lamport which Dartagnan solves in less than 5 secs while Nidhugg and RCMC timeout. For both Power and ARM, Nidhugg again outperforms all tools when the number of executions is small. However for benchmarks with a big number of executions (above 80K), Dartagnan performs better. For the Lamport and Szymanski benchmarks, our tool outperforms Nidhugg by at least one order of magnitude. Table II suggests that approaches based on SAT/SMT encodings have a lot of potential for large programs. Dartagnan can currently handle four million executions in less than 20 secs while Nidhugg and RCMC need 15 and 6 minutes respectively.

The results on state inclusion are given in Table III. The SAT column reports whether a counterexample to inclusion was found (✔) or not (✘). When Herd returns a result, we report on the number of delta executions (Δ). This corresponds to an upper bound on the maximal number of iterations

Benchmark	TSO ⊆ SC					
	SAT	Herd	Porthos	Δ	It	S.U.
Parker	✔	**0.15**	0.70	3	1	0.21
Dekker	✔	T/O	**12.31**	-	1	**>292.44**
Peterson	✔	9.96	**1.31**	12	1	**7.60**
Burns	✔	610.65	**2.00**	53	1	**305.32**
Bakery	✔	T/O	**10.78**	-	2	**>333.95**
Lamport	✔	T/O	**10.64**	-	3	**>338.34**
Szymanski	✔	T/O	**101.32**	-	1	**>35.53**

Benchmark	Power ⊆ TSO					
	SAT	Herd	Porthos	Δ	It	S.U.
Parker	✔	**0.15**	2.46	3	2	0.06
Dekker	✘	T/O	**108.89**	-	0	**>33.06**
Peterson	✘	9.94	**6.33**	0	0	**1.57**
Burns	✘	578.55	**6.12**	18	1	**94.53**
Bakery	✘	T/O	**836.44**	-	43	**>4.30**
Lamport	-	T/O	T/O	-	-	-
Szymanski	✘	T/O	**940.75**	-	0	**>3.82**

Benchmark	ARM ⊆ TSO					
	SAT	Herd	Porthos	Δ	It	S.U.
Parker	✔	**0.15**	1.90	3	1	0.07
Dekker	✘	T/O	**134.43**	-	0	**>26.77**
Peterson	✘	10.28	**6.51**	0	0	**1.57**
Burns	✘	546.90	**7.89**	18	1	**69.31**
Bakery	-	T/O	T/O	-	-	-
Lamport	-	T/O	T/O	-	-	-
Szymanski	✘	T/O	**850.44**	-	0	**>4.23**

TABLE III. State inclusion of mutual exclusion algorithms.

Porthos might perform. As it can be seen from Table II, in general this number is several orders of magnitude smaller than the total number of executions. The cases reporting zero iterations correspond to the set of executions coinciding for both memory models. For most of the cases, Porthos is at least one order of magnitude faster than Herd. For TSO, the speed-up (S.U. column) can be up-to two orders of magnitude.

VI. Related Work

The influence of memory models on the semantics of concurrent programs has been studied at least since 2007. Initially, hardware architectures have been addressed [7], [15], [22], [31], [36], [41], [42], followed by programming languages, in particular C11 and C++11 [18], [19], [34]. Recently, an axiomatic memory model for the Linux kernel has been introduced [14]. These semantic studies form the basis for the development of verification tools.

As of today, none of the following tools (except Herd) consider the description of the memory model as an input. They all implement (at best few) concrete models. Nitpick [20], SATCheck [30], NemosFinder [50], and MemSAT [45] use SMT solvers. CBMC had been extended to support TSO and Power [11] but Power is no longer supported. CPPMem [19] and Herd enumerate all executions, making them less scalable. More efficient but technically involved and hard to generalize are Stateless Model Checkers, available for TSO, PSO, Power, ARM [2], [6] and C11 [32]. TRENCHER [21] looks for trace inclusion bugs between SC and TSO; it underapproximates state inclusion. It can also synthesize fences to enforce SC behaviors. Memorax shares this functionality and is complete for reachability under TSO [3], [4], [5]. Trace inclusion can be enforced not only for TSO but also for weaker memory models. The OFFENCE tool [13] does this, although it is limited to restoring SC behaviors of litmus tests. Another fence insertion tool is MUSKETEER [10]. It scales to large programs, but is also restricted to ensuring SC. The Fender and DFence tools [33], [35] use fence insertion to guarantee safety properties. They support TSO, PSO, and RMO.

A modular proof technique has been introduced recently [8]. It uses invariants to verify programs under a model given in CAT. Another tool based on CAT synthesizes programs differentiating two memory models [49]. However, this tool is of interest to memory model designers and not made for verification.

Porthos was originally designed to check trace inclusion. In [40], we showed that state inclusion has a higher complexity than trace inclusion. As a consequence, there is no polynomial encoding that reduces inclusion to a single SAT query. However, the experiments in Section V show that our oracle-based heuristic still performs well in programs where an exhaustive state exploration does not scale.

VII. Conclusion and Outlook

We have presented Dartagnan and Porthos, two modular Bounded Model Checkers for concurrent programs. The tools can check reachability and state equivalence under any (pair of) memory model(s) defined in the CAT language. Our method reduces reachability to satisfiability of a SMT formula using novel encoding techniques. Equivalence is tested using a guided search. We propose to use an oracle to find relevant candidate states, and show how to implement an efficient oracle based on SMT queries. We have performed experiments to compare our tools to several memory model-aware tools, and find them at least one order of magnitude faster for large programs.

We are currently developing methods to synthesize memory models from reachability results using our encoding techniques. The techniques include compact representations of relations by predicates as well as approximations of operations that are not precise but still sound.

Other verification tasks, such as synthesizing programs to compare memory models, could in principle also be solved by reducing them to SMT queries. We would like to explore this in the future.

Modern compilers perform various optimizations when mapping high-level code to assembly instructions. We plan to investigate whether such compiler mappings can be extracted from the compilation process, at least approximately.

Acknowledgements: We thank Natalia Gavrilenko for constructive feedback on the manuscript and the tool implementation.

References

[1] C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed: 23.04.2018.

[2] Parosh A. Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[3] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, volume 7460 of *LNCS*, pages 164–180. Springer, 2012.

[4] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *TACAS*, volume 7214 of *LNCS*, pages 204–219. Springer, 2012.

[5] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS*, volume 7795 of *LNCS*, pages 530–536. Springer, 2013.

[6] Parosh A. Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, volume 9780 of *LNCS*, pages 134–156. Springer, 2016.

[7] Jade Alglave. *A Shared Memory Poetics*. Thèse de doctorat, L'université Paris Denis Diderot, 2010.

[8] Jade Alglave. Simulation and invariance for weak consistency. In *SAS*, pages 3–22, 2016.

[9] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.

[10] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, volume 8559 of *LNCS*, pages 508–524. Springer, 2014.

[11] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[12] Jade Alglave and Luc Maranget. The diy7 tool suite. http://diy.inria.fr/.

[13] Jade Alglave and Luc Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.

[14] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *ASPLOS*, pages 405–418. ACM, 2018.

[15] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[16] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[17] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[18] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

[19] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

[20] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *PPDP*, pages 113–124, 2011.

[21] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[22] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.

[23] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

[24] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[25] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[26] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS*, volume 3253 of *LNCS*, pages 263–276. Springer, 2004.

[27] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, volume 7935 of *LNCS*, pages 84–104. Springer, 2013.

[28] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, volume 8931 of *LNCS*, pages 449–466. Springer, 2015.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[30] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for sc and tso. In *OOPSLA*, pages 20–36, 2015.

[31] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL*, pages 608–621. ACM, 2016.

[32] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018.

[33] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.

[35] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.

[36] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *LNCS*, pages 495–512. Springer, 2012.

[37] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and armv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016.

[38] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *CC*, pages 1–10. ACM, 2017.

[39] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for axiomatic memory models. PORTHOS: One tool for all models. *CoRR*, abs/1702.06704, 2017. Extended version of [40].

[40] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[41] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

[42] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391. ACM, 2009.

[43] Viggo Stoltenberg-Hansen, Edward R. Griffor, and Ingrid Lindstrom. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[44] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[45] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*, pages 341–350. ACM, 2010.

[46] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

[47] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220. ACM, 2015.

[48] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.

[49] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204. ACM, 2017.

[50] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE Computer Society, 2004.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Complete Test Sets And Their Approximations

Eugene Goldberg
eu.goldberg@gmail.com

***Abstract*—We use testing to check if a combinational circuit N always evaluates to 0 (written as $N \equiv 0$). We call a set of tests proving $N \equiv 0$ a complete test set (CTS). The conventional point of view is that to prove $N \equiv 0$ one has to generate a *trivial* CTS. It consists of all $2^{|X|}$ input assignments where X is the set of input variables of N. We use the notion of a Stable Set of Assignments (SSA) to show that one can build a *non-trivial* CTS consisting of less than $2^{|X|}$ tests. Given an unsatisfiable CNF formula $H(W)$, an SSA of H is a set of assignments to W that proves unsatisfiability of H. A trivial SSA is the set of all $2^{|W|}$ assignments to W. Importantly, real-life formulas can have non-trivial SSAs that are much smaller than $2^{|W|}$. In general, construction of even non-trivial CTSs is inefficient. We describe a much more efficient approach where tests are extracted from an SSA built for a projection of N on a *subset* of its variables. These tests can be viewed as an approximation of a CTS for N. We describe potential applications of our approach. We show experimentally that it can be used to facilitate hitting corner cases and expose bugs in sequential circuits overlooked due to checking “misdefined” properties.**

I. INTRODUCTION

Testing is an important part of verification flows. For that reason, any progress in understanding testing and improving its quality is of great importance. In this paper, we consider the following problem. Given a single-output combinational circuit N, find a set of input assignments (tests) proving that N evaluates to 0 for every test (written as $N \equiv 0$) or find a counterexample. We will call a set of input assignments proving $N \equiv 0$ a *complete test set* (*CTS*)[1]. We will call the set of all possible tests a *trivial CTS*. Typically, one assumes that proving $N \equiv 0$ involves derivation of the trivial CTS, which is infeasible in practice. Thus, testing is used only for finding an input assignment refuting $N \equiv 0$. We present an approach for building a non-trivial CTS consisting only of a subset of all possible tests[2]. In general, finding even a non-trivial CTS for a large circuit is impractical. We describe a much more efficient approach where an *approximation* of a CTS is generated.

The circuit N above usually describes a property ξ of a multi-output combinational circuit M, the latter being the *real object of testing*. For instance, ξ may state that M never produces some output assignments. To differentiate CTSs and their approximations from conventional test sets verifying M “as a whole”, we will refer to the former as *property-checking test sets*. Let $\Xi := \{\xi_1, \dots, \xi_k\}$ be the set of properties of M formulated by a designer. Assume that every property of Ξ holds and T_i is a test set generated to check property $\xi_i \in \Xi$. There are at least two reasons why applying T_i to M makes sense. First, if Ξ is *incomplete*[3], a test of T_i can expose a bug breaking a property of M that is not in Ξ. Second, if property ξ_i is defined *incorrectly*, a test of T_i may expose a bug breaking the correct version of ξ_i. On the other hand, if M produces proper output assignments for all tests of $T_1 \cup \dots \cup T_k$, one gets extra guarantee that M is correct. In Section VI, we list some other applications of property-checking test sets such as increasing the probability of hitting corner cases and testing properties of sequential circuits.

Let $N(X, Y, z)$ be a single-output combinational circuit where X and Y specify the sets of input and internal variables of N respectively and z specifies the output variable of N. Let $F_N(X, Y, z)$ be a formula defining the functionality of N (see Section III). We will denote the set of variables of circuit N (respectively formula H) as $\mathit{Vars}(N)$ (respectively $\mathit{Vars}(H)$). Every assignment[4] to $\mathit{Vars}(F_N)$ satisfying F_N corresponds to a consistent assignment[5] to $\mathit{Vars}(N)$ and vice versa. Then the problem of proving $N \equiv 0$ reduces to showing that formula $F_N \wedge z$ is unsatisfiable. From now on, we assume that all formulas mentioned in this paper are *propositional*. Besides, we will assume that every formula is represented in CNF i.e. as a conjunction of disjunctions of literals.

Our approach is based on the notion of a Stable Set of Assignments (SSA) introduced in [9]. Given formula $H(W)$, an SSA of H is a set P of assignments to variables of W that have two properties. First, every assignment of P falsifies H. Second, P is a transitive closure of some neighborhood relation between assignments (see Section II). The fact that H has an SSA means that the former is unsatisfiable. Otherwise, an assignment satisfying H is generated when building its SSA. If H is unsatisfiable, the set of all $2^{|W|}$ assignments is always an SSA of H . We will refer to it as *trivial*. Importantly, a real-life formula H can have a lot of SSAs whose size is much less than $2^{|W|}$. We will refer to them as *non-trivial*. As we show in Section II, the fact that P is an SSA of H is a *structural* property of the latter. That is this property cannot be expressed in terms of the truth table of H (as opposed to a *semantic* property of H). For that reason, if P is an SSA

---

[1] Term CTS is sometimes used to say that a test set invokes every event specified by a *coverage metric*. Our application of this term is quite different.

[2] In the case of black-box testing, i.e. when only *the number of input variables* of N is known, to prove $N \equiv 0$ one indeed has to enumerate all possible input assignments. In this paper, we consider white-box testing.

[3] That is M can be incorrect even if all properties of Ξ hold.

[4] By an assignment to a set of variables V, we mean a *full* assignment where every variable of V is assigned a value.

[5] An assignment to a gate G of N is called consistent if the value assigned to the output variable of G is implied by values assigned to its input variables. An assignment to variables of N is called consistent if it is consistent for every gate of N.

ISBN: 978-0-9835678-8-2.

for H, it may not be an SSA for another formula H' logically equivalent to H. So, a structural property is *formula-specific*.

We show that a CTS for N can be easily extracted from an SSA of formula $F_N \wedge z$. This makes a non-trivial CTS a structural property of circuit N that cannot be expressed in terms of its truth table. Building an SSA for a large formula is inefficient. So, we present a procedure constructing a simpler formula $H(V)$ implied by $F_N \wedge z$ (where $V \subset Vars(F_N \wedge z)$) and building an SSA of H. The existence of such an SSA means that H (and hence $F_N \wedge z$) is unsatisfiable. So, $N \equiv 0$ holds. Formula H is obtained from $F_N \wedge z$ by a resolution-based procedure where *no resolutions* on variables of V are allowed. So H *preserves* some structure of $F_N \wedge z$. A test set extracted from an SSA of H can be viewed as a way to verify a "projection" of N on variables of V. On the other hand, one can consider this set as an approximation of a CTS for N. We will refer to the procedure above as $SeSt$ ("Se-mantics and St-ructure"). $SeSt$ combines semantic and structural derivations, hence the name. The semantic part of $SeSt$ is[6] to derive H. Its structural part consists of constructing an SSA of H thus proving H unsatisfiable.

The contribution of this paper is as follows. First, we introduce the notion of non-trivial CTSs (Section III). Second, we present a method for efficient construction of property-checking tests that are approximations of CTSs (Sections IV and V). Third, we describe applications of such tests (Section VI). Fourth, we experimentally show the efficiency and effectiveness of property-checking tests (Section VII).

II. STABLE SET OF ASSIGNMENTS

A. *Definitions*

We will refer to a disjunction of literals as a *clause*. Let $\vec{p}$ be an assignment to a set of variables V. Let $\vec{p}$ falsify a clause C. Denote by $\boldsymbol{Nbhd(\vec{p}, C)}$ the set of assignments to V satisfying C that are at Hamming distance 1 from $\vec{p}$. (Here *Nbhd* stands for "Neighborhood"). Thus, the number of assignments in $Nbhd(\vec{p}, C)$ is equal to that of literals in C. Let $\vec{q}$ be another assignment to V (that may be equal to $\vec{p}$). Denote by $\boldsymbol{Nbhd(\vec{q}, \vec{p}, C)}$ the subset of $Nbhd(\vec{p}, C)$ consisting only of assignments that are farther from $\vec{q}$ than $\vec{p}$ is (in terms of the Hamming distance).

Example 1: Let $V = \{v_1, v_2, v_3, v_4\}$ and $\vec{p}$=0110. We assume that the values are listed in $\vec{p}$ in the order the corresponding variables are numbered i.e. $v_1 = 0$, $v_2 = 1, v_3 = 1, v_4 = 0$. Let $C = v_1 \vee \overline{v_3}$. (Note that $\vec{p}$ falsifies C.) Then $Nbhd(\vec{p}, C)$=$\{\vec{p}_1, \vec{p}_2\}$ where $\vec{p}_1$ = 1110 and $\vec{p}_2$=0100. Let $\vec{q}$ = 0000. Note that $\vec{p}_2$ is closer to $\vec{q}$ than $\vec{p}$ is. So $Nbhd(\vec{q}, \vec{p}, C)$=$\{\vec{p}_1\}$.

Definition 1: Let H be a formula[7] specified by a set of clauses $\{C_1, \dots, C_k\}$. Let $P = \{\vec{p}_1, \dots, \vec{p}_m\}$ be a set of assignments to $Vars(H)$ such that every $\vec{p}_i \in P$ falsifies H. Let Φ denote a mapping $P \rightarrow H$ where $\Phi(\vec{p}_i)$ is a clause C of H falsified by $\vec{p}_i$. We will call Φ an **AC-mapping** where "AC" stands for "Assignment-to-Clause".

Definition 2: Let H be a formula specified by a set of clauses $\{C_1, \dots, C_k\}$. Let $P = \{\vec{p}_1, \dots, \vec{p}_m\}$ be a set of assignments to $Vars(H)$. P is called a **Stable Set of Assignments**[8] (**SSA**) of H with **center** $\vec{p}_{init} \in P$ if there is an AC-mapping Φ such that for every $\vec{p}_i \in P$, $Nbhd(\vec{p}_{init}, \vec{p}_i, C) \subseteq P$ holds where $C = \Phi(\vec{p}_i)$.

Example 2: Let H consist of four clauses: $C_1 = v_1 \vee v_2 \vee v_3$, $C_2 = \overline{v}_1$, $C_3 = \overline{v}_2$, $C_4 = \overline{v}_3$. Let $P = \{\vec{p}_1, \vec{p}_2, \vec{p}_3, \vec{p}_4\}$ where $\vec{p}_1 = 000$, $\vec{p}_2 = 100$, $\vec{p}_3 = 010$, $\vec{p}_4 = 001$. Let Φ be an AC-mapping specified as $\Phi(\vec{p}_i) = C_i, i = 1, \dots, 4$. Since $\vec{p}_i$ falsifies C_i, $i = 1, \dots, 4$, Φ is a correct AC-mapping. P is an SSA of H with respect to Φ and center $\vec{p}_{init}$=$\vec{p}_1$. Indeed, $Nbhd(\vec{p}_{init}, \vec{p}_1, C_1)$=$\{\vec{p}_2, \vec{p}_3, \vec{p}_4\}$ where $C_1 = \Phi(\vec{p}_1)$ and $Nbhd(\vec{p}_{init}, \vec{p}_i, C_i)$ = $\emptyset$, where $C_i = \Phi(\vec{p}_i)$, $i = 2, 3, 4$. Thus, $Nbhd(\vec{p}_{init}, \vec{p}_i, \Phi(\vec{p}_i)) \subseteq P$, $i = 1, \dots, 4$.

B. *SSAs and satisfiability of a formula*

Proposition 1: Formula H is unsatisfiable iff it has an SSA.

The proof is given in [11]. A similar proposition is proved in [9] for "uncentered" SSAs (see Footnote 8).

```
$BuildPath(H, \Phi, \vec{p}_{init}, \vec{s})${
  $Path := nil$
  $\vec{p}_1 := \vec{p}_{init}$
  $i := 1$
  while ($\vec{p}_i \neq \vec{s}$) {
    $Path := Extend(Path, \vec{p}_i)$
    $C := \Phi(\vec{p}_i)$
    $v := FindVar(C, \vec{p}_i, \vec{s})$
    $\vec{p}_{i+1} := FlipVar(\vec{p}_i, v)$
    $i := i + 1$ }
  return($Path$) }
```

Fig. 1. *BuildPath* procedure

The set of all assignments to $Vars(H)$ forms the *trivial* uncentered SSA of H. Example 2 shows a *non-trivial* SSA. The fact that formula H has a non-trivial SSA P is its *structural* property. That is one cannot check whether P is an SSA of H if only the truth table of H is known. In particular, P may not be an SSA of a formula H' logically equivalent to H.

The relation between SSAs and satisfiability can be explained as follows. Suppose that formula H is satisfiable. Let $\vec{p}_{init}$ be an arbitrary assignment to $Vars(H)$ and $\vec{s}$ be a satisfying assignment that is the closest to $\vec{p}_{init}$ in terms of the Hamming distance. Let P be the set of all assignments to $Vars(H)$ that falsify H and Φ be an AC-mapping from P to H. Then $\vec{s}$ can be reached from $\vec{p}_{init}$ by procedure *BuildPath* shown in Figure 1. It generates a sequence of assignments $\vec{p}_1, \dots, \vec{p}_i$ where $\vec{p}_1$ = $\vec{p}_{init}$ and $\vec{p}_i$=$\vec{s}$. First, *BuildPath* checks if current assignment $\vec{p}_i$ equals $\vec{s}$. If so, then $\vec{s}$ has been reached. Otherwise, *BuildPath* uses clause $C = \Phi(\vec{p}_i)$ to generate next assignment. Since $\vec{s}$ satisfies C, there is a variable $v \in Vars(C)$ that is assigned differently in $\vec{p}_i$ and $\vec{s}$. *BuildPath* generates a new assignment $\vec{p}_{i+1}$ obtained from $\vec{p}_i$ by flipping the value of v.

---

[6] Implication $F_N \wedge z \rightarrow H$ is a *semantic* property of $F_N \wedge z$. To verify this property it suffices to know the truth table of $F_N \wedge z$.

[7] We use the set of clauses $\{C_1, \dots, C_k\}$ as an alternative representation of a CNF formula $C_1 \wedge \dots \wedge C_k$.

[8] In [9], the notion of "uncentered" SSAs was introduced. The definition of an uncentered SSA is similar to Definition 2. The only difference is that one requires that for every $p_i \in P$, $Nbhd(\vec{p}_i, C) \subseteq P$ holds instead of $Nbhd(\vec{p}_{init}, \vec{p}_i, C) \subseteq P$. The advantage of centered SSAs is that they are usually much smaller than uncentered SSAs.

BuildPath reaches $\vec{s}$ in k steps where k is the Hamming distance between $\vec{p}_{init}$ and $\vec{s}$. Importantly, *BuildPath* reaches $\vec{s}$ for *any* AC-mapping. Let P be an SSA of H with respect to center $\vec{p}_{init}$ and AC-mapping Φ. Then if *BuildPath* starts with $\vec{p}_{init}$ and uses Φ as an AC-mapping, it can reach only assignments of P. Since every assignment of P falsifies H, no satisfying assignment can be reached.

```
$BuildSSA(H)${
  $E = \emptyset$; $\Phi := \emptyset$
  $\vec{p}_{init} := PickInitAssgn(H)$
  $Q := \{\vec{p}_{init}\}$
  while ($Q \neq \emptyset$) {
    $\vec{p} := PickAssgn(Q)$
    $Q := Q \setminus \{\vec{p}\}$
    if ($SatAssgn(\vec{p}, H)$)
      return($\vec{p}, nil, nil, nil$)
    $C := PickFlsCls(H, \vec{p})$
    $R := Nbhd(\vec{p}_{init}, \vec{p}, C) \setminus E$
    $Q := Q \cup R$
    $E := E \cup \{\vec{p}\}$
    $\Phi := \Phi \cup \{(\vec{p}, C)\}$}
return($nil, E, \vec{p}_{init}, \Phi$) }
```

Fig. 2. *BuildSSA* procedure

A procedure for generation of SSAs called *BuildSSA* is shown in Figure 2. It accepts formula H and outputs either a satisfying assignment or an SSA of H, center $\vec{p}_{init}$ and AC-mapping Φ. *BuildSSA* maintains two sets of assignments denoted as E and Q. Set E contains the examined assignments i.e. those whose neighborhood is already explored. Set Q specifies assignments that are queued to be examined. Q is initialized with an assignment $\vec{p}_{init}$ and E is originally empty. *BuildSSA* updates E and Q in a *while* loop. First, *BuildSSA* picks an assignment $\vec{p}$ of Q and checks if it satisfies H. If so, $\vec{p}$ is returned as a satisfying assignment. Otherwise, *BuildSSA* removes $\vec{p}$ from Q and picks a clause C of H falsified by $\vec{p}$. The assignments of $Nbhd(\vec{p}_{init}, \vec{p}, C)$ that are not in E are added to Q. After that, $\vec{p}$ is added to E as an examined assignment, pair $(\vec{p}, C)$ is added to Φ and a new iteration begins. If Q is empty, E is an SSA with center $\vec{p}_{init}$ and AC-mapping Φ.

III. COMPLETE TEST SETS

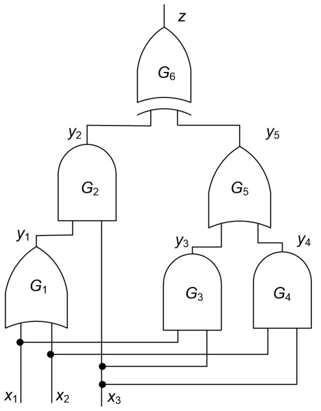


Fig. 3. Example of circuit $N(X,Y,z)$

Let $N(X,Y,z)$ be a single-output combinational circuit where X and Y specify the input and internal variables of N respectively and z specifies the output variable of N. Let N consist of gates $G_1,\dots,G_k$. Then N can be represented as $F_N = F_{G_1} \wedge \dots \wedge F_{G_k}$ where $F_{G_i}, i=1,\dots,k$ is a CNF formula specifying the consistent assignments of gate G_i. Proving $N \equiv 0$ reduces to showing that formula $F_N \wedge z$ is unsatisfiable.

Example 3: Circuit N shown in Figure 3 represents equivalence checking of expressions $(x_1 \vee x_2) \wedge x_3$ and $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ specified by gates G_1, G_2 and G_3, G_4, G_5 respectively. Formula F_N is equal to $F_{G_1} \wedge \dots \wedge F_{G_6}$ where, for instance, $F_{G_1} = C_1 \wedge C_2 \wedge C_3$, $C_1 = x_1 \vee x_2 \vee \overline{y}_1$, $C_2 = \overline{x}_1 \vee y_1$, $C_3 = \overline{x}_2 \vee y_1$. Every assignment satisfying F_{G_1} corresponds to a consistent assignment to gate G_1 and vice versa. For instance, $(x_1 = 0, x_2 = 0, y_1 = 0)$ satisfies F_{G_1} and is a consistent assignment to G_1 since the latter is an OR gate. Formula $F_N \wedge z$ is unsatisfiable since $(x_1 \vee x_2) \wedge x_3 \equiv (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. Thus, $N \equiv 0$.

Let $\vec{x}$ be a test i.e. an assignment to X. The set of assignments to $Vars(N)$ sharing the same assignment $\vec{x}$ to X forms a cube of $2^{|Y|+1}$ assignments. (Recall that $Vars(N) = X \cup Y \cup \{z\}$). Denote this set as $Cube(\vec{x})$. Only one assignment of $Cube(\vec{x})$ specifies the correct execution trace produced by N under $\vec{x}$. All other assignments can be viewed as "erroneous" traces under test $\vec{x}$.

Definition 3: Let T be a set of tests $\{\vec{x}_1,\dots,\vec{x}_k\}$ where $k \leq 2^{|X|}$. We will say that T is a **Complete Test Set (CTS)** for N if $Cube(\vec{x}_1) \cup \dots \cup Cube(\vec{x}_k)$ contains an SSA for formula $F_N \wedge z$.

```
$SeSt(G,V)${
  $H := \emptyset$
  foreach ($C \in G$)
    if ($Vars(C) \subseteq V$)
      $H := H \cup \{C\}$
  while (true) {
    $(\vec{v},P) := BuildSSA(H)$
    if ($P \neq nil$)
      return($nil, H, P$)
    $(C,\vec{s}) := GenCls(G,V,\vec{v})$
    if ($\vec{s} \neq nil$)
      return($\vec{s}, nil, nil$)
    $H := H \cup \{C\}$ }
```

Fig. 4. *SeSt* procedure

If T satisfies Definition 3, set $Cube(\vec{x}_1) \cup \dots \cup Cube(\vec{x}_k)$ "contains" a proof that $N \equiv 0$ and so T can be viewed as complete. If $k = 2^{|X|}$, T is the *trivial* CTS. In this case, $Cube(\vec{x}_1) \cup \dots \cup Cube(\vec{x}_k)$ contains the trivial SSA consisting of all assignments to $Vars(F_N \wedge z)$. Given an SSA P of $F_N \wedge z$, one can easily generate a CTS by extracting all different assignments to X that are present in the assignments of P.

Example 4: Formula $F_N \wedge z$ of Example 3 has an SSA of 21 assignments to $Vars(F_N \wedge z)$. They have only 5 different assignments to $X = \{x_1, x_2, x_3\}$. The set $\{101, 100, 011, 010, 000\}$ of those assignments is a CTS for N.

Definition 3 is meant for circuits that are not "too redundant". Highly-redundant circuits are discussed in [12], [11].

IV. *SeSt* PROCEDURE

A. Motivation

Building an SSA for a large formula is inefficient. So, constructing a CTS of N from an SSA of $F_N \wedge z$ is impractical. To address this problem, we introduce a procedure called *SeSt* (a short for "Semantics and Structure"). Given formula $F_N \wedge z$ and a set of variables $V \subseteq Vars(F_N \wedge z)$, *SeSt* generates a simpler formula $H(V)$ implied by $F_N \wedge z$ at the same time trying to build an SSA for H. If *SeSt* succeeds in constructing such an SSA, formula H is unsatisfiable and so is $F_N \wedge z$. Then a set of tests T is extracted from this SSA. As we show in Subsection V-A, one can view T as an approximation of a CTS for N (if $X \subseteq V$) or an "approximation of approximation" of a CTS (if $X \not\subseteq V$).

Example 5: Consider the circuit N of Figure 3 where $X = \{x_1, x_2, x_3\}$. Assume that $V = X$. Application of *SeSt* to $F_N \wedge z$ produces $H(X) = (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge x_3$. *SeSt* also generates an SSA of H of four assignments to X:

$\{000, 001, 011, 101\}$ with center $\vec{p}_{init}$=000. (We omit the AC-mapping here.) These assignments form an approximation of a CTS for N.

B. Description of SeSt

```
$GenCls(G, V, \vec{v})\{$
$G_{\vec{v}} := GenForm(F, \vec{v})$
$(\vec{s}, R) := ChkSat(G_{\vec{v}})$
if ($\vec{s} \neq nil$)
    return($nil, \vec{s} \cup \vec{v}$)
$V' := Analyze(R, G_{\vec{v}}, G)$
$C := FormCls(V', \vec{v})$
return($C, nil$)
```

Fig. 5. *GenCls* procedure

The pseudocode of *SeSt* is shown in Figure 4. *SeSt* accepts formula G (in our case, $G := F_N \wedge z$) and a set of variables $V \subseteq Vars(G)$. *SeSt* outputs an assignment satisfying G or formula $H(V)$ implied by G and an SSA of H. Initially, H consists of the clauses of G depending only on variables of V (if any). Then a *while* loop is performed. First, *SeSt* tries to build an SSA for the current formula H by calling *BuildSSA* (line 6). If H is unsatisfiable, *BuildSSA* computes an SSA P returned by *SeSt* along with H (line 8). Otherwise, *BuildSSA* returns an assignment $\vec{v}$ satisfying H. In this case, *SeSt* calls procedure *GenCls* to build a clause C falsified by $\vec{v}$. Clause C is obtained by resolving clauses of G on variables of $Vars(G) \setminus V$. (Hence C is implied by G.) If $\vec{v}$ can be extended to an assignment $\vec{s}$ satisfying G, *SeSt* terminates (lines 10-11). Otherwise, C is added to H and a new iteration begins.

Procedure *GenCls* is shown in Figure 5. First, *GenCls* generates formula $G_{\vec{v}}$ obtained from G by discarding clauses satisfied by $\vec{v}$ and removing literals falsified by $\vec{v}$. Then *GenCls* checks if there is an assignment $\vec{s}$ satisfying $G_{\vec{v}}$. If so, $\vec{s} \cup \vec{v}$ is returned as an assignment satisfying G. Otherwise, a proof R of unsatisfiability of $G_{\vec{v}}$ is produced. Then *GenCls* forms a set $V' \subseteq V$. A variable w is in V' iff a clause of $G_{\vec{v}}$ is used in proof R and its parent clause from G has a literal of w falsified by $\vec{v}$. Finally, clause C is generated as a disjunction of literals of V' falsified by $\vec{v}$. By construction, clause C is implied by G and falsified by $\vec{v}$.

V. BUILDING APPROXIMATIONS OF CTS

A. Two kinds of approximations of CTSs

As before, let $H(V)$ denote a formula implied by $F_N \wedge z$ that is generated by $SeSt$ and P denote an SSA for H. Projections of N can be of two kinds depending on whether $X \subseteq V$ holds. Let $X \subseteq V$ be true and T be the test set consisting of all different assignments to X present in the assignments of P. Using the reasoning of Section III one can show that T is a CTS for projection of N on V. Since $H(V)$ is essentially an abstraction of $F_N \wedge z$, one can view T an approximation of a CTS for N. For that reason, we will refer to T as a **CTS**$^{\mathbf{a}}$ of N where superscript "a" stands for "approximation".

Now assume $X \subseteq V$ is not true. Generation of a test set T from P for this case is described in the next section. Let us relate this case to that of $X \subseteq V$. Assume for the sake of simplicity that $V \cap X = \emptyset$. Let us consider computing a test set T' for a projection of N on set V' where $V' = X \cup V$. Let P' be an SSA for formula $H'(V')$ generated by $SeSt$. Every assignment of P' can be represented as ($\vec{x}$,$\vec{v}$) where $\vec{x}$ and $\vec{v}$ are assignments to X and V respectively. The assignments $(\vec{x}_1, \vec{v}), (\vec{x}_2, \vec{v}), \dots$ of P' sharing the same $\vec{v}$ specify all tests of T' corresponding to $\vec{v}$. On the other hand, since $V \cap X = \emptyset$, to generate T one has to a) use some *heuristic* for generating a test corresponding to $\vec{v}$ and b) *guess* how many tests corresponding to $\vec{v}$ one should generate. Thus, T is an approximation of T' that is itself a CTS$^{\text{a}}$ i.e. an approximation of a CTS. So, we will refer to T as **CTS**$^{\mathbf{aa}}$.

B. Construction of CTS^{aa}

```
$GenTests(F_N,X,P,tr_1,tr_2)\{$
1  $T := \emptyset$
2  for each $\vec{v} \in P$ {
3      $\vec{s} := SatAssgn(F_N, \vec{v})$
4      if ($\vec{s} \neq nil$) {
5          $AddTest(T, \vec{s}, X)$
6          for ($i = 1$;$i < tr_1$;$i$++){
7              $\vec{s} := SatAssgn(F_N, \vec{v})$
8              $AddTest(T, \vec{s}, X)$}
9      else
10         for ($i = 0$;$i < tr_2$;$i$++){
11             $F_N^* := Relax(F_N)$
12             $\vec{s} := SatAssgn(F_N^*, \vec{v})$
13             if ($\vec{s} = nil$) continue
14             $AddTest(T, \vec{s}, X)$}}
15 return($T$)}
```

Fig. 6. *GenTests* procedure

Consider extraction of a test set T from SSA P of formula $H(V)$ when $X \not\subseteq V$. Since V, in general, contains internal variables[9] of N, translation of P to a test set T needs a special procedure *GenTests* shown in Figure 6. As we mentioned in Subsection V-A, building a test $\vec{x}$ corresponding to an assignment $\vec{v}$ of P requires some heuristic. In *GenTests*, we use the following idea. One can view building an SSA (see Fig. 2) as a try to reach a satisfying assignment, if any. So, intuitively, every assignment of a good SSA falsifies a very small number of clauses of G. For that reason, when building a test $\vec{x}$ corresponding to $\vec{v}$, we look for an assignment to $Vars(F_N \wedge z)$ that contains $\vec{x}$ and $\vec{v}$ and falsifies as few clauses of $F_N \wedge z$ as possible.

Parameters tr_1 and tr_2 control the number of tests generated for one assignment of P (*tr* here stands for "tries"). For every $\vec{v} \in P$, *GenTests* checks if formula F_N is satisfiable under assignment $\vec{v}$ i.e. if there exists a test under which N assigns $\vec{v}$ to V. If so, *GenTests* calls procedure *AddTest* that forms a new test by extracting the values assigned to X in $\vec{s}$ and adds it to T. (Note that the only clause of $F_N \wedge z$ falsified by $\vec{s}$ is the unit clause z.) Then *GenTests* runs a *for* loop (lines 6-8) to generate $tr_1 - 1$ more tests producing the same assignment $\vec{v}$. We assume that the SAT-solver invoked in line 7 generates different satisfying assignments in different calls.

If F_N is unsatisfiable under $\vec{v}$, *GenTests* runs another *for* loop of tr_2 iterations (lines 10-14). In every iteration, *GenTests* relaxes F_N by removing the clauses specifying a small random subset of gates. If the relaxed version of F_N has a satisfying assignment $\vec{s}$ (line 12), a test is extracted from $\vec{s}$ and added to T. Note that $\vec{s}$ falsifies only a small number of clauses of $F_N \wedge z$, namely, a subset of clauses removed to relax F_N and possibly the unit clause z.

C. Finding a set of variables to project on

---

[9] If the special case $V \subset X$ holds, every assignment of P can be easily turned into a test by assigning values to variables of $X \setminus V$ (e.g. randomly).

```
$GenCut(N, Size)${
1  $G_{out} := OutGate(N)$
2  $Gts := \{G_{out}\}$
3  $Dpth(G_{out}) := 0$
4  $Inps := \emptyset$
5  while ($|Gts \cup Inps| < Size$) {
6    $G := MinDepth(Gts, Dpth)$
7    $Gts := Gts \setminus \{G\}$
8    $Seen(G) := true$
9    foreach $G' \in FanIn(G)${
10     if ($Seen(G')$) continue
11     if ($G' \in Inputs(N)$) {
12       $Inps = Inps \cup \{G'\}$
13       continue }
14     $Dpth(G') := Dpth(G) + 1$
15     $Gts := Gts \cup \{G'\}$}}
16 return($Gts \cup Inps$)}
```

Fig. 7. *GenCut* procedure

Intuitively, a good choice of the set V to project N on is a (small) coherent subset of variables of N reflecting its structure and/or semantics. One obvious choice of V is the set X of input variables of N. In this section, we describe generation of a set V whose variables form an internal cut of N denoted as *Cut*. Procedure *GenCut* for generation of set *Cut* consisting of *Size* gates is shown in Figure 7. Set V is formed from output variables of the cut gates.

The current cut is specified by $Gts \cup Inps$. Set *Gts* is initialized with the output gate G_{out} of circuit N and *Inps* is originally empty. *GenCut* computes the *depth* of every gate of *Gts*. The depth of G_{out} is set to 0. Set *Gts* is processed in a *while* loop (lines 5-15). In every iteration, a gate of the smallest depth is picked from *Gts*. Then *GenCut* removes gate G from *Gts* and examines the fan-in gates of G (lines 9-15). Let G' be a fan-in gate of G that has not been seen yet and is not a primary input of N. Then the depth of G' is set to that of G plus 1 and G' is added to *Gts*. If G' is a primary input of N it is added to *Inps*.

VI. APPLICATIONS OF PROPERTY-CHECKING TESTS

Given a multi-output circuit M, traditional testing is used to verify M "as a whole". In this paper, we describe generation of a test set meant for checking a *particular property* of M specified by a single-output circuit N. In this section, we present some applications of property-checking test sets.

A. Verification of corner cases

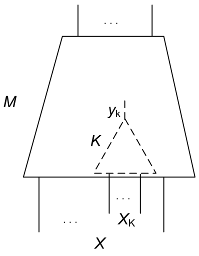


Fig. 8. Subcircuit K of circuit M

Let K be a single-output subcircuit of circuit M as shown in Figure 8. For the sake of simplicity, here, we consider the case where the set X_K of input variables of K is a subset of the set X of input variables of M. (The technique below can also be applied when input variables of K are *internal* variables of M.) Suppose K evaluates, say, to value 0 much more frequently then to 1. Then one can view an input assignment of M for which K evaluates to 1 as specifying a "corner case" i.e. a rare event. Hitting such a corner case by a random test can be very hard. This issue can be addressed by using a coverage metric that *requires* setting the value of K to both 0 and 1. (The task of finding a test for which K evaluates to 1 can be solved, for instance, by a SAT-solver.) The problem however is that hitting a corner case only once may be insufficient.

One can increase the frequency of hitting the corner case above as follows. Let N be a miter of circuits K' and K'' (see Figure 9) i.e. a circuit that evaluates to 1 iff K' and K'' are functionally inequivalent. Let K' and K'' be two copies of circuit K. So $N \equiv 0$ holds. Let test set T_K be extracted from an SSA built for a projection of N on a set $V \subset Vars(N)$. Set T_K can be viewed as a result of "squeezing" the truth table of K. Since this truth table is dominated by input assignments for which K evaluates to 0, this part of the truth table is *reduced the most*. So, one can expect that the ratio of tests of T_K for which K evaluates to 1 is higher than in the truth table of K. In Subsection VII-B, we substantiate this intuition experimentally. One can easily extend an assignment $\vec{x}_K$ of T_K to an assignment $\vec{x}$ to X e.g. by randomly assigning values to the variables of $X \setminus X_K$.

B. Testing sequential circuits

There are a few ways to apply property-checking tests meant for combinational circuits to verification of *sequential* circuits. Here is one of them based on bounded model checking [2]. Let M be a sequential circuit and ξ be a property of M. Let $N_k(X, Y, z)$ be a circuit such that $N_k \equiv 0$ holds iff ξ is true for k time frames. Circuit N_k is obtained by unrolling M k times and adding logic specifying property ξ. Set X consists of the subset X' specifying the state variables of M in the first time frame and subset X'' specifying the combinational input variables of M in k time frames.

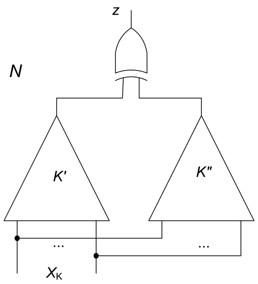


Fig. 9. The miter of circuits K' and K''

Having constructed N_k, one can build CTSs, CTS$^{\mathrm{a}}$s and CTS$^{\mathrm{aa}}$s for testing property ξ of M. The only difference here from the problem we have considered so far is as follows. Circuit M starts in a state satisfying some formula $I(X')$ that specifies the initial states. So, one needs to check if $N_k \equiv 0$ holds only for the assignments to X satisfying $I(X')$. A test here is an assignment $(\vec{x'}_1, \vec{x''}_1, \dots, \vec{x''}_k)$ where $\vec{x'}_1$ is an initial state and $\vec{x''}_i$, $1 \leq i \leq k$ is an assignment to the combinational input variables of i-th time frame. Given a test, one can easily compute the corresponding sequence of states $(\vec{x'}_1, \dots, \vec{x'}_k)$ of M. In Subsection VII-C, we give examples of building CTS$^{\mathrm{aa}}$s for testing sequential circuits.

C. Exposing bugs overlooked due to misdefining properties

One can use property-checking tests to mitigate the problem of incomplete specifications. By running tests generated for an incomplete set of properties of M, one can expose bugs overlooked due to missing some properties. An important special case of this problem is as follows. Let ξ be a property of M that holds. Assume that the correctness of M requires proving a slightly *different* property ξ' that *does not hold.* By running a test set T built for property ξ, one may expose a bug overlooked in formal verification due to proving ξ instead of

ξ'. In Subsection VII-C, we illustrate this idea experimentally. Note that the problem above has nothing to do with the complexity of proving ξ' false. The designer simply does not know that *there is* a problem and so can overlook a bug even if proving ξ' false is very easy.

VII. Experiments

In this section, we describe experiments with property-checking tests (PCT) generated by procedure *GenPCT* shown in Figure 10. *GenPCT* accepts a single-output circuit N and outputs a set of tests T. (For the sake of simplicity, we assume here that $N \equiv 0$ holds.) *GenPCT* starts with generating formula $F_N \wedge z$. Then it builds a set of variables $V \subseteq \mathit{Vars}(F_N \wedge z)$. Parameter *type* specifies whether *GenPCT* is supposed to generate a CTS, CTS$^{\text{a}}$ or CTS$^{\text{aa}}$. After that, *GenPCT* calls SeSt (see Fig. 4) to compute a formula $H(V)$ implied by $F_N \wedge z$ and its SSA.

```
$GenPCT(N, X, type, tr_1, tr_2)\{$
1  $F_N \wedge z := GenForm(N)$
2  $V := GenVars(F_N \wedge z, type)$
3  $(H, P) := SeSt(F_N \wedge z, V)$
4  if $(X \subseteq V)$
5    $T := ExtrTests(X, P)$
6  else $\{$
7    $RedVars := V \setminus Vars(H)$
8    $P := Drop(P, RedVars)$
9    $T := GenTests(F_N, X, P, tr_1, tr_2)\}$
10 return$(T)\}$
```

Fig. 10. *GenPCT* procedure

If $X \subseteq V$ holds (where X is the set of input variables of N), *GenPCT* computes T as the set of all different assignments to X present in assignments of P (line 5). Otherwise, *GenPCT* calls procedure *GenTests* (see Fig. 6). Every variable $w \in V \setminus \mathit{Vars}(H)$ is redundant in the sense that its value is the same in all assignments of P. So the values assigned to $V \setminus \mathit{Vars}(H)$ are dropped by *GenTests* (lines 7-8). If $V = \mathit{Vars}(F_N \wedge z)$, then $H(V)$ is $F_N \wedge z$ itself and *GenPCT* produces a CTS of N. Otherwise, according to definitions of Subsection V-A, *GenPCT* generates a CTS$^{\text{a}}$ (if $X \subseteq V$) or CTS$^{\text{aa}}$ (if $X \not\subseteq V$).

In the following subsections, we describe results of three experiments. In the first two experiments we used circuits specifying next state functions of latches of HWMCC-10 benchmarks. (The motivation was to employ realistic circuits.) In the third experiment, we used combinational circuits obtained by unfolding HWMCC-10 benchmarks. In our implementation of SeSt, as a SAT-solver, we used Minisat 2.0 [6], [17]. We also employed Minisat to run simulation. To compute the output value of N under test $\vec{x}$, we added unit clauses specifying $\vec{x}$ to formula $F_N \wedge z$ and checked its satisfiability.

A. Comparing CTSs, CTS^{a}s and CTS^{aa}s

The objective of the first experiment was to give examples of circuits with non-trivial CTSs and compare the efficiency of computing CTSs, CTS$^{\text{a}}$s and CTS$^{\text{aa}}$s. In this experiment, N was a miter specifying equivalence checking of circuits M' and M'' (see Figure 9). M'' was obtained from M' by optimizing the latter with ABC [15].

TABLE I

Computing CTSs, CTS^{a}s and CTS^{aa}s

name	la-tch	#inp vars	#ga-tes	CTS		CTS$^{\text{a}}$ or CTS$^{\text{aa}}$			
				$\lvert SSA\rvert$ (#tests) $\times 10^3$	time (s.)	test set type	$\lvert V\rvert$	$\lvert SSA\rvert$ (#tests) $\times 10^3$	time (s.)
bob3	L26	14	41	46 (2.0)	0.1	CTS$^{\text{a}}$	14	0.6 (0.6)	0.01
eijks258	L10	16	45	259 (8.2)	0.5	CTS$^{\text{a}}$	16	0.1 (0.1)	0.02
cmudme1	L230	19	50	2,184 (63)	5.4	CTS$^{\text{a}}$	19	13 (13)	0.1
mutexp0	L60	29	199	memout	$*$	CTS$^{\text{a}}$	29	659 (659)	26
pdtpmsmiim	L118	31	136	memout	$*$	CTS$^{\text{a}}$	31	936 (936)	4.2
abp4pold	L270	129	1,178	memout	$*$	CTS$^{\text{aa}}$	22	0.9 (0.5)	0.6
pj2009	L1318	366	25,160	memout	$*$	CTS$^{\text{aa}}$	22	0.6 (0.3)	51
mentorb..00	L8670	626	3,156	memout	$*$	CTS$^{\text{aa}}$	22	1.2 (0.6)	11
139454p0	L1676	791	19,843	memout	$*$	CTS$^{\text{aa}}$	22	0.1 (0.1)	99

The results of the first experiment are shown in Table I. The first two columns specify an HWMCC-10 benchmark and its latch whose next state function was used as M'. The next two columns give the number of input variables and that of gates in the miter N. The following pair of columns describe computing a CTS for N. The first column of this pair gives the size of the SSA P found by *GenPCT* in thousands. The number of tests in the set T extracted from P is shown in the parentheses in thousands. The second column of this pair gives the run time of *GenPCT* in seconds.

The last four columns of Table I describe results of computing test sets for a projection of N on a set of variables V. The first column of this group shows if CTS$^{\text{a}}$ or CTS$^{\text{aa}}$ was computed whereas the next column gives the size of V. The third column of this group provides the size of SSA P and the test set T extracted from P (in parentheses). Both sizes are given in thousands. The last column shows the run time of *GenPCT*. For the first five examples, we used a projection of N on X, thus constructing a CTS$^{\text{a}}$ of N. For the last four examples we computed a projection of N on an internal cut (see Subsection V-C) thus generating a CTS$^{\text{aa}}$ of N. *GenPCT* was called with $tr_1 = 1, tr_2 = 5$ (see Fig. 6 and 10).

For the first three examples, *GenPCT* managed to build non-trivial CTSs that are smaller than $2^{|X|}$. For instance, the trivial CTS for example *bob3* consists of 2^{14}=16,384 tests, whereas *GenPCT* found a CTS of 2,004 tests. (So, to prove M' and M'' equivalent it suffices to run 2,004 out of 16,384 tests.) For the other examples, *GenPCT* failed to build a non-trivial CTS due to exceeding the memory limit (1.5 Gbytes). On the other hand, *GenPCT* built a CTS$^{\text{a}}$ or CTS$^{\text{aa}}$ for all nine examples of Table I. Note, however, that CTS$^{\text{a}}$s give only a moderate improvement over CTSs. For the last four examples *GenPCT* failed to compute a CTS$^{\text{a}}$ of N due to memory overflow whereas it had no problem computing an CTS$^{\text{aa}}$ of N. So CTS$^{\text{aa}}$s can be computed efficiently even for large circuits. Further, we show that CTS$^{\text{aa}}$s are also very effective.

B. Testing corner cases

In the second experiment, we generated CTS$^{\text{a}}$s and CTS$^{\text{aa}}$s to test corner cases (see Subsection VI-A). First, we formed a circuit K that evaluates to 0 for almost all input assignments. So, the assignments for which K evaluates to 1 are corner cases[10]. We compared the frequency of hitting corner cases by random tests and by tests of a set T built by *GenPCT* as

---

[10] We assume here that K is a subcircuit of some circuit M. The input assignments for which K evaluates to 1 are corner cases for M.

follows. Let N be a miter of copies K' and K'' (see Figure 9). The set T was generated using a projection of N either on the set X of input variables or an internal cut of N.

TABLE II

Testing corner cases

name	la-tch	#inp vars	#and vars	#ga-tes	random testing		testing by CTS^{a} and CTS^{aa}				
					#te-sts	#hits %	test set	$\|V\|$	#te-sts	#hits %	time (s.)
pd..gigamax5	L46	43	10	512	10^5	0.02	CTS^{a}	43	547	7.1	0.2
pd..gigamax5	L46	63	30	512	10^8	0	CTS^{a}	63	1,243	3.0	0.2
pdtvisbpb1	L48	46	10	108	10^5	0.04	CTS^{a}	46	398	9.0	0.01
pdtvisbpb1	L48	66	30	108	10^8	0	CTS^{a}	66	736	3.1	0.03
abp4pold	L270	139	10	637	10^5	0.02	CTS^{aa}	35	2,047	8.5	0.9
abp4pold	L270	159	30	637	10^8	0	CTS^{aa}	55	5,256	3.3	2.1
mentorbm1p00	L8670	636	10	1,630	10^5	0.1	CTS^{aa}	35	594	11	3.7
mentorbm1p00	L8670	656	30	1,630	10^8	0	CTS^{aa}	55	2,009	4.7	8.7

To build circuit K, we extracted the circuit R specifying the next state function of a latch of a HWMCC-10 benchmark and composed it with an n-input AND gate as shown in Figure 11. The circuit K outputs 1 only if R evaluates to 1 and the first $n-1$ inputs variables of the AND gate are set to 1 too. So the input assignments for which K evaluates to 1 are "corner cases".

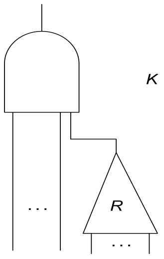


Fig. 11. Circuit K whose output value is biased to 0

The results of the experiment are given in Table II. The first two columns name the benchmark and latch whose next state function was used as circuit R. The next three columns give the total number of input variables of K, the value of n in the n-input AND gate fed by R and the number of gates in circuit K. The following pair of columns describes the performance of random testing. The first column of this pair gives the total number of tests. The next column shows the percentage of times circuit K evaluated to 1 (and so a corner case was hit). The last five columns of Table II describe the results of *GenPCT*. The first column of the five indicates whether a CTS^{a} or CTS^{aa} was generated. The second column gives the size of set V on which a projection of N was computed. CTS^{a}s were generated with $V = X$. When computing CTS^{aa}s, the set V formed an internal cut of N and parameters tr_1 and tr_2 were both set to 1. The next column shows the size of the test set. The fourth column gives the percentage of times a corner case was hit. The last column shows the total run time.

The examples of Table II were generated in pairs that shared the same circuit R and were different only in the size of the AND gate fed by R. For instance, in the first and second entry of Table II, circuit K was obtained by composing the same circuit R extracted from benchmark *pdtvisgigamax5* with 10-input and 30-input AND gates respectively. Table II shows that for circuits with a 10-input AND gate, random testing hit corner cases but the percentage of those events was much lower than for CTS^{a}s and CTS^{aa}s. On the other hand, even 100 millions of random tests failed to hit a single corner case for examples with a 30-input AND gate in sharp contrast to CTS^{a}s and CTS^{aa}s.

C. Testing properties defined incorrectly

TABLE III

*Testing "misdefined" properties. CTS^{aa}s were computed for $|V| = 20$. Test sets with a counterexample are shown **in bold**.*

name	#ti-me fra-mes	#inp vars	#ga-tes $\times 10^3$	cov. met. tests		random tests		testing by CTS^{aa}		
				#tests	time (s.)	#tests	time (s.)	#iter	#tests	time (s.)
bobcount	19	38	1.6	740	0.4	$\mathbf{1.0*10^7}$	294	1	**3,339**	1.1
boblivea	5	65	8.0	3,778	7.2	$\mathbf{9.7*10^3}$	2.1	100	9,982	74
p..gigamax0	4	88	4.3	2,150	6.3	$\mathbf{1.4*10^6}$	158	20	**923**	3.7
kenflashp01	2	108	2.5	1,076	0.8	10^8	1,625	48	**6,027**	1.7
pdtpmsudc8	10	110	3.7	2,066	2.5	$6.8*10^7$	5,000	100	51,123	283
eijks526	39	117	18	**8,976**	70	$4.5*10^6$	5,000	1	**183**	31
kenopp1	3	129	1.7	1,202	0.5	10^8	695	13	**1,344**	0.4
vis..cellp01	5	135	14	**4,581**	16	$8*10^7$	5,000	13	**1,354**	4.4
cmugigamax	5	159	3.1	1,826	2.3	10^8	2,671	100	8,985	13
eijks5378	6	209	17	8,318	56	$\mathbf{3.4*10^4}$	58	1	**387**	3.6
eijks208o	25	250	4.0	1,506	3.6	$\mathbf{1.9*10^7}$	2,207	3	**1,811**	4.9
eijks420	18	324	6.6	1,115	3.7	$\mathbf{4.1*10^6}$	1,140	86	**26,199**	82
n..guidancep1	6	504	10	7,922	27	$2.1*10^7$	5,000	6	**378**	2.3
pdt..feistel	12	816	115	**68,006**	4,066	$3.9*10^6$	5,000	5	**804**	49
nusmvtcasp2	7	1,029	19	11,510	82	$4.5*10^7$	5,000	38	**3,549**	53
cmuperiodic	34	1,220	51	**30,999**	760	$9.5*10^6$	5,000	85	**5,611**	240
pj2002	4	4,054	137	**61,113**	3.868	$0.6*10^6$	5,000	2	**161**	7.9

The objective of the third experiment was to expose bugs overlooked due to incorrect definition of properties (see Subsection VI-C). In contrast to the previous two experiments, here we employed "complete" HWMCC-10 benchmarks, each benchmark specifying a safety property ξ of a sequential circuit M. In our experiment, we used benchmarks with *true* properties. We assumed that ξ was defined incorrectly and formed a new property ξ' of M that failed. Property ξ' served as the "real" property to check. It was obtained by changing the functionality of a gate of M involved in specifying property ξ. The fact that ξ' indeed failed was established by running IC3 [3]. Let k denote the length of the counterexample found by IC3 for ξ'. We unrolled the transition relation of M k times to generate single-output circuits N_k and N'_k. These circuits evaluated to 1 iff no counterexample of length k existed for ξ and ξ' respectively. By construction, $N_k \equiv 0$ held whereas $N'_k \equiv 0$ did not.

In our experiment, we compared three different methods of breaking property ξ'. In the first method, we used testing driven by a coverage metric. Namely, we generated a test set T aimed at setting the output[11] of every gate G of N_k both to 0 and 1. Then we applied T to N'_k to disprove $N'_k \equiv 0$. Note that a single test sets the output of every gate of N_k to 0 or 1. To make T stronger, when processing a gate G of N_k we tried to find a new test setting the output of G to $b \in \{0,1\}$, even if this goal was "inadvertently" achieved earlier. In the

---

[11] In [11], we give results for the coverage metric based on stuck-at faults.

second method, we simply applied random tests[12] to N'_k until a counterexample was generated or a resource was exceeded. In the third method, we applied *GenPCT* to circuit N_k to generate a CTS^{aa} T. Then we used T to break $N'_k \equiv 0$.

A sample of 17 benchmarks is shown in Table III. When compiling this sample we dropped the easy examples solved by all three methods. The first column of Table III lists names of benchmarks. The second column specifies the value of k in N_k and N'_k. The third column gives the number of input variables in N_k (and N'_k) minus[13] the number of latches in M. The fourth column of Table III shows the number of gates in N_k and N'_k (in thousands). The following pair of columns describes the performance of testing driven by the coverage metric above (the number of tests and the run time required to generate and run them). The next two columns provide the results of random testing limited to 100 million tests and the runtime of 5,000 secs.

The final three columns describe the results of CTS^{aa}s. The first column of the three gives the number of iterations we tried when building a CTS^{aa}. Each iteration was a separate run of *GenPCT* generating a different set of tests due to randomization of internal procedures[14]. CTS^{aa}s were built for a projection of N_k on a set of variables V forming an internal cut of N_k. *GenPCT* was run with $tr_1 = 20$ and $tr_2 = 5$. Iterating *GenPCT* went on until $N'_k \equiv 0$ was broken or the number of iterations reached 100. The final two columns describe the total number of tests and run time (over all iterations).

The results of Table III show the high efficiency and effectiveness of CTS^{aa}s on the examples we tried. In particular, for four examples (*kenflashp01*, *kenoppl*, *nusmvguidancep1* and *nusmvtcasp2*) a CTS^{aa} was the only test set to break $N'_k \equiv 0$. Our experiment suggests that one can run the procedure below to check if a bug is overlooked due to misdefining a true property ξ of circuit M. (This procedure does not require knowledge of the "right" property ξ'.) 1) Pick a number k (by an educated guess) to form circuit N_k. 2) Pick a number p of tests to build when proving $N_k \equiv 0$. Run *GenPCT* in a loop until a set T of p tests is generated. 3) Make sure that M correctly behaves on tests of T "as a whole" e.g. by checking that the properties of M related to ξ hold for T.

VIII. BACKGROUND

As we mentioned earlier, traditional testing checks if a circuit M is correct as a whole. This notion of correctness means satisfying a conjunction of *many* properties of M. For this reason, one tries to spray tests uniformly in the space of all input assignments. To improve the effectiveness of testing, one can try to run many tests at once as it is done in symbolic simulation [4]. To avoid generation of tests that for some reason should be or can be excluded, a set of constraints can be used [13]. Another method of making testing more reliable is to generate tests exciting a particular set of events specified by a coverage metric [16]. Our approach is different from those above in that it is aimed at testing a particular property of M.

The method of testing introduced in [10] is based on the idea that tests should be treated as a "proof encoding" rather than a sample of the search space. (The relation between tests and proofs have been also studied in software verification, e.g. in [7], [8], [1]). In this paper, we take a different point of view where testing becomes a *part* of a formal proof namely the part that performs structural derivations.

Reasoning about SAT in terms of random walks was pioneered in [14]. The centered SSAs we introduce in this paper bear some similarity to sets of assignments generated in derandomization of Schöning's algorithm [5].

The first version of $SeSt$ procedure is presented in report [12]. It has a much tighter integration between the structural part (computation of SSAs) and semantic part (derivation of formula H implied by the original formula). The advantage of the new version of $SeSt$ described in this paper is twofold. First, it is much simpler than $SeSt$ of [12]. In particular, any resolution based SAT-solver that generates proofs can be used to implement the new $SeSt$. Second, the simplicity of the new version makes it much easier to achieve the level of scalability where $SeSt$ becomes practical.

IX. CONCLUSION

We consider the problem of finding a Complete Test Set (CTS) for a combinational circuit N that is a test set proving $N \equiv 0$. We use the machinery of stable sets of assignments to derive non-trivial CTSs i.e. those that do not include all possible input assignments. Computing a CTS for a large circuit N is inefficient. So, we present a procedure that generates a test set for a "projection" of N on a subset V of variables of N. Depending on the choice of V, this procedure generates a test set CTS^{a} that is an approximation of an CTS or a test set CTS^{aa} that is an approximation of CTS^{a}. We give experimental results showing that CTS^{aa}s can be efficiently computed even for large circuits and are effective in solving verification problems.

X. ACKNOWLEDGMENT

This research was supported in part by NSF grants CCF-1117184 and CCF-1319580.

---

[12] Even in a random test, the values assigned to the input variables of N_k and N'_k corresponding to state variables of circuit M had to satisfy the predicate specifying the initial states of M (see Subsection VI-B).

[13] The HWMCC-10 benchmarks have only one initial state. So in every test generated in our experiment, the input variables of N_k and N'_k corresponding to the state variables of M were simply set to a constant value.

[14] In particular, a different center was used for the SSA of formula H implied by $F_{N_k} \wedge z$. Formula H was also different in every run of *GenPCT* due to randomization of SAT-calls invoked in *GenCls* (line 2 of Fig. 5).

REFERENCES

[1] N. Beckman, A. Nori, S. Rajamani, R. Simmons, S. Tetali, and A. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, July 2010.

[2] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[3] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[4] R. Bryant. Symbolic simulation—techniques and applications. In *DAC-90*, pages 517–521, 1990.

[5] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic (22/(k+1))n algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69 – 83, 2002.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, Santa Margherita Ligure, Italy, 2003.

[7] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *TAP*, pages 169–188, 2007.

[8] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Integrated Formal Methods*, pages 20–32, 2005.

[9] E. Goldberg. Testing satisfiability of cnf formulas by computing a stable set of points. In *Proc. of CADE-02*, pages 161–180, 2002.

[10] E. Goldberg. On bridging simulation and formal verification. In *VMCAI-08*, pages 127–141, 2008.

[11] E. Goldberg. Complete test sets and their approximations. Technical Report arXiv:1808.05750 [cs.LO], 2018.

[12] E. Goldberg. Generation of complete test sets. Technical Report arXiv:1804.00073 [cs.LO], 2018.

[13] N. Kitchen and A.Kuehlmann. Stimulus generation for constrained random simulation. In *ICCAD-07*, pages 258–265, 2007.

[14] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *32nd Annual Symposium of Foundations of Computer Science*, pages 163–169, Oct 1991.

[15] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, 2017. http://www.eecs.berkeley.edu/∼alanmi/abc.

[16] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, Jul 2001.

[17] Minisat2.0. http://minisat.se/MiniSat.html.

ISBN: 978-0-9835678-8-2.

Expansion-Based QBF Solving Without Recursion

Roderick Bloem, Nicolas Braud-Santoni, Vedad Hadzic,
TU Graz
Uwe Egly, Florian Lonsing,
TU Wien
Martina Seidl,
JKU Linz

Abstract—**In recent years, expansion-based techniques have been shown to be very powerful in theory and practice for solving quantified Boolean formulas (QBF), the extension of propositional formulas with existential and universal quantifiers over Boolean variables. Such approaches partially expand one type of variable (either existential or universal) and pass the obtained formula to a SAT solver for deciding the QBF. State-of-the-art expansion-based solvers process the given formula quantifier-block wise and recursively apply expansion until a solution is found.**

In this paper, we present a novel algorithm for expansion-based QBF solving that deals with the whole quantifier prefix at once. Hence recursive applications of the expansion principle are avoided. Experiments indicate that the performance of our simple approach is comparable with the state of the art of QBF solving, especially in combination with other solving techniques.

I. INTRODUCTION

Efficient tools for deciding the satisfiability of Boolean formulas (SAT solvers) are the core technology in many verification and synthesis approaches [45]. However, verification and synthesis problems are often beyond the complexity class NP as captured by SAT, requiring more powerful formalisms like *quantified Boolean formulas* (QBFs). QBFs extend propositional formulas by universal and existential quantifiers over Boolean variables [32] resulting in a decision problem that is PSPACE-complete. Applications from verification and synthesis [8], [13], [14], [18], [20], [24], realizability checking [19], bounded model checking [16], [48], and planning [17], [41] motivate the quest for efficient QBF solvers.

Unlike for SAT, where *conflict-driven clause learning* (CDCL) is the single dominant solving approach for practical problems, two dominant approaches exist for QBF solving. On one hand, CDCL has been successfully extended to QCDCL that enables clause and cube learning [21], [35], [47]. On the other hand, *variable expansion* has become very popular. In short, expansion-based solvers eliminate one kind of variables by assigning them truth values and solve the resulting propositional formula with a SAT solver. For QBFs with one quantifier alternation (2QBF), a natural approach is to use two SAT solvers: one that deals with the existentially quantified variables and another one that deals with the universally quantified variables. For generalising this SAT-based approach to QBFs with an arbitrary number of quantifier alternations, expansion is recursively applied per quantifier block, requiring multiple SAT solvers. As noted by Rabe and Tentrup [39], these CEGAR-based approaches show poor performance for formulas with many quantifier alternations in general.

In this paper, we present a novel solving algorithm based on non-recursive expansion for QBFs with arbitrary quantifier prefixes using only two SAT solvers. Our approach of non-recursive expansion is theoretically (i.e., from a proof complexity perspective) equivalent to approaches that apply recursive expansion since both non-recursive and recursive expansion rely on the $\forall$Exp+Res proof system [5]. However, the non-recursive expansion has practical implications such as a modified search strategy. That is, the use of recursive or non-recursive expansion results in different search strategies for the proof. With respect to proof search, there is an analogy to, e.g., implementations of resolution-based CDCL SAT solvers that employ different search heuristics.

In addition, we implemented a hybrid approach that combines clause learning with non-recursive expansion-based solving for exploiting the power of QCDCL. Our experiments indicate that this hybrid approach performs very well, especially on formulas with multiple quantifier alternations.

This paper is structured as follows. After a review of related work in the next section, we introduce the necessary preliminaries in Section III. After a short recapitulation of expansion in Section IV, our novel non-recursive expansion-based algorithm is presented in Section V. Implementation details are discussed in Section VI together with a short discussion of the hybrid approach. In Section VII we compare our approach to state-of-the-art solvers.

II. RELATED WORK

Already the early QBF solvers Qubos [2] and Quantor [2] incorporate selective quantifier expansion for eliminating one kind of quantification to reduce the given QBF to a propositional formula. The resulting propositional formula is then solved by calling a SAT solver once. Qubos and Quantor impressively demonstrated the power of expanding universal variables but also showed its enormous memory consumption. As a pragmatic compromise, bounded universal expansion was introduced for efficient preprocessing [11], [22], [23], [46].

The first approach which uses two alternate SAT solvers A and B for solving 2QBF, i.e., QBFs of the form $\forall U \exists E.\phi$, was presented in [40]. Solver A is initialised with ϕ, B with the empty formula. Both propositional formulas are incrementally refined with satisfying assignments found by the other solver. If A finds its formula unsatisfiable, then the QBF is false. Otherwise, the negation of the universal part of the satisfying assignment is passed to solver B. If solver B finds its formula unsatisfiable, then the QBF is true. Otherwise, the existential part of the satisfying assignment is passed to solver A. Janota and Marques-Silva generalised the idea of alternating SAT solvers [31] such that one solver deals with the existentially

ISBN: 978-0-9835678-8-2.

quantified variables and one solver deals with the universally quantified variables exclusively. Solver A gets *instantiations* of ϕ in which the universal variables are assigned, and solver B gets instantiations of $\neg\phi$ in which the existential variables are assigned. The satisfying assignment found by one solver is used to obtain a new instantiation for the other. This loop is repeated until one solver returns unsatisfiable. This approach realises a natural application of the counter-example guided abstraction refinement (CEGAR) paradigm [15]. A detailed survey on 2QBF solving is given in [3].

A significant advancement of expansion-based solving for QBF with an arbitrary number of quantifier alternations was made with the solver RAReQS [26], [27], which recursively applies the previously discussed 2QBF approach [31] for each quantifier alternation. The approach turned out to be highly competitive.[1] For formalising this solving approach the calculus ∀Exp+Res was introduced [5], and proof-theoretical investigations revealed the orthogonal strength of ∀Exp+Res and Q-resolution [33], the QBF variant of the resolution calculus that forms the basis for QCDCL-based solvers. Research on the proof complexity of QBF has identified an exponential separation between Q-resolution and the ∀Exp+Res system. There are families of QBFs for which any Q-resolution proof has exponential size, in contrast to ∀Exp+Res proofs of polynomial size, and vice versa. Hence these two systems have orthogonal strength.

Recent work successfully combines machine learning with this CEGAR approach [25]. Motivated by the success of expansion-based QBF solving, several other approaches [10], [30], [39], [42]–[44] have been presented that are based on levelised SAT solving, i.e., one SAT solver is responsible for the variables of one quantifier block. In this paper, we also introduce a solving approach that is based upon propositional abstraction but considers the whole quantifier prefix at once.

III. PRELIMINARIES

The QBFs considered in this paper are in prenex normal form $\Pi.\phi$ where Π is a quantifier prefix $Q_1x_1Q_2x_2\dots Q_nx_n$ over the set of variables $X = \{x_1,\dots,x_n\}$ with $Q_i \in \{\forall,\exists\}$ and $x_i \neq x_j$ for $i \neq j$. The propositional formula ϕ contains only variables from X. Unless stated otherwise, we do not make any assumptions on the structure of ϕ. Sometimes $\Pi.\phi$ is in *prenex conjunctive normal form* (PCNF), i.e., Π is a prefix as introduced before and ϕ is a conjunction of clauses. A clause is a disjunction of literals, and a literal is a variable or the negation of a variable. The prefix imposes the order $<_\Pi$ on the elements of X such that $x_i <_\Pi x_j$ if $i < j$. By U_Π (E_Π) we denote the set of universally (existentially) quantified variables of the prefix Π. If clear from the context we omit the subscript Π. We assume the standard semantics of QBF. A QBF consisting of only the syntactic truth constant $\bot$ ($\top$) is false (true). A QBF $\forall x\Pi.\phi$ is true if $\Pi.\phi[x \leftarrow \top]$ and $\Pi.\phi[x \leftarrow \bot]$ are both true, where $\phi[x \leftarrow t]$ is the substitution of x by t in ϕ. A QBF $\exists x\Pi.\phi$ is true if $\Pi.\phi[x \leftarrow \top]$ or $\Pi.\phi[x \leftarrow \bot]$ is true.

Given a set of variables X, we call a function $\sigma\colon X \rightarrow \{\top,\bot,\epsilon\}$ an assignment for X. If there is an $x \in X$ with $\sigma(x) = \epsilon$ then σ is a *partial* assignment, otherwise σ is a *full* assignment of X. Informally, $\sigma(x) = \epsilon$ means that σ does not assign a truth value to variable x. A restriction $\sigma|_Y\colon Y \rightarrow \{\top,\bot,\epsilon\}$ of assignment $\sigma\colon X \rightarrow \{\top,\bot,\epsilon\}$ to $Y \subseteq X$ is defined by $\sigma|_Y(x) = \sigma(x)$ if $x \in Y$, otherwise $\sigma|_Y(x) = \epsilon$. By Σ_X we denote the set of all full assignments $\sigma\colon X \rightarrow \{\top,\bot\}$. Let ϕ be a propositional formula over X. By $\sigma(\phi)$ we denote the application of assignment $\sigma\colon X \rightarrow \{\top,\bot,\epsilon\}$ on ϕ, i.e., $\sigma(\phi)$ is the formula obtained by replacing variables $x \in X$ by $\sigma(x)$ if $\sigma(x) \in \{\top,\bot\}$ and performing standard propositional simplifications. Let ϕ,ψ be propositional formulas over the set of variables X. If for every full assignment $\sigma \in \Sigma_X$, $\sigma(\phi) = \sigma(\psi)$ then ϕ and ψ are equivalent. Let $\tau\colon X \rightarrow \{\top,\bot,\epsilon\}$ and $\sigma\colon Y \rightarrow \{\top,\bot,\epsilon\}$ be assignments such that for every $x \in X \cap Y$, $\tau(x) = \sigma(x)$ if $\tau(x) \neq \epsilon$ and $\sigma(x) \neq \epsilon$. Then the composite assignment of σ and τ is denoted by $\sigma\tau\colon X \cup Y \rightarrow \{\top,\bot,\epsilon\}$ and for every propositional formula ϕ over $X \cup Y$, it holds that $\sigma\tau(\phi) = \tau\sigma(\phi) = \sigma(\tau(\phi)) = \tau(\sigma(\phi))$. Furthermore, $\sigma\sigma = \sigma$ for any assignment σ.

Example 1. *Let $\sigma\colon X \rightarrow \{\top,\bot,\epsilon\}$ be an assignment over variables $\{a,b,x,y\}$ defined by $\sigma(a) = \top$, $\sigma(b) = \epsilon$, $\sigma(x) = \top$, and $\sigma(y) = \epsilon$. The restriction $\tau = \sigma|_Y$ of σ to $Y = \{x,y\}$ is given by $\tau(a) = \epsilon$, $\tau(b) = \epsilon$, $\tau(x) = \top$, $\tau(y) = \epsilon$. For the propositional formula $\phi = (x \vee a \vee y) \wedge (\neg x \vee \neg a \vee y) \wedge (\neg y \vee b)$, the application of σ and τ on ϕ gives us $\sigma(\phi) = y \wedge (\neg y \vee b)$ and $\tau(\phi) = (\neg a \vee y) \wedge (\neg y \vee b)$.*

IV. EXPANSION

In the following, we introduce the notation and terminology used for describing expansion-based QBF solving in general, and the algorithm introduced in the next section in particular. We first define the notion of *instantiation* that is inspired by the axiom rule of the calculus ∀Exp+Res [29].

Definition 1. *Let $\Pi.\phi$ be a QBF with prefix $\Pi = Q_1x_1\dots Q_nx_n$ over the set of variables $X = \{x_1,\dots,x_n\}$ and $\sigma\colon Y \rightarrow \{\top,\bot,\epsilon\}$ with $Y \subseteq X$ an assignment. If $Y \subset X$, we extend the domain of σ to X by setting $\sigma(x) = \epsilon$ if $x \notin Y$. The* instantiation of ϕ by σ*, denoted by ϕ^σ, is obtained from ϕ as follows:*

1) *all variables $x \in X$ with $\sigma(x) \neq \epsilon$ are set to $\sigma(x)$;*
2) *all variables $x \in X$ with $\sigma(x) = \epsilon$ are replaced by x^ω where annotation ω is uniquely defined by the sequence $\sigma(x_{k_1})\sigma(x_{k_2})\dots\sigma(x_{k_m})$ such that the set formed from the variables x_{k_i} contains all variables of X with $x_{k_i} <_\Pi x$. Furthermore, $x_{k_i} <_\Pi x_{k_j}$ if $k_i < k_j$.*

If we instantiate a QBF $\Pi.\phi$ with the full assignment $\sigma\colon U_\Pi \rightarrow \{\top,\bot\}$ of the universal variables, we obtain a propositional formula that contains only (possibly annotated) variables from E_Π. The dual holds for the instantiation by a full assignment $\sigma\colon E_\Pi \rightarrow \{\top,\bot\}$.

---

[1] http://www.qbflib.org

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Example 2. *Given the QBF* $\forall a \exists x \forall b \exists y.\phi$ *with* $\phi = ((x \vee a \vee y) \wedge (\neg x \vee \neg a \vee y) \wedge (\neg y \vee b))$*. Then* $U = \{a, b\}$ *and* $E = \{x, y\}$*. Let* $\sigma\colon U \to \{\top, \bot, \epsilon\}$ *be defined by* $\sigma(a) = \top$ *and* $\sigma(b) = \bot$*. Then* $\phi^{\sigma} = (\neg x^{\top} \vee y^{\top\bot}) \wedge \neg y^{\top\bot}$*. Further, let* $\tau\colon E \to \{\top, \bot, \epsilon\}$ *with* $\tau(x) = \bot$ *and* $\tau(y) = \bot$*. Then* $\phi^{\tau} = a$*. Note that* a *is not annotated because it occurs in the first quantifier block.*

Sometimes we want to remove the annotations again from an assignment or an instantiated formula. Therefore, we introduce the following notation. Let ϕ^{σ} be an instantiation by assignment $\sigma\colon X \to \{\top, \bot, \epsilon\}$ and X^{σ} the set of annotated variables. If we have an assignment $\tau\colon X^{\sigma} \to \{\top, \bot, \epsilon\}$, then we define $\tau^{-\sigma}\colon X \to \{\top, \bot\}$ by $\tau^{-\sigma}(x) = \tau(x^{\sigma})$ for $x^{\sigma} \in X^{\sigma}$. If we have an instantiated formula ϕ^{σ}, the $(\phi^{\sigma})^{-\sigma}$ is the formula obtained by replacing every annotated variable $x^{\sigma} \in X^{\sigma}$ by x. In general, $(\phi^{\sigma})^{-\sigma} \neq \phi$.

Lemma 1. *Let* $\Pi.\phi$ *be a QBF with variables* X *and* $\sigma\colon X \to \{\top, \bot, \epsilon\}$ *be a partial assignment. Then* $(\phi^{\sigma})^{-\sigma}$ *and* $\sigma(\phi)$ *are equivalent.*

Proof. By induction over the formula structure. For the base case let $\phi = x$ with $x \in X$. If $\sigma(x) = \epsilon$, $\sigma(\phi) = x$ and $\phi^{\sigma} = x$. Then $(\phi^{\sigma})^{-\sigma} = x$. Otherwise, $\phi^{\sigma} = \sigma(x)$. Obviously, $\sigma(\phi) = \sigma(x) = (\sigma(x))^{-\sigma} \in \{\top, \bot\}$. The induction step naturally follows from the semantics of the logical connectives. $\square$

Example 3. *Reconsider the propositional formula* ϕ *and assignments* σ, τ *from above (Example 2). Then* $(\phi^{\sigma})^{-\sigma} = ((\neg x^{\top} \vee y^{\top\bot}) \wedge \neg y^{\top\bot})^{-\sigma} = (\neg x \vee y) \wedge \neg y$*. Furthermore,* $(\phi^{\tau})^{-\tau} = (a)^{-\tau} = a$*.*

Finally, we specify the semantics of a QBF in terms of universal and existential expansion on which expansion-based QBF solving is founded.

Lemma 2. *Let* $\Phi = \Pi.\phi$ *be a QBF with universal variables* U*. There is a set of assignments* $A \subseteq \Sigma_U$ *with* $\bigwedge_{\alpha \in A} \phi^{\alpha}$ *is unsatisfiable if and only if* Φ *is false.*

The lemma above has a dual version for true QBFs. This duality plays a prominent role in our novel solving algorithm.

Lemma 3. *Let* $\Phi = \Pi.\phi$ *be a QBF with existential variables* E*. There is a set of assignments* $S \subseteq \Sigma_E$ *with* $\bigvee_{\sigma \in S} \phi^{\sigma}$ *is valid if and only if* Φ *is true.*

V. A Non-Recursive Algorithm for Expansion-Based QBF Solving

The pseudo-code in Figure 1 summarises the basic idea of our novel approach for solving the QBF $\Pi.\phi$ with universal variables U and existential variables E.

First, an arbitrary assignment α_0 for the universal variables is selected in Line 1. The instantiation ϕ^{α_0} is handed over to a SAT solver. If ϕ^{α_0} is unsatisfiable, then $\Pi.\phi$ is false and the algorithm returns. Otherwise, $\tau\colon E^{\alpha_0} \to \{\top, \bot\}$ is a satisfying assignment of ϕ^{α_0}. Let σ_1 denote the assignment $\tau^{-\alpha_0}$. Then $\alpha_0\sigma_1$ is a satisfying assignment of ϕ.

```
input : QBF $\Pi.\phi$ with universal variables $U$ and
        existential variables $E$
output: truth value of $\Pi.\phi$
1  $A_0 := \{\alpha_0\}$, where $\alpha_0\colon U \to \{\top, \bot\}$ is an arbitrary
   assignment
2  $S_0 := \emptyset$
3  $i := 1$
4  while true do
5      $(isUnsat, \tau) :=$ SAT($\bigwedge_{\alpha \in A_{i-1}} \phi^{\alpha}$)
6      if isUnsat then return false;
7      $S_i := S_{i-1} \cup \{(\tau|_{E^{\alpha}})^{-\alpha} \mid \alpha \in A_{i-1}\}$

8      $(isUnsat, \rho) :=$ SAT($\bigwedge_{\sigma \in S_i} \neg\phi^{\sigma}$)
9      if isUnsat then return true;
10     $A_i := A_{i-1} \cup \{(\rho|_{U^{\sigma}})^{-\sigma} \mid \sigma \in S_i\}$

11     $i$++
12 end
```

Figure 1: Non-Recursive Expansion-Based Algorithm

Next, the propositional formula $\neg\phi^{\sigma_1}$ is handed over to a SAT solver for checking the validity of ϕ^{σ_1}. If $\neg\phi^{\sigma_1}$ is unsatisfiable, then $\Pi.\phi$ is true and the algorithm returns. If $\neg\phi^{\sigma_1}$ is satisfiable, then $\rho\colon U^{\sigma_1} \to \{\top, \bot\}$ is a satisfying assignment of $\neg\phi^{\sigma_1}$. Let α_1 denote the assignment $\rho^{-\sigma_1}$. Then $\alpha_1\sigma_1$ is a satisfying assignment for $\neg\phi$. The following lemma shows that α_0 and α_1 are different.

Lemma 4. *Let* $\Pi.\phi$ *be a QBF with universal variables* U *and existential variables* E*. Further, let* $\alpha\colon U \to \{\top, \bot\}$ *be an assignment such that the instantiation* ϕ^{α} *is satisfiable and has the satisfying assignment* $\tau\colon E^{\alpha} \to \{\top, \bot\}$*. Let* $\sigma\colon E \to \{\top, \bot\}$ *with* $\sigma = \tau^{-\alpha}$*. Then* α *falsifies* $(\neg\phi^{\sigma})^{-\sigma}$*.*

Proof. Since ϕ^{α} is satisfied by τ, ϕ is satisfied by the composite assignment $\alpha\tau^{-\alpha} = \alpha\sigma$, and therefore $\neg\phi$ is falsified by $\alpha\sigma$. Then α falsifies $\sigma(\neg\phi)$. According to Lemma 1 $\sigma(\neg\phi)$ is equivalent to $(\neg\phi^{\sigma})^{-\sigma}$. Then α also falsifies $(\neg\phi^{\sigma})^{-\sigma}$. $\square$

In the next round of the algorithm, the propositional formula $\phi^{\alpha_0} \wedge \phi^{\alpha_1}$ is handed over to a SAT solver. If this formula is unsatisfiable, $\Pi.\phi$ is false and the algorithm returns. Otherwise, it is satisfiable under some assignment $\tau\colon E^{\alpha_0} \cup E^{\alpha_1} \to \{\top, \bot\}$, then at least one new assignment $\sigma_2\colon E \to \{\top, \bot\}$ with $\sigma_2 \neq \sigma_1$ can be extracted from $\tau|_{E^{\alpha_i}}$ with $0 \leq i \leq 1$. This assignment is then used for obtaining a new propositional formula $\phi^{\sigma_1} \vee \phi^{\sigma_2}$. To show the validity of this formula, its negation is passed to a SAT solver. If this formula is unsatisfiable, $\Pi.\phi$ is true and the algorithm returns. Otherwise, it is satisfiable under the assignment $\rho\colon U^{\sigma_1} \cup U^{\sigma_2} \to \{\top, \bot\}$. A new assignment $\alpha_2\colon U \to \{\top, \bot\}$ with $\alpha_2 \neq \alpha_1 \neq \alpha_0$ is obtained from $\rho|_{A^{\sigma_i}}$ with $1 \leq i \leq 2$. This assignment is then used in the next round of the algorithm. In this way, the propositional formulas $\bigwedge_{\alpha \in \Sigma_U} \phi^{\alpha}$ and $\bigvee_{\sigma \in \Sigma_E} \phi^{\sigma}$ are generated. If $\bigwedge_{\alpha \in A} \phi^{\alpha}$ is unsatisfiable for some $A \subseteq \Sigma_U$, by Lemma 2

$\Pi.\phi$ is false. Dually, if $\bigvee_{\sigma \in S} \phi^{\sigma}$ is valid for some $S \subseteq \Sigma_E$, by Lemma 3 $\Pi.\phi$ is true. The algorithm iteratively extends the sets A and S by adding parts of satisfying assignments of ϕ to S and parts of falsifying assignments to A. In particular, A is extended by assignments of the universal variables and S is extended by assignments of the existential variables. The order in which assignments are considered depends on the used SAT solver.

Example 4. *We show how to solve the QBF $\forall a \exists x \forall b \exists y.\phi$ with $E = \{x, y\}$, $U = \{a, b\}$, and $\phi = ((a \vee x \vee y) \wedge (\neg a \vee \neg x \vee y) \wedge (b \vee \neg y))$ with the algorithm presented above. This formula can be solved in two iterations:*

Init: We start with some random assignment $\alpha_0 \colon U \to \{\top, \bot\}$, for example with $\alpha_0(a) = \top$ and $\alpha_0(b) = \bot$.

Iteration 1: The formula $\phi^{\alpha_0} = (\neg x^{\top} \vee y^{\top\bot}) \wedge \neg y^{\top\bot}$ is passed to a SAT solver and found satisfiable under the assignment $\tau \colon E^{\alpha_0} \to \{\top, \bot\}$ with $\tau(x^{\top}) = \bot$ and $\tau(y^{\top\bot}) = \bot$. By removing the variable annotations we get assignment $\sigma_1 = (\tau|_{E^{\alpha_0}})^{-\alpha_0}$, where $\sigma_1 \colon E \to \{\top, \bot\}$ with $\sigma_1(x) = \bot$ and $\sigma_1(y) = \bot$. Based on this assignment we obtain $\phi^{\sigma_1} = a$. The formula $\neg\phi^{\sigma_1}$ is passed to a SAT solver. It is satisfiable and has the satisfying assignment $\rho \colon U^{\sigma_1} \to \{\top, \bot\}$ with $\rho(a) = \bot$ and $\rho(b^{\bot}) = \top$, which we then reduce to $\alpha_1 = (\rho|_{U^{\sigma_1}})^{-\sigma_1}$ where $\alpha_1 \colon U \to \{\top, \bot\}$ with $\alpha_1(a) = \bot$ and $\alpha_1(b) = \top$.

Iteration 2: The formula $\phi^{\alpha_0} \wedge \phi^{\alpha_1} = (\neg x^{\top} \vee y^{\top\bot}) \wedge \neg y^{\top\bot} \wedge (x^{\bot} \vee y^{\bot\top})$ is passed to a SAT solver in the second iteration. It is satisfiable and one satisfying assignment is $\tau \colon E^{\alpha_0} \cup E^{\alpha_1} \to \{\top, \bot\}$ with $\tau(x^{\top}) = \bot, \tau(x^{\bot}) = \top, \tau(y^{\top\bot}) = \bot, \tau(y^{\bot\top}) = \bot$. From τ, we can extract the assignment $\sigma_2 = (\tau|_{E^{\alpha_1}})^{-\alpha_1}$ where $\sigma_2 \colon E \to \{\top, \bot\}$ with $\sigma_2(x) = \top$ and $\sigma_2(y) = \bot$. Note that for any choice of τ, $\sigma_2 \neq \sigma_1$. Next, we construct $\phi^{\sigma_1} \vee \phi^{\sigma_2} = a \vee \neg a$. This formula is a tautology, so its negation that is passed to a SAT solver is unsatisfiable, hence $\Pi.\phi$ is true.

The soundness of our algorithm immediately follows from Lemmas 2 and 3: the algorithm returns false (true) if, in some iteration i, it finds that the current partial expansion $\bigwedge_{\alpha \in A_{i-1}} \phi^{\alpha}$ (respectively $\bigwedge_{\sigma \in S_i} \neg\phi^{\sigma}$) is unsatisfiable.

Theorem 1. *The algorithm shown in Figure 1 is sound.*

For showing that the algorithm also terminates, we argue that sets A_i and S_i increase in iteration $i+1$. To this end, we have to relate the variables of the QBF, the annotated variables as well as their assignments. Before we give the proof, we first consider another example in which we illustrate how the different assignments are related.

Example 5. *We show one possible run of the algorithm presented above for the QBF $\Phi := \forall a \exists x \forall b \exists y.\phi$ with*

$$\phi := (a \wedge b \wedge \neg x \wedge \neg y) \vee (\neg a \wedge x \wedge (b \leftrightarrow y))$$

and how it iteratively generates the sets Σ_U and Σ_E. Figure 2 shows the expansion trees that are implicitly built during the search. An expansion tree relates the variables of the partial expansion of Φ constructed from A_i (left column) and S_i (right column). Solid edges indicate that the variable on the top has been set by an assignment from A_i or S_i, and dotted edges indicate that the variable has to be assigned a value by the SAT solver. The order of the (annotated) variables in the expansion tree respects the order of the (original) variables in the prefix.

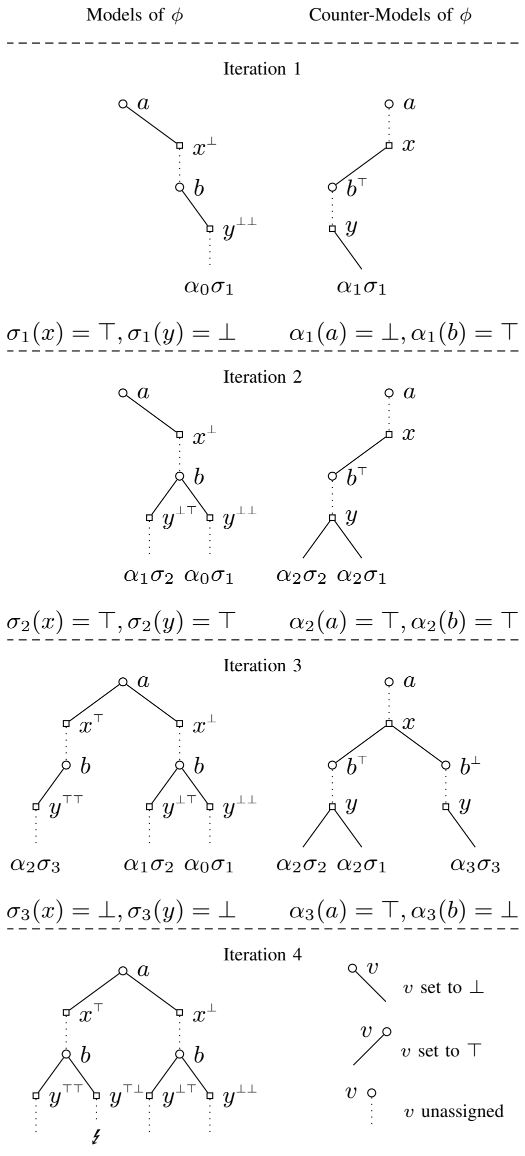


Figure 2: Expansion trees relating the assignments found during solving the QBF $\forall a \exists x \forall b \exists y.\phi$ in Example 5, with initial assignment $\alpha_0(a) = \bot, \alpha_0(b) = \bot$. The assignments shown in the leaves of the trees satisfy (left trees) or falsify (right trees) ϕ.

Init: For the initialisation of A_0, an arbitrary assignment $\alpha_0 \colon U \to \{\top, \bot\}$ is chosen. Let $\alpha_0(a) = \bot$ and $\alpha_0(b) = \bot$.

Iteration 1: $\phi^{\alpha_0} := x^{\bot} \wedge \neg y^{\bot\bot}$ is satisfiable. Assignment $\sigma_1 \colon E \to \{\top, \bot\}$, with $\sigma_1(x) = \top$ and $\sigma_1(y) = \bot$, is extracted from model $\tau \colon E^{\alpha_1} \to \{\top, \bot\}$ and added to S_1. Now $\phi^{\sigma_1} := \neg a \wedge \neg b^{\top}$ is checked for validity. Assignment $\alpha_1 \colon U \to \{\top, \bot\}$, with $\alpha_1(a) = \bot$ and $\alpha_1(b) = \top$, obtained *from counter-example $\rho\colon U^{\sigma_1} \to \{\top, \bot\}$ is added to A_1.*

Iteration 2: Next, $\phi^{\alpha_0} \wedge \phi^{\alpha_1}$ with $\phi^{\alpha_1} := x^{\bot} \wedge y^{\bot\top}$ is checked. From model $\tau\colon E^{\alpha_0} \cup E^{\alpha_1} \to \{\top, \bot\}$, again σ_1 can be extracted for ϕ^{α_0}. For ϕ^{α_1} a new assignment σ_2 which is not in S_1 is found and added to S_2. In particular, we get $\sigma_2\colon E \to \{\top, \bot\}$ with $\sigma_2(x) = \top$ and $\sigma_2(y) = \top$. When the validity of $\phi^{\sigma_1} \vee \phi^{\sigma_2}$ with $\phi^{\sigma_2} := \neg a \wedge b^{\top}$ is checked, we get a counter-example $\rho\colon U^{\sigma_1} \cup U^{\sigma_2} \to \{\top, \bot\}$, from which $\alpha_2\colon U \to \{\top, \bot\}$, with $\alpha_2(a) = \top$ and $\alpha_2(b) = \top$, can be extracted. Assignment α_2 is added to A_2 leading to a new path in the left expansion tree (Iteration 3 in Figure 2).

Iteration 3: Next, $\phi^{\alpha_0} \wedge \phi^{\alpha_1} \wedge \phi^{\alpha_2}$ with $\phi^{\alpha_2} := \neg x^{\top} \wedge \neg y^{\top\top}$ is checked. From model $\tau\colon E^{\alpha_0} \cup E^{\alpha_1} \cup E^{\alpha_2} \to \{\top, \bot\}$, $\sigma_3\colon E \to \{\top, \bot\}$ is extracted, satisfying ϕ^{α_2}. This assignment is different from both σ_1 and σ_2: $\sigma_3(x) = \bot$ and $\sigma_3(y) = \bot$. This again results in a new branch of the expansion tree (see left expansion tree of Iteration 4 in Figure 2). The resulting formula $\phi^{\sigma_1} \vee \phi^{\sigma_2} \vee \phi^{\sigma_3}$ with $\phi^{\sigma_3} := a \wedge b^{\bot}$ is not valid, and from the counter-example $\rho\colon U^{\sigma_1} \cup U^{\sigma_2} \cup U^{\sigma_3} \to \{\top, \bot\}$ we get $\alpha_3\colon U \to \{\top, \bot\}$ with $\alpha_3(a) = \top$ and $\alpha_3(b) = \bot$.

Iteration 4: Finally, the full expansion $\phi^{\alpha_0} \wedge \phi^{\alpha_1} \wedge \phi^{\alpha_2} \wedge \phi^{\alpha_3}$ with $\phi^{\alpha_3} := \bot$ is not satisfiable, meaning that the original formula $\forall a \exists x \forall b \exists y.\phi$ is false.

In the example above we saw that new assignments are generated in each iteration because A_i and S_i build models and counter-models of ϕ. The following definition formalises the relationship between A_i and S_i.

Definition 2. *Let $\Pi.\phi$ be a QBF over universally quantified variables U and existentially quantified variables E. Further, let $A \subseteq \{\alpha \mid \alpha\colon U \mapsto \{\top, \bot\}\Delta\}$ and $S \subseteq \{\sigma \mid \sigma\colon E \mapsto \{\top, \bot\}\Delta\}$. If for every assignment $\sigma \in S$, there exists an assignment $\alpha \in A$ such that $\alpha\sigma(\neg\phi)$ is true, then we say that A* completes *S. If for every assignment $\alpha \in A$, there exists an assignment $\sigma \in S$ such that $\alpha\sigma(\phi)$ is true, then we say that S* completes *A.*

We now show that S_i completes A_{i-1} and A_i completes S_i if the algorithm does not terminate in iteration i because of the unsatisfiability of the respective expansion.

Lemma 5. *Let $\Pi.\phi$ be a QBF over universally quantified variables U and existentially quantified variables E. Further, let A_{i-1} and A_i with $A_{i-1} \subseteq A_i$ be two sets of full assignments to the universal variables and let S_i be a set of full assignments to the existential variables obtained by iteration i during an execution of the algorithm shown in Figure 1.*

(1) If $\bigwedge_{\alpha \in A_{i-1}} \phi^{\alpha}$ is satisfiable, then S_i completes A_{i-1}, i.e., for every $\mu \in A_{i-1}$, there is an assignment $\nu \in S_i$ such that $\mu\nu(\phi)$ is true.

(2) If $\bigwedge_{\sigma \in S_i} \neg\phi^{\sigma}$ is satisfiable, then A_i completes S_i, i.e., for every $\nu \in S_i$, there is an assignment $\mu \in A_i$ such that $\nu\mu(\neg\phi)$ is true.

Proof. By contradiction. For (1), assume there is an assignment $\mu \in A_{i-1}$ such that there is no assignment $\nu \in S_i$ with $\mu\nu(\phi)$ is true. By assumption $\bigwedge_{\alpha \in A_{i-1}} \phi^{\alpha}$ is satisfiable, so there is a satisfying assignment τ with $\tau|_{E^{\mu}}(\phi^{\mu})$ is true. Then also $\mu(\tau|_{E^{\mu}})^{-\mu}(\phi)$ is true. But $(\tau|_{E^{\mu}})^{-\mu} \in S_i$. For (2), assume that there is an assignment $\mu \in S_i$ such that there is no $\nu \in A_i$ with $\mu\nu(\neg\phi)$ is true. The rest of the argument is similar as in (1). ☐

Next, we show that the addition of new assignments A' to a set A of universal assignments forces a set S of existential assignments to increase if some completion criteria hold.

Lemma 6. *Let $\Phi = \Pi.\phi$ be a QBF over universally quantified variables U and existentially quantified variables E. Further, let $A \cup A'$ be a set of universal assignments such that $A \cap A' = \emptyset$ and $A' \neq \emptyset$. Let S be a set of existential assignments and assume that $\bigwedge_{\sigma \in S} \neg\phi^{\sigma}$ has the satisfying assignment ρ, $A' \subseteq \{(\rho|_{U^{\sigma}})^{-\sigma} \mid \sigma \in S\}$.*

If S completes A, and $A \cup A'$ completes S, and $\bigwedge_{\alpha \in A \cup A'} \phi^{\alpha}$ evaluates to true under assignment τ, then there exists an assignment $\nu \in \{(\tau|_{E^{\alpha}})^{-\alpha} \mid \alpha \in A \cup A'\}$ with $\nu \notin S$.

Proof. By induction over the number of variables in Π.

Base Case. Assume that Φ has only one variable, i.e., $\Pi = Qx$. Note that $|A'| = 1$ because x is outermost in the prefix and A' is obtained from sub-assignments of ρ. If $Q = \forall$, then the elements of A are full assignments of ϕ, and S is either empty, or it contains the empty assignment $\omega\colon \emptyset \mapsto \{\top, \bot\}$. Let $A' = \{\mu\}$. If S is empty, so is A (because S has to complete A). If τ is a satisfying assignment of ϕ^{μ}, then $\nu = \tau = \omega$ is the empty assignment and $\nu \notin S$. Otherwise, $\omega \in S$. If there is an assignment $\alpha \in A$, then $\phi^{\alpha} \wedge \phi^{\mu}$ is a full expansion of Φ. If this full expansion is true, then $\neg\phi$ is unsatisfiable. Otherwise, $\phi^{\alpha} \wedge \phi^{\mu}$ is unsatisfiable. In both cases, the necessary preconditions for the lemma are not fulfilled. If $A = \emptyset$, then $\mu\omega(\neg\phi)$ is true. Then ϕ^{μ} is unsatisfiable, again violating a precondition. If $Q = \exists$, then $\mu = \omega$ and $A = \emptyset$. If $S = \emptyset$ and $\phi^{\omega} = \phi$ has the satisfying assignment τ, then $\nu = \tau$ and $\nu \notin S$. Otherwise, if there is an assignment $\sigma \in S$, then $\omega\sigma(\neg\phi)$ is true, because $A \cup \{\mu\} = \{\omega\}$ completes S. Hence, if assignment τ satisfies ϕ^{μ}, then $\nu = \tau$, so $\nu \notin S$.

Induction Step. Assume the lemma holds for QBFs with n variables. We show that it also holds for QBFs with $n+1$ variables. Let $\Phi = Qx\Pi.\phi$ be a QBF over existential variables E and universal variables U with $\Pi = Q_1x_1 \ldots Q_nx_n$ and $A \cup A'$ and S be as required (S completes A, $A \cup A'$ completes S, $\bigwedge_{\alpha \in A \cup A'} \phi^{\alpha}$ has a satisfying assignment τ, and $\bigwedge_{\sigma \in S} \neg\phi^{\sigma}$ has a satisfying assignment ρ from which A' is obtained).

If $Q = \forall$, then all assignments $\alpha \in A'$ assign the same value t to x, i.e., $\alpha(x) = t$, because these assignments are extracted from assignment ρ and since x is the outermost variable of the prefix of Φ, $\rho(x) = t$. Further, let $A^t = \{\alpha \in A \mid \alpha(x) = t\}$. It is easy to argue that for $\Pi.\phi[x \leftarrow t]$ together with the assignment sets $A^t \cup A'$ and S the induction hypothesis applies, i.e., there is an assignment $\nu \notin S$ with $\nu \in \{(\tau'|_{E^{\alpha}})^{-\alpha} \mid \alpha \in A^t \cup A'\}$ where τ' is the part of τ that satisfies $\bigwedge_{\alpha \in A^t \cup A'} (\phi[x \leftarrow t])^{\alpha}$. Obviously, $\nu \in \{(\tau|_{E^{\alpha}})^{-\alpha} \mid \alpha \in A \cup A'\}$.

If $Q = \exists$, assume that $\tau(x) = t$. Let $\{\sigma \in S \mid \sigma(x) = t\} \subseteq S^t \subseteq S$, and let $A^t \subseteq A$ such that the induction hypothesis applies to $\Pi.\phi[x \leftarrow t]$, $A^t \cup A'$, and S^t. Let τ^t be those sub-assignments of τ that satisfy $\bigwedge_{\alpha \in A^t} \phi^\alpha$. Then there is an assignment ν that can be extracted from τ^t with $\nu \notin S^t$. Since $\nu(x) = t$, $\nu \notin S$. This concludes the proof. ☐

This property also holds in the other direction, i.e., adding a set S' of new assignments to S will force the set A to increase.

Lemma 7. *Let $\Phi = \Pi.\phi$ be a QBF over universally quantified variables U and existentially quantified variables E. Further, let $S \cup S'$ be a set of existential assignments such that $S \cap S' = \emptyset$, $S' \neq \emptyset$, let A be a set of universal assignments, $\bigwedge_{\alpha \in A} \phi^\alpha$ has the satisfying assignment τ, $S' \subseteq \{(\tau|_{E^\alpha})^{-\alpha} \mid \alpha \in A\}$.*

If A completes S and $S \cup S'$ completes A and $\bigwedge_{\sigma \in S \cup S'} \neg\phi^\sigma$ evaluates to true under assignment ρ, then there exists an assignment $\nu \in \{(\rho|_{U^\sigma})^{-\sigma} \mid \sigma \in S \cup S'\}$ with $\nu \notin A$.

Proof. The proof is analogous to the proof of Lemma 6. ☐

Now that we have identified the relations between the sets of universal and existential assignments, we use them to show that the algorithm from Figure 1 terminates.

Theorem 2. *The algorithm shown in Figure 1 terminates for any QBF $\Phi = \Pi.\phi$.*

Proof. By induction over the number of iterations i, we argue that sets $A_{i-1} \subset A_i$ and $S_{i-1} \subset S_i$.

Base Case. Let $i = 1$ and $A_0 = \{\alpha_0\}$. $S_0 \subset S_1$, because $S_0 = \emptyset$ and $\sigma_1 \in S_1$ is a satisfying assignment of ϕ^{α_0} (if ϕ^{α_0} is unsatisfiable, the algorithm terminates). $A_0 \subset A_1$ directly follows from Lemma 4.

Induction Step. For $i > 1$, we argue that $S_i \subset S_{i+1}$. By induction hypothesis the theorem holds for iteration i, i.e., $A_i = A_{i-1} \cup A'$ with $A_{i-1} \cap A' = \emptyset$ and $A' \neq \emptyset$ and $S_i = S_{i-1} \cup S'$ with $S_{i-1} \cap S' = \emptyset$ and $S' \neq \emptyset$. Because of Lemma 5, S_i completes A_{i-1}, and A_i completes S_i. Furthermore, if $\bigwedge_{\sigma \in S_i} \neg\phi^\sigma$ is satisfiable under some assignment ρ (otherwise the algorithm would terminate), by construction $A' \subseteq \{(\rho|_{U^\sigma})^{-\sigma} \mid \sigma \in S_i\}$. Hence, Lemma 6 applies and if $\bigwedge_{\alpha \in A_i} \phi^\alpha$ is satisfiable under some assignment τ (otherwise the algorithm would immediately terminate), then there is an assignment $\nu \in \{(\tau|_{E^\alpha})^{-\alpha} \mid \alpha \in A_i\}$ with $\nu \notin S_i$.

The argument for $A_i \subset A_{i+1}$ is similar and uses the property shown in Lemma 7. ☐

Note that the algorithm presented above does not make any assumptions on the formula structure, i.e., for a QBF $\Pi.\phi$ it is not required that ϕ is in conjunctive normal form. Without any modification, our algorithm also works on formulas in PCNF—as SAT solvers typically process formulas in CNF only, we focus on this representation for the rest of the paper.

We conclude this section by arguing that the $\forall$Exp+Res [5], [28], [29] calculus yields the theoretical foundation of our algorithm for refuting a formula $\Pi.\phi$ in PCNF with universal variables U. The $\forall$Exp+Res calculus consists of two rules, the axiom rule

$$\frac{}{C^{\alpha}}$$

where C is a clause of ϕ and $\alpha\colon U \to \{\top, \bot\}$ is a universal assignment as well as the resolution rule

$$\frac{C_1 \vee x^\omega \qquad C_2 \vee \neg x^\omega}{C_1 \vee C_2}$$

A derivation in $\forall$Exp+Res is a sequence of clauses where each clause is either obtained by the axiom or derived from previous clauses by the application of the resolution rule. A refutation of a PCNF $\Pi.\phi$ is a derivation of the empty clause.

The application of the axiom instantiates the universal variables of one clause of ϕ. If enough of these instantiations can be found in order to derive the empty clause by the application of the resolution rule, the QBF $\Pi.\phi$ is false. Our algorithm in Figure 1 does not instantiate individual clauses, but all clauses of ϕ at once with a particular assignment of the universal variables. Hence, when the SAT solver finds $\psi_\forall = \bigwedge_{\alpha \in A_i} \phi^\alpha$ unsatisfiable for some A_i, not necessarily all clauses of $\psi_\forall$ are required to derive the empty clause via resolution, but only the minimal unsatisfiable core of $\psi_\forall$, i.e., a subset of the clauses such that the removal of any clause would make this formula satisfiable.

Proposition 1. *Let $\Pi.\phi$ be a false QBF. Further, let $\psi_\forall = \bigwedge_{\alpha \in A_i} \phi^\alpha$ be obtained by the application of the algorithm in Figure 1. Further, let $\psi'_\forall$ be an unsatisfiable core of $\psi_\forall$. Then there is a $\forall$Exp+Res refutation such that all clauses that are introduced by the axiom rule occur in $\psi'_\forall$.*

VI. IMPLEMENTATION

The algorithm described in Section V is realised in the solver Ijtihad[2] The most recent version of Ijtihad is available at

https://extgit.iaik.tugraz.at/scos/ijtihad

The solver is implemented in C++ and currently processes formulas in PCNF available in the QDIMACS format. For accessing SAT solvers, Ijtihad uses the IPASIR interface [4], which makes changing the SAT solver very easy. The SAT solver used in all of our experiments is Glucose [1]. Although the base implementation does reasonably well, we have realised various optimizations to make Ijtihad even more viable in practice. Some of them are discussed in the following.

For solving a QBF $\Pi.\phi$, the basic algorithm shown in Figure 1 adds instantiations of ϕ to $\psi_\forall = \bigwedge_{\alpha \in A_{i-1}} \phi^\alpha$ and $\psi_\exists = \bigwedge_{\sigma \in S_i} \neg\phi^\sigma$ in each iteration i until the formula is decided. The calls to the SAT solver in Line 5 and Line 8 are done incrementally, i.e., we create two instances of the SAT solver and provide them with the clauses stemming from new instantiations of ϕ at each iteration. For simplicity, we omit indices of sets A and S and refer to an arbitrary iteration of the execution of the algorithm in the following discussion.

Figure 5 relates set sizes of A and S as well as the accumulated time that one SAT solver needs to solve $\psi_\forall$

---

[2] The name Ijtihad refers to the effort of solving cases in Islamic law (for details see https://en.wikipedia.org/wiki/Ijtihad).

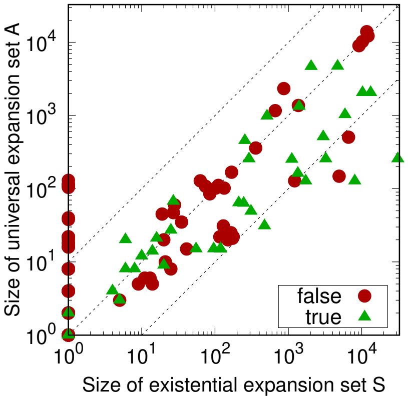


Figure 3: Sizes of sets S and A

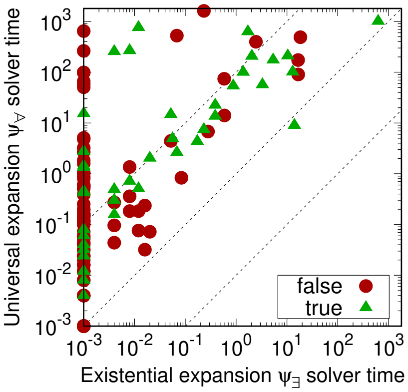


Figure 4: Time used for solving $\psi_\exists$ and $\psi_\forall$

Figure 5: Set sizes and time consumed during SAT calls for solved instances from QBFEVAL’17 preprocessed by Bloqqer

with the time the other SAT solver needs to solve $\psi_\exists$ for the formulas of the PCNF track of QBFEVAL’17 (preprocessed with Bloqqer [6]). We also distinguish between true and false formulas. In Figure 3 we see that for true formulas, set S tends to be larger than A, while for false instances the picture is less clear. Figure 4 shows the overall time needed for solving $\psi_\forall$ (y-axis) and $\psi_\exists$ (x-axis). In almost all cases, the solver that handles $\psi_\forall$ needs more time than the solver that handles $\psi_\exists$. This may be founded on the observation that many QBFs have considerably more existential variables than universal variables [37], hence the instantiations added to $\psi_\forall$ are much larger than the instantiations added to $\psi_\exists$.

In Line 1 of Figure 1, the set of universal assignments A is initialised with *one* arbitrary assignment α_0. Obviously, the set A may also be initialized with multiple assignments. In our current implementation, we initialize A with the assignments that set the variables of one universal quantifier block to $\bot$ and the variables of all other universal quantifier blocks to $\top$. The impact of various initialization heuristics remains to be investigated in future work.

In Line 7 and Line 10 our algorithm increases the size of S and A in each iteration of the main loop, as argued in Theorem 2. In the worst case, this leads to an exponential increase in space consumption. Although we detect shared clauses among the instantiations, that alone is not enough to significantly reduce the space consumption. However, some of the assignments found in an earlier iteration could become obsolete after better assignments were found. It is therefore beneficial to empty either S or A and then reconstruct them from $\psi_\forall$ and $\psi_\exists$, similarly to what is done in Line 7 and Line 10. We evaluated several heuristics for scheduling these set resets, and we found that resetting periodically and close to the memory limit works best. The regular resetting of one set has a similar effect as restarts in SAT solvers, and we observed a considerable improvement in performance, especially in terms of memory consumption. Our implementation periodically resets the set A, since experiments indicate that the resulting formula $\psi_\forall$ is much harder to solve than $\psi_\exists$ as seen in Figure 4. Besides the aforementioned imbalance between universal and existential variables, it is also likely due to the structure of $\psi_\exists$ which is a conjunction of formulas in disjunctive normal form. Note that this reset of A does not affect the termination argument presented in Theorem 2, since the sets A and S still complete each other.

Finally, we extended the presented approach with orthogonal reasoning techniques like QCDCL [21] for exploiting the different strengths of $\forall$Exp+Res and Q-resolution, yielding a hybrid solver that smoothly integrates both solving paradigms. To this end, we implemented the prototypical solver called Heretic which pursues the following idea: The main loop of the algorithm shown in Figure 1 (Lines 4-12) is extended in a sequential portfolio style such that a QCDCL solver is periodically called. After each call, all clauses that were learned through QCDCL are added to $\Pi.\Phi$, making them available in further iterations. These new clauses potentially exclude assignments that would otherwise be possible and that could result in more iterations of the main loop.

The solver Heretic extends Ijtihad by additional invocations of the QCDCL solver DepQBF [36]. About every 30 seconds, DepQBF is called and run for about 30 seconds. The learnt clauses are obtained via the API of DepQBF. Leveraging learned cubes is subject to future work.

VII. EVALUATION

We evaluate non-recursive expansion as implemented in our solvers Ijtihad and its hybrid variant Heretic on the benchmarks from the PCNF track of the QBFEVAL’17 competition. All experiments were carried out on a cluster of Intel Xeon CPUs (E5-2650v4, 2.20 GHz) running Ubuntu 16.04.1 with a CPU time limit of 1800 seconds and a memory limit of 7 GB. We considered the following top-performing solvers from QBFEVAL’17: Qute [38], Rev-Qfun [25], RAReQS [26], CAQE [39], [43], DynQBF [12], GhostQ [26], [34], DepQBF [36], QESTO [30], and QSTS [9], [10]. Our experiments are based on original benchmarks without preprocessing and benchmarks preprocessed using Bloqqer [6], [23] with a timeout of two hours.[3] We included the 76 formulas already

---

[3] We refer to an online appendix [7] for additional experiments.

Solver	*S*	$\bot$	$\top$	*Time*
Rev-Qfun	220	145	75	572K
GhostQ	194	120	74	617K
CAQE	170	128	42	656K
RAReQS	167	133	34	660K
DepQBF	167	121	46	666K
Heretic	163	133	30	664K
QSTS	152	116	36	687K
Ijtihad	150	127	23	684K
Qute	130	91	39	720K
QESTO	109	86	23	761K
DynQBF	72	38	34	826K

TABLE I: Original instances.

Solver	*S*	$\bot$	$\top$	*Time*
RAReQS	256	180	76	508K
CAQE	251	168	83	522K
Heretic	245	172	73	522K
Rev-Qfun	219	148	71	568K
Ijtihad	217	156	61	564K
QSTS	208	151	57	585K
QESTO	196	137	59	610K
DepQBF	183	117	66	633K
GhostQ	163	100	63	670K
Qute	154	109	45	682K
DynQBF	151	95	56	684K

TABLE II: Preprocessing by Bloqqer.

Solver	*S*	$\bot$	$\top$	*Time*
Heretic	92	81	11	195K
DepQBF	89	74	15	205K
CAQE	88	73	15	204K
RAReQS	82	78	4	211K
QSTS	81	69	12	216K
Ijtihad	80	73	7	212K
Rev-Qfun	78	75	3	224K
Qute	70	60	10	236K
QESTO	51	44	7	269K
GhostQ	45	39	6	279K
DynQBF	15	13	2	332K

TABLE III: 197 original instances with four or more quantifier blocks.

solved by Bloqqer in both benchmark sets.

Tables I and II show the total numbers of solved instances (*S*), solved unsatisfiable ($\bot$) and satisfiable ones ($\top$), and total CPU time including timeouts. In the following, we focus on a comparison of our solvers Ijtihad and Heretic with RAReQS (cf. Figure 6). Unlike our solvers, RAReQS is based on a recursive implementation of expansion.

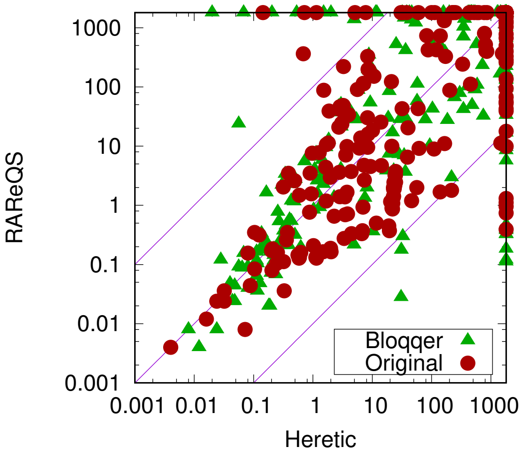


Figure 6: Scatter plots of the run times of Heretic and RAReQS on original instances (related to Table I) and on instances preprocessed by Bloqqer (related to Table II).

In general, preprocessing has a considerable impact on the number of solved instances. The difference in solved instances between Ijtihad and RAReQS is 17 on original instances (Table I), and becomes larger on preprocessed instances (Table II).

Notably Heretic, despite its simple design, significantly outperforms Ijtihad on the two benchmark sets. Moreover, Heretic is ranked third on preprocessed instances (Table II) and thus is on par with state-of-the-art solvers. On the two benchmark sets, the gap in solved instances between RAReQS and Heretic is considerably smaller than the one between RAReQS and Ijtihad.

We report on memory consumption of expansion-based solvers. While RAReQS, Ijtihad, and Heretic run out of memory on 42, 61, and 39 original instances (Table I), respectively, these numbers drop to 17, 41, and 24, respectively, with preprocessing (Table II). The average memory footprint is 1718 MB, 1836 MB, and 1842 MB for RAReQS, Ijtihad, and Heretic, respectively, and 1056 MB, 1311 MB, and 1187 MB on preprocessed instances. Interestingly, Ijtihad has a smaller median memory footprint than RAReQS without (792 MB vs. 802 MB) and with preprocessing (286 MB vs. 364 MB).

The strength of Heretic becomes obvious for formulas that have four or more quantifier blocks (i.e., three or more quantifier alternations), cf. [37]. As shown in Table III, Heretic outperforms all other solvers on these instances. We made a similar observation on preprocessed formulas.

Moreover, Heretic solves only four original instances less than DepQBF (Table I), and outperforms DepQBF on preprocessed instances (Table II). These results indicate the potential of combining the orthogonal proof systems ∀Exp+Res as implemented in Ijtihad and Q-resolution as implemented in DepQBF in a hybrid solver such as Heretic.

	R	*vs.*	*I*	*R*	*vs.*	*H*	*I*	*vs.*	*H*	*D*	*vs.*	*H*
	<	=	>	<	=	>	<	=	>	<	=	>
N	27	140	10	26	141	22	5	145	18	65	102	61
B	56	200	17	38	218	27	7	210	35	17	166	79

TABLE IV: Pairwise comparison of RAReQS (R), Ijtihad (I), Heretic (H), and DepQBF (D) by instances without (N) and with preprocessing by Bloqqer (B) that were solved by only one solver of the considered pair (<, >) or by both (=).

Although RAReQS outperforms both Ijtihad and Heretic on the two given benchmark sets (Tables I and II), RAReQS failed to solve certain instances that were solved by Ijtihad and Heretic. Table IV shows related statistics. E.g., on preprocessed instances (row “B”), 218 instances were solved by both RAReQS and Heretic (column “*R vs. H*), 38 only by RAReQS, and 27 only by Heretic. Summing up these numbers yields a total of 283 solved instances (more than any individual solver on preprocessed instances in Table II) that could have been solved by a *hypothetical solver* combining RAReQS and Heretic. This observation underlines the strength of expansion in general and, in particular, of the hybrid approach implemented in Heretic. Heretic solved a significant amount of instances not solved by RAReQS, it clearly outperformed Ijtihad on all benchmarks (column "*I vs. H*") and DepQBF on preprocessed ones ("*D vs. H*").

VIII. CONCLUSION

We presented a novel non-recursive algorithm for expansion-based QBF solving that uses only two SAT solvers for incrementally refining the propositional abstraction and the negated propositional abstraction of a QBF. We gave a concise proof of termination and soundness and demonstrated with several experiments that our prototype compares well with the state of the art. In addition to non-recursive expansion, we also studied the impact of combining Q-resolution and ∀Exp+Res in a hybrid approach. To this end, we coupled a QCDCL solver and non-recursive expansion to make clauses derived by the QCDCL solver available to the expansion solver. Experimental results indicated that the hybrid approach significantly outperforms our implementation of non-recursive expansion indicating the potential of combining expansion-based approaches with Q-resolution which gives rise to an exciting direction of future work. Further, our current implementation supports only formulas in conjunctive normal form while in theory, our approach does not make any assumptions on the structure of the propositional part of the QBF. We also plan to investigate how this formula structure can be exploited for efficiently processing the negation of the formula.

ISBN: 978-0-9835678-8-2.

REFERENCES

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404 (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf

[2] Ayari, A., Basin, D.A.: QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In: FMCAD. LNCS, vol. 2517, pp. 187–201. Springer (2002)

[3] Balabanov, V., Jiang, J.R., Scholl, C., Mishchenko, A., Brayton, R.K.: 2QBF: Challenges and Solutions. In: SAT. LNCS, vol. 9710, pp. 453–469. Springer (2016)

[4] Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT Race 2015. Artif. Intell. 241, 45–65 (2016), http://dx.doi.org/10.1016/j.artint.2016.08.007

[5] Beyersdorff, O., Chew, L., Janota, M.: On unification of QBF resolution-based calculi. In: Proc. of the 39th Int. Symbosium on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 8635, pp. 81–93. Springer (2014)

[6] Biere, A., Lonsing, F., Seidl, M.: Blocked Clause Elimination for QBF. In: CADE. LNCS, vol. 6803, pp. 101–115. Springer (2011)

[7] Bloem, R., Braud-Santoni, N., Hadzic, V., Egly, U., Lonsing, F., Seidl, M.: Expansion-Based QBF Solving Without Recursion. CoRR abs/1807.08964 (2018), https://arxiv.org/abs/1807.08964, FMCAD 2018 proceedings version with appendix

[8] Bloem, R., Könighofer, R., Seidl, M.: SAT-Based Synthesis Methods for Safety Specs. In: VMCAI. LNCS, vol. 8318, pp. 1–20. Springer (2014)

[9] Bogaerts, B., Janhunen, T., Tasharrofi, S.: SAT-to-SAT in QBFEval 2016. In: QBF Workshop. CEUR Workshop Proceedings, vol. 1719, pp. 63–70. CEUR-WS.org (2016)

[10] Bogaerts, B., Janhunen, T., Tasharrofi, S.: Solving QBF Instances with Nested SAT Solvers. In: Beyond NP. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)

[11] Bubeck, U., Kleine Büning, H.: Bounded Universal Expansion for Preprocessing QBF. In: SAT. LNCS, vol. 4501, pp. 244–257. Springer (2007)

[12] Charwat, G., Woltran, S.: Dynamic Programming-based QBF Solving. In: QBF Workshop. CEUR Workshop Proceedings, vol. 1719, pp. 27–40. CEUR-WS.org (2016)

[13] Cheng, C., Hamza, Y., Ruess, H.: Structural Synthesis for GXW Specifications. In: CAV. LNCS, vol. 9779, pp. 95–117. Springer (2016)

[14] Cheng, C., Lee, E.A., Ruess, H.: autoCode4: Structural Controller Synthesis. In: TACAS. LNCS, vol. 10205, pp. 398–404. Springer (2017)

[15] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50(5), 752–794 (2003)

[16] Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: SAT. LNCS, vol. 3569, pp. 408–414. Springer (2005)

[17] Egly, U., Kronegger, M., Lonsing, F., Pfandler, A.: Conformant Planning as a Case Study of Incremental QBF Solving. Ann. Math. Artif. Intell. 80(1), 21–45 (2017)

[18] Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of Bounded Synthesis. In: TACAS. LNCS, vol. 10205, pp. 354–370. Springer (2017)

[19] Finkbeiner, B., Tentrup, L.: Detecting Unrealizability of Distributed Fault-tolerant Systems. Logical Methods in Computer Science 11(3) (2015)

[20] Gascón, A., Tiwari, A.: A Synthesized Algorithm for Interactive Consistency. In: NASA Formal Methods. LNCS, vol. 8430, pp. 270–284. Springer (2014)

[21] Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with Quantified Boolean Formulas. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 761–780. IOS Press (2009)

[22] Giunchiglia, E., Marin, P., Narizzano, M.: sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In: SAT. LNCS, vol. 6175, pp. 85–98. Springer (2010)

[23] Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause Elimination for SAT and QSAT. JAIR 53, 127–168 (2015)

[24] Heyman, T., Smith, D., Mahajan, Y., Leong, L., Abu-Haimed, H.: Dominant Controllability Check Using QBF-Solver and Netlist Optimizer. In: SAT. LNCS, vol. 8561, pp. 227–242. Springer (2014)

[25] Janota, M.: Towards Generalization in QBF Solving via Machine Learning. In: AAAI. AAAI Press (2018)

[26] Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with Counterexample Guided Refinement. Artif. Intell. 234, 1–25 (2016)

[27] Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with Counterexample Guided Refinement. In: SAT. LNCS, vol. 7317, pp. 114–128. Springer (2012)

[28] Janota, M., Marques-Silva, J.: On Propositional QBF Expansions and Q-Resolution. In: SAT. LNCS, vol. 7962, pp. 67–82. Springer (2013)

[29] Janota, M., Marques-Silva, J.: Expansion-Based QBF Solving versus Q-Resolution. Theor. Comput. Sci. 577, 25–42 (2015)

[30] Janota, M., Marques-Silva, J.: Solving QBF by Clause Selection. In: IJCAI. pp. 325–331. AAAI Press (2015)

[31] Janota, M., Silva, J.P.M.: Abstraction-Based Algorithm for 2QBF. In: SAT. LNCS, vol. 6695, pp. 230–244. Springer (2011)

[32] Kleine Büning, H., Bubeck, U.: Theory of Quantified Boolean Formulas. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 735–760. IOS Press (2009)

[33] Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Inf. Comput. 117(1), 12–18 (1995)

[34] Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A Non-Prenex, Non-Clausal QBF Solver with Game-State Learning. In: SAT. LNCS, vol. 6175, pp. 128–142. Springer (2010)

[35] Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: TABLEAUX. LNCS, vol. 2381, pp. 160–175. Springer (2002)

[36] Lonsing, F., Egly, U.: DepQBF 6.0: A Search-Based QBF Solver Beyond Traditional QCDCL. In: CADE. LNCS, vol. 10395, pp. 371–384. Springer (2017)

[37] Lonsing, F., Egly, U.: Evaluating QBF Solvers: Quantifier Alternations Matter. In: CP. LNCS, vol. 11008, pp. 276–294. Springer (2018)

[38] Peitl, T., Slivovsky, F., Szeider, S.: Dependency Learning for QBF. In: SAT. LNCS, vol. 10491, pp. 298–313. Springer (2017)

[39] Rabe, M.N., Tentrup, L.: CAQE: A Certifying QBF Solver. In: FMCAD. pp. 136–143. IEEE (2015)

[40] Ranjan, D.P., Tang, D., Malik, S.: A Comparative Study of 2QBF Algorithms. In: SAT (2004)

[41] Rintanen, J.: Asymptotically Optimal Encodings of Conformant Planning in QBF. In: AAAI. pp. 1045–1050. AAAI Press (2007)

[42] Tentrup, L.: Non-prenex QBF solving using abstraction. In: SAT. LNCS, vol. 9710, pp. 393–401. Springer (2016)

[43] Tentrup, L.: On Expansion and Resolution in CEGAR Based QBF Solving. In: CAV. LNCS, vol. 10427, pp. 475–494. Springer (2017)

[44] Tu, K., Hsu, T., Jiang, J.R.: QELL: QBF Reasoning with Extended Clause Learning and Levelized SAT Solving. In: SAT. LNCS, vol. 9340, pp. 343–359. Springer (2015)

[45] Vizel, Y., Weissenbacher, G., Malik, S.: Boolean Satisfiability Solvers and Their Applications in Model Checking. Proceedings of the IEEE 103(11), 2021–2035 (2015)

[46] Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre - An Effective Preprocessor for QBF and DQBF. In: TACAS. LNCS, vol. 10205, pp. 373–390. Springer (2017)

[47] Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: CP. LNCS, vol. 2470, pp. 200–215. Springer (2002)

[48] Zhang, W.: QBF Encoding of Temporal Properties and QBF-Based Verification. In: IJCAR. LNCS, vol. 8562, pp. 224–239. Springer (2014)

ISBN: 978-0-9835678-8-2.

Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction

Peter Backeman, Philipp Rümmer, Aleksandar Zeljić
Department of Information Technology, Uppsala University, Sweden

***Abstract*—The inference of program invariants over machine arithmetic, commonly called bit-vector arithmetic, is an important problem in verification. Techniques that have been successful for unbounded arithmetic, in particular Craig interpolation, have turned out to be difficult to generalise to machine arithmetic: existing bit-vector interpolation approaches are based either on eager translation from bit-vectors to unbounded arithmetic, resulting in complicated constraints that are hard to solve and interpolate, or on bit-blasting to propositional logic, in the process losing all arithmetic structure. We present a new approach to bit-vector interpolation, as well as bit-vector quantifier elimination (QE), that works by lazy translation of bit-vector constraints to unbounded arithmetic. Laziness enables us to fully utilise the information available during proof search (implied by decisions and propagation) in the encoding, and this way produce constraints that can be handled relatively easily by existing interpolation and QE procedures for Presburger arithmetic. The lazy encoding is complemented with a set of native proof rules for bit-vector equations and non-linear (polynomial) constraints, this way minimising the number of cases a solver has to consider.**

I. INTRODUCTION

Craig interpolation is a commonly used technique to infer invariants or contracts in verification. Over the last 15 years, efficient interpolation techniques have been developed for a variety of logics and theories, including propositional logic [1], [2], uninterpreted functions [1], [3], [4], first-order logic [5], [6], [7], algebraic data-types [8], linear real arithmetic [1], non-linear real arithmetic [9], Presburger arithmetic [10], [4], [11], and arrays [12], [13], [14].

A theory that has turned out notoriously difficult to handle in Craig interpolation is bounded machine arithmetic, commonly called bit-vector arithmetic. Decision procedures for bit-vectors are predominantly based on bit-blasting, in combination with sophisticated preprocessing and simplification methods, which implies that also extracted interpolants stay on the level of propositional logic and are difficult to map back to compact high-level bit-vector constraints. An alternative interpolation approach translates bit-vector constraints to unbounded integer arithmetic formulas [15], but is limited to linear constraints and tends to produce integer formulas that are hard to solve and interpolate, due to the necessary introduction of additional variables and large coefficients to model wrap-around semantics correctly.

In this paper, we introduce a new Craig interpolation method for bit-vector arithmetic, focusing on arithmetic bit-vector operations including addition, multiplication, and division. Like [15], we compute interpolants by reducing bit-vectors to unbounded integers; unlike in earlier approaches, we define a calculus that carries out this reduction lazily, and can therefore dynamically choose between multiple possible encodings of the bit-vector operations. This is done by initially representing bit-vector operations as uninterpreted predicates, which are expanded and replaced by Presburger arithmetic expressions on demand. The calculus also includes native rules for non-linear constraints and bit-vector equations, so that formulas can often be proven without having to resort to a full encoding as integer constraints. Our approach gives rise to both Craig interpolation and quantifier elimination (QE) methods for bit-vector constraints, with both procedures displaying competitive performance in our experiments.

Reduction of bit-vectors to unbounded integers has the additional advantage that integer and bit-vector formulas can be combined efficiently, including the use of conversion functions between both theories, which are difficult to support using bit-blasting. This combination is of practical importance in software verification, since programs and specifications often mix machine arithmetic with arbitrary-precision numbers; tools might also want to switch between integer semantics (if it is known that no overflows can happen) and bit-vector semantics for each individual program instruction.

The contributions of the paper are: 1) a new calculus for non-linear integer arithmetic, which can eliminate quantifiers (in certain cases) and extract Craig interpolants (Section III); 2) a corresponding calculus for arithmetic bit-vector constraints (Section IV); 3) an experimental evaluation using SMT-LIB and model checking benchmarks (Section V).

A. Related Work

Most SMT solvers handle bit-vectors using bit-blasting and SAT solving, and usually cannot extract interpolants for bit-vector problems. The exception is MATHSAT [16], which uses a layered approach [15] to compute interpolants: MATHSAT first tries to compute interpolants by keeping bit-vector operations uninterpreted; then using a restricted form of quantifier elimination; then by eager encoding into linear integer arithmetic (LIA); and finally through bit-blasting. Our approach has some similarities to the LIA encoding, but can choose simpler encodings thanks to laziness, and also covers non-linear arithmetic constraints.

Other related work has focused on fragments of bit-vector logic. In [17], an algorithm is given for reconstructing bit-vector interpolants from bit-level interpolants, however restricted to the case of bit-vector equalities. An interpolation

ISBN: 978-0-9835678-8-2.

procedure based on a set of tailor-made (but incomplete) rewriting rules for bit-vectors is given in [18].

II. Preliminaries: The Base Logic

We formulate our approach on top of a simple logic of Presburger arithmetic constraints combined with uninterpreted predicates, introduced in [19] and extended in [4], [10] to support Craig interpolation. Let x range over an infinite set X of variables, c over an infinite set C of constants, p over a set P of uninterpreted predicates with fixed arity, and α over the set $\mathbb{Z}$ of integers. The syntax of terms and formulae is defined by the following grammar:

$$\phi ::= t = 0 \,\Big|\, t \leq 0 \,\Big|\, p(t, \ldots, t) \,\Big|\, \phi \wedge \phi \,\Big|\, \phi \vee \phi \,\Big|\, \neg\phi \,\Big|\, \forall x.\phi \,\Big|\, \exists x.\phi$$

$$t ::= \alpha \,\Big|\, c \,\Big|\, x \,\Big|\, \alpha t + \cdots + \alpha t$$

The symbol t denotes terms of linear arithmetic. Substitution of a term t for a variable x in ϕ is denoted by $[x/t]\phi$; we assume that variable capture is avoided by renaming bound variables as necessary. For simplicity, we sometimes write $s = t$ as a shorthand of $s - t = 0$, inequality $s \leq t$ for $s - t \leq 0$, and $\forall c.\phi$ as a shorthand of $\forall x.[c/x]\phi$ if c is a constant. The abbreviation *true* (*false*) stands for the equality $0 = 0$ ($1 = 0$), and the formula $\phi \rightarrow \psi$ abbreviates $\neg\phi \vee \psi$. Semantic notions such as structures, models, satisfiability, and validity are defined as is common (e.g., [20]), but we assume that evaluation always happens over the universe $\mathbb{Z}$ of integers; bit-vectors will later be defined as a subset of the integers.

A. A Sequent Calculus for the Base Logic

For checking whether a formula in the base logic is satisfiable or valid, we work with the calculus presented in [19], a part of which is shown in Fig. 1. If Γ, Δ are sets of formulae, then $\Gamma \vdash \Delta$ is a *sequent*. A sequent is *valid* if the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. Positions in Δ that are underneath an even/odd number of negations are called positive/negative; and vice versa for Γ. Proofs are trees growing upward, in which each node is labelled with a sequent, and each non-leaf node is related to the node(s) directly above it through an application of a calculus rule. A proof is *closed* if it is finite and all leaves are justified by an instance of a rule without premises. Soundness of the calculus implies that the root of a closed proof is a valid sequent.

In addition to propositional and quantifier rules in Fig. 1, the calculus in [19] also includes rules for equations and inequalities in Presburger arithmetic; the details of those rules are not relevant for this paper. The calculus is complete for quantifier-free formulas in the base logic, i.e., for every valid quantifier-free sequent a closed proof can be found. It is well-known that the base logic including quantifiers does not admit complete calculi [21], but as discussed in [19] the calculus can be made complete (by adding slightly more sophisticated quantifier handling) for interesting undecidable fragments, for instance for sequents $\vdash \phi$ with only existential quantifiers.

For quantifier-free input formulas, proof search can be implemented in depth-first style following the core concepts of DPLL(T) [22]: rules with multiple premises correspond to *decisions* and explore the branches one by one; rules with a single premise represent *propagation* or *rewriting;* and logging of rule applications is used in order to implement conflict-driven learning and proof extraction. For experiments, we use the implementation of the calculus in Princess.[1]

$$\dfrac{\begin{array}{c}\Gamma, \phi \vdash \Delta \\ \Gamma, \psi \vdash \Delta\end{array}}{\Gamma, \phi \vee \psi \vdash \Delta}\ \vee\text{-LEFT} \qquad \dfrac{\begin{array}{c}\Gamma, \phi \vdash \Delta \\ \Gamma, \psi \vdash \Delta\end{array}}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\text{-RIGHT}$$

$$\dfrac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\text{-LEFT} \qquad \dfrac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \vee\text{-RIGHT}$$

$$\dfrac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \neg\text{-LEFT} \qquad \dfrac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\text{-RIGHT}$$

$$\dfrac{*}{\Gamma, \phi \vdash \phi, \Delta}\ \text{CLOSE}$$

$$\dfrac{\Gamma, [x/t]\phi, \forall x.\phi \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \forall\text{-LEFT} \qquad \dfrac{\Gamma, [x/c]\phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta}\ \exists\text{-LEFT}$$

$$\dfrac{\Gamma \vdash [x/t]\phi, \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\text{-RIGHT} \qquad \dfrac{\Gamma \vdash [x/c]\phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \forall\text{-RIGHT}$$

Fig. 1. A selection of the basic calculus rules for propositional logic (upper box) and quantifier rules (lower box). In the rules $\exists$-LEFT and $\forall$-RIGHT, c is a constant that does not occur in the conclusion.

B. Quantifier Elimination in the Base Logic

The sequent calculus can eliminate quantifiers in Presburger arithmetic, i.e., in the base logic without uninterpreted predicates, since the arithmetic calculus rules are designed to systematically eliminate constants. To illustrate this use case, suppose ϕ is a formula without uninterpreted predicates and without constants c, but possibly containing variables x. Formula ϕ furthermore only contains $\forall/\exists$ under an even/odd number of negations, i.e., all quantifiers are effectively universal. To compute a quantifier-free formula ψ that is equivalent to ϕ, we can construct a proof with root sequent $\vdash \phi$, and keep applying rules until no further applications are possible in any of the remaining open goals $\{\Gamma_i \vdash \Delta_i \mid i = 1, \ldots, n\}$. In this process, rules $\exists$-LEFT and $\forall$-RIGHT can introduce fresh constants, which are subsequently isolated and eliminated by the arithmetic rules. To find ψ, it is essentially enough to extract the constant-free formulas $\Gamma_i^v \subseteq \Gamma_i$, $\Delta_i^v \subseteq \Delta_i$ in the open goals, and construct $\psi = \bigwedge_{i=1}^{n}(\bigwedge \Gamma_i^v \rightarrow \bigvee \Delta_i^v)$.

The full calculus [19] is moreover able to eliminate arbitrarily nested quantifiers, and can be used similarly to prove validity of sequents with quantifiers. A recent independent evaluation [23] showed that the resulting proof procedure is competitive with state-of-the-art SMT solvers and theorem provers on a wide range of quantified integer problems.

$$\frac{\Gamma, \lfloor\phi\rfloor_L \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_L \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_L \vdash \Delta \blacktriangleright I \vee J} \ \vee\text{-LEFT}_L$$

$$\frac{\Gamma, \lfloor\phi\rfloor_R \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_R \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_R \vdash \Delta \blacktriangleright I \wedge J} \ \vee\text{-LEFT}_R$$

$$\frac{\Gamma, \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\phi \wedge \psi\rfloor_D \vdash \Delta \blacktriangleright I} \ \wedge\text{-LEFT}_D$$

$$\frac{\Gamma \vdash \lfloor\phi\rfloor_D, \Delta \blacktriangleright I}{\Gamma, \lfloor\neg\phi\rfloor_D \vdash \Delta \blacktriangleright I} \ \neg\text{-LEFT}_D$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \mathit{false}} \ \text{CLOSE}_{LL}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \phi} \ \text{CLOSE}_{LR}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_R \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \neg\phi} \ \text{CLOSE}_{RL}$$

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_L, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright \forall_{Rt}\, I} \ \forall\text{-LEFT}_L$$

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_R, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright \exists_{Lt}\, I} \ \forall\text{-LEFT}_R$$

$$\frac{\Gamma, \lfloor[x/c]\phi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\exists x.\phi\rfloor_D \vdash \Delta \blacktriangleright I} \ \exists\text{-LEFT}_D$$

Fig. 2. The upper box presents a selection of interpolating rules for propositional logic, while the lower box shows rules for quantifiers. Parameter D stands for either L or R. The quantifier $\forall_{Rt}$ denotes universal quantification over all constants occurring in t but not in $\Gamma_L \cup \Delta_L$; likewise, $\exists_{Lt}$ denotes existential quantification over all constants occurring in t but not in $\Gamma_R \cup \Delta_R$. In $\exists$-LEFT$_D$, c is a constant that does not occur in the conclusion.

C. Craig Interpolation in the Base Logic

Given formulas A and B such that $A \wedge B$ is unsatisfiable, Craig interpolation can determine a formula I such that the implications $A \Rightarrow I$ and $B \Rightarrow \neg I$ hold, and non-logical symbols in I occur in both A and B [24]. An interpolating version of our sequent calculus has been presented in [4], [10], and is summarised in Fig. 2. To keep track of the partitions A, B, the calculus operates on labelled formulas $\lfloor\phi\rfloor_L$ (with L for "left") to indicate that ϕ is derived from A, and similarly formulas $\lfloor\phi\rfloor_R$ for ϕ derived from B. If Γ, Δ are finite sets of *L*/*R*-labelled formulas, and I is an unlabelled formula, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent.*

Semantics of interpolating sequents is defined using projections $\Gamma_L =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_L \in \Gamma\}$ and $\Gamma_R =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_R \in \Gamma\}$, which extract the *L*/*R*-parts of a set Γ of labelled formulae. A sequent $\Gamma \vdash \Delta \blacktriangleright I$ is *valid* if 1) the sequent $\Gamma_L \vdash I, \Delta_L$ is valid, 2) the sequent $\Gamma_R, I \vdash \Delta_R$ is valid, and 3) the constants and uninterpreted predicates/functions in I occur in both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$. As a special case, note that the sequent $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$ is valid iff I is an interpolant of $A \wedge B$. Soundness of the calculus guarantees that the root of a closed interpolating proof is a valid interpolating sequent.

To solve an interpolation problem $A \wedge B$, a prover typically first constructs a proof of $A, B \vdash \emptyset$ using the ordinary calculus from Section II-A. Once a closed proof has been found, it can be lifted to an interpolating proof: this is done by replacing the root formulas A, B with $\lfloor A\rfloor_L, \lfloor B\rfloor_R$, respectively, and recursively assigning labels to all other formulas as defined by the rules from Fig. 2. Then, starting from the leaves, intermediate interpolants are computed and propagated back to the root, leading to an interpolating sequent $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$.

III. SOLVING NON-LINEAR CONSTRAINTS

We extend the base logic in two steps: in this section, symbols and rules are added to solve *non-linear diophantine problems;* a second extension is then done in Section IV to handle *arithmetic bit-vector constraints.* Both constructions preserve the ability of the calculus to eliminate quantifiers (under certain assumptions) and derive Craig interpolants.

For non-linear constraints, we assume that the set P of uninterpreted predicates contains a distinguish ternary predicate $\times$, with the intended semantics that the third argument represents the result of multiplying the first two arguments, i.e., $\times(s, t, r) \Leftrightarrow s \cdot t = r$. The predicate $\times$ is clearly sufficient to express arbitrary polynomial constraints by introducing a $\times$-literal for each product in a formula, at the cost of introducing a linear number of additional constants or existentially quantified variables. We make the simplifying assumption that $\times$ only occurs in negative positions; that means, top-level occurrences will be on the left-hand side of sequents. Positive occurrences can be eliminated thanks to the equivalence $\neg\times(s, t, r) \Leftrightarrow \exists x.(\times(s, t, x) \wedge x \neq r)$.

A. Calculus Rules for Non-Linear Constraints

We now introduce different classes of calculus rules to reason about the $\times$-predicate. The rules are necessarily incomplete for proving that a sequent is valid, but they are complete for finding counterexamples: if ϕ is a satisfiable quantifier-free formula with $\times$ as the only uninterpreted predicate, then it is possible to construct a proof for $\phi \vdash \emptyset$ that has an open and unprovable goal in pure Presburger arithmetic (by systematically splitting variable domains, Section III-A4).

1) Deriving Implied Equalities with Gröbner Bases: The first rule applies standard algebra methods to infer new equalities from multiplication literals. To avoid the computation of more and more complex terms in this process, we restrict the calculus to the inference of *linear* equations that can be derived through computation of a Gröbner basis.[2] Given a set $\{\times(s_i, t_i, r_i)\}_{i=1}^n$ of $\times$-literals and a set $\{e_j = 0\}_{j=1}^m$ of linear equations, the generated ideal $I = (\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ over rational numbers is the smallest set of rational polynomials that contains $\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m$, is closed under addition, and closed under multiplication with arbitrary rational polynomials [25]. Any $f \in I$ corresponds to an

---

[1] http://www.philipp.ruemmer.org/princess.shtml

[2] The set of *all* linear equations implied by a set of $\times$-literals over integers is clearly not computable, by reduction of Hilbert's 10th problem.

equation $f = 0$ that logically follows from the literals, and can therefore be added to a proof goal:

$$\frac{\Gamma, \{\times(s_i, t_i, r_i)\}_{i=1}^n, \{e_j = 0\}_{j=1}^m, f = 0 \;\vdash\; \Delta}{\Gamma, \{\times(s_i, t_i, r_i)\}_{i=1}^n, \{e_j = 0\}_{j=1}^m \;\vdash\; \Delta} \;\times\text{-EQ}$$

$$\text{if } f \text{ is linear, has integer coefficients, and } f \in I$$

To see how this rule can be applied practically, note that the subset of linear polynomials in I forms a rational vector space, and therefore has a finite basis. It is enough to apply ×-EQ for terms $f_1, \ldots, f_k$ corresponding to any such basis, since linear arithmetic reasoning (in the base logic) will then be able to derive all other linear polynomials in I. To compute a basis $f_1, \ldots, f_k$, we can transform $\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m$ to a Gröbner basis using Buchberger's algorithm [26], and then apply Gaussian elimination to find linear basis polynomials (or directly by choosing a suitable monomial order).

Example 1: Consider the square of a sum: $(x + y)^2 = x^2 + 2xy + y^2$. This can be proven in the following way. We begin by rewriting the equation to normal form, let $\Pi = \{\times(x, x, c_1), \times(x, y, c_2), \times(y, y, c_3), \times(x{+}y, x{+}y, c_4)\}$:

$$\frac{\begin{array}{c} * \\ \vdots \\ \Pi, c_1 + 2c_2 + c_3 - c_4 = 0 \;\vdash\; c_4 = c_1 + 2c_2 + c_3 \end{array}}{\Pi \;\vdash\; c_4 = c_1 + 2c_2 + c_3} \;\times\text{-EQ}$$

Here, the ×-EQ-step is motivated by the fact that the Gröbner basis derived from Π contains the linear polynomial $c_1 + 2c_2 + c_3 - c_4$, from which the desired equation can be derived using linear reasoning.

2) Interval Constraint Propagation (ICP): Our main technique for inequality reasoning in the presence of ×-predicates is interval constraint propagation (ICP) [27], which computes greatest fixed-points over-approximating the ranges of constants or free variables. Due to lack of space we do not introduce ICP in full detail, but only assume that $\mathit{Prop}_{\{\phi_1,\ldots,\phi_n\}}$ is a monotonic function describing the propagation of bounds information implied by equalities, inequalities, and ×-literals $\phi_1, \ldots, \phi_n$, and gfp $\mathit{Prop}_{\{\phi_1,\ldots,\phi_n\}}$ is its greatest fixed-point. The ICP rule adds resulting bounds for a constant or variable $c \in C \cup X$:

$$\frac{\Gamma, \phi_1, \ldots, \phi_n,\; l \le c, c \le u \;\vdash\; \Delta}{\Gamma, \phi_1, \ldots, \phi_n \;\vdash\; \Delta} \;\times\text{-ICP}$$

$$\text{if } (\text{gfp}\, \mathit{Prop}_{\{\phi_1,\ldots,\phi_n\}})(c) = [l, u]$$

Example 2: From two inequalities $x \ge 5$ and $y \ge 5$, the rule ×-ICP can derive $(x + y)^2 \ge 100$:

$$\frac{\times(x + y, x + y, c_4), x \ge 5, y \ge 5, 100 \le c_4 \;\vdash}{\times(x + y, x + y, c_4), x \ge 5, y \ge 5 \;\vdash} \;\times\text{-EQ}$$

The slightly different problem $x + y \ge 10 \rightarrow (x + y)^2 \ge 100$ cannot be proven in the same way, since ICP will not be able to deduce bounds for x or y from $x + y \ge 10$.

3) Cross-Multiplication of Inequalities: While ICP is highly effective for approximating the range of constants, and quickly detecting inconsistencies, it is less useful for inferring relationships between multiple constants that follow from multiplication literals. We cover such inferences using a *cross-multiplication* rule that resembles procedures used in ACL2 [28]. The rule captures the fact that if s, t are both non-negative, then also the product $s \cdot t$ is non-negative.

Like in Section III-A1, we prefer to avoid the introduction of new multiplication literals during proof search, and only add $s \cdot t \ge 0$ if the term $s \cdot t$ can be expressed linearly. For this, we again write $I = (\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ for the ideal induced by equations and ×-literals:

$$\frac{\Gamma, s \le 0, t \le 0,\; -f \le 0 \;\vdash\; \Delta}{\Gamma, s \le 0, t \le 0 \;\vdash\; \Delta} \;\times\text{-CROSS}$$

$$\text{if } f \text{ is linear, has integer coefficients, and } s \cdot t - f \in I$$

The term f can practically be found by computing a Gröbner basis of I, and reducing the product $s \cdot t$ to check whether an equivalent linear term exists.

4) Interval Splitting: If everything else fails, as last resort it can become necessary to systematically split over the possible values of a variable or constant $c \in C \cup X$:

$$\frac{\Gamma, c \le \alpha - 1 \;\vdash\; \Delta \qquad \Gamma, c \ge \alpha \;\vdash\; \Delta}{\Gamma \;\vdash\; \Delta} \;\times\text{-SPLIT}$$

The $\alpha \in \mathbb{Z}$ can in principle be chosen arbitrarily in the rule, but in practice a useful strategy is to make use of the range information derived for ×-ICP: when no ranges can be tightened any further using ×-ICP, instead ×-SPLIT can be applied to split one of the intervals in half.

5) ×-Elimination: Finally, occurrences of × can be eliminated whenever a formula is subsumed by other literals in a goal, again writing $I = (\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$:

$$\frac{\Gamma \;\vdash\; \Delta}{\Gamma, \times(s, t, r) \;\vdash\; \Delta} \;\times\text{-ELIM}$$

$$\text{if } s \cdot t - r \in I$$

Note that ×-ELIM only eliminates non-linear ×-literals, whereas ×-EQ only introduces linear equations, so that the application of the two rules cannot induce cycles.

B. Quantifier Elimination for Non-Linear Constraints

Due to necessary incompleteness of calculi for Peano arithmetic, quantifiers can in general not be eliminated in the presence of the × predicate, even when considering formulas that do not contain other uninterpreted predicates. By combining the QE approach in Section II-B with the rules for × that we have introduced, it is nevertheless possible to reason about quantified non-linear constraints in many practical cases, and sometimes even get rid of quantifiers. This is possible because the rules in Section III-A are not only sound, but even *equivalence transformations:* in any application of the rules, the conjunction of the premises is equivalent to the conclusion.

ISBN: 978-0-9835678-8-2.

Similarly as in [29], QE is always possible if sufficiently many constants or variables in a formula ϕ range over *bounded* domains: if there is a set $B \subseteq C \cup X$ of symbols with bounded domain such that in each literal $\times(s, t, r)$ either s or t contain only symbols from B. In this case, proof construction will terminate when applying the rule $\times$-SPLIT only to variables or constants with bounded domain. This guarantees that eventually every literal $\times(s, t, r)$ can be turned into a linear equation using $\times$-EQ, and then be eliminated using $\times$-ELIM, only leaving proof goals with pure Presburger arithmetic constraints. The boundedness condition is naturally satisfied for bit-vector formulas.

C. Craig Interpolation for Non-Linear Constraints

To carry over the Craig interpolation approach from Section II-C to non-linear formulas, interpolating versions of the calculus rules for the $\times$-predicate are needed. For this, we follow the approach used in [4] (which in turn resembles the use of theory lemmas in SMT in general): when translating a proof to an interpolating proof, we replace applications of the $\times$-rules with instantiation of an equivalent theory axiom QAx. Suppose a non-interpolating proof contains a rule application

$$\cfrac{\begin{array}{c}\vdots \\ \Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta\end{array} \quad \cdots \quad \begin{array}{c}\vdots \\ \Gamma, \Gamma', \Gamma_n \vdash \Delta_n, \Delta', \Delta\end{array}}{\begin{array}{c}\Gamma, \Gamma' \vdash \Delta', \Delta \\ \vdots\end{array}}\ R$$

in which Γ', Δ' are the formulas assumed by the rule application, Γ, Δ are side formulas not required or affected by the application, and $\Gamma_1, \Delta_1, \ldots, \Gamma_n, \Delta_n$ are newly introduced formulas in the individual branches.

The (unquantified) theory axiom Ax corresponding to the rule application expresses that the conjunction of the premises has to imply the conclusion; the quantified theory axiom $QAx =_{\text{def}} \forall S.\, Ax$ in addition contains universal quantifiers for all constants $S \subseteq C$ occurring in Ax.

$$Ax \;=_{\text{def}}\; \bigwedge_{i=1}^{n} \big(\bigwedge \Gamma_i \to \bigvee \Delta_i \big) \;\to\; \big(\bigwedge \Gamma' \to \bigvee \Delta' \big)$$

Ax and QAx are specific to the *application* of R: the axioms for two distinct applications of R will in general be different formulas. QAx is defined in such a way that the effect of R can be simulated by introducing QAx in the antecedent, instantiating it with the right constants, and applying propositional rules:

$$\cfrac{\cfrac{\begin{array}{c} * \\ \vdots \\ \Gamma, \Gamma', \bigwedge \Gamma' \to \bigvee \Delta' \vdash \Delta', \Delta \end{array} \qquad \cfrac{\Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \quad \cdots}{\vdots}}{\Gamma, \Gamma', Ax \vdash \Delta', \Delta}}{\Gamma, \Gamma', \forall S. Ax \vdash \Delta', \Delta}\ \forall\text{-LEFT}*$$

This construction leads to a proof using only the standard rules from Section II-A, which can be interpolated as discussed earlier. Since QAx is a valid formula not containing any constants, it can be introduced in a proof at any point, and labelled $\lfloor QAx \rfloor_L$ or $\lfloor QAx \rfloor_R$ on demand.

The obvious downside of this approach is the possibility of quantifiers occurring in interpolants. The interpolating rules $\forall$-LEFT$_{L/R}$ (Fig. 2) have to introduce quantifiers $\forall_{Rt}$/$\exists_{Lt}$ for local symbols occurring in the substituted term t; whether such quantifiers actually occur in the final interpolant depends on the applied $\times$-rules, and on the order of rule application. For instance, with $\times$-SPLIT it is always possible to choose the label of QAx so that no quantifiers are needed, whereas $\times$-EQ might mix symbols from left and right partitions in such a way that quantifiers become unavoidable. In our implementation we approach this issue pragmatically. We leave proof search unrestricted, and might thus sometimes get proofs that do not give rise to quantifier-free interpolants; when that happens, we afterwards apply QE to get rid of the quantifiers. QE is always possible for bit-vector constraints, see Section IV-D.[3]

IV. SOLVING BIT-VECTOR CONSTRAINTS

We now define the extension of the base logic to bit-vector constraints. The main idea of the extension is to represent bit-vectors of width w as integers in the interval $\{0, \ldots, 2^w - 1\}$, and to translate bit-vector operations to the corresponding operation in Presburger arithmetic (or possible the $\times$-predicate for non-linear formulas), followed by an integer remainder operation to map the result back to the correct bit-vector domain. Since the remainder operation tends to be a bottleneck for interpolation, we keep the operation symbolic and initially consider it as an uninterpreted predicate bmod_a^b. The predicate is only gradually reduced to Presburger arithmetic by applying the calculus rules introduced later in this section.

Formally, we introduce a set $P_{bv} = \{\mathit{bmod}_a^b \mid a, b \in \mathbb{Z}, a < b\}$ of binary predicates. The semantics of bmod_a^b is to relate any whole number $x \in \mathbb{Z}$ to its remainder modulo $b - a$ in the interval $\{a, \ldots, b - 1\}$:

$$\begin{aligned} \mathit{bmod}_a^b(s, r) \;&\Leftrightarrow\; a \leq r < b \;\wedge\; \exists z.\; r = s + (b - a) \cdot z \\ &\Leftrightarrow\; a \leq r < b \;\wedge\; r \equiv s \pmod{b - a} \end{aligned}$$

We also introduce short-hand notations for the casts to the unsigned and signed bit-vector domains:

$$\mathit{ubmod}_w =_{\text{def}} \mathit{bmod}_0^{2^w}, \quad \mathit{sbmod}_w =_{\text{def}} \mathit{bmod}_{-2^{w-1}}^{2^{w-1}} .$$

A. Translating Bit-Vector Constraints to the Core Language

For the rest of the section, we use the base logic augmented with $\times$ and bmod_a^b-predicates as the *core language* to which bit-vector constraints are translated. For presentation, the translation focuses on a subset of the arithmetic bit-vector operations, BVOP $= \{\mathsf{bvadd}_w, \mathsf{bvmul}_w, \mathsf{bvudiv}_w, \mathsf{bvneg}_w, \mathsf{ze}_{w+w'}, \mathsf{bvule}_w, \mathsf{bvsle}_w\}$. All operations are sub-scripted with the bit-width of the operands; the zero-extend function $\mathsf{ze}_{w+w'}$ maps bit-vectors of width w to width $w + w'$. Semantics

---

[3] Non-linear integer arithmetic in general does not admit quantifier-free interpolants. For instance, $(x > 1 \wedge x = y^2) \wedge x = z^2 + 1$ is unsatisfiable, but no quantifier-free interpolants exist, regardless of whether divisibility predicates $\alpha \mid t$ are allowed or not.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{bvadd}_w(s,t) = r & \mathrel{-\!\triangleright} & \mathit{ubmod}_w(s+t, r) & \mathsf{bvneg}_w(s) = r & \mathrel{-\!\triangleright} & \mathit{ubmod}_w(-s, r) \\
\mathsf{bvmul}_w(s,t) = r & \mathrel{-\!\triangleright} & \exists x.\ \big(\times(s,t,x) \wedge \mathit{ubmod}_w(x,r)\big) & \mathsf{ze}_{w+w'}(s) = r & \mathrel{-\!\triangleright} & s = r \\
\mathsf{bvsle}_w(s,t) & \mathrel{-\!\triangleright} & \exists x,y.\ (\mathit{sbmod}_w(s,x) \wedge \mathit{sbmod}_w(t,y) \wedge x \leq y) & \mathsf{bvule}_w(s,t) & \mathrel{-\!\triangleright} & s \leq t \\
\neg\mathsf{bvsle}_w(s,t) & \mathrel{-\!\triangleright} & \exists x,y.\ (\mathit{sbmod}_w(s,x) \wedge \mathit{sbmod}_w(t,y) \wedge x > y) & \neg\mathsf{bvule}_w(s,t) & \mathrel{-\!\triangleright} & s > t
\end{array}$$

$$\mathsf{bvudiv}_w(s,t) = r \quad \mathrel{-\!\triangleright} \quad \big(t = 0 \wedge r = 2^w - 1\big) \vee \big(t \geq 1 \wedge \exists x.\, (\times(t,r,x) \wedge s - t < x \leq s)\big)$$

Fig. 3. Rules translating bit-vector operations into the core language. The rules only apply in negative positions.

follows the FixedSizeBitVectors[4] theory of the SMT-LIB [30]. Other arithmetic operations, for instance bvsdiv_w or bvsmod_w, can be handled in the same way as shown here, though sometimes the number of cases to be considered is larger.

The translation from bit-vector constraints ϕ to core formulas ϕ_{core} has two parts: first, BVOP occurrences in a formula ϕ have to be replaced with equivalent expressions in the core language; second, since the core language only knows the sort of unbounded integers, type information has to be made explicit by adding domain constraints.

a) BVOP *elimination:* Like in Section III, we assume that the bit-vector formula ϕ has already been brought into a flat form by introducing additional constants or quantified variables: the operations in BVOP must not occur nested, and functions only occur in equations of the form $f(\bar{s}) = t$ in negative positions. The translation from ϕ to ϕ' is then defined by the rewriting rules in Fig. 3. Since the rules for the predicate bvsle_w distinguish between positive and negative occurrences, we assume that rules are only applied to formulas in negation normal-form, and only in negative positions.

The rules for bvadd_w, bvneg_w, $\mathsf{ze}_{w+w'}$, and bvule_w simply translate to the corresponding Presburger term, if necessary followed by remainder ubmod_w. Multiplication bvmul_w is mapped similarly to the $\times$-predicate defined in Section III, adding an existential quantifier to store the intermediate product. Since rules are only applied in negative positions, the quantified variable can later be replaced with a Skolem constant. An optimised rule could be defined for the case that one of the factors is constant, avoiding the use of the $\times$-predicate. Translation of bvsle_w simply maps the operands to a signed bit-vector domain $\{-2^{w-1}, \ldots, 2^{w-1} - 1\}$. The rule for unsigned division bvudiv_w distinguishes the cases that the divisor t is zero or positive (as required by SMT-LIB), and maps the latter case to standard integer division.

b) Domain constraints: Bit-vector variables/constants x of width w occurring in ϕ are interpreted as unbounded integer variables in ϕ_{core}, which therefore has to contain explicit assumptions about the ranges of bit-vector variables. We use the abbreviation $\mathit{in}_w(x) =_{\mathrm{def}} (0 \leq x < 2^w)$ and define

$$\phi_{core} \;=\; \Big(\bigwedge_{x \in S} \mathit{in}_{w_x}(x)\Big) \to \phi'$$

where $S \subseteq C \cup X$ is the set of free variables and constants occurring in ϕ, w_x is the bit-width of $x \in S$, and ϕ' is the result of applying rules from Fig. 3 to ϕ. Similar constraints are used to express quantification over bit-vectors, for instance $\exists x.\ (\mathit{in}_w(x) \wedge \ldots)$ and $\forall x.\ (\mathit{in}_w(x) \to \ldots)$.

Example 3: We consider the SMT-LIB QF_BV problem `challenge/multiplyOverflow.smt2`, a bit-vector formula that is known to be hard for most SMT solvers since it contains both multiplication and division. In experiments, neither Z3 nor CVC4 could prove the formula within 10min. In our notation, the problem amounts to showing validity of the following implication, with a, b ranging over bit-vectors of width 32:

$$\begin{aligned}
&\mathsf{bvule}_{32}(b, \mathsf{bvudiv}_{32}(2^{32} - 1, a)) \to \\
&\qquad \mathsf{bvule}_{64}\big(\mathsf{bvmul}_{64}(\mathsf{ze}_{32+32}(a), \mathsf{ze}_{32+32}(b)), 2^{32} - 1\big)
\end{aligned}$$

As a flat formula, with additional constants c_1 of width 32 and c_2, c_3, c_4 of width 64, the implication takes the form:

$$\begin{aligned}
&\big(\mathsf{bvudiv}_{32}(2^{32} - 1, a) = c_1 \wedge \mathsf{bvmul}_{64}(c_3, c_4) = c_2 \wedge {} \\
&\quad \mathsf{ze}_{32+32}(a) = c_3 \wedge \mathsf{ze}_{32+32}(b) = c_4 \wedge \mathsf{bvule}_{32}(b, c_1)\big) \to \\
&\mathsf{bvule}_{64}(c_2, 2^{32} - 1)
\end{aligned}$$

The final formula ϕ_{core} is obtained by application of the rules in Fig. 3, and adding domain constraints:

$$\begin{aligned}
&\big(\mathit{in}_{32}(a) \wedge \mathit{in}_{32}(b) \wedge \mathit{in}_{32}(c_1) \wedge \mathit{in}_{64}(c_2) \wedge \mathit{in}_{64}(c_3) \wedge \mathit{in}_{64}(c_4)\big) \wedge \\
&\left(\begin{array}{l} \big(a = 0 \wedge c_1 = 2^{32} - 1\big) \vee \\ \big(a \geq 1 \wedge \exists x.(\times(a, c_1, x) \wedge 2^{32} - 1 - a < x \leq 2^{32} - 1)\big) \end{array}\right) \wedge \\
&\quad \exists z.\ (\times(c_3, c_4, z) \wedge \mathit{ubmod}_{64}(z, c_2)) \wedge a = c_3 \wedge b = c_4 \wedge b \leq c_1\big) \\
&\to c_2 \leq 2^{32} - 1
\end{aligned}$$

B. Preprocessing and Simplification

An encoded formula ϕ_{core} tends to contain a lot of redundancy, in particular nested or unnecessary occurrences of the bmod_a^b predicates. As an important component of our calculus, and in line with the approach in other bit-vector solvers, we therefore apply simplification rules both during preprocessing and during the solving phase ("inprocessing"). The most important simplification rules are shown in Fig. 4. Our implementation in addition applies rules for Boolean and Presburger connectives.

The notation $\Pi : \phi \mathrel{-\!\triangleright} \phi'$ expresses that formula ϕ can be rewritten to ϕ', given the set Π of formulas as context. The structural rules in the upper half of Fig. 4 define how formulas are traversed, and how the context Π is extended to Π, Lit' when encountering further literals. We apply the structural rules modulo associativity and commutativity of $\wedge, \vee$, and prioritise LIT-$\wedge$-RW and LIT-$\vee$-RW over the other

---

[4] http://www.smtlib.org/theories-FixedSizeBitVectors.shtml

$$\frac{\Pi : \phi \rightarrowtriangle \phi' \qquad \Pi : \psi \rightarrowtriangle \psi'}{\Pi : \phi \circ \psi \rightarrowtriangle \phi' \circ \psi'}\ \circ\text{-RW}$$

$$\frac{\Pi : \mathit{Lit} \rightarrowtriangle \mathit{Lit}' \qquad \Pi, \mathit{Lit}' : \phi \rightarrowtriangle \phi'}{\Pi : \mathit{Lit} \wedge \phi \rightarrowtriangle \mathit{Lit}' \wedge \phi'}\ \text{LIT-}\wedge\text{-RW}$$

$$\frac{\Pi : \mathit{Lit} \rightarrowtriangle \mathit{Lit}' \qquad \Pi, \neg\mathit{Lit}' : \phi \rightarrowtriangle \phi'}{\Pi : \mathit{Lit} \vee \phi \rightarrowtriangle \mathit{Lit}' \vee \phi'}\ \text{LIT-}\vee\text{-RW}$$

$$\frac{\Pi : \phi \rightarrowtriangle \phi'}{\Pi : \neg\phi \rightarrowtriangle \neg\phi'}\ \neg\text{-RW} \qquad \frac{\Pi : \phi \rightarrowtriangle \phi'}{\Pi : Qx.\phi \rightarrowtriangle Qx.\phi'}\ Q\text{-RW}$$

$$\frac{\left\lfloor \frac{\mathit{lbound}(\Pi,s)-a}{b-a} \right\rfloor = k = \left\lfloor \frac{\mathit{ubound}(\Pi,s)-a}{b-a} \right\rfloor}{\Pi : \mathit{bmod}_a^b(s,r) \rightarrowtriangle s = r + k \cdot (b-a)}\ \text{BOUND-RW}$$

$$\frac{s + (b-a)\cdot t \prec s}{\Pi : \mathit{bmod}_a^b(s,r) \rightarrowtriangle \mathit{bmod}_a^b(s + (b-a)\cdot t, r)}\ \text{COEFF-RW}$$

$$\frac{\begin{array}{c}\mathit{bmod}_{a'}^{b'}(s',r') \in \Pi, \quad (b-a) \mid k \cdot (b'-a'), \\ s + k\cdot(s'-r') \prec s\end{array}}{\Pi : \mathit{bmod}_a^b(s,r) \rightarrowtriangle \mathit{bmod}_a^b(s + k\cdot(s'-r'), r)}\ \text{BMOD-RW}$$

Fig. 4. Simplification rules for bit-vector formulas. In $\circ$-RW, ϕ and ψ are not literals, and $\circ \in \{\wedge, \vee\}$. In LIT-$\wedge$-RW and LIT-$\vee$-RW, the formula Lit is a literal. In Q-RW, x must not occur in Π, and $Q \in \{\forall, \exists\}$. In COEFF-RW, all constants or variables in t also occur in s.

rules. Simplification is iterated until a fixed-point is reached and no further rewriting is possible. The connection between rewriting rules and the sequent calculus is established by the following rules:

$$\frac{\Gamma, \phi' \vdash \Delta}{\Gamma, \phi \vdash \Delta}\ \text{RW-LEFT} \qquad \frac{\Gamma \vdash \phi', \Delta}{\Gamma \vdash \phi, \Delta}\ \text{RW-RIGHT}$$

$$\text{if } \Gamma \cup \{\neg\psi \mid \psi \in \Delta\} : \phi \rightarrowtriangle \phi'$$

The lower half of Fig. 4 shows three of the bit-vector-specific rules. Rule BOUND-RW defines elimination of bmod_a^b-predicates that do not require any case splits; the definition of the rule assumes functions $\mathit{lbound}(\Pi, s)$ and $\mathit{ubound}(\Pi, s)$ that derive lower and upper bounds of a term s, respectively, given the current context Π. The two functions can be implemented by collecting inequalities (and possibly type information available for predicates) in Π to obtain an over-approximation of the range of s.

Rule COEFF-RW reduces coefficients in $\mathit{bmod}_a^b(s, r)$ by adding a multiple of the modulus $b - a$ to s. The rule assumes a well-founded order $\prec$ on terms to prevent cycles during simplification. One way to define such an order is to choose a total well-founded order $\prec$ on the union $C \cup X$ of variables and constants, extend $\prec$ to expressions $\alpha \cdot x$ by sorting coefficients as $0 \prec 1 \prec -1 \prec 2 \prec \cdots$, and finally extend $\prec$ to arbitrary terms $\alpha_1 t_1 + \cdots + \alpha_n t_n$ as a multiset order [19].

The same order $\prec$ is used in BMOD-RW, defining how $\mathit{bmod}_a^b(s, r)$ can be rewritten in the context of a second literal $\mathit{bmod}_{a'}^{b'}(s', r')$. The rule is useful to optimise the translation of nested bit-vector operations. Assuming $\mathit{bmod}_{a'}^{b'}(s', r')$, the value of $s' - r'$ is known to be a multiple of $b' - a'$, and therefore $k \cdot (s' - r')$ is a multiple of $b - a$ provided that $b - a$ divides $k \cdot (b' - a')$. This implies that the truth value of $\mathit{bmod}_a^b(s, r)$ is not affected by adding $k \cdot (s' - r')$ to s.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} * \\ \vdots \\ \ldots, a \geq 1, e < 2^{32}, b \leq c_1, d \geq 2^{32}, e - d - c_1 + b \geq 0 \ \vdash \end{array}}{\ldots, \times(a,b,d), \times(a,c_1,e), a \geq 1, e < 2^{32}, b \leq c_1, d \geq 2^{32} \ \vdash}\ \times\text{-CROSS}}{\ldots, 0 \leq d, d \leq 2^{64} - 2^{33} + 1, d = c_2 \ \vdash}\ (b)}{\ldots, 0 \leq d, d \leq 2^{64} - 2^{33} + 1, \mathit{ubmod}_{64}(d, c_2) \ \vdash}\ \text{RW-LEFT}}{\ldots, \mathit{in}_{32}(a), \mathit{in}_{32}(b), \times(a,b,d), \mathit{ubmod}_{64}(d, c_2) \ \vdash}\ \times\text{-ICP}}{\vdash \ \phi_{core}}\ (a)$$

Fig. 5. Proof tree for Example 5, with the sequences (a) and (b) of rule applications not shown in detail.

Our implementation uses various further simplification rules, for instance to eliminate $\times$ or bmod_a^b whose result is never used; we skip those for lack of space.

Example 4: The expression $\mathsf{bvadd}_{32}(\mathsf{bvadd}_{32}(a, b), c)$ corresponds to $\mathit{ubmod}_{32}(a + b, r_1) \wedge \mathit{ubmod}_{32}(r_1 + c, r_2)$ in the core language. Using BMOD-RW, the formula can be rewritten to $\mathit{ubmod}_{32}(a + b, r_1) \wedge \mathit{ubmod}_{32}(a + b + c, r_2)$, provided that $a + b + c \prec r_1 + c$.

Example 5: We continue Ex. 3 and show that ϕ_{core} is valid, focusing on the $a \geq 1$ case of bvudiv_{32}. The proof (Fig. 5) consists of three core steps: 1) using $\times$-ICP, from the constraints $\mathit{in}_{32}(a)$, $\mathit{in}_{32}(b)$, $\times(a, b, d)$ the inequalities $0 \leq d$ and $d \leq 2^{64} - 2^{33} + 1$ can be derived; 2) therefore, using RW-LEFT and BOUND-RW, the literal $\mathit{ubmod}_{64}(d, c_2)$ can be rewritten to $d = c_2$, capturing the fact that 64-bit multiplication cannot overflow for unsigned 32-bit operands; 3) using $\times$-CROSS, from the inequalities $a \geq 1$ and $b \leq c_1$ and the products $\times(a, b, d)$, $\times(a, c_1, e)$ we can derive $e - d - c_1 + b \geq 0$. The proof branch can then be closed using standard arithmetic reasoning. The implementation of our procedure can easily find the outlined proof automatically.

C. Splitting Rules for bmod_a^b

In general, formulas will of course also contain occurrences of bmod_a^b that cannot be eliminated just by simplification. We introduce two calculus rules for reasoning about such general literals $\mathit{bmod}_a^b(s, r)$. The first rule makes the assumption that lower and upper bounds of s are available, and are reasonably tight, so that an explicit case analysis can be carried out; the rule generalises BOUND-RW to the situation in which the factors l, u do not coincide:

$$\frac{\left\{\Gamma, a \leq r < b, s = r + i \cdot (b-a) \ \vdash \ \Delta\right\}_{i=l}^{u}}{\Gamma, \mathit{bmod}_a^b(s, r) \ \vdash \ \Delta}\ \text{BMOD-SPLIT}$$

assuming $\left\lfloor \frac{\mathit{lbound}(\Pi,s)-a}{b-a} \right\rfloor = l$ and $\left\lfloor \frac{\mathit{ubound}(\Pi,s)-a}{b-a} \right\rfloor = u$ with $\Pi = \Gamma \cup \{\neg\psi \mid \psi \in \Delta\}$.

If the bounds l, u are too far apart, the number of cases created by BMOD-SPLIT would become unmanageable, and it

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE I

Comparison of Eldarica configurations and CPAchecker. For each family, the table shows the number of safe/unsafe results, the average time, the required number of CEGAR iterations, and the average size of computed interpolants for Eldarica.

Categories	Total	Eldarica math				Eldarica ilp32				CPAchecker -32		
		Solved	Time	Iter.	P. Size	Solved	Time	Iter.	P. Size	Solved	Time	Iter.
All	551	293	21.0	11.1	1.0	217	28.0	13.6	1.4	180	30.6	28.5
		101	73.4	31.8	1.0	117	49.7	21.7	1.2	168	48.6	3.9
HOLA	46	44	11.4	8.9	1.1	21	11.0	5.8	2.0	12	84.1	87.4
		0				4	6.0	0.0	1.3	4	11.4	0.0
llreve	21	16	13.1	16.1	1.1	8	17.4	27.3	1.6	7	26.5	75.7
		5	7.4	7.6	1.1	4	8.5	5.8	1.1	5	37.3	7.0
VeriMAP	155	132	5.8	2.3	1.0	100	5.9	3.6	1.1	87	12.2	18.5
		21	8.4	4.4	1.0	41	11.6	2.4	1.5	33	24.8	1.3
SVCOMP	329	101	46.1	22.9	1.0	88	58.1	25.7	1.3	74	44.0	26.3
		75	96.0	41.1	1.0	68	77.7	35.5	1.1	126	56.5	4.5

is better to choose a direct encoding of the remainder operation in Presburger arithmetic:

$$\frac{\Gamma, a \le r < b, s = r + (b-a)\cdot c \ \vdash\ \Delta}{\Gamma, \mathit{bmod}_a^b(s,r)\ \vdash\ \Delta}\ \textsc{bmod-const}$$

where c is assumed to be a fresh constant. Rule BMOD-CONST corresponds to the encoding chosen in [15].

In practice, it turns out to be advantageous to prioritise rule BMOD-SPLIT over BMOD-CONST, as long as the number of cases does not become too big. This is because each of the premises of BMOD-SPLIT tends to be significantly simpler to solve (and interpolate) than the conclusion; in addition, splitting one bmod_a^b literal often allows subsequent simplifications that eliminate other bmod_a^b occurrences.

Example 6: We consider one of the examples from [15], the interpolation problem $A \wedge B$ defined by

$$A = \neg \mathsf{bvule}_8(\mathsf{bvadd}_8(y_4, 1), y_3) \wedge y_2 = \mathsf{bvadd}_8(y_4, 1)$$
$$B = \mathsf{bvule}_8(\mathsf{bvadd}_8(y_2, 1), y_3) \wedge y_7 = 3 \wedge y_7 = \mathsf{bvadd}_8(y_2, 1)$$

where all variables range over unsigned 8-bit bit-vectors. An eager encoding into LIA would typically add variables to handle wrap-around semantics, e.g., mapping $y_4' = \mathsf{bvadd}_8(y_4, 1)$ to $y_4' = y_4 + b1 - 2^8\sigma_1 \wedge 0 \le y_4' < 2^8 \wedge 0 \le \sigma_1 \le 1$. Additional variables tend to be hard for interpolation, and the LIA interpolant presented in [15] is the formula $I_{LIA} = -255 \le y_2 - y_3 + 256\lfloor -1\frac{y_2}{256}\rfloor$; the formula can be mapped back to a pure bit-vector formula if needed.

We outline how our calculus proves the unsatisfiability of $A \wedge B$. Translation of the formulas to the core language gives:

$$A_{\text{core}} = \begin{array}{l}\psi_A \wedge \mathit{ubmod}_w(y_4 + 1, c_1) \wedge \\ c_1 > y_3 \wedge y_2 = c_1\end{array}$$

$$B_{\text{core}} = \begin{array}{l}\psi_B \wedge \mathit{ubmod}_w(y_2 + 1, c_2) \wedge \\ c_2 \le y_3 \wedge y_7 = 3 \wedge y_7 = c_2\end{array}$$

where $\psi_A = \mathit{in}_8(y_2) \wedge \mathit{in}_8(y_3) \wedge \mathit{in}_8(y_4) \wedge \mathit{in}_8(c_1)$ and $\psi_B = \mathit{in}_8(y_2) \wedge \mathit{in}_8(y_3) \wedge \mathit{in}_8(y_7) \wedge \mathit{in}_8(c_2)$ are the domains. The main reasoning step is application of the rule BMOD-SPLIT to $\mathit{ubmod}_w(y_2 + 1, c_2)$, using the bounds $\mathit{lbound}(\Pi, y_2 + 1) = 4$ and $\mathit{ubound}(\Pi, y_2 + 1) = 256$ that follow from $A_{\text{core}}, B_{\text{core}}$:

$$\frac{\begin{array}{c}\ldots, 0 \le c_2 < 256, y_2 + 1 = c_2 \ \vdash \\ \ldots, 0 \le c_2 < 256, y_2 + 1 = c_2 + 256 \ \vdash\end{array}}{\ldots, \mathit{ubmod}_w(y_2 + 1, c_2)\ \vdash}\ \textsc{bmod-split}$$

Due to $y_7 = 3 \wedge y_7 = c_2$, the cases reduce to $y_2 = 2$ and $y_2 = 258$, and immediately contradict $A_{\text{core}}, B_{\text{core}}$.

D. Quantifier Elimination and Craig Interpolation

Since the bit-vector rules in this section are all equivalence transformations, QE for bit-vectors can be done exactly as described in Section III-B. As the ranges of all symbols are now bounded, it is guaranteed that any formula will eventually be reduced to Presburger arithmetic, so that we obtain complete QE for (arithmetic) bit-vector constraints.

Similarly, the interpolation approach from Section III-C carries over to bit-vectors, with theorem axioms being generated for each of the rules defined in this section. Since the translation of bit-vector formulas to the core language happens upfront, also interpolants are guaranteed to be in the core language, and can be mapped back to bit-vector formulas if necessary (e.g., as in [15]). Interpolants might contain quantifiers, in which case QE can be applied (as described in the first paragraph), so that we altogether obtain a complete procedure for quantifier-free interpolation of arithmetic bit-vector formulas. For interpolation problems from software verification, it happens rarely, however, that QE is needed.

In our implementation, we restrict the use of the simplification rules RW-LEFT and RW-RIGHT when computing proofs for the purpose of interpolation. Unrestricted use could quickly mix up the vocabularies of the individual partitions in an interpolation problem $A \wedge B$, and thus increase the likelihood of quantifiers in interpolants. Instead we simplify A, B separately upfront using rules in Fig. 4, and apply RW-LEFT, RW-RIGHT only when the modified formula ϕ is a literal.

Example 7: We continue Example 6, and show how our calculus finds the simpler interpolant $I'_{LIA} = y_3 < y_2$ for the interpolation problem $A \wedge B$. The core step is to turn the

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE II
PERFORMANCE ON SMT-LIB BV AND QF_BV PROBLEMS. FOR EACH FAMILY, THE FIRST/SECOND ROW GIVES SAT/UNSAT PROBLEMS.

Category	PRINCESS		Z3		CVC4	
	Total	Time	Total	Time	Total	Time
Automizer	16	158.2	16	0.1	14	0.1
	127	215.1	137	0.0	137	0.3
keymaera	5	268.6	108	6.9	34	1.0
	3771	2.5	3923	0.3	3921	0.1
psyco	2	2.5	132	0.1	132	1.5
	3	141.6	62	0.2	62	0.5
tptp	15	2.3	17	0.0	17	0.0
	54	1.7	56	0.0	56	0.0
RND	2	40.8	40	6.9	25	40.7
	5	188.5	28	6.7	22	13.2
RNDPRE	2	7.4	20	19.0	22	26.9
	14	53.9	36	14.1	26	29.3
model	16	1.9	144	0.0	73	10.8
	0		0		0	
Heizmann	13	49.8	15	37.8	18	18.1
	27	155.5	17	50.7	108	8.3
ranking	0		34	4.4	32	1.5
	5	12.0	19	19.5	13	0.4
fixpoint	25	94.9	36	0.5	54	14.2
	26	85.0	73	0.6	75	2.3
QFBV	334	2.3	2701	11.6	2632	17.4
	164	16.4	1967	29.7	1919	19.3

application of BMOD-SPLIT into an explicit axiom; after slight simplifications, this axiom is:

$$Ax = \begin{array}{l}\big(\mathit{ubmod}_w(y_2+1, c_2) \wedge 3 \le y_2 < 256 \wedge \mathit{in}_8(c_2)\big) \to \\ \big(y_2+1 = c_2 \vee y_2 + 1 = c_2 + 256\big)\end{array}$$

The axiom mentions all assumptions made by the rule, including the bounds $3 \le y_2 < 256$ that determine the number of resulting cases (or, alternatively, the formulas $c_1 > y_3, y_2 = c_1, c_2 \le y_3, y_7 = 3, y_7 = c_2$ from which the bounds derive). The axiom also includes domain constraints like $\mathit{in}_8(c_2)$ for occurring symbols, which later ensures that possible quantifiers in interpolants range over bounded domains. The quantified axiom is $QAx = \forall y_2, c_2.\, Ax$, and can be used to construct an interpolating proof:

$$\dfrac{\begin{array}{c}*\\ \vdots\\ \begin{array}{l}\lfloor c_1 > y_3\rfloor_L, \lfloor y_2 = c_1\rfloor_L, \lfloor c_2 \le y_3\rfloor_R,\\ \lfloor y_7 = 3\rfloor_R, \lfloor y_7 = c_2\rfloor_R, \lfloor y_2 + 1 = c_2\rfloor_R\end{array} \vdash \emptyset \blacktriangleright y_3 < y_2\end{array}}{\mathcal{P}}$$

$$\dfrac{\dfrac{\cdots \quad \mathcal{P} \quad \cdots}{\lfloor A_{\text{core}}\rfloor_L, \lfloor B_{\text{core}}\rfloor_R, \lfloor Ax\rfloor_R \vdash \emptyset \blacktriangleright y_3 < y_2}\ \vee\text{-LEFT}_R}{\lfloor A_{\text{core}}\rfloor_L, \lfloor B_{\text{core}}\rfloor_R, \boxed{\lfloor QAx\rfloor_R} \vdash \emptyset \blacktriangleright y_3 < y_2}\ \forall\text{-LEFT}_R$$

We only show one of the cases, $\mathcal{P}$, resulting from splitting the axiom $\lfloor Ax\rfloor_R$ using the rules from Fig. 2. The final interpolant $y_3 < y_2$ records the information needed from A_{core} to derive a contradiction in the presence of $y_2 + 1 = c_2$; the branch is closed using standard arithmetic reasoning [10].

V. EXPERIMENTS

We have implemented the procedures in the PRINCESS theorem prover. PRINCESS also partly supports operators like shift and bit-wise and/or. All experiments were done using PRINCESS version 2018-05-25 on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime was limited to 10min wall clock time, and heap space 2GB.

a) SAT Checking on BV and QF_BV Problems: Results on SMT-LIB benchmarks are given in Table II. We compare our implementation with Z3 4.8.0 and CVC4 1.6. Our procedure can solve a similar number of problems as Z3 and CVC4 on many of the BV families. Although our procedure is not specifically designed for QF_BV, we include overall numbers for completeness (excluding the families ASP and Sage). However, the overwhelming majority of the QF_BV benchmarks contains bit-wise operations not fully supported by PRINCESS yet. QF_BV families on which our procedure does well include Example 3 and the PSPACE family.

b) Verification of C Programs: Since it is difficult to compare interpolation procedures outside of an application, we present results of running the ELDARICA version 2.0-alpha3 model checker[5] on a benchmark set of 551 C programs, using the implementation of our calculus in PRINCESS as interpolation procedure (Table I). The benchmarks are the programs used in [31] for evaluating different predicate generation strategies. The programs use only arithmetic operations, no arrays or heap data structures. For this paper, we interpret the programs as operating either on the mathematical integers (*math*), or on signed 32-bit bit-vectors (*ilp32*) with wrap-around semantics. Both configurations were running a parallel portfolio of two interpolation strategies (ELDARICA option `-abstractPO`): straightforward interpolation to compute predicates, and the interpolation abstraction technique [32]. The experiments show that our interpolation approach for bit-vectors can solve almost as many programs as the existing interpolation methods for mathematical integers, with a similar number of CEGAR iterations, and with interpolants of comparable size. The scatter plot in Fig. 6 indeed shows very similar runtimes for the two configurations.

As comparison, we also ran CPACHECKER 1.7 [33] on the benchmarks, using options `-predicateAnalysis -32` and MATHSAT as solver; MATHSAT uses the interpolation method from [15]. As can be seen in Table I, our method is competitive with CPACHECKER on all considered families, in particular for the safe programs. We remark, however, that we are comparing different verification systems here. Although both ELDARICA and CPACHECKER apply CEGAR and interpolation, there are many factors affecting the results.

VI. CONCLUSIONS

We have presented a new calculus for Craig interpolation and quantifier elimination in bit-vector arithmetic. While the experimental results in model checking are already promising, we believe that there is still a lot of room for extension and

---

[5] https://github.com/uuverifiers/eldarica

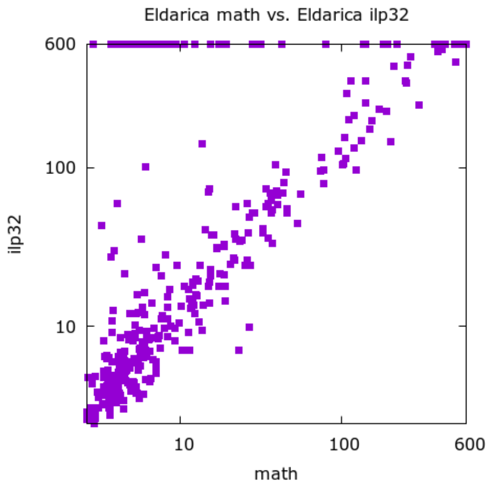


Fig. 6. Scatter plot comparing runtime of math and ilp32 semantics on the C benchmarks.

improvement of the approach. This includes more powerful propagation and simplification rules, and more sophisticated strategies to apply the splitting rules ×-SPLIT and BMOD-SPLIT. Future work also includes the extension of our calculus to bit-wise operations like bvand, bvor, or bvxor, for which we plan to add further uninterpreted predicates to our setting to preserve laziness as far as possible.

Acknowledgements: We thank the reviewers for helpful comments. This work was supported by the Swedish Research Council (VR) under grant 2014-5484, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

REFERENCES

[1] K. L. McMillan, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, 2005.

[2] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *VMCAI*, ser. LNCS. Springer, 2010.

[3] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli, "Ground interpolation for the theory of equality," in *TACAS*, ser. LNCS. Springer, 2009.

[4] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "Beyond quantifier-free interpolation in extensions of Presburger arithmetic," in *VMCAI*, ser. LNCS. Springer, 2011, pp. 88–102.

[5] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 413–427.

[6] L. Kovács and A. Voronkov, "Interpolation and symbol elimination," in *CADE*, 2009, pp. 199–213.

[7] M. P. Bonacina and M. Johansson, "On interpolation in automated theorem proving," *J. Autom. Reasoning*, vol. 54, no. 1, pp. 69–97, 2015.

[8] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *SIGSOFT'06/FSE-14*. New York, NY, USA: ACM, 2006, pp. 105–116.

[9] L. Dai, B. Xia, and N. Zhan, "Generating non-linear interpolants by semidefinite programming," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 364–380.

[10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "An interpolating sequent calculus for quantifier-free Presburger arithmetic," *Journal of Automated Reasoning*, vol. 47, pp. 341–367, 2011.

[11] A. Griggio, T. T. H. Le, and R. Sebastiani, "Efficient interpolant generation in satisfiability modulo linear integer arithmetic," *Logical Methods in Computer Science*, vol. 8, no. 3, 2010.

[12] R. Bruttomesso, S. Ghilardi, and S. Ranise, "Quantifier-free interpolation of a theory of arrays," *Logical Methods in Computer Science*, vol. 8, no. 2, 2012.

[13] N. Totla and T. Wies, "Complete instantiation-based interpolation," *J. Autom. Reasoning*, vol. 57, no. 1, pp. 37–65, 2016.

[14] J. Hoenicke and T. Schindler, "Efficient interpolation for the theory of arrays," *CoRR*, vol. abs/1804.07173, 2018.

[15] A. Griggio, "Effective word-level interpolation for software verification," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 28–36.

[16] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *TACAS*, ser. LNCS, vol. 7795, 2013.

[17] D. Kroening and G. Weissenbacher, "Lifting propositional interpolants to the word-level," in *FMCAD*. IEEE Computer Society, 2007, pp. 85–89.

[18] ——, "An interpolating decision procedure for transitive relations with uninterpreted functions," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 6405. Springer, 2009, pp. 150–168.

[19] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *LPAR*, ser. LNCS, vol. 5330. Springer, 2008, pp. 274–289.

[20] M. C. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd ed. Springer-Verlag, New York, 1996.

[21] J. Y. Halpern, "Presburger arithmetic with unary predicates is Π^1_1 complete," *Journal of Symbolic Logic*, vol. 56, 1991.

[22] 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*, vol. 53, no. 6, pp. 937–977, 2006.

[23] A. Reynolds, T. King, and V. Kuncak, "Solving quantified linear arithmetic by counterexample-guided instantiation," *Formal Methods in System Design*, vol. 51, no. 3, pp. 500–532, 2017.

[24] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, September 1957.

[25] S. Lang, *Algebra (3. ed.)*. Addison-Wesley, 1993.

[26] B. Buchberger, "An algorithm for finding the basis elements in the residue class ring modulo a zero dimensional polynomial ideal," Ph.D. dissertation, 3 2006.

[27] P. Van Hentenryck, D. McAllester, and D. Kapur, "Solving polynomial systems using a branch and prune approach," *SIAM J. Numer. Anal.*, vol. 34, no. 2, pp. 797–827, Apr. 1997.

[28] J. Warren A. Hunt, R. B. Krug, and J. S. Moore, "Linear and nonlinear arithmetic in ACL2." in *Proceedings, Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference*, ser. LNCS, vol. 2860. Springer, 2003, pp. 319–333.

[29] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT modulo linear arithmetic for solving polynomial constraints," *J. Autom. Reasoning*, vol. 48, no. 1, pp. 107–131, 2012.

[30] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[31] Y. Demyanova, P. Rümmer, and F. Zuleger, "Systematic predicate abstraction using variable roles," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. Lecture Notes in Computer Science, C. Barrett, M. Davies, and T. Kahsai, Eds., vol. 10227, 2017, pp. 265–281.

[32] J. Leroux, P. Rümmer, and P. Subotic, "Guiding craig interpolation with domain-specific abstractions," *Acta Inf.*, vol. 53, no. 4, pp. 387–424, 2016.

[33] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," *CoRR*, vol. abs/0902.0019, 2009.

Analyzing the Fundamental Liveness Property of the Chord Protocol

Julien Brunel
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
julien.brunel@onera.fr

David Chemouil
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
david.chemouil@onera.fr

Jeanne Tawa
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
jeanne.tawa@onera.fr

***Abstract*—Chord is a protocol that provides a scalable distributed hash table over an underlying peer-to-peer network. Since it combines data structures, asynchronous communications, concurrency, and fault tolerance, it features rich structural and temporal properties that make it an interesting target for formal specification and verification. Previous work has mainly focused on automatic proofs of safety properties or manual proofs of the full correctness of the protocol (a liveness property). In this paper, we report on analyzing automatically the correctness of Chord with the Electrum language (developed in former work) on small instance of networks. In particular, we were able to find various corner cases in previous work and showed that the protocol was not correct as described there. We fixed all these issues and provided a version of protocol for which we were not able to find any counterexample using our method.**

***Index Terms*—Chord protocol, distributed systems, formal specification and verification, Electrum**

I. INTRODUCTION

Peer-to-peer systems are distributed systems without hierarchical organization or centralized control. They are an alternative to the traditional client-server model and enjoy interesting properties in terms of scalability, robustness and cost. Chord [1]–[3] is a one of the most popular peer-to-peer systems. It is a protocol and algorithm for a peer-to-peer distributed hash table (DHT). A DHT stores key-value pairs by assigning keys to different nodes (basically computers) in the network. Chord addresses the efficient and robust localization of data in such a network. When Chord was initially presented, three main qualities were highlighted: its simplicity, its provable performance and its provable correctness. Although the first two claims are true, proving the Chord correctness turns out to be a hard task, as showed by numerous works by P. Zave [4]–[8].

In Chord, each node has an identifier and can reach other nodes using pointers to other identifiers. The nodes and their pointers form a topology which is essential to ensure the correct localization of data in the network. Because of the fact autonomous nodes may join or leave the network (or fail) at any time, the topology is always evolving. A key aspect of the Chord protocol consists in the definition of maintenance operations that are in charge of repairing the network topology so that the data stored in any node keeps being reachable from any other node, despite failures, joins and departures.

Thus, the correctness of Chord deals with the network topology. In fact, the nodes and their successor pointers have to form a ring, so that each node is accessible from any other node. Since nodes can join and leave the network, the ring topology cannot always be ensured. That is why the the correctness property of Chord is expressed as follows: *if, from a certain instant, there is no subsequent join, departure or failure, then the network is ensured to recover a ring topology eventually, and keep it*. So, the correctness of Chord is not only about the structure of the system, but also about its temporal evolution: it is in fact a liveness property. This twofold nature is one of the reasons for the hardness to prove Chord correctness.

We recently developed Electrum [9], a specification language based on First-Order Linear Temporal Logic, with which both structural and temporal properties can easily be defined and checked. The Electrum language is inspired by Alloy [10] for its structural concepts and by Linear Temporal Logic [11] for its temporal concepts.

In this article, we propose a formal description of the Chord protocol in Electrum and focus on proving its correctness. We show the following benefits of our approach:

- the Electrum ability to deal with structural aspects makes the specification of the network topology straightforward;
- the Electrum ability to deal with temporal aspects fits with the specification of the network evolution (throughout the execution of the maintenance operations) and makes the specification of the correctness property, which is a liveness property, direct;
- the automatic verification of the full correctness property is performed for the first time (only for a limited number of nodes though)
- thanks to the quick feedback to the user, we have been able to the detect several shortcomings and corner cases in the previous formalization of the protocol, and to clearly identify temporal hypotheses on the ordering of the maintenance operations (fairness properties) that are necessary to ensure the correctness.

The rest of this paper is structured as follows. In Sect. II we briefly present the Chord protocol. In sect. III, we give an overview of Electrum, and formalize Chord in sect. IV. In Sect. V, we evaluate the formal verification of our Chord

ISBN: 978-0-9835678-8-2.

model. In Sect. VI, we highlight important aspects of our study and compare to related work. We then conclude in Sect. VII.

II. The Chord Protocol

Chord is a distributed lookup protocol which addresses an essential issue of peer-to-peer applications: the efficient localization of the network node that stores the desired data. An important quality that probably explains the popularity of Chord is its simplicity. Indeed, Chord makes no use of synchronization or timing constraints on distributed nodes, and each atomic operation involves a single node. As claimed by the authors, this simplicity makes Chord easy to implement and extend. Other interesting features of Chord are its provable performance and its scalability. However, contrary to another claim, proving the correctness of Chord. *i.e.*, the reachability of the data, is not an easy task.

A. *The Network Structure*

In a Chord network, each node has an identifier (the m-bit hash of its IP address). Pairs of keys and associated data are stored in nodes. Every node has a *successor list* of pointers to other nodes. We refer to the first element of this list as the *successor*. The goal of having a list of successors instead of a single one is to be robust to the failures: if a node leaves the network, its predecessor still has successors in the network. Besides, each node also has a pointer to its *predecessor*. This is useful in the execution of the Chord maintenance operations.

When a network is structured as a ring according to the relation induced by the *successor* pointers and when the order of identifiers complies with the order of the *successor* pointers, then each node is accessible from any other node, *i.e.* any data is accessible from any node. We say that such a network is in an *ideal* state.

Since nodes can join and leave the network at any time, the ring structure cannot be continuously ensured. For instance, nodes joining a ring create an appendage. The maintenance operations aim to recover a ring structure eventually, despite the fact nodes join and leave the network.

B. *Network Properties*

The authors of Chord have provided explicit properties of the network that ensure correct data delivery [2]. They define in particular the *ideal state* of a network, which we have introduced informally in the previous section, and a temporary imperfect state, which we call a *valid state* following [4]. As our study only deals with the correctness of the protocol, we do not present the quantitative and probabilistic properties mentioned in the original Chord articles.

Let us first present some notations and preliminary definitions. In the following, we will denote the successor (resp. predecessor) of a node n by $n.\text{SUCCESSOR}$ (resp. $n.\text{PREDECESSOR}$). A Chord network is *locally consistent* if, for any node n, we have $(n.\text{SUCCESSOR}).\text{PREDECESSOR} = n$. A Chord network is *globally consistent* if, for each node n_1, there is no node n_2 in the same ring as n_1 such that $n_1 < n_2 < n_1.\text{SUCCESSOR}$. A Chord network is *loopy* if it is locally consistent but globally inconsistent.

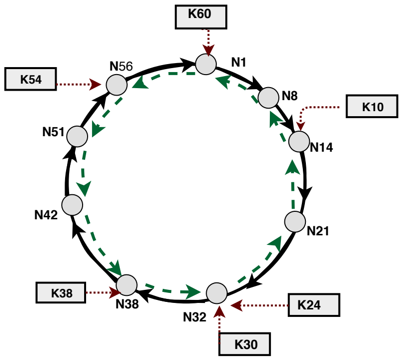


Fig. 1. A Chord network in an ideal state (successor pointers are shown as a bold arrows, predecessor pointers as dashed arrows and key/node mappings as dotted arrows).

Definition 1: A Chord network is in an *ideal* state if:

- *ring:* the successor relation forms a single ring of nodes (every node is in the ring);
- *non-loopiness:* the ring is locally and globally consistent;
- *successor list validity:* the successor list (of size k) of each node n contains the first k nodes that follow n in the ring.

Fig. 1 shows a Chord network in an ideal state, with nine nodes and storing six key-data pairs (we only represent the keys). Each key is stored in the node with the least identifier among the nodes having a greater identifier than the key. For example, key K10 is stored in node N14.

As explained above, joins and fails of nodes force the network in a non-ideal state. But the maintenance operations of Chord aim at recovering from such non-ideal states.

In order to characterize these non-ideal states, we introduce the notion of *valid* states (following [2] and [4]) which allow some nodes not to be in the ring, but in *appendages* of the ring. For a node n in the ring, there may be a tree of nodes rooted at n, consisting of nodes that have recently joined the network and are not yet in the ring. We refer to this tree as n's appendage and denote it A_n.

Definition 2: A Chord network is in a *valid* state if:

- *connectivity:* a subset of nodes form a ring following the successor relation (there is only one such ring), the rest of the nodes are part of appendages, which are connected to the ring;
- *non-loopiness:*
  - the ring is non-loopy;
  - and for every node n' in an appendage A_n, the path of successors from n' to n is increasing (in the sense of the identifier order).
- *successor list validity:*
  - if n is in the ring, then $n.\text{SUCCESSOR}$ is the first live ring node following n (according to the identifier order);

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

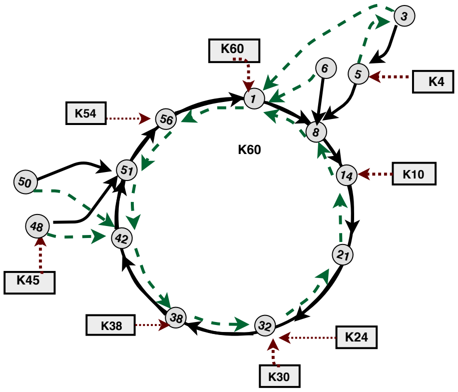


Fig. 2. A Chord network in a valid state.

- if n' is in appendage A_n, then n is the first live ring node following n' (according to the identifier order);
- if the successor list of n.SUCCESSOR skips over a live node n', then n' is not in n successor list.

Fig. 2 shows a Chord network in a validstate.

From these two definitions, the correctness of the Chord protocol can be expressed as follows.

Correctness of the Chord protocol: Starting from a network that is initially valid, in any execution state, if there are no subsequent join or fail events, then the network will eventually become ideal and remain ideal.

C. Chord Events

The operations of the Chord protocol consist of four events (`join`, `fail`, `stabilize` and `rectify`) each of which changes the state of at most one node. A `join` operation occurs when a node joins the network. We then refer to this node as a *member*, or a *live* node. When a node joins the network, it contacts a network member and takes the successor list of this member as its own successor list. It also considers this member as its predecessor. Diagrams (a) and (b) in Fig. 3 show successor and predecessor pointers in a network where node 6 joins by contacting node 8.

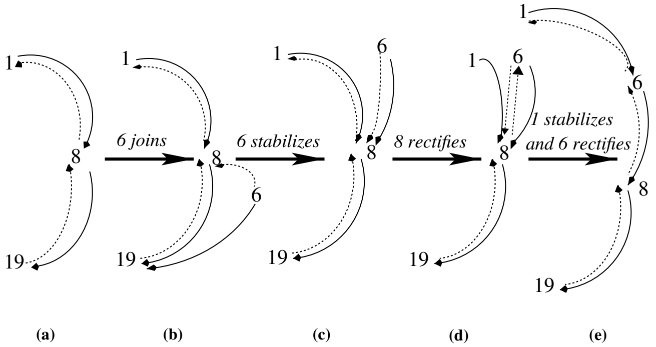


Fig. 3. Chord Events: predecessor pointers are shown as dashed arrows and successor pointers as solid arrows

A `join` event may break the ideal property of the network. In order to recover, every node performs a `stabilize` operation periodically. When a member *stabilizes*, it contacts its successor and asks it about its predecessor identifier. If the predecessor identifier is a better candidate for being its successor than its current successor (according to the identifier order) then it takes this predecessor as its new successor. In diagram (c) of Fig. 3, node 6 stabilizes and takes node 8 as its new successor. The contact a node establishes with its successor during stabilization is also an opportunity to update its full successor list with information from its successor.

After stabilization, the stabilized node *notifies* its successor about its identity. The notified member then executes a `rectify` operation. A notified member must adopt the notifying member as its new predecessor if the notifying member is closer to itself than its current predecessor, or if its current predecessor is dead (see below). Diagram (d) in Fig. 3 shows a `rectify` operation made by node 8 after the notification by node 6.

Finally, a node can leave the network in case of a `fail` operation. Such a node is no longer a member and is referred to as a *dead* node. It obviously does not inform any other node about its departure and still appears in the successor list of other nodes.

a) Operating assumptions: Chord relies on an important assumption, which states that each member always has at least one live successor. In practice, this depends on the size of the successor list, and on the ratio between the occurrence of maintenance operations and the occurrence of failures. For instance, let us suppose that a given live node n has a successor list of size 3, and that all three successors of n fail before any stabilisation and rectification occur, then the network is no longer in a valid state (the ring structure is broken) and the protocol is not able to recover from such a situation.

Another assumption is the perfect communication between a node and its successor, in the sense that each node necessarily answers a query in bounded time. This allows for perfect detection of failures (a successor that does not answer a query before a deadline is considered dead).

III. ELECTRUM IN A NUTSHELL

Electrum [9] is a dynamic extension of Alloy [10] based upon Linear Temporal Logic (LTL). It preserves the flexibility of Alloy while easing the specification of behavioral properties and enabling verification on traces with a bounded or an unbounded number of states.

In Electrum, as in classic class-based modelling, structure is introduced through the declaration of *signatures*, which denote sets of indivisible, immutable and uninterpreted *atoms*; and *fields* (between signatures) that denote flat n-ary relations between sets. Signatures and fields may be constrained by simple *multiplicity* constraints.

Unlike in Alloy however, fields and signatures may either be declared as *static* (by default) or *variable*: the former hold the same valuation throughout a given time trace, while the latter are mutable and hence may see their valuation change at every step of a trace.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

If needed, more constraints may be imposed on a specification as *facts*, which are just axioms (*i.e.* statements that every instance of the specification conforms to).

Constraints (formulas) are expressed in a logic comprising both connectives (and quantifiers) of First-Order Logic (FOL) and LTL, with relational expressions as a term language. The latter are built by composing signatures and fields with common set-theoretic operators and relational operators such as the join `.` of two relations or the transitive closure `^` of a relation. Moreover, every relational expression may be primed, referring thus to its valuation in the succeeding state. For ease of specification, parameterized, named expressions and constraints may also be introduced as functions and (resp.) predicates.

Analysis instructions consist of `run` and `check` commands restricted by *scopes* that determine the maximum number of atoms that will be considered for every signature. A `run` instructs the Analyzer to search for an instance (a model, in the logical sense) satisfying a given predicate; while a `check` instructs the Analyzer to prove a given assertion (introduced with the `assert` keyword) valid *in the given scope*.

Electrum Analyzer[1], an extension of Alloy Analyzer, offers two alternative model-checking techniques: the first implements *bounded model-checking* (BMC) [12], [13] over Alloy itself, thus bounding the number of states in a trace (this is expressed with a bound over a fake `Time` signature). The second one relies on the compilation to the NuSMV [14] and nuXmv [15] model-checkers, relying on *unbounded* model-checking (UMC) algorithms.

IV. FORMALIZATION OF THE PROTOCOL

We now present the main aspects of the formalization of the Chord protocol[2], taking inspiration in both the presentation of Chord in [2], referred to as PODC in the rest of this section, and in P. Zave's recent work [4].

A. Data Structures

The main concept in our model is that of a *node*, which corresponds to a Chord node identifier (we conflate Chord nodes, their IP address, and their identifier). As explained before, node (identifiers) are ordered totally.

Recall that a node also maintain a list of successors: its purpose is to recover from failures, and its length defines a threshold for fault tolerance for Chord. To ensure that each node always remains connected to the network after a failure, the minimum length of this list is 2. For the sake of readability, we only show a model with successor list of size 2. Actually, we "unfold" this list and represent it as the datum of two fields `fst` ("first") and `snd` ("second"). We made this choice because using lists here would make the use of *explicit* quantification over all possible lists necessary, a fact that is easily overlooked and that, more importantly, is costly in terms of space.

Finally, to ensure maintenance operations, each node also holds a pointer `prdc` to its predecessor in the network. These three fields may mutate, depending on various events happening in the network, hence they are marked as **var**iable. Technically, each of this field denotes a partial function from nodes to nodes, which is specified using the `lone` multiplicity (meaning "0 or 1"):

```
open util/ordering[Node] // total ordering on nodes
sig Node {
  var fst, snd, prdc: lone Node,
  var todo: Status → Node }
```

The `todo` field, also present in the declaration, represents *pending operations* that the node will have to perform over another node (hence this field denotes a ternary relation): its use will be detailed later. There are two kinds of such operations, described by a so-called *status* (its formalization is a way of saying that it is an enumeration):

```
abstract sig Status {}
one sig Stabilizing, Rectifying extends Status {}
```

A node is a *member* of the Chord network if its successor pointers effectively point to some nodes (*i.e.* the pointers are not null). This is neatly expressed by introducing a *variable subset signature* that takes its elements among nodes but the valuation of which may change at every instant[3]:

```
var sig members in Node {}
fact membersDef {
  always members = {n: Node | some n.fst && some n.snd}}
```

Now, at every instant, the successor of a node is the first living node among its successors. We specify this as a partial function `succ` which states that the successor of a node is its `fst` field if this field is a member, and its `snd` field otherwise.

```
fun succ: Node → lone Node {
  { m1, m2: members | m1.fst in members ⇒ m2 = m1.fst
                                     else m2 = m1.snd } }
```

Using this definition, we can define *ring members* as members belonging to the cycle maintained by Chord. This is once again expressed as a variable subset signature, the elements of which are those that can all reach themselves through the *transitive closure* (`^`) of `succ`:

```
var sig ring in members {}
fact ringDef {
  always ring = { m : members | m in m.^succ } }
```

Finally, the set of *appendages* can simply be defined as those members that are not ring members:

```
fun appendages: set Node { members − ring }
```

B. Network Properties

As nodes are arranged into a cyclic network, their ordering must take into account the fact that the successor of the largest node identifier is the smallest one. Besides, we will often need to compare nodes by checking whether one node is between two others. This is reflected by the following definitions:

---

[1] Cf. https://haslab.github.io/Electrum.

[2] The full model is available at https://doi.org/10.5281/zenodo.1322052.

[3] `always` is the classic G (or □) connective of LTL; `some` applied to an expression means "not null"; and `.` is the relational join akin, here, to function application

```
fun nextNode: Node → Node {
  { n, m: Node | no next[n]
      implies m = first else m = next[n] } }
pred between [n1, nb, n2: Node] { // 'lt' is '<'
  lt[n1, n2] implies (lt[n1, nb] and lt[nb, n2])
    else (lt[n1, nb] or lt[nb, n2]) }
```

In Sect. II, we defined the key properties of Chord networks, called `Valid` and `Ideal`. The former is the conjunction of five properties: (1) there is *at least* one ring; (2) there is *at most* one ring; (3) any appendage node can reach a ring member by following successor pointers; (4) non-loopiness: there cannot be a ring member between a ring member and its successor; (5) successor list validity: the first successor of any member is between the member itself and the member's second successor.

```
pred valid { atLeastOneRing and atMostOneRing and
  orderedRing and connectedAppendages and
  orderedSuccessors }
pred atLeastOneRing { some ring }
pred atMostOneRing { all m1, m2: ring | m1 in m2.^succ }
pred connectedAppendages {
  all m1: appendages | some m2: ring | m2 in m1.^succ }
pred orderedRing { // = non−loopy
  all disj m1, m2, mb: ring |
  // 'disj' = 'all different'
    m2 = m1.succ implies not between[m1, mb, m2] }
pred orderedSuccessors { // successor list validity
  all m: members | between[m, m.fst, m.snd] }
```

An *ideal* network is a *valid* one *s.t.* (1) every member is in the ring (*i.e.* there are no appendages); (2) the `fst` and `prdc` functions are mutual inverses (local consistency) (3) the successor list of any member of the network contains the first 2 nodes that follow it in the network.

```
pred ideal {
  valid and no appendages and fst = ~prdc
  // '~' means 'transpose'
  all m: members { m.snd + m.fst in members
                   m.snd = m.fst.fst } }
```

C. Chord Events

1) An Action Layer: To model Chord events, we rely on an experimental *action* layer recently added to Electrum [16] that makes specification of transition systems much leaner. Actions are introduced by the keyword `act` and may take arguments. Their body is a conjunction of constraints referring to the current instant or the one following it immediately. The set of possible traces is automatically defined; notice that it implements (as of writing this article) an interleaving model of time: at every instant, exactly one action happens. Finally, an action comes with a `modifies` clause that contains the names of variable signatures and fields that the action may modify: an implicit fact then states that all other ones remain invariant under this action (this is usually called the action *frame condition*).

2) Communication Model: Following [4], our operations are atomic actions that may read *and* modify variables on at most two nodes. Compared to an asynchronous model, this is a *shared-state* abstraction that, in particular, hides the fact that nodes communicate through queued messages. Notice finally that the PODC paper states that communication is assumed to be reliable.

3) Events: The `join` action modifies `fst`, `snd` and `prdc` fields, and `members` and `ring` variable signatures. Under a `join` action, the joining node `new` must not be a member already. In PODC, the informal description of this event states that `new` contacts any node of the network and then makes a query to find a node `m` such that `new` is between `m` and its first successor. In our model, we abstract this query, assuming there is an oracle to determine this `m`. This abstraction does not affect the correctness of the protocol. Indeed, seeking the best position for the incoming node is an implementation and performance detail. Then `new` gets its pointers `fst` and `snd` from `m` and takes the latter as its predecessor[4].

```
act join [new: Node]
modifies fst, snd, prdc, members, ring {
  new not in members
  some m: members {
    between[m, new, m.fst] and fst' = fst ++ new→m.fst
    snd' = snd ++ new→m.snd and prdc' = prdc ++ new→m}}
```

Failures (or leavings) may happen too. When a member fails, it should empty all its fields. Besides, we take as an hypothesis that a node failure should not happen if it would leave another *member* with absolutely no live successors, meaning we forbid too many failures from happening on the same node to the point where it would completely break the network (this models the PODC failure assumptions: the protocol is indeed not able to fix networks split in several components that are mutually unreachable). Here, as there are only two successors per node, this requirement is easily modelled by stating that any node which points at the failing node using the `succ` relation keeps at least one of its two successors live when the failure happens.

Stabilization consists in fixing the first successor of a node. As in [4], stabilization is split here into two actions, depending on whether the concerned node has pending operations to do. This is shown in its `todo` field. We remark that, contrary to P. Zave, we may store *several* pending operations for a given node (otherwise, the field could be overwritten, which leads to a benign bug that we found during our analyses).

When there is not any pending operation for this node `m` (`stabilizeFromFst` action), it may contact its first successor. If the latter is dead (not a member), then `m` should update its `fst` field with its `snd` successor. The latter must also, in that case, be updated: to maintain the atomicity of the action, `m` should not contact any other node to get a new value for its `snd`. The solution here is just to take the immediate successor in the ring ordering. If it corresponds to no node, it will be fixed later by other events.

Otherwise, if the `fst` is a member, its value does not have to change but we update the `snd` as a small optimization. Besides,

---

[4] In the formalization, *e.g.* in the fourth line, `fst'` represents the value of `fst` in the *next* instant; `new→m.fst` is the pairing of `new` and `m.fst`; and ++ stands for the relational *override*. All in all, the line says that `fst'` is the current `fst` except in `new` where `fst'` will yield `m.fst`.

we check if `fst`'s predecessor is not null and is better than `m`'s first successor: if that is the case, the second form of stabilization (`stabilizeFromFstPrdc`) should be programmed; otherwise, `m` asks its first successor to program a future rectification with itself, meaning the first successor must ensure that `m` is its predecessor.

```
act stabilizeFromFst[m: Node]
modifies fst, snd, todo, members, ring {
  m in members
  no m.todo.Node // no pending operation
  m.fst not in members implies {
    todo' = todo and fst' = fst ++ m→m.snd
    snd' = snd ++ m→nextNode[m.snd] }
  else {
    fst' = fst and snd' = snd ++ m→m.fst.fst
    (some m.fst.prdc and between[m, m.fst.prdc, m.fst])
      implies todo' = todo + m→Stabilizing→m.fst.prdc
      else todo' = todo + m.fst→Rectifying→m } }
```

The second form of stabilization (`stabilizeFromFstPrdc`) may happen when a stabilization operation is pending: this is the case when a better candidate has been found for `m.fst` (during an `stabilizeFromFst` event). The first thing to do is to check whether the candidate would indeed, still, make a better `fst`. Besides, if the candidate is not even a member anymore, the operation must be cancelled. Otherwise, the `fst` field must be updated with the candidate (and the `snd` field is updated as well, with the candidate's `fst` field). Finally, this candidate is also told to rectify its predecessor later with `m` itself.

```
act stabilizeFromFstPrdc [m, newFst: Node]
modifies fst, snd, todo, members, ring {
  m in members and between[m, newFst, m.fst]
  m→Stabilizing→newFst in todo
  newFst not in members implies {
    todo' = todo − m→Stabilizing→newFst
    fst' = fst and snd' = snd }
  else {
    fst' = fst ++ m→newFst
    snd' = snd ++ m→ newFst.fst
    todo' = todo − (m→Stabilizing→newFst)
            + (newFst→Rectifying→m) } }
```

Finally, the `rectify` action aims at fixing a node `m`'s predecessor: it may only happen if a rectification has been programmed. There are then three possibilities: (1) if `m`'s predecessor is null or if the new candidate is better, then the predecessor should be updated to the candidate; (2) otherwise, if the current predecessor is not a member, the update is done too; (3) otherwise, `m`'s predecessor is left as is.

D. *Traces*

As explained at the beginning of this section, the shape of traces is automatically set by the Electrum action layer. At any instant, exactly one event happens.

A Chord network must run indefinitely. Nevertheless when the network becomes ideal, if there are no more join and fail, the network will deadlock. To avoid this concern, we also add a `skip` action, which is a silent action that leaves everything unchanged and does nothing.

```
act skip {} // does nothing, modifies nothing
```

Notice that we also impose that, in every trace, there are always at least three live nodes: this is due to the fact that the network should always have a size strictly greater than the size of successor lists.

E. *Initial State*

Concerning the initial state, we specify that the ring is the ideal state (and no node has pending operations and non-member nodes do not have a predecessor).

```
fact init { no nonMembers.prdc and no todo and ideal }
```

Notice that it is stronger than the original claim made in the PODC paper (proved wrong in [6]). [4] exhibits an invariant stronger than validity. Although Electrum alleviates us from expressing such an invariant, we still need to ensure that the initial state satisfies it. Saying that the network starts in an ideal state avoids formulating that property explicitly: it is in our view not that strong an hypothesis as a Chord network may start with a very small size.

F. *Correctness*

1) Basic Properties: Using the Electrum Analyzer, we checked that the specification is consistent (*i.e.* it admits a model, in the logical sense) and that all branches of all actions are realizable.

2) Correctness Statement: The correctness property for the Chord protocol is a *liveness* property. The translation from the PODC paper is straightforward thanks to LTL: it states that if there are, eventually, never any join or fail events, then, eventually, the network will become ideal and remain so (recall the inital state is set in Sect. IV-E). This is expressed by an Electrum *assertion*:

```
assert correctness {
  (eventually always not (join or fail))
    implies eventually always ideal }
```

3) Fairness: Checking this assertion with the Electrum Analyzer yields a counterexample that manifests itself as the endless repetition of the `skip` action in some non-ideal state. This is to be expected as the correctness property is a *liveness* property: any action that may cause starvation to the ones meant to fix the network will be a problem. Classically, the solution is to add *fairness* constraints on the said actions. Here we use strong fairness constraints, saying for instance that if the guard of `rectify` is infinitely often satisfied, then the effect of `rectify` will be satisfied infinitely often:

```
pred rectifyEnabled[m, n: Node] {
  m in members and  m→Rectifying→n in todo }
fact fairness {
  all n, m : Node |
    (always eventually rectifyEnabled[n,m])
      implies (always eventually rectify[n,m]) }
```

We added such constraints for all stabilization and rectification actions. Doing so excludes the kind of starvation described above. Actually, it corresponds to a requirement in the PODC paper that says that nodes should perform these actions "periodically".

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

4) Corner Cases: During our analysis of correctness, we were able to find a few benign corner cases in P. Zave's model. They were all found in a matter of seconds, simply by checking the correctness property (the ease of finding them comes in our view from the fact that the use of the LTL layer of Electrum helps circumvent the risk of overlooking some verifications to be done). This led to make a few simple fixes w.r.t. her model (*e.g.* using a `todo` field to gather several pending operations instead of only one).

5) Liveness Bug: However, the correctness property is still wrong: checking the assertion in the Electrum Analyzer yields a trace with six time instants, the last one looping back in its predecessor (we recall that traces are infinite and represented as finite traces with a back loop from the last state to a former one). We present in Fig. 4 these last two steps only. In the first one, a `stabilizeFromFst[Node$3]` event is performed: `Node$3` contacts it immediate successor (which is in the ring) and learns from it that `Node$0` may be a better first successor. Then it programs a `stabilizeFromFstPrdc` action for itself and this new candidate.

In the following instant, the said action is performed but, as `Node$0` happens not be a member, the stabilization operation is cancelled. A rectification on `Node$1` would be needed here, for it to take `Node$3` as its predecessor, but a thorough analysis show that there is no way to trigger it by any of the stabilization actions.

6) Fixed Model: We fix this by adding another rectification action that is *not* triggered by other actions but done "periodically" by the nodes themselves (*i.e.* we also add a strong fairness constraint for this new action; note that such an operation was actually present in the original Chord papers). As nodes cannot guess who they should take as a new predecessor, they should just set their predecessor pointer to null if it points to a non member. The bet here is that, by other operations, the said node will eventually find a correct predecessor.

```
act rectifyNull[m: Node] modifies prdc, members, ring {
  m in members
  m.prdc not in members
    implies prdc' = prdc − m→m.prdc else prdc' = prdc }
```

Checking the correctness assertion, once this has been added, yields no counter-example anymore.

V. EVALUATION OF RESULTS

This section presents the evaluation of various properties as well as that of the final correctness property with the Electrum Analyzer. The verification is performed on a GNU/Linux-based work station featuring an Intel Xeon E5-2699 providing 512 GB RAM (time-out was set to 5 h.).

Depending on the analyses to perform, we relied on the bounded and unbounded model-checking techniques (BMC and UMC) provided by the tool: the former relying on either a translation to BMC over Minisat (performed by an Electrum extension of Alloy's Kodkod pivot solver) or to the BMC mode of nuXmv (`check_ltlspec_bmc_inc` algorithm); and the latter through the ultimate compilation to the nuXmv model checker running the `check_ltlspec_klive` procedure [15]. As it is usually far more efficient, we always favored using the BMC technique when we were expecting to end up with an instance or a counter-example to a given property.

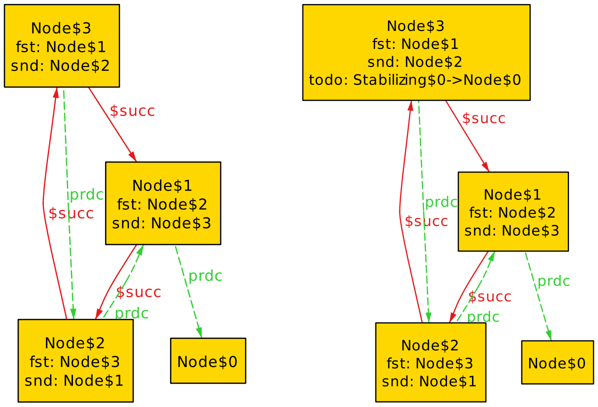


Fig. 4. Counterexample to `correctness` (loop part). Step 1 of the loop: a `stabilizeFromFst[Node$3]` is done; step 2 of the loop: a `stabilizeFromFstPrdc[Node$3, Node$0]`, then loop back to step 1.

TABLE I
TIME (S.) FOR CORRECTNESS ANALYSES (BUGGY AND FIXED CASES).

Prop.	Scope	M 10	M 15	XB 10	XB 15	XU
buggy	4	5	5	9	9	36
fixed	4	69	1 316	21	260	558
fixed	5	1 060	t/o	549	t/o	t/o
fixed	6	11 506	t/o	6 930	t/o	t/o
fixed	7	t/o	t/o	t/o	t/o	t/o

Although we do not report in detail on all properties due to lack of space, we first checked that our model is consistent (*i.e.* it admits an instance) and that all action branches are realizable. All these analyses ended positively in at most 10 s. using the BMC mode of the Electrum Analyzer.

We present in Table I the results for finding the liveness bug of Sect. IV-F5 and to check the correctness property for the fixed model (time in seconds; "t/o" means "time-out"; M is for BMC over Minisat, XB is for nuXmv in BMC mode, and XU is for nuXmv in UMC mode; for bounded modes, we considered bounds of 10 and 15 states).

As can be seen, correctness can be checked by the bounded analyzer for networks with 4-6 nodes and a time bound of 10 (4 nodes for a time bound of 15), while the unbounded Electrum analyzer yields a result for networks with 4 members only (taking into consideration that a basic ring already contains at least three nodes). This limitation in the size of the network with nuXmv is compensated by the fact that verification is exhaustive. For equal network sizes, the bounded version of Electrum Analyzer is faster than the unbounded one, as can be expected for valid properties. Finally, in bounded mode, nuXmv is faster than the Electrum implementation of BMC over Minisat, for the fixed model.

VI. DISCUSSION

In this section, we would like to stress some important aspects of this work.

First, we modelled the Chord protocol in a very straightforward way thanks to the ease of use of Electrum. Compared with previous work, first-order relational logic, temporal logic and the action setting (with automatic handling of frame conditions and interleaving), combined with the push-button approach and visual feedback of the Electrum Analyzer, allowed us to quickly implement and test various approaches. Our model is inspired by important work of P. Zave, in particular its last incarnation [4], in that it essentially implements the same algorithm. But, we argue that our model is simpler, in particular because we do not have to deal with the details of state representation and because expressing complex temporal formulas over infinite traces is immediate.

Compared to P. Zave's analysis with SPIN [5], our modelling is also more straightforward as the author had to resort to various C programs to handle the *graph* notions present in Chord as well as visualization.

Second, due to this very reliance on LTL, we did not have to look for an inductive invariant to study the protocol. The search for the said invariant has been very arduous [4]–[8], but also illuminating: an invariant is not only a means of verification but also a way to understand a protocol better and to provide indications to developers. Notice also that Zave's invariant can actually be checked in Electrum exactly as in Alloy, with performances in the same order of magnitude. For these reasons, we think that Electrum may be used in early analysis and provide some help into finding the said invariant.

Third, we were able to find some corner cases in the Alloy model as well as the manual proof of [4] that were confirmed by P. Zave. In particular, the claimed invariant is indeed one but it is not strong enough to prove the protocol correctness. Fortunately, these issues were rather easy to fix.

Fourth, it is sometimes claimed that liveness "in the abstract" is not that important as what one really longs for is *bounded liveness*, which is actually a safety property. Our work confirms that straightforward temporal specification in LTL and pure liveness analysis are useful to find various issues (including a too weak invariant).

Finally, although limited to very small networks, our analysis is, up to our knowledge, the first "push-button" analysis of the actual correctness property of Chord, which is a liveness property.

We have insisted in this paper on P. Zave's work as this has been an important one but also because it served as a basis to lots of other works. However, the main recent work [17] relies on manual proofs in Coq and proves a safety property over Raft. [18] features an interesting mostly-automated approach but also focuses on an invariant proof. More recent work by the same authors [19] addresses liveness properties (for other protocols) but still with manual interaction.

In another line of work, [20] relied on π-calculus and for a bisimulation proof of the correctness of a very simple version of Chord (without failures). The proof was purely manual.

Besides, other distributed system protocols have been formally studied using "high-level" specification languages. For instance, Pastry was analyzed using TLA$^+$ [21] and other work used Event-B [22] to partly verify other protocols. However, these studies are limited to the verification of safety properties.

VII. CONCLUSION

This work presented the specification and verification of the Chord distributed protocol. We highlighted the usefulness of a lightweight modeling method that allows modeling and verifying dynamic systems with rich structural properties, as exemplified by Electrum (in particular with its action layer). Electrum allowed a simple and straightforward modeling of both structural and temporal properties of Chord, with a rather high abstraction level and without losing the key concepts of the protocol.

The analysis of the Chord model with the Electrum Analyzer is fully automated (on a bounded domain), which implies that the cost of entry is rather lower than many other formal methods. This analysis allowed us to find a few minor issues in the most important previous work (in which we took inspiration) and to show that the invariant there is not strong enough. We were able to fix all issues. Up to our knowledge, this is the first work analyzing the correctness of Chord, a liveness property, in a "push-button" way.

As of now, this analysis is admittedly limited in the size of networks, in particular for unbounded model checking. This was expected to us as one of the reasons to study Chord, for us, was to get a challenging test bed for our unbounded model-checking back-end. On the other hand, even with small networks and bounded model-checking, we were able to find various shortcomings in previous work, which confirms the interest on working even with small instances.

In the future, we will work both on improving Electrum and its analysis tools, and on the Chord protocol. For the former aspect, we will investigate smarter verification techniques, both on the bounded and unbounded sides (the latter is in particular, as of now, implemented in a naive way). For the latter aspect, we will investigate imperfect detection of failures.

ACKNOWLEDGMENT

The authors are grateful to Pamela Zave for her explanations and comments. We also thank Nuno Macedo for adding or fixing some Electrum Analyzer features needed for our study. Finally we thank the reviewers for their very useful remarks.

Work financed by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation (COMPETE2020) and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT) within project POCI-01-0145-FEDER-016826; and within the French Research Agency project FORMEDICIS (ANR-16-CE25-0007).

REFERENCES

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[2] D. Liben-Nowell, H. Balakrishnan, and D. Karger, “Analysis of the evolution of peer-to-peer systems,” in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM, 2002, pp. 233–242.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.

[4] P. Zave, “Reasoning about identifier spaces: How to make chord correct,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1144–1156, Dec 2017.

[5] ——, “A practical comparison of Alloy and SPIN,” *Formal Aspects of Computing*, vol. 27, no. 2, p. 239, 2015.

[6] ——, “Using lightweight modeling to understand Chord,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[7] ——, “Why the Chord ring-maintenance protocol is not correct,” AT&T Research, Tech. Rep, Tech. Rep., 2011.

[8] ——, “Lightweight Modeling of Network Protocols in Alloy,” 2009.

[9] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, “Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations,” in *Foundations of Software Engineering*, 2016.

[10] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[11] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[13] A. Cunha, “Bounded model checking of temporal formulas with Alloy,” in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 303–308.

[14] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, “NUSMV: A new symbolic model verifier,” in *Computer Aided Verification, 11th International Conference, CAV ’99, Trento, Italy, July 6-10, 1999, Proceedings*, 1999, pp. 495–499.

[15] 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 *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 334–342.

[16] J. Brunel, D. Chemouil, A. Cunha, T. Hujsa, N. Macedo, and J. Tawa, “Proposition of an action layer for electrum,” in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2018, pp. 397–402.

[17] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 357–368.

[18] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 614–630.

[19] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, “Reducing liveness to safety in first-order logic,” *PACMPL*, vol. 2, no. POPL, pp. 26:1–26:33, 2018.

[20] R. Bakhshi and D. Gurov, “Verification of peer-to-peer algorithms: A case study,” *Electronic Notes in Theoretical Computer Science*, vol. 181, pp. 35–47, 2007.

[21] S. Merz, T. Lu, and C. Weidenbach, “Towards Verification of the Pastry Protocol using TLA^+,” in *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, vol. 6722, 2011.

[22] J. Risson, K. Robinson, and T. Moors, “Fault tolerant active rings for structured peer-to-peer overlays,” in *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*. IEEE, 2005, pp. 18–25.

ISBN: 978-0-9835678-8-2.

k-FAIR = *k*-LIVENESS + FAIR
Revisiting SAT-based Liveness Algorithms

Alexander Ivrii, Ziv Nevo, Jason Baumgartner IBM Corporation

***Abstract*—We revisit the two main SAT-based algorithms for checking liveness properties of finite-state transition systems: the *k*-LIVENESS algorithm of [1] and the FAIR algorithm of [2]. These approaches are fundamentally different. *k*-LIVENESS works by translating the liveness property together with fairness constraints to the form FGq, and then bounding the number of times the variable q can evaluate to false. FAIR works by finding an over-approximation R of reachable states, so that no state in R is contained on a fair cycle. Each technique has unique strengths on different problems. In this paper, we present a new algorithm *k*-FAIR that builds upon both techniques, synergistically leveraging their strengths. Experiments demonstrate that this combined approach is stronger than running both in parallel.**

I. INTRODUCTION

Efficient verification of liveness properties remains an important unsolved problem. A common approach is based on the *liveness-to-safety* translation [3] that converts liveness properties to safety properties, enabling the use of any safety checking technique. In practice this translation works very well especially for failing properties, though suffers from the severe performance penalty of doubling the number of state variables. More direct SAT-based approaches have thus been proposed: FAIR [2] and *k*-LIVENESS [1]. In this paper, we revisit these approaches, and present a new algorithm *k*-FAIR that combines the strengths of both in a way that outperforms running them in parallel.

A liveness property can be converted to form FGq, meaning that on every path variable q must eventually evaluate to true forever [1]. A counterexample would illustrate q evaluating to false infinitely often. As the state-space of hardware models is finite, such a counterexample may be represented as a lasso-shaped trace, consisting of a prefix from an initial state to a $\neg q$-state s, and a repeating loop suffix from s back to itself.

Given a property FGq, *k*-LIVENESS [1] attempts to bound the number of times that q can evaluate to false. Effectively, this technique checks a sequence of safety properties p_k which evaluate to false when q evaluates to false at least $k + 1$ times. Initially $p_0 = q$, and p_{k+1} is obtained from p_k by adding "absorbing logic" that masks one occurrence of $\neg q$. If $\neg p_k$ is proven valid, FGq is clearly valid. A bounded counterexample to $\neg p_k$ does not guarantee the existence of a counterexample for higher bounds, though if it exhibits a repeated state sequence within which q evaluates to false, it is a valid unbounded counterexample. Given a finite state space, for suitably-large k, either $\neg p_k$ will be proven or will yield a valid unbounded counterexample. *k*-LIVENESS is thus sound and complete. As noted in [4], in practice unbounded counterexamples can often be detected even for small values of k. Given the close relation between models being checked for increasing k, an *incremental* model checker such as IC3 [5], [6] offers the advantage of reusing information such as bounded and absolute invariants between each query.

FAIR [2] is an iterative algorithm that incrementally learns information about reachable states and the SCC-closed regions of the state space. Roughly speaking, a *reachability assertion* R indicates that all the states on a potential lasso-shaped counterexample belong to R, while a *wall* W states that all states on the loop suffix of a potential counterexample either together belong to W or together belong to the complement of W. If one side of the wall W has no reachable states, the wall actually represents a constraint on all states on the loop of a potential counterexample, called *stabilizing constraints* in [1]. Specializing to liveness properties of form FGq, FAIR uses a SAT-solver to obtain a $\neg q$-state s, subject to the previously-discovered reachability assertions and walls. If this query is unsatisfiable, then FGq holds. Otherwise, FAIR tries to compute lasso-shaped counterexample for s, checking whether s is reachable from an initial state, then whether s can eventually transition to itself. If both queries are satisfiable, the liveness property fails. Otherwise, FAIR requires the underlying safety model checker to produce an inductive proof of unsatisfiability. If s is not reachable from an initial state, this proof represents a new reachability assertion. If s cannot transition to itself, this proof represents a new wall. [2] suggests several methods to discover new walls, including a method to generalize s to a set of states s_{gen}, so that no state in s_{gen} has a loop back to itself; equivalently, $\neg s_{gen}$ is a new stabilizing constraint. In either case, the algorithm makes progress and must eventually terminate with a conclusive verification result.

These two algorithms have different strengths. When FGq is valid, *k*-LIVENESS works well when a small value of k is sufficient to prove unsatisfiability; otherwise the underlying safety queries become unscalable as k becomes large. FAIR works well when inductive proofs restrict large portions of the search space; otherwise, too many iterations are required.

In this paper, we propose a new algorithm *k*-FAIR that combines ideas from both approaches. Similarly to *k*-LIVENESS, we pose a safety query that checks for a trace on which q evaluates to false at least k times. If unsatisfiable, the liveness property is proven. If satisfiable, we check whether the bounded counterexample has a repeated $\neg q$-state; if so, the liveness property is disproven. Otherwise, we select a $\neg q$-state s from the trace, and (similar to FAIR) check if it can eventually transition back to itself. If so, the liveness property

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

is disproven. Otherwise, we extract a new stabilizing constraint $c = \neg s_{gen}$, by generalizing s to a larger set of states s_{gen} without a self-loop. These stabilizing constraints are used to restrict every occurrence of $\neg q$ in future checks for a trace on which $c \rightarrow q$ evaluates to false at least k times. Note that when a new stabilizing constraint is discovered, there is no need to increase k for completeness, enabling convergence with smaller bounds than k-LIVENESS.

In Section II, we describe details for making these restrictions more efficient with IC3 queries, and for more-efficient detection of new stabilizing constraints. Originally [2] suggests to periodically look for *single-literal* stabilizing constraints. [1] improves upon this technique by considering all nets in a circuit as candidate constraints, and using the liveness signal q in a stronger way; however, this is purely a preprocessing technique. k-FAIR uses the best of both worlds: applying the method of [1] periodically, using the external reachability invariants and stabilizing constraints to strengthen the induction hypothesis.

II. k-FAIR

A. Algorithm Overview

Algorithm 1 k-FAIR

```
Input: Liveness property $FGq$
Data: Reachability invariants $R$, Stabilizing constraints $S$
 1: $OnLoop \leftarrow$ CreateOnLoopReg ()
 2: $r \leftarrow$ register with init = $true$ and next = $(OnLoop \rightarrow q)$
 3: $p \leftarrow r$, $k \leftarrow 0$, $R \leftarrow \emptyset$, $S \leftarrow \emptyset$
 4: while $true$ do
 5:     if (*) then
 6:         $(st, S) \leftarrow$ StabilizingConstraints$(R, S)$
 7:         if $(st = $ UNSAT$)$ then
 8:             return PASS
 9:     $(st, \alpha, R) \leftarrow$ Run_kLIVENESS$(p, R, S)$
10:     if $(st = $ UNSAT$)$ then
11:         return PASS
12:     if $\alpha$ has a state repetition with $\neg r$ then
13:         return FAIL
14:     if (*) then
15:         $s \leftarrow$ Select last state on $\alpha$ with $\neg r$
16:         $(st, \beta, S) \leftarrow$ Run_FAIR$(s, R, S)$
17:         if $(st = $ SAT$)$ then
18:             return FAIL
19:     if (*) then
20:         $p \leftarrow$ AbsorbingLogic$(p, r)$, $k$++
```

Our k-FAIR algorithm is depicted in Algorithm 1. It accepts a liveness property FGq (which embeds fairness constraints), and returns PASS or FAIL with counterexample. The algorithm incrementally updates two important structures: *reachability invariants* R (that constrain all states on a potential lasso-shaped counterexample), and *stabilizing constraint*s S (that constrain all states on the loop of a potential lasso-shaped counterexample). In practice, each constraint in R and S is a clause (disjunction) over registers and internal nets.

Lines 1–3. Function `CreateOnLoopReg` creates a new register $OnLoop$ that is initialized to 0, which nondeterministically changes its value to 1 after which it remains 1 forever. We create a new register r with next-state function $OnLoop \rightarrow q$. It is easy to see that the validity of FGq is equivalent to the validity of FGr, and a counterexample to FGr is a counterexample to FGq. Intuitively, $OnLoop$ allows to efficiently pass information to the underlying safety model checker, while register r simplifies implementation details. Variable p represents the value of the current safety property. For clarity, index k corresponds to the safety property p_k.

Lines 5–8. Algorithm `StabilizingConstraints` derives new stabilizing constraints, accepting R and S and updating S. This function is similar to [1], except that it additionally uses R and S to restrict both current- and the next-states in the SAT-solver. Additionally, we have found it useful to reason about the original fairness constraints instead of q when looking for nets that stabilize to a constant value. Theoretically, this allows `StabilizingConstraints` to discover more stabilizing constraints as the sets R and S are extended elsewhere, justifying the value of running this function periodically. In cases, these new stabilizing constraints exclude *all* reachable states, in which case the algorithm terminates with PASS.

Lines 9–11. Function `Run_kLIVENESS`(p, R, S) checks whether an initial state can reach a $\neg p$-state, subject to constraints $R \wedge (OnLoop \rightarrow S)$. Equivalently, this checks for a path from an initial state, on which $OnLoop \wedge \neg q$ occurs at least k times under these constraints. In particular, the stabilizing constraints S must hold on every state after the first occurrence of $OnLoop \wedge \neg q$. This is slightly stronger than suggested in [1], where the stabilizing constraints are only used to restrict the $\neg q$-states. This function returns the verification status $st \in \{\texttt{SAT}, \texttt{UNSAT}\}$, counterexample α for $st = \texttt{SAT}$, and additional reachability invariants R discovered in the process. As in [1], we use an incremental IC3-engine, which reuses bounded and absolute invariants between runs; this allows to neglect explicitly passing R to this engine. Instead of synthesizing $(OnLoop \rightarrow S)$ using new logic, we have extended the IC3-engine to accept *clausal constraints* over registers and internal nets. In particular, for each clause $c \in S$, we pass the clausal constraint $\neg OnLoop \vee c$. If the verification status st returned by `Run_kLIVENESS` is `UNSAT`, the algorithm terminates with PASS.

Lines 12–13. If the safety query returns `SAT`, then as suggested in [4] we analyze the counterexample α to check if it exhibits a state repetition on which r evaluates to false. If so, the counterexample is valid and the algorithm terminates with FAIL. Additionally, we may manipulate α using the trace manipulation techniques described in [4] to improve the likelihood of producing a valid counterexample from α.

Lines 14–18. First, we select a $\neg r$-state s from α; by construction there are at least $k+1$ such states. In practice, the last such state is most effective, though any (or multiple) may be selected. Function `Run_FAIR`(s, R, S) checks for a path from s back to itself, subject to constraints $R \wedge S$. If the result is `SAT`, a valid counterexample β exists and the algorithm terminates with FAIL. (A valid counterexample to FGr can be constructed by concatenating α and β). If $st =$ `UNSAT`, we use the technique of [2] to generalize s to a larger set of states s_{gen}, none of which can transition to s_{gen}. In this case, we update S by adding a new stabilizing constraint $\neg s_{gen}$. As in [2], we use an IC3-engine, which produces inductive invariants. To avoid trivial 0-length paths, we introduce an additional register Z with initial value 0 and next-state function 1, and the actual query checks whether $s \wedge \neg Z$ can reach $s \wedge Z$. Note that passing sets R and S to IC3 is not required for correctness, so using them most efficiently in the underlying IC3-engine poses a complex implementation choice. In our experience, having many redundant clausal constraints may slow down IC3 (hurting ability to reduce proof obligations by ternary simulation or alternative techniques). In our implementation, we pass S as clausal constraints and R as *clausal invariants*, the difference being that clausal invariants are ignored when reducing proof obligations.

Lines 19–20. If `Run_FAIR` was executed, a new stabilizing constraint was detected hence the algorithm made progress. Contrary to k-LIVENESS, adding absorbing logic is not required for completeness: we may continue with the same value of k (or even reduced k). In fact, FAIR can be seen as an instance of this algorithm when k is always zero.

B. Comparison to k-LIVENESS *and to* FAIR

k-FAIR effectively combines the strengths of k-LIVENESS and FAIR. If `Run_FAIR` is never executed (if the **if**-condition on line 14 is always false), then k-FAIR closely corresponds to k-LIVENESS modulo the ability to detect new stabilizing constraints via reachability invariants from IC3. k-FAIR can be viewed as k-LIVENESS extended with an additional technique to look for unbounded counterexamples. On the other hand, if `AbsorbingLogic` is never executed (if the **if**-condition on line 19 is always false), k-FAIR corresponds to an alternative implementation of FAIR. Though instead of using a SAT-solver to find candidate $\neg q$-states s and checking if they are reachable from an initial state, we search for such reachable states directly. Additionally, when s cannot reach itself, we only borrow the method of [2] that discovers stabilizing constraint and not a more general wall constraint. Arguably, this makes our implementation simpler, but may lose some potential power enabled by more general constraints.

III. EXPERIMENTS

In this section we present our experimental results. The techniques described in this paper are implemented in the IBM formal verification tool *Rulebase: Sixthsense Edition* [7]. All experiments are executed on a 2.00 GHz Linux machine with an Intel Xeon E7540 processor, 16GB of RAM, and 3 hours time-limit. We used all single-property liveness benchmarks from the 2011–2017 Hardware Model Checking Competitions [8], as well as various proprietary industrial testcases. For a more realistic setup, the benchmarks are preprocessed using standard logic synthesis techniques (similar to ABC [9] commands *rewrite*, *lcorr* and *ssw*).

TABLE I
SUMMARY OF EXPERIMENTAL RESULTS

	PASS solved	PASS time	FAIL solved	FAIL time
***k*-FAIR-fair**	108	338,475	89	351,634
***k*-FAIR-b50**	111	301,592	94	321,260
***k*-FAIR-b5**	122	166,655	104	240,458
***k*-FAIR-klive**	123	173,077	97	245,543
***k*-FAIR-klive-pre**	117	250,431	100	225,971
LTS-BMC	-	-	114	94,103
LTS-IC3	116	225,059	99	226,321
VBS	131	37,270	117	29,315
VBS without **klive**	130	43,661	117	29,349
VBS without **b5,b50**	131	37,510	116	31,326
***k*-FAIR-fair** & ***k*-FAIR-klive-pre**	124	175,581	107	154,646
LTS-IC3 & ***k*-FAIR-klive**	131	37,270	100	206,171
LTS-BMC & ***k*-FAIR-b5**	122	166,655	117	57,482

TABLE II
PROVEN k VALUE FOR COMMONLY-SOLVED BENCHMARKS

***k*-FAIR-**	**fair**	**b50**	**b5**	**klive**	**klive-pre**
average k	0	0.50	1.84	6.02	10.09

A. Review of results

The configurations evaluated include: The first five configurations are different variants of Algorithm 1. In ***k*-FAIR-fair**, `Run_FAIR` runs on every iteration of the main loop but the counter k is never incremented: this is "pure FAIR" mode. In ***k*-FAIR-b50** and ***k*-FAIR-b5**, `Run_FAIR` runs on every iteration of the loop, while the counter k is incremented on, respectively, every 50th and 5th iteration. In ***k*-FAIR-klive** and ***k*-FAIR-klive-pre**, `Run_FAIR` never runs, and the counter is incremented on every iteration of the loop: this is the "pure k-LIVENESS" mode. In all five variants, `StabilizingConstraints` runs on the first iteration of the main loop as preprocessing. Additionally, in the first four variants, `StabilizingConstraints` runs periodically (either each time the counter increments, or on every 50th iteration of the loop, whichever happens first). In ***k*-FAIR-klive-pre**, `StabilizingConstraints` does not run again, corresponding most closely to [1]. The last two configurations **LTS-BMC** and **LTS-IC3** correspond to the liveness-to-safety translation, followed by BMC (Bounded Model Checking) [10] and IC3, respectively.

Table I summarizes the experiments. Columns "PASS solved" and "FAIL solved" show the number of passing and failing instances, respectively, solved by a specific configuration. Columns "PASS time" and "FAIL time" represent the cumulative time in seconds for passing and failing properties, respectively. Benchmarks solved by preprocessing alone, and

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE III
COMPARISON OF "PURE FAIR" IN k-FAIR AND FAIR IN *IImc*

	PASS solved	PASS time	FAIL solved	FAIL time
***k*-FAIR-fair**	108 (18)	208,875	89 (21)	70,834
IImc-fair	101 (11)	277,657	70 (2)	242,762

those not solved by any configuration, are excluded from further consideration, leaving a total of 131 passing and 117 failing testcases. As BMC cannot prove properties, results are shown only for failing properties. Row "**VBS**" corresponds to the *virtual best* of all configurations. The last five rows represent selected portfolios of the configurations above. For example, row "**VBS** without **klive**" corresponds to running in parallel all configurations except **klive**. Row "***k*-FAIR-fair** & ***k*-FAIR-klive-pre**" corresponds to running in parallel the two configurations ***k*-FAIR-fair** and ***k*-FAIR-klive-pre**.

B. Overall summary

Simple liveness-to-safety followed by BMC is a very strong falsification strategy, solving all but 3 failing testcases. Interestingly, these 3 are solved by ***k*-FAIR-b5** (with one unique solve), and in each case the counterexample is detected by `Run_FAIR` vs. the state repetition check. This may be because candidate states returned by `Run_kLIVENESS` for large k have a higher chance to belong to a valid counterexample. The best two-engine parallel portfolio consists of **LTS-BMC** and ***k*-FAIR-b5**, solving *all* failing properties with a runtime improvement of 1.6 vs. **LTS-BMC** alone. A parallel portfolio running all seven configurations improves total runtime by an additional factor of 1.9.

For passing properties, the "pure k-LIVENESS approach with an incremental detection of stabilizing constraints" performs best (yielding one unique solve), outperforming both the "pure FAIR" approach, the liveness-to-safety followed by IC3, and the "standard k-LIVENESS approach" ***k*-FAIR-klive-pre**. A best two-engine portfolio consists of **LTS-IC3** and ***k*-FAIR-klive**, solving all passing properties in the smallest possible time.

C. Examining k sufficient for proof

There are 100 passing testcases (out of 131) solved by all five variants of k-FAIR. In Table II we restrict to these testcases and report the values of k sufficient for proof, averaged over all the testcases. Not surprisingly, this value is 0 in "pure FAIR" mode, and gradually increases to 6.02 as the variant changes to "pure k-LIVENESS." This table shows that stabilizing constraints based upon `Run_FAIR` reduce the value of k needed to obtain a proof. Without the incremental detection of stabilizing constraints based on `StabilizingConstraints`, the sufficient value of k is even larger.

D. Comparing *k*-FAIR-fair to IImc-fair

As an additional experiment, we compare ***k*-FAIR-fair** – our "pure FAIR" approach, and **IImc-fair** – the original FAIR algorithm of [2]. **IImc-fair** uses the implementation in *IImc* [11] with command *iimc -t fair -v1 --fair_timeout 10800*. The results are summarized in Table III. As before, we present data only for testcases solved by at least one configuration. The numbers in parentheses represent unique solves. Overall ***k*-FAIR-fair** performs substantially better than **IImc-fair**, both on passing and failing properties, though both variants have unique value. Unfortunately, a detailed comparison is difficult, as the two techniques are implemented in very different verification frameworks, and the improvements may be due to a large number of different factors, including an improved method in [1] to find stabilizing constraints, only looking for loops from a priori reachable states, and the syntactic check for a state repetition (for failing properties). In any case *the adaptation of FAIR presented in this paper seems as a viable alternative to the implementation in IImc [11].*

IV. CONCLUSION AND FURTHER WORK

In this paper we presented the algorithm k-FAIR, which combines the strengths of the prominent SAT-based algorithms for liveness verification: k-LIVENESS and FAIR. We experimented with several variants of k-FAIR and demonstrated that a combined portfolio approach brings unique value.

Fine-tuning the algorithm is likely to offer additional performance improvements. Each of the main methods `StabilizingConstraints`, `Run_kLIVENESS` or `Run_FAIR` may be the key to success, but may also be the bottleneck of the approach. Carefully balancing the effort spent on each component (e.g., by suitably imposing resource limits, or by increasing or decreasing k more aggressively) is a subject of further research. Another promising direction consists of tuning the underlying IC3-engine towards the safety queries posed by the algorithm. For example, one could attempt to leverage the fact that all safety queries made by `Run_kLIVENESS` (except for possibly the very last one) are satisfiable, while all safety queries made by `Run_FAIR` (except for possibly the very last one) are unsatisfiable. Additionally, one could attempt to devise better methods to pass constraints, invariants, etc. to the IC3-engine, and to use these more efficiently in the IC3-engine itself.

REFERENCES

[1] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, 2012.
[2] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*, 2011.
[3] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electr. Notes Theor. Comput. Sci.*, vol. 66, no. 2, 2002.
[4] G. Aleksandrowicz, J. Baumgartner, A. Ivrii, and Z. Nevo, "Generalized counterexamples to liveness properties," in *FMCAD*, 2013.
[5] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Jan. 2011.
[6] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011.
[7] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.
[8] Hardware Model Checking Competition 2017. http://fmv.jku.at/hwmcc17.
[9] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification.* http://people.eecs.berkeley.edu/ alanmi/abc/.

ISBN: 978-0-9835678-8-2.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *TACAS*, March 1999.
[11] A. R. Bradley and F. Somenzi and Z. Hassan, *IImc: an Incremental Inductive model checker*. https://github.com/mgudemann/iimc.

ISBN: 978-0-9835678-8-2.

Temporal Prophecy for Proving Temporal Properties of Infinite-State Systems

Oded Padon*, Jochen Hoenicke†, Kenneth L. McMillan‡, Andreas Podelski†, Mooly Sagiv* and Sharon Shoham*
*Tel Aviv University, Israel †University of Freiburg, Germany ‡Microsoft Research, USA

Abstract—**Various verification techniques for temporal properties transform temporal verification to safety verification. For infinite-state systems, these transformations are inherently imprecise. That is, for some instances, the temporal property holds, but the resulting safety property does not. This paper introduces a mechanism for tackling this imprecision. This mechanism, which we call *temporal prophecy*, is inspired by prophecy variables. Temporal prophecy refines an infinite-state system using first-order linear temporal logic formulas, via a suitable tableau construction. For a specific liveness-to-safety transformation based on first-order logic, we show that using temporal prophecy strictly increases the precision. Furthermore, temporal prophecy leads to robustness of the proof method, which is manifested by a cut elimination theorem. We integrate our approach into the Ivy deductive verification system, and show that it can handle challenging temporal verification examples.**

1. Introduction

There are various techniques in the literature that transform the problem of verifying liveness of a system to the problem of verifying safety of a different system. These transformations compose the system with a device that has the known property that some safety condition σ implies liveness. The classical example of this is proving termination of a while loop with a ranking function. In this case, the device evaluates a chosen function r on loop entry, where the range of r is a well-founded set. The safety property σ is that r decreases at every iteration, which implies that the loop must terminate.

A related transformation, due to Armin Biere [5], applies to finite-state (possibly parameterized) systems. The safety property σ is, in effect, that no state occurs twice, from which we can infer termination. In the infinite-state case, this can be generalized using a function f that projects the program state onto a finite set. We can think of this as a ranking that tracks the set of unseen values of f and is ordered by set inclusion. However, the property that no value of f occurs twice is simpler to verify, since the composed device can non-deterministically guess the recurring value. In general, the effectiveness of a liveness-to-safety transformation depends strongly on the difficulty of the resulting safety proof problem.

Other methods can be seen as instances of this general approach. For example, the Terminator tool [11] might be seen as combining the ranking and the finite projection approaches. Another approach by Fang *et al.* applies a collection of ad-hoc devices with known safety-to-liveness properties to prove liveness of parameterized protocols [16]. Of greatest interest here, a recent paper by Padon *et al.* uses a dynamically chosen finite projection that depends on a finite prefix of the system's execution [30]. The approach of [28] also has some similar characteristics.

In the case of infinite-state systems, these transformations from liveness verification to safety verification are not precise reductions. That is, while safety implies liveness, a counterexample to the safety property σ does not in general imply a counterexample to liveness. For example, in the projection method, a terminating infinite-state system may have runs whose length exceeds the finite range of any chosen projection f, forcing some value to repeat.

In this paper, we show that the precision of a liveness-to-safety transformation can be usefully increased by the addition of *prophecy variables*. These variables are expressed as first-order LTL formulas. For example, suppose we augment the state of the system with a variable $r_{\Box p}$ that tracks the truth value of the proposition $\Box p$, which is true when p holds in all future states. We can soundly add two constraints to the transition system. To the transition relation, we add $r_{\Box p} \leftrightarrow (p \wedge r_{\Box p}')$, where $r_{\Box p}'$ denotes the value of the prophecy variable in the post-state. We also add the fairness constraint that $r_{\Box p} \vee \neg p$ holds infinitely often. These constraints are typical of tableau constructions that convert a temporal formula to a symbolic automaton. As we show in this paper, the additional information they provide refines the trace set of the transformed system, potentially eliminating false counterexamples.

In particular, we will show how to integrate tableau-based prophecy with the liveness-to-safety transformation of [30] that uses a history-based finite projection, referred to as *dynamic abstraction*. We show that the precision of this transformation is consequently increased. The result is that we can prove properties that otherwise would not be directly provable using the technique.

This paper makes the following contributions:

1) Introduce the notion of temporal prophecy, including prophecy formulas and prophecy witnesses, *via* a first-order LTL tableau construction.
2) Show that temporal prophecy increases the proof power (i.e., precision) of the safety-to-liveness transformation based on dynamic abstraction, and further show that the properties provable with tem-

ISBN: 978-0-9835678-8-2.

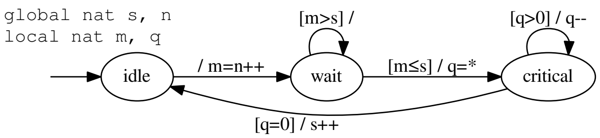


Figure 1. The ticket mutual exclusion protocol. Edges are labeled by condition / action.

poral prophecy are closed under *first-order reasoning*, with cut elimination as a special case.

3) Integrate the liveness-to-safety transformation based on dynamic abstraction and temporal prophecy into the Ivy deductive verification system, deriving the prophecy formulas from an inductive invariant provided by the user (for proving the safety property).
4) Demonstrate the effectiveness of the approach on some challenging examples that cannot be handled by the transformation without temporal prophecy.
5) Demonstrate that prophecy witnesses can eliminate quantifier alternations in the verification conditions generated for the safety problem obtained after the transformation, facilitating decidable reasoning.

2. Illustrative Example

We illustrate our approach using the ticket protocol for ensuring mutual exclusion with non starvation among multiple threads, depicted in Fig. 1. The ticket protocol may be run by any number of threads, and also allows dynamic spawning of threads. The protocol is an idealized version of spinlocks used in the Linux kernel [13]. In the protocol, each thread can be in one of three states: idle, waiting to enter the critical section, or in the critical section. The right to enter the critical section is determined by a ticket number. A global variable n, records the next available ticket, and a global variable s, records the ticket currently being served. Each thread has a local variable m that records the ticket it holds. A thread only enters the critical section when $m \leq s$. Once a thread enters the critical section, it handles tasks that accumulated in its task queue, and stays in the critical section until its queue is empty (tasks are only added to the queue when the thread is outside the critical section). In Fig. 1, this is modeled by the task counter q, a thread-local variable which is non-deterministically set when a thread enters the critical section (to account for the unbounded, but finite, number of tasks), and is then decremented in each step. When $q = 0$ the thread leaves the critical section, and increments s to allow other threads to be served.

The protocol is designed to satisfy the following first-order temporal property:

$$(\forall x. \Box \Diamond \mathit{scheduled}(x)) \to \forall y. \Box \left(\mathit{wait}(y) \to \Diamond \mathit{critical}(y)\right)$$

That is, if every process is scheduled infinitely often, then every waiting process eventually enters its critical section. (Note that we encode fairness assumptions as part of the temporal property.)

Insufficiency of liveness-to-safety transformations. While the temporal property is clearly satisfied by the ticket protocol, proving it is challenging for liveness-to-safety transformations. First, due to the unbounded values obtained by the ticket number and the task counter, and also due to dynamic spawning of threads, this example does not belong to the class of parameterized systems [32], where a simple lasso argument is sound (and complete) for proving liveness. Second, while using a finite abstraction can recover soundness, no fixed finite abstraction is precise enough to show the absence of a lasso-shaped counterexample in this example. The reason is that a thread can go to the waiting state (*wait*) with any number of threads waiting “ahead of it in line”.

For cases where no finite abstraction is sufficiently precise to prove liveness, we may instead apply the liveness-to-safety transformation of [30]. This transformation relaxes the requirement of proving absence of lassos over a fixed finite abstraction, and instead requires one to prove absence of lassos over a *dynamic* finite abstraction that is only determined after some prefix of the trace (allowing for better precision). Soundness is maintained since the abstraction is still finite. Technically, the technique requires to prove that no *abstract lasso* exists, where an abstract lasso is a finite execution prefix that (i) visits a *freeze point*, at which a finite projection (abstraction) of the state space is fixed, (ii) the freeze point is followed by two states that are equal in the projection. We refer to these as the *repeating states*, and (iii) all fairness constraints are visited both before the freeze point and between the repeating states.

Unlike fixed finite abstractions, dynamic abstractions allow us to prove that an eventuality holds if there is a finite upper bound on the number of steps required *at the time the eventuality is asserted* (the freeze point). The bound need not be fixed *a priori*. Unfortunately, due to the non-determinism introduced by the task counter q, each of the k threads ahead of t in line could require an unbounded number of steps to leave the critical section, and this number is not yet determined when t makes its request. As a result, there is an abstract lasso which freezes the abstraction when t makes its request, after which some other thread t_0 enters the critical section and loops, decrementing its task counter q. Since the value of the task counter of t_0 is not captured in the abstraction, the loop does not change the abstract state. This spurious abstract lasso prevents this liveness-to-safety transformation from proving the property.

Temporal prophecy to the rescue. The key to fixing this problem is to predict the future to the extent that a bound on the steps required for progress is determined at the freeze point. Surprisingly, this is accomplished by the use of one temporal prophecy variable corresponding to the truth value of the following formula:

$$\exists x. \Diamond \Box \mathit{critical}(x).$$

If this formula is initially true, there is some thread t_0 that eventually enters the critical section and stays there. At this point, we can prove it eventually exits (a contradiction)

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

because the number of steps needed for this is bounded by the current task counter of t_0. Operationally, the freeze point is delayed until $\Box \mathit{critical}(x)$ holds at which point t_0's task counter is captured in the finite projection, ruling out an abstract lasso. On the other hand if the prophecy variable is initially false, then all threads are infinitely often out of the critical section. With this fairness constraint, thread t requires only a finite number of steps to be served, determined by the number of threads with lesser tickets. Operationally, the extra fairness constraint extends the lasso loop until the abstract state must change, ruling out an abstract lasso.

Though the liveness-to-safety transformation via dynamic abstraction and abstract lasso detection cannot handle the problem as given, introducing suitable temporal prophecy eliminates the spurious abstract lassos. Some spurious lassos are eliminated by postponing the freeze point, thus refining the finite abstraction, and others are eliminated by additional fairness constraints on the lasso loop. This example is explained in greater detail in § 4.3.

3. Preliminaries

In this section, we present the first-order formalism for specifying infinite-state systems and their properties, as well as a tableau construction for first-order LTL formulas.

3.1. Transition Systems in First-Order Logic

A first-order logic transition system is a triple (Σ, ι, τ), where Σ is a first-order vocabulary that contains only relation symbols and constant symbols (functions can be encoded by relations), ι is a closed formula over Σ defining the set of initial states, and τ is a closed formula over $\Sigma \uplus \Sigma'$, where $\Sigma' = \{\ell' \mid \ell \in \Sigma\}$, defining the transition relation. The constants in Σ represent the program variables.

A state of the transition system is a first-order structure, $s = (\mathcal{D}, \mathcal{I})$, over Σ, where $\mathcal{D}$ denotes the (possibly infinite) domain of the structure and $\mathcal{I}$ denotes the interpretation function. The set of initial states is the set of all states s such that $s \models \iota$, and the set of transitions is the set of all pairs of states (s, s') with the same domain such that $(s, s') \models \tau$. In the latter, (s, s') denotes a structure over the vocabulary $\Sigma \uplus \Sigma'$ with the same domain as s and s' in which the symbols in Σ are interpreted as in s, and the symbols in Σ' are interpreted as in s'.

For a state $s = (\mathcal{D}, \mathcal{I})$ over Σ, and for $D \subseteq \mathcal{D}$, we denote by $s|_D$ the partial structure by projecting s to D, i.e., $s|_D = (D, \mathcal{I}|_D)$, where $\mathcal{I}|_D$ interprets only constants $c \in \Sigma$ for which $\mathcal{I}(c) \in D$ (making it a partial interpretation), and for every relation symbol $r \in \Sigma$ of arity k, $\mathcal{I}|_D(r) = \mathcal{I}(r) \cap D^k$. For a vocabulary $\Sigma' \subseteq \Sigma$, we denote by $s|_{\Sigma'}$ the state over Σ' obtained by restricting the interpretation function to the symbols in Σ', i.e., $s|_{\Sigma'} = (\mathcal{D}, \mathcal{I}')$, where for every symbol $\ell \in \Sigma'$, $\mathcal{I}'(\ell) = \mathcal{I}(\ell)$.

A (finite or infinite) *trace* of (Σ, ι, τ) is a sequence of states $\pi = s_0, s_1, \ldots$ where $s_0 \models \iota$ and $(s_i, s_{i+1}) \models \tau$ for every $0 \leq i < |\pi|$. Every state along the trace has its own interpretation of the constant and relation symbols, but they all share the same domain.

We note that first-order transition systems are Turing-complete. Furthermore, tools such as Ivy [31] provide modeling languages that are closer to imperative programming languages and compile to a first-order transition system. This makes it easier for a user to provide a first-order specification of the transition system they wish to verify.

Safety. Given a vocabulary Σ, a safety property P is a set of sequences of states over Σ, such that for every sequence of states $\pi \notin P$, there exists a finite prefix π' of π, such that π' and all of its extensions are not in P. A transition system over Σ satisfies P if all of its traces are in P.

3.2. First-Order Linear Temporal Logic (FO-LTL)

To specify temporal properties of first-order transition systems we use First-Order Linear Temporal Logic (FO-LTL), which combines LTL with first-order logic [1]. For simplicity, we consider only the "globally" ($\Box$) temporal operator. The tableau construction extends to other operators as well, and so does our approach.

Syntax. Given a first-order vocabulary Σ, FO-LTL formulas are defined by:

$$\begin{aligned} f &::= r(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg f \mid f_1 \vee f_2 \mid \exists x.f \mid \Box f \\ t &::= c \mid x \end{aligned}$$

where r is an n-ary relation symbol in Σ, c is a constant symbol in Σ, x is a variable, each t_i is a term over Σ and $\Box$ denotes the "globally" temporal operator. We also use the standard shorthand for the "eventually" temporal operator: $\Diamond f = \neg\Box\neg f$, and the usual shorthands for logical operators (e.g., $\forall x.f = \neg\exists x.\neg f$).

Semantics. FO-LTL formulas over Σ are interpreted over infinite sequences of states (first-order structures) over Σ. Atomic formulas are interpreted over states, the temporal operators are interpreted as in traditional LTL, and first-order quantifiers are interpreted over the shared domain $\mathcal{D}$ of all states in the trace. Formally, the semantics is defined w.r.t. an infinite sequence of states $\pi = s_0, s_1, \ldots$ and an assignment σ that maps variables to $\mathcal{D}$ — the shared domain of all states in π. We define $\pi^i = s_i, s_{i+1}, \ldots$ to be the suffix of π starting at index i. The semantics is defined as follows.

$$\begin{aligned} \pi, \sigma \models r(t_1, \ldots, t_n) &\Leftrightarrow s_0, \sigma \models r(t_1, \ldots, t_n) \\ \pi, \sigma \models t_1 = t_2 &\Leftrightarrow s_0, \sigma \models t_1 = t_2 \\ \pi, \sigma \models \neg\psi &\Leftrightarrow \pi, \sigma \not\models \psi \\ \pi, \sigma \models \psi_1 \vee \psi_2 &\Leftrightarrow \pi, \sigma \models \psi_1 \text{ or } \pi, \sigma \models \psi_2 \\ \pi, \sigma \models \exists x.\psi &\Leftrightarrow \text{exists } d \in \mathcal{D} \text{ s.t. } \pi, \sigma[x \mapsto d] \models \psi \\ \pi, \sigma \models \Box\psi &\Leftrightarrow \text{forall } i \geq 0,\ \pi^i, \sigma \models \psi \end{aligned}$$

When the formula has no free variables, we omit σ. A first-order transition system (Σ, ι, τ) satisfies a closed FO-LTL formula φ over Σ if all of its traces satisfy φ.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

3.3. Tableau for FO-LTL

As part of our liveness-to-safety transformation, we use a standard tableau construction for FO-LTL formulas that results in a first-order transition system with *fairness constraints*. Unlike the classical construction, we define the tableau for a set of formulas, not necessarily a single temporal formula.

For an FO-LTL formula φ, we denote by $sub(\varphi)$ the set of subformulas of φ, defined in the usual way. In the sequel, we consider a finite set A of FO-LTL formulas that is closed under subformulas, i.e. for every $\varphi \in A$, $sub(\varphi) \subseteq A$. Note that A may contain formulas with free variables.

Definition 1 (Tableau vocabulary). Given a finite set A as above over a first-order vocabulary Σ, the *tableau vocabulary* for A, denoted Σ_A, is obtained from Σ by adding a fresh relation symbol $r_{\Box\varphi}$ of arity k for every formula $\Box\varphi \in A$ with k free variables.

Recall that $\Box$ is the only primitive temporal operator we consider (a similar construction can be done for other operators). The symbols added in Σ_A will be used to "label" states by temporal subformulas that are satisfied by all outgoing fair traces. To translate temporal formulas over Σ to first-order formulas over Σ_A we use the following definition.

Definition 2. For a FO-LTL formula $\varphi \in A$ (over Σ), its first-order representation, denoted $\mathrm{FO}[\varphi]$, is a first-order formula over Σ_A, defined inductively, as follows.

$$\begin{aligned}
\mathrm{FO}[\varphi] &= \varphi \quad \text{if } \varphi = r(t_1, \ldots, t_n) \text{ or } \varphi = t_1 = t_2 \\
\mathrm{FO}[\Box\psi(\overline{x})] &= r_{\Box\psi(\overline{x})}(\overline{x}) \\
\mathrm{FO}[\neg\psi] &= \neg\mathrm{FO}[\psi] \\
\mathrm{FO}[\psi_1 \vee \psi_2] &= \mathrm{FO}[\psi_1] \vee \mathrm{FO}[\psi_2] \\
\mathrm{FO}[\exists x.\psi] &= \exists x.\mathrm{FO}[\psi]
\end{aligned}$$

Note that $\mathrm{FO}[\varphi]$ has the same free variables as φ. We can now define the tableau for A as a transition system.

Definition 3 (Tableau transition system). The *tableau transition system* for A is the first-order transition system $T_A = (\Sigma_A, \mathit{true}, \tau_A)$, where τ_A (defined over $\Sigma_A \uplus \Sigma_A{}'$) is defined as follows:

$$\tau_A = \bigwedge_{\Box\varphi \in A} \forall\overline{x}.\ (r_{\Box\varphi}(\overline{x}) \leftrightarrow (\mathrm{FO}[\varphi(\overline{x})] \wedge r_{\Box\varphi}{}'(\overline{x}))).$$

Note that the original symbols in Σ (and Σ') are not constrained by τ_A, and may change arbitrarily with each transition. However, the $r_{\Box\varphi}$ relations are updated in accordance with the property that $\pi, \sigma \models \Box p$ iff $s_0, \sigma \models p$ and $\pi^1, \sigma \models \Box p$ (where s_0 is the first state of π and p is a first-order formula over Σ).

Definition 4 (Fairness). A sequence of states $\pi = s_0, s_1, \ldots$ over Σ_A is *A-fair* if for every temporal formula $\Box\varphi(\overline{x}) \in A$ and for every assignment σ, there are infinitely many i's for which $s_i, \sigma \models \mathrm{FO}[\Box\varphi(\overline{x}) \vee \neg\varphi(\overline{x})]$.

Note that $\Box\varphi(\overline{x}) \vee \neg\varphi(\overline{x})$, used above, is equivalent to $\Diamond\neg\varphi(\overline{x}) \to \neg\varphi(\overline{x})$. So the definition of fairness ensures an eventuality cannot be postponed forever. In the sequel, the set A is always clear from the context (e.g., from the vocabulary), hence we omit it and simply say that π is fair.

The next claims summarize the properties of the tableau; Lemma 1 states that the FO-LTL formulas over Σ that hold in the outgoing traces of a tableau state correspond to the first-order formulas over Σ_A that hold in the state; Lemma 2 states that every sequence of states over Σ has a representative trace in the tableau; finally, Thm. 1 states that a transition system satisfies a FO-LTL formula iff its product with the tableau of the negated formula has no fair traces.

Lemma 1. In a fair trace $\pi = s_0, s_1, \ldots$ of T_A (over Σ_A), for every FO-LTL formula $\psi(\overline{x}) \in A$, for every assignment σ and for every index $i \in \mathbb{N}$, we have that $s_i, \sigma \models \mathrm{FO}[\psi(\overline{x})]$ iff $\pi^i, \sigma \models \psi(\overline{x})$.

Lemma 2. Every infinite sequence of states $\hat{s}_0, \hat{s}_1, \ldots$ over Σ can be extended to a fair trace $\pi = s_0, s_1, \ldots$ of T_A (over Σ_A) s.t. for every $i \in \mathbb{N}$, $s_i|_\Sigma = \hat{s}_i$.

Definition 5 (Product system). Given a transition system $T_S = (\Sigma, \iota, \tau)$, a closed FO-LTL formula φ over Σ, a finite set A of FO-LTL formulas over Σ closed under subformulas such that $\neg\varphi \in A$, we define the *product system* of T_S and $\neg\varphi$ over A as the first-order transition system $T_P = (\Sigma_P, \iota_P, \tau_P)$ given by $\Sigma_P = \Sigma_A$, $\iota_P = \iota \wedge \mathrm{FO}[\neg\varphi]$ and $\tau_P = \tau \wedge \tau_A$, where $T_A = (\Sigma_A, \mathit{true}, \tau_A)$ is the tableau for A.

Theorem 1. Let T_P be the product system of T_S and $\neg\varphi$ over A as defined in Def. 5. Then $T_S \models \varphi$ iff T_P has no fair traces.

Intuitively, the product system augments the states of T_S with temporal formulas from A, splitting each state into many (often infinitely many) states according to the future behavior of its outgoing traces. Note that Thm. 1 holds already when $A = sub(\neg\varphi)$. However, as we will see, taking a larger set A is useful for proving fair termination via the liveness-to-safety transformation.

4. Liveness-to-Safety with Temporal Prophecy

In this section we present our liveness proof approach using temporal prophecy and a liveness-to-safety transformation. As in earlier approaches, our transformation (i) uses a tableau construction to construct a product transition system equipped with fairness constraints such that the latter has no fair traces iff the temporal property holds of the original system, and (ii) defines a safety property over the product transition system such that safety implies that no fair traces exist (note that the opposite direction does not hold).

The gist of our liveness-to-safety transformation is that we augment the construction of the product transition system with two forms of prophecy detailed in § 4.2. We then use the definition of the safety property from [30]. In the sequel, we first present the safety property and then present the augmentation with temporal prophecy, whose goal is to "refine" the product system such that it will be safe.

4.1. Safety Property: Absence of Abstract Lassos

Given a transition system $T_W = (\Sigma_W, \iota_W, \tau_W)$ with $\Sigma_W \supseteq \Sigma_A$ (e.g., the product system from Def. 5), we define a notion of an abstract lasso, whose absence is a safety property that implies that T_W has no A-fair traces. This section recapitulates material from [30].

The definition of an abstract lasso is based on a dynamic abstraction that is fixed at some point along the trace, henceforth called the *freeze point*. The abstraction function is defined by projecting a state (a first-order structure) into a finite subset of its domain. This finite subset is defined by the union of the *footprints* of all states encountered until the freeze point, where the footprint of a state includes the interpretation it gives all constants from Σ_W. Intuitively, the footprint includes all elements "exposed" in the state, including those "touched" by outgoing transitions.

Definition 6 (Footprint). For a state $s = (\mathcal{D}, \mathcal{I})$ over Σ_W, we define the footprint of s as $f(s) = \{\mathcal{I}(c) \mid c \in \Sigma_W\}$. For a sequence of states $\pi = s_0, s_1, \ldots$ over Σ_W, and an index $i < |\pi|$, we define the footprint of $s_0, \ldots, s_i$ as $f(s_0, \ldots, s_i) = \bigcup_{j=0}^{i} f(s_j)$.

Importantly, the footprint of a finite trace is always finite. As a result, an abstraction function that maps each state to the result of projecting it to the footprint of the trace until the freeze point has a finite range.

Definition 7 (Fair Segment). Let $\pi = s_0, s_1, \ldots$ be a sequence of states over Σ_W. For $0 \leq i \leq j < |\pi|$, we say the segment $[i, j]$ is fair if for every formula $\Box\psi(\overline{x}) \in A$, and for every assignment σ where every variable is assigned to an element of $f(s_0, \ldots, s_i)$, there exists $i \leq k \leq j$ s.t. $s_k, \sigma \models \mathrm{FO}\left[(\Box\psi(\overline{x})) \vee \neg\psi(\overline{x})\right]$.

Definition 8 (Abstract Lasso). A finite trace $s_0, \ldots, s_n$ of T_W is an *abstract lasso* if there are $0 \leq i \leq j < k \leq n$ s.t. the segments $[0, i]$ and $[j, k]$ are fair, and $s_j|_{f(s_0,\ldots,s_i)} = s_k|_{f(s_0,\ldots,s_i)}$.

Intuitively, in the above definition, i is the *freeze point*, where the abstraction is fixed. The states s_j and s_k are the "repeating states" – states that are indistinguishable by the abstraction that projects them to the footprint $f(s_0, \ldots, s_i)$. The segment between j and k, respectively, the segment between 0 and i, meet all the fairness constraints restricted to elements in $f(s_0, \ldots, s_j)$, respectively, in $f(s_0)$. Fairness of the segment $[0, i]$ is needed to prevent the freeze point from being chosen too early, thus creating spurious abstract lassos. Note that the absence of abstract lassos is a safety property.

Lemma 3. If T_W has no abstract lassos then it also has no fair traces.

Proof: Assume to the contrary that T_W has a fair trace $\pi = s_0, s_1, \ldots$. Let i be the first index such that $[0, i]$ is fair (such an index must exist since the set $f(s_0)$, which determines the relevant fairness constraints is finite). Since $f(s_0, \ldots, s_i)$ is also finite, there must exist an infinite subsequence π' of π^i such that for every s, s' in this subsequence $s|_{f(s_0,\ldots,s_i)} = s'|_{f(s_0,\ldots,s_i)}$. Let $j \geq i$ be the index in π of the first state in π'. $f(s_0, \ldots, s_j)$ is also finite, hence there exists $k' > j$ such that the segment $[j, k']$ of π is fair. Take k to be the index in π of the first state of $\pi^{k'}$ that is also in π'. Since π' is infinite, such a k must exist. Since $k \geq k'$, the segment $[j, k]$ is also fair. This defines an abstract lasso $s_0, \ldots, s_i, \ldots, s_j, \ldots, s_k$, in contradiction. □

4.2. Augmenting the Transition System with Temporal Prophecy

In this section we explain how our liveness-to-safety transformation constructs $T_W = (\Sigma_W, \iota_W, \tau_W)$, to which we apply the safety property of § 4.1. Our construction exploits both *temporal prophecy formulas* and *prophecy witnesses*, explained below. For the rest of this section we fix a first-order transition system $T_S = (\Sigma, \iota, \tau)$ and a closed FO-LTL formula φ over Σ that we wish to verify in T_S.

Temporal Prophecy Formulas. First, given a set A of (not necessarily closed) FO-LTL formulas closed under subformula that contains $\neg\varphi$, we construct the product system $T_P = (\Sigma_P, \iota_P, \tau_P)$ defined in Def. 5. By Thm. 1, $T_S \models \varphi$ iff T_P has no fair traces. Note that classical tableau constructions are defined with $A = \mathit{sub}(\neg\varphi)$, and we allow A to include more formulas. These formulas act as "temporal prophecy variables" in the sense that they split the states of T_S, according to the future behavior of outgoing traces.

While the liveness-to-safety transformation is already sound with $A = \mathit{sub}(\neg\varphi)$, one of the chief observations of this work is that temporal prophecy formulas improve its precision. These additional formulas in A split the states of T_S into more states in T_P, and they cause some nondeterminism of the future trace to be "pulled backwards" (the outgoing traces contain less non-determinism). For example, if $r_{\Box\varphi}$ holds for some elements in the current state, then φ must continue to hold for these elements in the future of the trace. Similarly, for elements where $r_{\Box\varphi}$ does not hold, there will be some time in the future of the trace where φ would not hold for them.

This is exploited by the liveness-to-safety transformation in three ways, eliminating spurious abstract lassos. First, having more temporal formulas in A refines the definition of a fair segment, and postpones the freeze point, thus making the abstraction defined by the footprint up to the freeze point more precise. For example, if $r_{\Box\varphi}$ does not hold for a ground formula φ in the initial state, then the freeze point would be postponed until after φ does not hold for the first time. Second, it strengthens the requirement on the looping segment $s_j \ldots s_k$, in a similar way. Third, the additional relations in $\Sigma_P (= \Sigma_A)$ are part of the state as considered by the transformation, and a difference in these relations (projected to the footprint up to the freeze point) is a valid difference. These three ways all played a role in the examples considered in our evaluation.

Prophecy Witnesses. The notion of an abstract lasso, used to define the safety property, considers a finite abstraction according to the footprint, which depends on the constants of the vocabulary. To increase the precision of the

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

abstraction, we augment the vocabulary with fresh constants that serve as *prophecy witnesses* for existential properties.

To illustrate the idea, consider the formula $\psi(x) = \Diamond\Box p(x)$ where x is a free variable. If ψ holds for some element, it is useful to include in the vocabulary a constant that serves as a witness for $\psi(x)$, and whose interpretation will be taken into account by the abstraction. If ψ holds for some x, the interpretation of the constant will be taken from such an x. Otherwise, this constant will be allowed to take any value.

Temporal prophecy witnesses not only refine the abstraction, they can also be used in the inductive invariant. In particular, as demonstrated in the TLB Shootdown example (see § 6), in some cases this allows to avoid quantifier alternation cycles in the verification conditions, leading to decidability of VC checking.

Formally, given a set $B \subseteq A$, we construct $T_W = (\Sigma_W, \iota_W, \tau_W)$ as follows. We extend Σ_P to Σ_W by adding fresh constant symbols $c_1, \ldots, c_n$ for every formula $\psi(x_1, \ldots, x_n) \in B$. We denote by C the set of new constants, i.e., $\Sigma_W = \Sigma_P \cup C$. The transition relation formula is extended to keep the new constants unchanged, i.e. $\tau_W = \tau_P \wedge \bigwedge_{c \in C} c = c'$, and we define ι_W by

$$\iota_W = \iota_P \wedge \mathrm{FO}\left[(\exists x_1, \ldots, x_n.\psi(x_1, \ldots, x_n)) \to \psi(c_1, \ldots, c_n)\right]$$

Namely, $c_1, \ldots, c_n$ are required to serve as witnesses for $\psi(x_1, \ldots, x_n)$ in case it holds in the initial state for some elements, and otherwise they may get any interpretation at the initial state, after which their interpretation remains unchanged. Adding these fresh constants and their defining formulas to the initial state is a conservative extension, in the sense that every fair trace of T_P can be extended to a fair trace of T_W (fairness of traces over $\Sigma_W \supseteq \Sigma_A$ is defined as in Def. 4), and every fair trace of T_W can be projected to a fair trace of T_P. As such we have the following:

Lemma 4. Let $T_P = (\Sigma_P, \iota_P, \tau_P)$ and $T_W = (\Sigma_W, \iota_W, \tau_W)$ be defined as above. Then T_P has no fair traces iff T_W has no fair traces.

The overall soundness of the liveness-to-safety transformation is given by the following theorem.

Theorem 2 (Soundness). Given a first-order transition system T_S and a closed FO-LTL formula φ both over Σ, and given a set of temporal prophecy formulas A over Σ that contains $\neg\varphi$ and is closed under subformula, and a set of temporal prophecy witness formulas $B \subseteq A$, if T_W as defined above does not contain an abstract lasso, then $T_S \models \varphi$.

4.3. The Ticket Example

In this section we show in greater detail how prophecy increases the power of the liveness-to-safety transformation. As an illustration we return to the ticket example of Fig. 1. As explained in § 2, in this example the liveness-to-safety transformation without temporal prophecy fails (similarly to [30, §5.2]), but it succeeds when adding suitable temporal prophecy.

To model the ticket example as a first-order transition system, we use a vocabulary with two sorts: *thread* and *number*. The first represents threads, and the second represents ticket values and counter values. The vocabulary also includes a static binary relation symbol $\leq$: *number*, *number*, with suitable first-order axioms to make it a total order. (for more details about modeling systems in first-order logic see e.g. [31].) The state of the system is modeled by unary relations for the program counter: *idle*, *wait*, *critical*, constant symbols of sort *number* for the global variables n, s, and functions from *thread* to *number* for the local variables m, c. The vocabulary also includes a unary relation *scheduled*, which holds the last scheduled thread.

Next we show that when adding the temporal prophecy formula $\exists x.\Diamond\Box \mathit{critical}(x)$ to the tableau construction, no abstract lasso exists in the augmented transition system, hence the liveness-to-safety transformation succeeds to prove the property. Formally, in this case, A includes the following two formulas and their subformulas:

$$\neg\left((\exists x.\neg\Box\neg\Box\neg \mathit{scheduled}(x)) \vee \neg\exists x.\neg\Box\left(\neg \mathit{wait}(x) \vee \neg\Box\neg \mathit{critical}(x)\right)\right)$$
$$\exists x.\neg\Box\neg\Box \mathit{critical}(x)$$

And $B = \{\neg\Box\left(\neg \mathit{wait}(x) \vee \neg\Box\neg \mathit{critical}(x)\right), \neg\Box\neg\Box \mathit{critical}(x)\}$. Therefore, Σ_W extends the original vocabulary with the following 6 unary relations:

$$r_{\Box\neg \mathit{scheduled}(x)}, r_{\Box\neg\Box\neg \mathit{scheduled}(x)}, r_{\Box\neg \mathit{critical}(x)},$$
$$r_{\Box\neg \mathit{wait}(x) \vee \neg\Box\neg \mathit{critical}(x)}, r_{\Box \mathit{critical}(x)}, r_{\Box\neg\Box \mathit{critical}(x)}$$

as well as two constants for prophecy witnesses: c_1 for $\neg\Box\left(\neg \mathit{wait}(x) \vee \neg\Box\neg \mathit{critical}(x)\right)$, and c_2 for $\neg\Box\neg\Box \mathit{critical}(x)$.

We now explain why there is no abstract lasso. To do this, we show that the tableau construction, combined with the dynamic abstraction and the fair segment requirements, result in the same reasoning that was presented informally in § 2.

First, observe that from the definition of c_1 and the negation of the liveness property (both assumed by ι_W), we have that the initial state $s_0 \models \mathrm{FO}\left[\neg\Box\left(\neg \mathit{wait}(c_1) \vee \neg\Box\neg \mathit{critical}(c_1)\right)\right]$. For brevity, denote $p = (\neg \mathit{wait}(c_1) \vee \neg\Box\neg \mathit{critical}(c_1))$, so we have $s_0 \models \mathrm{FO}\left[\neg\Box p\right]$, i.e., $s_0 \models \neg r_{\Box p}$. Since c_1 is also in the footprint of the initial state, the fair segment requirement ensures that the freeze point can only happen after encountering a state satisfying: $\mathrm{FO}\left[(\Box p) \vee \neg p\right] \equiv r_{\Box p} \vee \mathrm{FO}\left[\neg p\right]$. Recall that the transition relation of the tableau (τ_A), ensures $(r_{\Box p}) \leftrightarrow (\mathrm{FO}\left[p\right] \wedge {r_{\Box p}}')$. Therefore, on update from a state satisfying $\neg r_{\Box p}$ to a state satisfying $r_{\Box p}$ can only happen if the pre-state satisfies $\mathrm{FO}\left[\neg p\right]$. Therefore, the freeze point must come after encountering a state that satisfies $\mathrm{FO}\left[\neg p\right] \equiv \mathit{wait}(c_1) \wedge r_{\Box\neg \mathit{critical}(c_1)}$. From the freeze point onward, τ_A will ensure both $r_{\Box\neg \mathit{critical}(c_1)}$ and $\neg \mathit{critical}(c_1)$ continue to hold, so c_1 will stay in *wait* (since the protocol does not allow to go from *wait* to anything but *critical*). So, we see that the mechanism of the tableau, combined with the temporal prophecy witness and the fair segment requirement, ensures that the freeze point happens after c_1

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

makes a request that is never granted. This will ensure that the footprint used for the dynamic abstraction will include all threads ahead of c_1 in line, i.e., those with smaller ticket numbers.

As for c_2, the initial state will either satisfy $\mathrm{FO}\left[\neg\Box\neg\Box \mathit{critical}(c_2)\right]$ or it would satisfy $\mathrm{FO}\left[\neg\exists x.\neg\Box\neg\Box \mathit{critical}(x)\right]$. In the first case, by an argument similar to the one used above for c_1, the freeze point will happen after c_2 enters the critical section and then stays in it. Therefore, the footprint used for the dynamic abstraction will include all numbers smaller than q of c_2 when it enters the critical section[1]. Since c_2 is required to be scheduled between the repeating states (again by the tableau construction and the fair segment requirement), its value for q will be decreased, and this will be visible in the dynamic abstraction. Thus, in this case, an abstract lasso is not possible.

In the second case the initial state satisfies $\mathrm{FO}\left[\neg\exists x.\neg\Box\neg\Box \mathit{critical}(x)\right]$. By a similar argument that combines the tableau with the fair segment requirement for the repeating states, we will obtain that between the repeating states, any thread in the footprint of the first repeating state, must both be scheduled and visit a state outside the critical section. In particular, this includes all threads that are ahead of c_1 in line. This entails a change to the program counter of one of them (the one that had a ticket number equal to the service number at the first repeating state), which will be visible in the abstraction. Thus, an abstract lasso is not possible in this case either.

5. Closure Under First-Order Reasoning

The transformation from temporal verification to safety verification developed in § 4 introduces an abstraction, and incurs a loss of precision. That is, for some systems and properties, liveness holds but the safety of the resulting system does not hold, no matter what temporal prophecy is used. (This is unavoidable for a transformation from arbitrary FO-LTL properties to safety properties [30].) However, in this section, we show that the set of instances for which the transformation can be made precise (via temporal prophecy) is closed under first-order reasoning. This is unlike the transformation of [30]. It shows that the use of temporal prophecy results in a particular kind of robustness.

We consider a proof system in which the above transformation is performed and the resulting safety property is checked by an oracle. That is, for a transition system T_S and a temporal property φ (a closed FO-LTL formula), we write $T_S \vdash \varphi$ if there exist finite sets of FO-LTL formulas A and B satisfying the conditions of Thm. 2, such that resulting transition system T_W is safe, i.e., does not contain an abstract lasso. We now show that the relation $\vdash$ satisfies a powerful closure property.

Theorem 3 (Closure under first-order reasoning). Let T_S be a transition system, and $\psi, \varphi_1, \ldots, \varphi_n$ be closed FO-LTL formulas, such that $\mathrm{FO}\left[\varphi_1 \wedge \ldots \wedge \varphi_n\right] \models \mathrm{FO}\left[\psi\right]$. If $T_S \vdash \varphi_i$ for all $1 \leq i \leq n$, then $T_S \vdash \psi$.

The condition that $\mathrm{FO}\left[\varphi_1 \wedge \ldots \wedge \varphi_n\right] \models \mathrm{FO}\left[\psi\right]$ means that $\varphi_1 \wedge \ldots \wedge \varphi_n$ entails ψ when using only first-order reasoning, and treating temporal operators as uninterpreted. The theorem states that provability using the liveness-to-safety transformation is closed under such reasoning. Two special cases of Thm. 3 given by the following corollaries:

Corollary 1 (Modus Ponens). If T_S is a transition system and φ and ψ are closed FO-LTL formulas such that $T_S \vdash \varphi$ and $T_S \vdash \varphi \to \psi$, then $T_S \vdash \psi$.

Corollary 2 (Cut). If T_S is a transition system and φ and ψ are closed FO-LTL formulas such that $T_S \vdash \varphi \to \psi$ and $T_S \vdash \neg\varphi \to \psi$, then $T_S \vdash \psi$.

Proof of Thm. 3: In the proof we use the notation $T_W(T_S, \varphi, A, B)$ to denote the transition system constructed for T_S and φ when using A, B as temporal prophecy formulas. Likewise, we refer to the vocabulary, initial states and transition relation formulas of the transition system as $\Sigma_W(T_S, \varphi, A, B)$, $\iota_W(T_S, \varphi, A, B)$, and $\tau_W(T_S, \varphi, A, B)$, respectively. Let $(A_1, B_1), \ldots, (A_n, B_n)$ be such that $T_W(T_S, \varphi_i, A_i, B_i)$ has no abstract lasso, for every $1 \leq i \leq n$. Now, let $A = \bigcup_{i=1}^{n} A_i$ and $B = \bigcup_{i=1}^{n} B_i$. We show that $T_W(T_S, \psi, A, B)$ has no abstract lasso. Assume to the contrary that $s_0, \ldots, s_i, \ldots, s_j, \ldots, s_k, \ldots, s_n$ is an abstract lasso for $T_W(T_S, \psi, A, B)$. Since $s_0 \models \iota_W(T_S, \psi, A, B)$, we know that $s_0 \models \neg\mathrm{FO}\left[\psi\right]$, and since $\mathrm{FO}\left[\varphi_1 \wedge \ldots \wedge \varphi_n\right] \models \mathrm{FO}\left[\psi\right]$, there must be some $1 \leq \ell \leq n$ s.t. $s_0 \models \neg\mathrm{FO}\left[\varphi_\ell\right]$. Denote $\Sigma' = \Sigma_W(T_S, \varphi_\ell, A_\ell, B_\ell)$. Now, $s_0|_{\Sigma'}, \ldots, s_i|_{\Sigma'}, \ldots, s_j|_{\Sigma'}, \ldots, s_k|_{\Sigma'}, \ldots, s_n|_{\Sigma'}$ is an abstract lasso of $T_W(T_S, \varphi_\ell, A_\ell, B_\ell)$, which is a contradiction. To see that, we first simplify the notation and denote $s_m|_{\Sigma'}$ by $\hat{s}_m$. The footprint $f(s_0, \ldots, s_i)$ contains more elements than the footprint $f(\hat{s}_0, \ldots, \hat{s_i})$, since $\Sigma_W(T_S, \psi, A, B) \supseteq \Sigma_W(T_S, \varphi_\ell, A_\ell, B_\ell)$. Therefore, given that $s_j|_{f(s_0,\ldots,s_i)} = s_k|_{f(s_0,\ldots,s_i)}$, we have that $\hat{s}_j|_{f(\hat{s}_0,\ldots,\hat{s_i})} = \hat{s}_k|_{f(\hat{s}_0,\ldots,\hat{s_i})}$ as well. Moreover, the fairness constraints in $T_W(T_S, \varphi_\ell, A_\ell, B_\ell)$, determined by A_ℓ, are a subset of those in $T_W(T_S, \psi, A, B))$, determined by A, so the segments $[0, i]$ and $[j, k]$ are also fair in $T_W(T_S, \varphi_\ell, A_\ell, B_\ell)$. □

The proof of Thm. 3 sheds more light on the power of using temporal prophecy formulas that are not subformulas of the temporal property to prove. In particular, the theorem does not hold if A is restricted to subformulas of the temporal proof goal.

6. Implementation & Evaluation

We have implemented our approach for temporal verification and integrated it into the Ivy deductive verification system [31]. This allows the user to model the transition system in the Ivy language (which internally translates into

---

[1] When modeling natural numbers in first-order logic, the footprint is adjusted to include all numbers lower than any constant (still being a finite set).

a first-order transition system), and express temporal properties directly in FO-LTL. In our implementation, the safety property that results from the liveness-to-safety transformation is proven by a suitable inductive invariant, provided by the user. To facilitate this process, Ivy internally constructs a suitable monitor for the safety property, i.e., the absence of abstract lasso's in T_W. The user then provides an inductive invariant for T_W composed with this monitor. The monitor keeps track of the footprint and the fairness constraints, and non-deterministically selects the freeze point and repeated states of the abstract lasso. Similar to the construction of [5], the monitor keeps a shadow copy of the "saved state", which is the first of the two repeated states. These are maintained via designated relation symbols (in addition to Σ_W). The user's inductive invariant must then prove that it is impossible for the monitor to detect an abstract lasso.

Mining Temporal Prophecy from the Invariant. As presented in previous sections, our liveness-to-safety transformation is parameterized by sets of formulas A and B. In the implementation, these sets are implicit, and are extracted automatically from the inductive invariant provided by the user. Namely, the inductive invariant provided by the user contains temporal formulas, and also prophecy witness constants, where every temporal formula $\Box\varphi$ is a shorthand (and is internally rewritten to) $r_{\Box\varphi}$. The set A to be used in the construction is defined by all the temporal subformulas that appear in the inductive invariant (and all their subformulas), and the set B is defined according to the prophecy witness constants that are used in the inductive invariant.

In particular, the user's invariant may refer to the satisfaction of each fairness constraint $\mathrm{FO}\left[\Box\varphi \vee \neg\varphi\right]$ for $\Box\varphi \in A$, both before the freeze point and between the repeated states, via a convenient syntax provided by Ivy.

Interacting with Ivy. If the user provides an inductive invariant that is not inductive, Ivy presents a graphical counterexample to induction. This guides the user to adjust the inductive invariant, which may also lead to new formulas being added to A or B, if the user adds new temporal formulas or prophecy witnesses to the inductive invariant. In this process, the user's mental image is of a liveness-to-safety transformation where A and B include all (countably many) FO-LTL formulas over the system's vocabulary, so the user is free to use any temporal formula, or prophecy witness for any formula. However, since the user's inductive invariant is a finite formula, the liveness-to-safety transformation needs only to be applied to finite A and B, and the infinite A and B are just a mental model.

We have used our implementation to prove liveness for several challenging examples, summarized in Fig. 2. We focused on examples that were beyond reach for the liveness-to-safety transformation of [30]. In [30], such examples were handled using a nesting structure. Our experience shows that with temporal prophecy, the invariants are simpler than with a nesting structure (for additional comparison with nesting structure see § 7). For all examples we considered, the verification conditions are in a decidable fragment of first-order logic which is supported by Z3 (the stratified extension of EPR [19], [31]). Interestingly, for the TLB shootdown example, the proof presented in [30] (using a nesting structure) required non-stratified quantifier alternation, which is eliminated by the use of temporal prophecy witnesses. Due to the decidability of verification conditions, Z3 behaves predictably, and whenever the invariant is not inductive it produces a finite counterexample to induction, which Ivy presents graphically. Our experience shows that the graphical counterexamples provide valuable guidance towards finding an inductive invariant, and also for coming up with temporal prophecy formulas as needed. Below we provide more detail on each example.

Protocol	**# A**	**# B**	**# LOC**	**# C**	**FO-LTL**	**t** [sec]
Ticket w/ Task Queues	1	2	90	60	22%	9.4
Alternating Bit Protocol	4	1	143	70	40%	32
TLB Shootdown	6	3	468	102	49%	283

Figure 2. Protocols for which we verified liveness. For each protocol, **# A** reports the number of temporal prophecy formulas used. **# B** reports the number of prophecy witnesses used. **# LOC** reports the number of lines of code for the system model (without proof) in Ivy's modeling language. **# C** reports the number of conjectures used in the inductive invariant (a typical conjecture is one or few lines). **FO-LTL** reports the fraction of the conjectures that use temporal formulas. Finally, **t** reports the run time (in seconds) for checking the verification conditions using Ivy and Z3. The experiments were performed on a laptop running 64-bit Linux, with a Core-i7 1.8 GHz CPU, using Z3 version 4.6.0.

Ticket. The ticket example has been discussed in § 1, and § 4.3 contains more details about its proof with temporal prophecy, using a single temporal prophecy formula and two prophecy witness constants. To give a flavor of what the proof looks like in Ivy, we present a couple of the conjectures that make up the inductive invariant for the resulting system, in Ivy's syntax. In Ivy, the prefix `l2s` indicates symbols that are introduced by the liveness-to-safety transformation. Some conjectures are needed to state that the footprint used in the dynamic abstraction contains enough elements. An example of such a conjecture is:

```
l2s_frozen & (globally critical(c2)) ->
    forall N.  N <= q(c2) -> l2s_a(N)
```

This conjecture states that after the freeze point (indicated by the special symbol `l2s_frozen`), if the prophecy witness `c2` (which is the prophecy witness defined for $\Diamond\Box \mathit{critical}(x)$) is globally in the critical section, then the finite domain of the frozen abstraction (stored in the unary relation `l2s_a`) contains all numbers up the `c2`'s value for `q`. Other conjectures are needed to show that the current state is different from the saved state. One example is:

```
l2s_saved & (globally critical(c2)) &
   ~($l2s_w X. scheduled(X))(c2) ->
       q(c2) ~= ($l2s_s X. q(X))(c2)
```

The special operator `$l2s_w` lets the user query whether a fairness constraint has been encountered, and `$l2s_s` exposes to the user the saved state (both syntactically λ-like binders). This conjecture states that after we saved a shadow state (indicated by `l2s_save`), if the prophecy witness `c2` is globally in the critical section, and if we

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

have encountered the fairness constraints associated with $\mathit{scheduled}(x) \vee \Box\neg\mathit{scheduled}(x)$ instantiated for `c2` (which can only happen after `c2` has been scheduled), then the current value `c2` has for `q` is different from the same value in the shadow state.

Alternating Bit Protocol. The alternating bit protocol is a classic communication algorithm for transition of messages using lossy first-in-first-out (FIFO) channels. The protocol uses two channels: a data channel from the sender to the receiver, and an acknowledgment channel from the receiver to the sender. The sender and the receiver each have a state bit, and messages include a bit that functions as a "sequence number". We assume that the sender has an (infinite) array of values to send, which is filled by some independent process. The liveness property we wish to prove is that every value entered into the sender array is eventually received by the receiver.

The protocol is live under fair scheduling assumptions, as well as standard fairness constraints for the channels: if messages are infinitely often sent, then messages are infinitely often received. This makes the structure of the temporal property more involved. Formally, the liveness property we prove is:

$$\begin{aligned}
&(\Box\Diamond \mathit{sender\_scheduled}) \wedge (\Box\Diamond \mathit{receiver\_scheduled}) \wedge \\
&((\Box\Diamond \mathit{data\_sent}) \to (\Box\Diamond \mathit{data\_received})) \wedge \\
&((\Box\Diamond \mathit{ack\_sent}) \to (\Box\Diamond \mathit{ack\_received})) \to \\
&\forall x. \Box(\mathit{sender\_array}(x) \neq \bot \to \Diamond \mathit{receiver\_array}(x) \neq \bot))
\end{aligned}$$

This property cannot be proven without temporal prophecy. However, it can be proven using 4 temporal prophecy formulas: $\{\Diamond\Box\,(\mathit{sender\_bit} = s \wedge \mathit{receiver\_bit} = r) \mid s, r \in \{0,1\}\}$. Intuitively, these formulas make a distinction between traces in which the sender and receiver bits eventually become fixed, and traces in which they change infinitely often.

TLB Shootdown. The TLB shootdown algorithm [6] is used (e.g. in the Mach operating system) to maintain consistency of Translation Look-aside Buffers (TLB) across processors. When some processor (dubbed the initiator) changes the page table, it interrupts all other processors currently using the page table (dubbed the responders) and waits for them to receive the interrupt before making changes. The liveness property we prove is that no processor can become stuck either as an initiator or as a responder (formally, it will respond or initiate infinitely often). This liveness depends on fair scheduling assumptions, as well as strong fairness assumptions for the page table locks used by the protocol. We use one witness for the process that does not satisfy the liveness property. Another witness is used for a pagemap that is never unlocked, if this exists. A third witness is used for a process that possibly gets stuck while holding the lock blocking the first process. We use six prophecy formulas to case split on when some process may get stuck. Two of them are used for the two loops in the initiator to distinguish the cases whether the process that hogs the lock gets stuck there. They are of the form $\Diamond\Box pc(c_2) \in \{i_3, \ldots, i_8\}$. Two are used for the two lock instructions to indicate that the first process gets stuck: $\Diamond\Box pc(c_1) = i_2$. And two are used for the second and third witness to indicate whether such a witness exists, e.g., $\Diamond\Box plock(c_3)$. Compared to the proof of [30], our proof is simpler due to the temporal prophecy, and avoids non-stratified quantifier alternation, resulting in decidable verification conditions.

7. Related Work

Prophecy variables were first introduced in [2], in the context of refinement mappings. There, prophecy variables are required to range over a finite domain to ensure soundness. Our notion of prophecy via first-order temporal formulas and witness constants does not meet this criterion, but is still sound as assured by Thm. 2. In [25], LTL formulas are used to define prophecy variables in a way that is similar to ours, but only to show refinement between finite-state processes. We use temporal prophecy defined by FO-LTL formulas in the context of infinite-state systems. Furthermore, we consider a liveness-to-safety transformation (rather than refinement mappings), which can be seen as a proof system for FO-LTL.

The liveness-to-safety transformation based on dynamic abstraction, but *without temporal prophecy*, was introduced in [30]. There, a *nesting structure* was used to increase the power of the transformation. A nesting structure is defined by the user (via first-order formulas), and has the effect of splitting the transition system into levels (analogous to nested loops) and proving each level separately. Temporal prophecy as we introduce here is more general, and in particular, any proof that is possible with a nesting structure, is also possible with temporal prophecy (by adding a temporal prophecy formula $\Diamond\Box\delta$ for every nesting level, defined by δ). Moreover, the nesting structure does not admit cut elimination or closure under first-order reasoning, and is therefore less robust.

One effect of prophecy is to split cases in the proof on some aspect of the future. This very general idea occurs in various approaches to liveness, particularly in the large body of work on lexicographic or disjunctive rankings for termination [4], [7], [8], [11], [12], [14], [18], [20], [21], [23], [26], [27], [33], [34], [35], [36], [37], [38]. In the work of [22], the partitioning of the space of potentially infinite executions is based on the *a priori* decomposition of regular expressions for iterated loop segments. Often the partitioning here amounts to a split according to a fairness condition ("command a is taken infinitely often or it is not"). The partitioning is constructed dynamically (and represented explicitly through a union of Buchi automata) in [24] (for termination), in [15] (for liveness), and in [17] (for liveness of parameterized systems). None of these works uses a temporal tableau construction to partition the space of futures, however.

Here, we use prophecy to, in effect, partially determinize a system by making non-deterministic choices earlier in an execution. This same effect was used for a different purpose in refining an abstraction from LTL to ACTL [10] and checking CTL* properties [9]. The prophecy in this case relates only to the next transition and is not expressed

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

temporally. The method of "temporal case splitting" in [29] can also be seen as a way to introduce prophecy variables to increase the precision of an abstraction, though in that case the transformation was to finite-state liveness, not infinite-state safety. Moreover, it only introduces temporal witnesses.

We have considered only proof methods that transform liveness to safety (which includes the classical ranking approach for while loops). There are approaches, however, which do *not* transform liveness to safety. For example, the approaches in [3], [14], [39] are essentially forms of widening in a CTL-style backwards fixpoint iteration. It is not clear to what extent temporal prophecy might be useful in increasing the precision of such abstractions, but it may be an interesting topic for future research.

8. Conclusion

We have seen that the addition of prophecy variables in the form of temporal formulas can increase the precision of liveness-to-safety tranformations for infinite-state systems. The prophecy variables are derived from additional temporal formulas that in our implementation were mined from the invariants a user provides to prove the safety property. This approach is effective for proving challenging examples. By increasing the precision of the dynamic abstraction, it avoided the need to decompose the proof into nested termination arguments, reducing the human effort of proof construction. Though completeness is not possible, we saw that the additional expressiveness of temporal prophecy provides a cut elimination property. While we considered temporal prophecy using a particular liveness-to-safety construction (based on dynamic abstraction), it seems reasonable to expect that the tableau-based approach would apply to other constructions and abstractions, including constructions based on rankings and well-founded relations. Because our approach relies on an inductive invariant supplied by the user, it requires the user to understand the liveness-to-safety transformation and it requires both cleverness and a deep understanding of the protocol. For this reason, a possible avenue for future research would be to explore invariant synthesis techniques, and in particular ones that account for refinement due to temporal prophecy.

Acknowledgements. We thank the anonymous referees for insightful comments which improved this paper. Padon was supported by Google under a PhD fellowship. Padon and Sagiv were supported by the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement no. [321174-VSSC]. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, and by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University. This material is based upon work supported by the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259.

References

[1] M. Abadi, "The power of temporal proofs," *Theor. Comput. Sci.*, vol. 65, no. 1, pp. 35–83, 1989. [Online]. Available: https://doi.org/10.1016/0304-3975(89)90138-2

[2] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P

[3] P. A. Abdulla, B. Jonsson, A. Rezine, and M. Saksena, "Proving liveness by backwards reachability," in *CONCUR*, ser. Lecture Notes in Computer Science, vol. 4137. Springer, 2006, pp. 95–109.

[4] D. Babic, A. J. Hu, Z. Rakamaric, and B. Cook, "Proving termination by divergence," in *SEFM*, 2007, pp. 93–102.

[5] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electr. Notes Theor. Comput. Sci.*, vol. 66, no. 2, pp. 160–177, 2002.

[6] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation lookaside buffer consistency: A software approach," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS III. New York, NY, USA: ACM, 1989, pp. 113–122. [Online]. Available: http://doi.acm.org/10.1145/70082.68193

[7] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 413–429.

[8] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI*, 2006, pp. 415–426.

[9] B. Cook, H. Khlaaf, and N. Piterman, "On automation of ctl* verification for infinite-state systems," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 13–29. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_2

[10] B. Cook and E. Koskinen, "Making prophecies with decision predicates," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 399–410. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926431

[11] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, no. 5, pp. 88–98, 2011.

[12] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *TACAS*, 2013, pp. 47–61.

[13] J. Corbet, "Ticket spinlocks," https://lwn.net/Articles/267968/, 2008.

[14] P. Cousot and R. Cousot, "An abstract interpretation framework for termination," in *POPL*, 2012, pp. 245–258.

[15] D. Dietsch, M. Heizmann, V. Langenfeld, and A. Podelski, "Fairness modulo theory: A new approach to LTL software model checking," in *CAV*, ser. Lecture Notes in Computer Science, vol. 9206. Springer, 2015, pp. 49–66.

[16] Y. Fang, K. L. McMillan, A. Pnueli, and L. D. Zuck, "Liveness by invisible invariants," in *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, ser. Lecture Notes in Computer Science, E. Najm, J. Pradat-Peyre, and V. Donzeau-Gouge, Eds., vol. 4229. Springer, 2006, pp. 356–371. [Online]. Available: https://doi.org/10.1007/11888116_26

[17] A. Farzan, Z. Kincaid, and A. Podelski, "Proving liveness of parameterized programs," in *LICS*. ACM, 2016, pp. 185–196.

[18] P. Ganty and S. Genaim, "Proving termination starting from the end," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 397–412.

ISBN: 978-0-9835678-8-2.

[19] Y. Ge and L. D. Moura, “Complete instantiation for quantified formulas in satisfiabiliby modulo theories,” in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 306–320.

[20] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, “Automated termination proofs with AProVE,” in *RTA*, 2004, pp. 210–220.

[21] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing software verifiers from proof rules,” in *PLDI*, 2012, pp. 405–416.

[22] S. Gulwani, S. Jain, and E. Koskinen, “Control-flow refinement and progress invariants for bound analysis,” in *PLDI*, 2009, pp. 375–385.

[23] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani, “Alternation for termination,” in *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, 2010, pp. 304–319.

[24] M. Heizmann, J. Hoenicke, and A. Podelski, “Termination analysis by learning terminating programs,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 797–813.

[25] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck, “Network invariants in action,” in *Proceedings of the 13th International Conference on Concurrency Theory*, ser. CONCUR ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 101–115. [Online]. Available: http://dl.acm.org/citation.cfm?id=646737.701938

[26] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger, “Termination analysis with compositional transition invariants,” in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, 2010, pp. 89–103.

[27] W. Lee, B. Wang, and K. Yi, “Termination analysis with algorithmic learning,” in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, pp. 88–104.

[28] R. Manevich, B. Dogadov, and N. Rinetzky, “From shape analysis to termination analysis in linear time,” in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 9779. Springer, 2016, pp. 426–446.

[29] K. L. McMillan, “A methodology for hardware verification using compositional model checking,” *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 279–309, 2000. [Online]. Available: https://doi.org/10.1016/S0167-6423(99)00030-1

[30] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, “Reducing liveness to safety in first-order logic,” *PACMPL*, vol. 2, no. POPL, pp. 26:1–26:33, 2018. [Online]. Available: http://doi.acm.org/10.1145/3158114

[31] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 614–630.

[32] A. Pnueli and E. Shahar, “Liveness and acceleration in parameterized verification,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 328–343.

[33] A. Podelski and A. Rybalchenko, “Transition invariants,” in *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, 2004, pp. 32–41.

[34] ——, “Transition predicate abstraction and fair termination,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 132–144.

[35] C. Urban, “The abstract domain of segmented ranking functions,” in *SAS*, ser. Lecture Notes in Computer Science, vol. 7935. Springer, 2013, pp. 43–62.

[36] C. Urban, A. Gurfinkel, and T. Kahsai, “Synthesizing ranking functions from bits and pieces,” in *TACAS*, ser. Lecture Notes in Computer Science, vol. 9636. Springer, 2016, pp. 54–70.

[37] C. Urban and A. Miné, “An abstract domain to infer ordinal-valued ranking functions,” in *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 412–431.

[38] ——, “A decision tree abstract domain for proving conditional termination,” in *SAS*, ser. Lecture Notes in Computer Science, vol. 8723. Springer, 2014, pp. 302–318.

[39] ——, “Inference of ranking functions for proving temporal properties by abstract interpretation,” *Computer Languages, Systems & Structures*, vol. 47, pp. 77–103, 2017.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Automatic Synchronization for GPU Kernels

Sourav Anand
UC San Diego

Nadia Polikarpova
UC San Diego

***Abstract*—We present a technique for automatic synthesis of efficient and provably correct synchronization in GPU kernels. Our technique relies on an off-the-shelf correctness oracle and achieves efficient synthesis by leveraging the *race location* information provided by the oracle in order to encode optimal synchronization synthesis as a MaxSAT problem. We have implemented our technique in a tool called AUTOSYNC that works on kernels written in CUDA and uses a static verifier GPUVERIFY as the correctness oracle. An evaluation on 18 realistic kernels from the GPUVERIFY benchmark suite shows that AUTOSYNC is able to synthesize optimal synchronization placements, and synthesis times are reasonable (20 seconds for our largest benchmark).**

I. INTRODUCTION

Recent years have seen increasing use of *graphics processing units* (GPUs) for speeding up general-purpose computations. GPU computations are highly parallel—with thousands of threads running concurrently—which creates ample opportunity for data races. To prevent races, programmers add *barrier synchronization* statements to their GPU code. Because synchronization incurs a performance penalty, a GPU programmer is faced with a challenging task of finding a placement of barrier statements that is both *correct* (eliminates all data races) and *optimal* (incurs the least overhead).

In response to this challenge, several verification techniques have been proposed [1], [2], [3], [4], [5], [6] for detecting data races in GPU code or proving their absence. These techniques would alert the programmer that a barrier statement is missing, but they neither suggest where to place the barrier, nor check whether the current placement is optimal. In this work we propose a computer-aided approach to GPU programming, where the programmer omits barrier statements from their code altogether, and our technique automatically synthesizes a correct and optimal barrier placement.

Barrier Synthesis. One approach to barrier synthesis is to search the space of all possible barrier placements, using an existing GPU verification tool [6], [4] as a black-box *correctness oracle*; among all correct placements, we can then select an optimal one according to some *cost model*. The benefit of such a black-box approach is that it automatically takes advantage of any current and future advances in GPU verification. Brute-force enumeration, however, would be prohibitively expensive for most programs, since the number of possible placements grows exponentially with the size of the program, and verifying each candidate placement is an expensive operation on its own.

In this paper we show how to leverage the *race location* provided by the oracle and the conservative *operational semantics* used in verification in order to avoid considering most invalid placements and thereby make the synthesis practical. Moreover, we demonstrate how to encode this information together with the cost model as a system of *soft Boolean constraints*, which allows our technique to delegate the bulk of the search to MaxSAT solvers. Our technique is *sound* and *complete* relative to the correctness oracle.

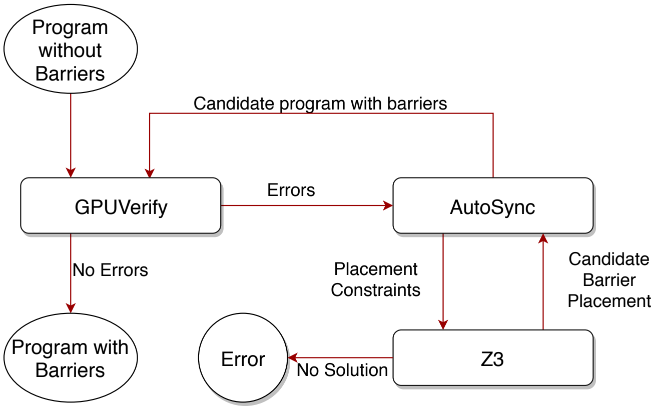


Fig. 1: The AUTOSYNC workflow

AUTOSYNC. We have implemented this constraint-based approach to barrier synthesis in a tool called AUTOSYNC (Fig. 1). The tool takes as input GPU programs—or *kernels*—written in the popular CUDA programming model, and uses the sound static verifier GPUVERIFY [6] as the correctness oracle. For constraint-based search, the tool relies on the νZ MaxSAT solver [7], which is part of Z3 [8].

Evaluation. We have evaluated AUTOSYNC on a series of small but challenging micro-benchmarks, as well as 18 realistic CUDA kernels from the GPUVERIFY benchmark suite. Our evaluation shows that in all these benchmarks, AUTOSYNC is able to recover a barrier placement that is at least as optimal as the one originally provided by the developer. Surprisingly, in 5 cases the automatically generated placement is *strictly better* than the original. Moreover, synthesis times are moderate and range from 1 to under 30 seconds. AUTOSYNC and all our benchmarks are available at www.souravanand.com/autosync.html.

II. MOTIVATING EXAMPLES

This section goes through a series of examples of data races in GPU programs, showcases the challenges of finding correct and optimal barrier placements, and provides the intuition for how AUTOSYNC addresses these challenges.

ISBN: 978-0-9835678-8-2.

```
1 x = A[tid+1];
2 x = x+11;
3 A[tid] += x;
```

```
1 x = A[tid + 1];
2 x = x+11;
3 __syncthreads();
4 A[tid] += x;
```

Fig. 2: (left) A kernel with a race, and (right) the correctly synchronized version of the kernel.

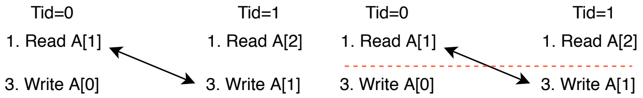


Fig. 3: Execution traces of kernels in Fig. 2 (left) and (right); the arrow depicts a data race; the dotted line depicts a barrier.

A. *Straight-line Code*

In the CUDA programming model, programmers describe a GPU computation as a *kernel*: a template to be executed by each GPU thread, implicitly parametrized by a unique thread id. For example, a simple kernel in Fig. 2 (left) instructs each thread to read from a shared array `A` at a distinct index, which depends on the thread's id **`tid`**, and then write into the array at the preceding index.

Fig. 3 (left) depicts an execution of this kernel by two threads with ids 0 and 1. This execution exhibits a read-write race: since the two threads are not synchronized, the read from `A[1]` by thread 0 is racing with the write to `A[1]` by thread 1. Eliminating this race requires adding a *barrier* statement `__syncthreads()` between the two racing instructions, as shown in Fig. 2 (right). A barrier requires all the threads to reach it before any thread can continue execution. When thread 1 encounters the barrier, it is forced to wait until thread 0 encounters the same barrier; hence the read from `A[1]` is now guaranteed to happen before the write to the same location.

Barrier synthesis. Given the kernel in Fig. 2 (left), AutoSync first checks its correctness using GPUVerify (see Fig. 1), which reports a possible data race between lines 1 and 3. Based on this race location information, AutoSync generates a *placement constraint*:

$$L_1 \vee L_2$$

Here each L_i is a propositional variable that indicates whether a barrier should be inserted after line i. Although any solution to this constraint would eliminate the race, setting more than one L_i to 1 is suboptimal, since every barrier incurs a performance overhead. To avoid suboptimal solutions, AutoSync adds a *soft constraint* $\neg L_i$ for each line i in the program, which penalizes the solver for setting any L_i to 1. The resulting system of constraints is discharged by Z3's MaxSAT solver, producing the solution $\{L_2\}$[1]. The corresponding barrier placement, shown in Fig. 2 (right), is proven correct by GPUVerify, and the synthesis succeeds.

```
1 for(i=0; i<n; i++){
2   x = A[tid+1];
3   x = x+i;
4   A[tid] += x;
5 }
```

```
1 for(i=0; i<n; i++) {
2   __syncthreads();
3   x = A[tid+1];
4   x = x+i;
5   __syncthreads();
6   A[tid] += x;
7 }
```

Fig. 4: (left) A kernel with a race inside a loop, and (right) the correctly synchronized version of the kernel.

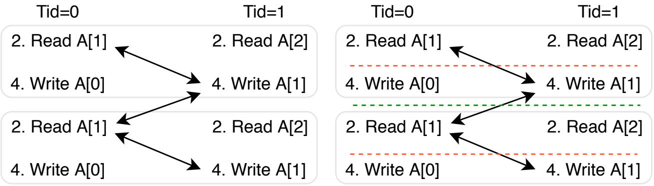


Fig. 5: Execution traces of kernels in Fig. 4 (left) and (right); each gray box corresponds to one loop iteration.

B. *Loops*

Given our first example, the reader might be wondering if all data races can be eliminated by simply inserting a barrier right before the second racing line. In the presence of loops, however, barrier placement becomes more challenging. Consider the kernel in Fig. 4 (left), which preforms a similar computation, but inside a loop. GPUVerify again reports a race between the two accesses to `A` (lines 2 and 4), however, adding a barrier between these two lines turns out to be *insufficient* to make the kernel race-free.

To see why, consider the execution trace of this kernel depicted in Fig. 5: a write to `A[1]` by thread 1 can race with a read executed by thread 0 either in the same loop iteration (*intra-iteration race*) or in a different one (*inter-iteration race*). Adding the "red" barrier between lines 2 and 4 forces the write to happen after the read in the same iteration, but imposes no order with the read from the next iteration. To synchronize this kernel correctly, a second barrier must be added *within* the loop body but *outside* the two racing lines ("green" barrier after line 1 in Fig. 4 and Fig. 5).

Barrier synthesis. Since the racing lines reported by GPUVerify are inside a loop, AutoSync generates the following system of placement constraints:

$$\begin{gathered} P^0 \vee P^1 \\ P^0 \Rightarrow (L_2 \vee L_3) \\ P^1 \Rightarrow (L_1 \vee L_4) \end{gathered}$$

Intuitively, the verification error does not contain enough information to determine the type of the race—intra-iteration, inter-iteration, or both—hence we encode the possibility of each race type using a fresh propositional variable (P^0 for intra-iteration and P^1 for inter-iteration). As before, AutoSync generates soft constraints $\neg L_i$ for all lines i, however, the

---

[1] We write a solution as a set of all variables set to 1.

```
1  x = 0;
2  if(tid%2==0) {
3    x = A[tid+2];
4  }
5  if(tid%6==0) {
6    A[tid] += x;
7  }
```

```
1  x = 0;
2  if(tid%2==0) {
3    x = A[tid+2];
4  }
5  __syncthreads();
6  if(tid%6==0){
7    A[tid] += x;
8  }
```

Fig. 6: (left) A kernel with a race between two different basic blocks, and (right) the correctly synchronized version of kernel.

```
1  x=A[tid+1];
2  y=B[tid+1];
3  A[tid] = x+y;
4  B[tid] = x-y;
```

```
1  x=A[tid+1];
2  y=B[tid+1];
3  __syncthreads();
4  A[tid] = x+y;
5  B[tid] = x-y;
```

Fig. 7: (left) A kernel with two data races, and (right) the correctly synchronized version that requires just one barrier.

lines inside the loop are given a *higher weight*. This forces the solver to prefer placing barriers outside the loop, whereby minimizing performance overhead.

Given these constraints, Z3 might return a solution $\{P^0, L_3\}$, which violates only one soft constraint and corresponds to adding the "red" barrier after line 3 alone. An attempt to verify this solution reveals that the race is still present. Hence, in a second iteration of barrier synthesis, AutoSync asks Z3 for a different assignment to the P variables, by adding a constraint $\neg(P^0 \wedge \neg P^1)$. The second solution, $\{P^1, L_1\}$ (the "green" barrier alone) does not solve the race either. In the third iteration, AutoSync further adds a constraint $\neg(\neg P^0 \wedge P^1)$, which forces the solver to set both P^j to 1 and results in the final solution $\{P^0, P^1, L_1, L_3\}$.

Nested loops. In general, if both racing statements are inside a loop nest of depth d, we have to consider $d + 1$ possibilities: one intra-iteration and d inter-iteration races as different depths. If the race is inter-iteration, placing a barrier at depth d is always sufficient, but "shallower" barriers incur less overhead.

C. Barrier Divergence

Conditionals also complicate barrier placement. Consider the kernel in Fig. 6 (left), where lines 3 and 6 are racing. Placing a barrier right after line 3 or right before line 6 (i.e. inside a conditional) would make that barrier unreachable for some of the threads executing the kernel, leading to undefined behavior due to so called *barrier divergence* [1]. The only correct solution is to insert the barrier between the two conditionals, as shown in Fig. 6 (right).

Luckily, GPUVerify detects and reports if a candidate barrier placement might cause barrier divergence. In response to this error, AutoSync adds a hard constraint that excludes all lines within the problematic **if**-block from consideration.

D. Multiple Races

When a kernel contains multiple data races, analyzing and eliminating each race independently might lead to a suboptimal barrier placement. Consider the kernel in Fig. 7 (left) with two pairs of racing lines: $\langle 1, 3\rangle$ and $\langle 2, 4\rangle$. Considering the two races independently might result in inserting two barrier statements, whereas in fact, a single barrier after line 2 eliminates both races. Such interactions between different races are difficult for programmers to reason about. AutoSync, on the other hand, generates the optimal placement in the first iteration, since $\{L_2\}$ is the least-cost solution to the placement constraints $[L_1 \vee L_2, L_2 \vee L_3]$.

```
kernel  ::= blk
blk     ::= {(ℓ : stmt)*}
stmt    ::= local name | name := expr | barrier
          | name := rd(expr) | wr(expr, expr)
          | if expr blk | while expr blk
expr    ::= name | tid | n | expr op expr
```

Fig. 8: The syntax of KPL

III. SYNTHESIS ALGORITHM

This section formalizes our synchronization synthesis algorithm for KPL (*Kernel Programming Language*), a core language we borrow from the work on GPUVerify [1].

A. Kernel Programming Language

Syntax. The syntax of KPL is presented in Fig. 8. Expressions `expr` are thread-local (do not access shared memory). Reading and writing from/to shared memory is accomplished via the statements **rd** and **wr**, respectively. A reserved variable **tid** gives the execution thread access to its unique id, which enables different threads to execute different behavior. Compared to the presentation in [1], we omit jump statements and **else** branches of conditionals (both can be desugared into our language in a standard way), and procedures, which—while not technically challenging—are currently not supported.

Each statement in a kernel is *labeled* with a unique label ℓ; $\mathrm{stmt}(\ell)$ denotes the statement with label ℓ. Labels of compound statements—**if** and **while**—double as labels of their enclosed blocks; the top-level block of the kernel has a reserved label $main$. A kernel's *label tree* is a tree whose nodes are statement labels, and a node's parent is the label of its enclosing block; in addition, we add a special *start node* ℓ_s as the leftmost child of each block[2]. We use $\mathrm{blks}(\ell)$ to denote the set of enclosing blocks of ℓ (its ancestors in the label tree); among those, $\mathrm{loops}(\ell)$ are the enclosing **while** blocks and $\mathrm{conds}(\ell)$ are the enclosing **if** blocks.

---

[2] Thus, to place a barrier at the beginning of the block, we place it after ℓ_s.

Sometimes we interpret these sets of blocks as sequences, ordered from the root downward. We define the *program text order* $\prec$ on labels as the *post-order* of the label tree. A *label interval* $[\ell_1, \ell_2)$ denotes the set of labels ℓ that lie between ℓ_1 and ℓ_2 in the program text ($\ell_1 \preceq \ell \prec \ell_2$) and share all enclosing blocks with at least one of the interval bounds ($\mathrm{blks}(\ell) \subseteq (\mathrm{blks}(\ell_1) \cup \mathrm{blks}(\ell_2))$). For example, on Fig. 6 (left):[3] $\mathrm{blks}(3) = \{\mathit{main}, 2{:}4\}$; $\mathrm{blks}(5{:}7) = \{\mathit{main}\}$; $[3, 6) = \{3, 2{:}4, 5{:}7_s\}$, while $[3, 5{:}7) = \{3, 2{:}4\}$ ($5{:}7_s$ and 6 are excluded, since they do not share their enclosing block 5:7 with any of the two bounds). Note that the set of all children of a block ℓ can be expressed as the interval $[\ell_s, \ell)$.

Semantics. Prior work [1] defined the semantics of KPL dubbed *synchronous, delayed visibility* (*SDV*). According to this semantics, all threads execute the kernel instructions *synchronously* (in lock step) but the effect of a `wr` statement may be *delayed*, i.e. not immediately visible to other threads. Importantly, the semantics of control structures models so-called *predicated execution*, illustrated informally in Fig. 9. Under predicated execution, the body of an `if` statement is always executed by all threads, but each statement in the body can be *enabled* or *disabled* for a given thread, depending on the value of a *predicate*—a thread-local Boolean variable initialized with the `if` guard; when a statement is disabled, it has no effect. Similarly, a `while` loop is executed the same number of times by all threads: it iterates as long as the loop guard holds for at least one thread. Due to synchronous predicated execution, at any point at run time all threads are always executing the same statement (which, however, might be disabled for some threads). More formally, we can define an execution *trace* as a sequence of instructions $\langle \ell_1, \vec{p}_1 \rangle, \ldots, \langle \ell_n, \vec{p}_n \rangle$, where each ℓ_i is the label of the statement being executed and each $\vec{p}_i = [p_i^1, \ldots, p_i^T]$ is a Boolean vector of predicate values (here T is the total number of threads). A kernel's set of *feasible traces* can be derived from the SDV operational semantics.

Races and synchronization. Delayed visibility leads to a potential *data race* when two distinct threads access the same shared memory location, and at least one of the accesses is a write. Executing a `barrier` statement makes the effect of all previous writes visible to all threads, eliminating a potential data race with any following reads or writes. More formally, we say that a trace $\ldots, \langle \ell_i, \vec{p}_i \rangle, \ldots, \langle \ell_j, \vec{p}_j \rangle, \ldots$ *exhibits a race between* ℓ_i *and* ℓ_j, if $\mathrm{stmt}(\ell_i)$ and $\mathrm{stmt}(\ell_j)$ are potentially conflicting shared memory accesses and $\forall k \in (i, j) : \mathrm{stmt}(\ell_k) \neq$ `barrier`. We say that a trace *exhibits barrier divergence at* ℓ if it contains a barrier instruction $\langle \ell, \vec{p} \rangle$ that is not uniformly enabled, i.e. if $\mathrm{stmt}(\ell) =$ `barrier` and $\exists t, u : p^t \neq p^u$. A kernel is *correctly synchronized* if none of its feasible traces exhibit races or barrier divergence. We define the *kernel synchronization problem* as follows: given a KPL kernel without barriers, find a subset $\mathcal{L}$ of its labels, such that inserting a `barrier` statement as the right sibling of every $\ell \in \mathcal{L}$ yields a correctly synchronized kernel.

```
                                          local p
if e {    local p    |  while e {  p := e
  s1      p := e     |    s1       while ∃ t : t.p {
  s2      p => s1    |    s2         p => s1
}         p => s2    |  }            p => s2
                                       p => p := e }
```

Fig. 9: Predicated form of conditionals (left) and loops (right)

B. Placement Constraints

Our approach to solving the kernel synchronization problem is to encode the set $\mathcal{L}$ as a solution to a system of Boolean *placement constraints* over the propositional variables L_ℓ, for each label ℓ in the kernel. Placement constraints are derived from race locations provided by the verification oracle. A *race location* is a pair of leaf labels $\langle \ell, \ell' \rangle$, such that $\ell \preceq \ell'$ and there exists a feasible trace tr that exhibits a race between ℓ and ℓ' *or* between ℓ' and ℓ. By definition, to eliminate the race in tr, it is sufficient to add a barrier instruction between the two racing instructions. Our key insight is that, thanks to SDV's synchronous predicated execution, this constraint on the barrier position *in the trace* translates into a constraint on its placement *in the program text.*

Consider a data race at location $\langle \ell, \ell' \rangle$. As we demonstrated in Sec. II, barrier placement depends on whether both racing statements are inside the same loop body. More precisely, we identify two types of data races: a *simple race* and a *loop race*.

Simple races arise when $\mathrm{loops}(\ell) \cap \mathrm{loops}(\ell') = \emptyset$. In this case, all occurrences of ℓ in any feasible trace tr precede all occurrences of ℓ', as illustrated in Fig. 3 (with $\ell = 1, \ell' = 3$). Hence, a simple race can always be fixed by placing a single barrier *anywhere* in the interval between the two racing statements, giving rise to the following placement constraint:

$$\bigvee \{L_i \mid i \in [\ell, \ell')\}$$

Loop races arise when both racing statements are inside a nest of loops of depth $d \geq 1$: $\mathrm{loops}(\ell) \cap \mathrm{loops}(\ell') = \{\ell^1, \ldots, \ell^d\}$. In this case, the occurrences of ℓ and ℓ' in tr are *interspersed*, as illustrated in Fig. 5 (with $\ell = 2, \ell' = 4$). Not all pairs of occurrences are necessarily conflicting, but the race location alone has insufficient information to discern which ones are. For barrier placement, we have $d + 1$ options that separate distinct subsets of conflicting instructions in tr.

The first option is to insert an *intra-iteration* barrier: a barrier inside the interval $[\ell, \ell')$. This barrier will separate every occurrence of ℓ in tr from the occurrence of ℓ' *within the same iteration* of the innermost loop ℓ^d, as illustrated by the red barrier in Fig. 5. Alternatively, we can insert an *inter-iteration* barrier: outside the two racing statements, but directly inside the body of one of their shared loops ℓ^j, $j \in [1, d]$. This barrier will separate every occurrence of ℓ from the occurrence of ℓ' in the previous iteration of ℓ^j, as illustrated by the green barrier in Fig. 5. A combination of an inter-iteration barrier at d and an intra-iteration barrier will separate every pair of

---

[3] Here each statement is labeled with its line number or line span.

occurrences of ℓ and ℓ' in tr, and hence is guaranteed to fix the race, but this is also the solution with most run-time overhead. In the interest of optimality, our algorithm explores all non-redundant combinations of intra- and inter-iteration barriers.

To this end, for a loop race $\langle \ell, \ell' \rangle$, we introduce additional propositional variables that encode the choice of placement options: $P^0_{\ell,\ell'}$ for the intra-iteration barrier and $P^j_{\ell,\ell'}$ with $j \in [1, d]$ for each inter-iteration barrier. The system of placement constraints for a loop race then includes a guarded constraint for each placement option:

$$P^0_{\ell,\ell'} \Rightarrow \bigvee \{L_i \mid i \in [\ell, \ell')\}$$
$$P^j_{\ell,\ell'} \Rightarrow \bigvee \{L_i \mid i \in [\ell^j_s, \ell) \cup [\ell', \ell^j)\}$$

Since only some placement options actually fix the race, the synthesis engine iterates through all possible P-assignments, calling the oracle to validate the corresponding candidate solution. In each iteration, the placement constraints also contain the negation of each previously encountered invalid P-assignment, including the initial assignment $P_{\ell,\ell'} = \vec{0}$, which corresponds to the input program without barriers. Finally, to avoid exploring redundant placement combinations, we add a constraint $\neg(P^j_{\ell,\ell'} \wedge P^k_{\ell,\ell'})$ for all $j, k \geq 1, j \neq k$, since an inter-iteration barrier in an inner loop always subsumes one in an outer loop.

Divergence. Given a candidate solution with a barrier at ℓ, where $\text{blks}(\ell) = \{main, \ell^1, \ldots, \ell^d\}$, the oracle reports barrier divergence at ℓ, if at least one of $\ell^1, \ldots, \ell^d$ has a thread-dependent guard (in which case the block might not be uniformly enabled for all threads). The synthesis engine responds by extending the placement constraints to disallow barriers inside the innermost block ℓ^d:

$$\bigwedge \{\neg L_i \mid i \in [\ell^d_s, \ell^d)\}$$

In the next iteration, the barrier will be placed outside of ℓ^d; iteration will continue as long as any of the remaining enclosing blocks have thread-dependent guards.

C. Cost Model

Our goal is to design a function $C \colon \mathcal{P}(\mathcal{L}) \to \mathbb{Q}^+$ such that the cost of a barrier placement $\mathcal{L}$ correlates with its overhead on the kernel execution time. Precise static analysis of execution time, however, is a hard problem; hence we opted for a simple cost model that approximates the number of barriers the kernel will encounter during its execution (we evaluate the adequacy of this model empirically in Sec. IV). More precisely, the cost of a placement is the sum of costs of all its barriers, and the cost of an individual barrier depends on the number of its enclosing loops and conditionals:

$$C(\mathcal{L}) = \sum_{\ell \in \mathcal{L}} C(\ell) \quad \text{where } C(\ell) = LC^{|\text{loops}(\ell)|} \times IC^{|\text{conds}(\ell)|}$$

Here, the constants $LC > 1$ and $0 < IC < 1$ conceptually represent the average number of times each loop is executed and the average proportion of times each conditional guard holds. In practice, the algorithm is not very sensitive to the precise values of these constants, since it rarely has to trade-off two solutions with different numbers of barriers. For example, in Fig. 4 (left) with $LC = 100$, $C(1{:}5) = 1$ and $C(2) = 100$.

Algorithm 1 The AUTOSYNC syntheis algorithm

```
 1: procedure SYNTHESIZE($kernel$)
 2:     $S$ = NEWSOLVER
 3:     for $\ell \in kernel$ do $S$.ASSERTSOFT($\neg L_\ell$, $C(\ell)$)
 4:     $races$ = GETRACES($kernel$)
 5:     for $\langle \ell, \ell' \rangle \in races$ do
 6:         $\{\ell^1, \ldots, \ell^d\}$ = $kernel$.loops($\ell$) $\cap$ $kernel$.loops($\ell'$)
 7:         $S$.ASSERT($P^0_{\ell,\ell'} \Rightarrow \vee L_i \mid i \in [\ell, \ell')$)
 8:         for $j \in [1, d]$ do
 9:             $S$.ASSERT($P^j_{\ell,\ell'} \Rightarrow \vee L_i \mid i \in [\ell^j_s, \ell) \cup [\ell', \ell^j)$)
10:             for $k \in [1, d], k \neq j$ do
11:                 $S$.ASSERT($\neg(P^j_{\ell,\ell'} \wedge P^k_{\ell,\ell'})$)
12:         $S$.ASSERT($\vee P^j_{\ell,\ell'} \mid j \in [0, d]$)
13:     return REFINE($kernel, races, S$)

14: procedure REFINE($kernel, races, S$)
15:     $kernel'$ = $kernel$
16:     while $races \neq \emptyset$ do
17:         if $S$.CHECK = UNSAT then
18:             return "No solution"
19:         $\mathcal{L}$ = $S$.MODEL
20:         $kernel'$ = INSERTBARRIERS($kernel, \mathcal{L}$)
21:         $divs$ = GETDIVERGENCES($kernel'$)
22:         if $divs \neq \emptyset$ then
23:             for $\ell \in divs$ do
24:                 $l^d$ = last($kernel'$.blks($\ell$))
25:                 $S$.ASSERT($\wedge \neg L_i \mid i \in [\ell^d_s, \ell^d)$)
26:         else
27:             $races$ = GETRACES($kernel'$)
28:             for $\langle \ell, \ell' \rangle \in races$ do
29:                 $S$.ASSERT($\vee(P^j_{\ell,\ell'} \neq \mathcal{L}[P^j_{\ell,\ell'}]$))
30:     return $kernel'$
```

D. Algorithm

Algorithm 1 describes the full barrier synthesis algorithm. The top-level procedure, SYNTHESIZE, takes as input a KPL $kernel$ and returns a correctly synchronized version of this kernel (or fails).

Initialization. We start by creating a fresh instance of a MaxSAT solver S and asserting soft constraints that penalize a barrier after any label ℓ proportionally to its cost (line 3). In lines 4–12, we query the oracle for the initial set of race locations $races$, and then generate initial placement constraints for each race. For a loop race at depth $d \geq 1$, we generate guarded placement constraints for an intra-iteration barrier (line 7) and all possible inter-iteration barriers (line 9); additionally, line 11 disallows redundant placements (multiple nested inter-iteration barriers), and line 12 forces the solver to place at least one barrier for the current loop race, since the solution with an empty set of barriers is known to be incorrect. When $d = 0$, we are dealing with a simple race; in this case, lines 7 and 12 together generate an appropriate placement constraint.

Refinement loop. After asserting the initial constraints we invoke REFINE. This procedure alternates between asking the solver for a placement $\mathcal{L}$ that satisfies the current constraints and asking the oracle whether $\mathcal{L}$ is valid; if not, the constraints are refined to exclude $\mathcal{L}$ *and* equivalent invalid placements.

The refinement loop starts by asking the solver whether the current set of placement constraints is satisfiable (line 17). If not, the algorithm terminates with failure; otherwise the least-cost placement $\mathcal{L}$ is obtained as the model of the constraints (line 19). Next, INSERTBARRIERS builds a candidate solution $kernel'$ by inserting a `barrier` statement as the right sibling of every $\ell \in \mathcal{L}$ into $kernel$. We assume that INSERTBARRIERS leaves the labels of existing statements unmodified and assigns fresh labels to the barrier statements.

On line 21 we query the oracle for the set $divs$ of barrier divergence locations in the candidate solution. If the barrier at label ℓ is diverging, we can safely exclude all statements in ℓ's innermost enclosing block from consideration (line 25).

In the absence of divergence, we query the oracle for the remaining set of $races$ (line 27). Note that each of these races $\langle \ell, \ell' \rangle$ must be a loop race for which the solver chose an invalid assignment to $P^j_{\ell,\ell'}$. In response, on line 29, we add a constraint that disables the current P-assignment, which prompts the solver to look for the next best combination of placement options in the next iteration.

The procedure terminates either when it finds a valid placement ($races = \emptyset$) or when the current constraints are unsatisfiable (line 18). The latter can happen for two reasons: (1) a race is of the form $\langle \ell, \ell \rangle$—a `wr` statement racing with itself—so the disjunction in line 7 is empty, or (2) a race is inside a block with a thread-dependent guard, so the divergence constraint in line 25 is inconsistent with the other placement constraints for this race. Such races cannot be eliminated by inserting barriers, and hence are out of scope.

E. Guarantees

Soundness. A synchronization synthesis algorithm is *sound* if every solution it returns is correctly synchronized. Since Algorithm 1 relies on the oracle to validate candidate placements, we obtain the soundness guarantee for free as long as the oracle is sound (which is true for GPUVERIFY).

Completeness. A synchronization synthesis algorithm is *complete* if it returns a valid placement as long as one exists. Algorithm 1 is *complete relative to the oracle*: it will discover a placement as long as there is one that the oracle can verify.

Proof. Consider a feasible trace tr that exhibits a race between its i-th and j-th instructions. If this race can be eliminated by barrier placement, it must be that $\exists k \in [i, j) : tr[k] = \langle \ell_k, \vec{1} \rangle$, i.e. a uniformly enabled instruction occurs between i and j, so the barrier can be inserted after ℓ_k. Let us define the set F_{ℓ_i,ℓ_j} of *feasible labels* as follows:

$$F_{\ell_i,\ell_j} = L^d \cap \begin{cases} [\ell_i, \ell_j) & \text{if } \ell_i \prec \ell_j \\ [\ell^d_s, \ell_j) \cup [\ell_i, \ell^d) & \text{if } \ell_j \prec \ell_i \end{cases}$$

where L^d is the set of children of $\mathrm{last}(\mathrm{blks}(\ell_i) \cap \mathrm{blks}(\ell_j))$. In other words, feasible labels are labels in the smallest enclosing block of the two racing instructions, which occur between i and j in the trace. Note that we can safely pick any label from F_{ℓ_i,ℓ_j} as the race solution ℓ_k, because (a) F_{ℓ_i,ℓ_j} is nonempty according to trace semantics, and (b) its labels are the least nested in $[i, j)$, hence they must be uniformly enabled if any $[i, j)$ instructions are.

We can now show that if every trace has a verifiable solution ℓ_k, then the constraints generated by Algorithm 1 never become inconsistent. We build a (non-optimal) model $\mathcal{L}$ of the placement constraints as follows: for every $\langle \ell_1, \ell_2 \rangle \in races$

$$\mathcal{L}[P^j_{\ell_1,\ell_2}] \iff j = 0 \lor j = d$$
$$\mathcal{L}[L_\ell] \iff \ell \in F_{\ell_1,\ell_2} \lor \ell \in F_{\ell_2,\ell_1}$$

This model obviously satisfies line 12; it satisfies lines 7 and 9 by definition of $F_{\ell,\ell'}$; it satisfies line 25 because labels in $F_{\ell,\ell'}$ cannot be divergent if a valid placement exists; finally, because we include at least one feasible label for both orderings of labels in every race, $\mathcal{L}$ is guaranteed to eliminate all races, hence no further constraints will be added in line 29. ☐

As explained in [1], the SDV semantics is *conservative*; in particular, it rejects some barrier placements that could be considered valid if we made more assumptions about the concrete GPU platform. Our synthesis algorithm benefits from SDV in two ways: on the one hand, soundness wrt. SDV guarantees that the resulting kernel will execute correctly on any GPU platform; on the other hand, SDV's synchronous predicated execution helps us prune the search space while maintaining relative completeness.

Termination Procedure REFINE terminates because every iteration eliminates at least one assignment to the propositional variables, and the number of variables is defined by the size of the original kernel. Moreover, the number of iterations is upper-bounded by $2 \times \sum_{\ell \in leaves} \mathrm{depth}(\ell)$.

Optimality. A synchronization synthesis algorithm is *optimal* (relative to a given cost metric) if it always finds the solution with the lowest cost among all valid solutions. Since Algorithm 1 relies on a MaxSAT solver to perform the search, and thanks to the soft constraints in line 3, it always finds the placement $\mathcal{L}$ with the minimal cost $C(\mathcal{L})$ among all models of the placement constraints. Not all valid placements, however, satisfy the constraints. Consider the following snippet:

```
for(i=0;i<20;i++){
  if (i<10 && tid%2==0) {
    x = rd(tid+i+1) }
  if (i<10) {
    x = x + 1 }
  if (i<10 && tid%6==0) {
    wr(tid+i,x) }
}
```

Here, the optimal placement—inside the middle **`if`**-statement—will not be discovered because as discussed above, $5 \notin [3, 7)$. The reason for the exclusion is that without analyzing the **`if`**-guards we cannot be sure that 5 occurs in every trace between each occurrence of 3 and 7. Hence, we do not provide a theoretical guarantee of optimality, but we have not encountered such examples in practice.

IV. IMPLEMENTATION AND EVALUATION

We have implemented the technique from Sec. III in a prototype tool called AutoSync. The implementation comprises 650 lines of Python code, and uses GPUVerify (revision 1937) and Z3 (version 4.6.0).

A. Research Questions

Our empirical evaluation aims to answer the following research questions:

1 Is AutoSync effective at synthesizing correct barrier placements?
2 Are the placements synthesized by AutoSync optimal? Does our cost model faithfully estimate execution time?
3 Is AutoSync efficient?

B. Experiment Setup

Benchmark selection. We evaluated AutoSync on the kernels from NVIDIA GPU Computing SDK v5.0 which is used by GPUVerify. We have selected 18 benchmarks from this benchmark suite, which (1) contained a barrier in the original program (2) were verifiable by GPUVerify within the timeout of five minutes, and (3) did not contain procedures, which are currently not supported by AutoSync.

For each benchmark, we compare the synthesized solution with a *baseline version*, which is correctly annotated with barrier statements by the developer, and can be verified by GPUVerify. In addition to the benchmark suite, we designed eight micro-benchmarks that exercise various challenging scenarios for barrier synthesis.

Running AutoSync. For each benchmark, we first remove all barrier statements from the baseline version, pass the resulting program to AutoSync, and check whether the barrier synthesis succeeded. If so, we manually compare the generated output with the baseline in terms of the number of barriers and their cumulative cost according to our cost model.

We also developed a *naive version* of barrier synthesis, which uses brute-force enumerative search. The naive version first inserts a barrier after each statement in the input kernel, then removes all barriers that lead to divergence, and finally, iterates over the remaining barriers, removing each barrier unless that causes a data race. The naive method is guaranteed to correctly synchronize the kernel, but it requires many more calls to the oracle, and serves as a baseline in our evaluation of the AutoSync's synthesis times.

All experiments were conducted on a machine with Intel i7-4700MQ CPU @ 2.40GHz and 8 GB RAM. Each timing presented in the results is the median of three runs. The cost model is evaluated on a p2.xlarge instance of AWS which runs the GPU kernels on NVIDIA K80 GPU.

Benchmark	LoC	N-V	AS-V	N-B	AS-B	N-Time	AS-Time
1-1-If.cu	11	6	4	1	1	7.1	3.6
1-1-loop:inter.cu	10	10	2	1	1	11.5	1.1
1-1-loop:intra.cu	7	6	3	1	1	9.5	3.1
1-1-main.cu	5	6	2	1	1	6.6	1.1
2-1-main.cu	6	7	2	1	1	8.0	1.1
2-1-loop-d2-intra.cu	9	8	5	1	1	12.8	6.4
2-2-loop-d2-both.cu	18	15	5	2	2	25.2	6.6
2-2-loop:both.cu	7	7	4	2	2	8.9	4.0

Loc: Lines of Code, N-*: Naive Method, AS-*: AutoSync, -V: Number of calls to GPUVerify, -B: Number of Barriers Inserted, -Time: Synthesis Time (sec)

TABLE I: Evaluation on Micro Benchmarks

C. Results

Micro-benchmarks. Tab. I present the evaluation result on the micro-benchmarks written by us. Benchmark names follow the convention "n-m-description.cu", where n is the number of data races present and m is the minimum number of barriers required to correctly synchronize the kernel. We make the following observations about the results:

- All micro-benchmarks are correctly synchronized by both the naive method and AutoSync.
- The number of barriers inserted in the synthesized kernel is the same as the expected minimum number of barriers.
- AutoSync is significantly more efficient than the naive method, because it performs fewer expensive calls to GPUVerify.
- AutoSync's synthesis time increases linearly with the number of calls to GPUVerify, which in practice is proportional to the maximum nesting depth of loop races present in the kernel. On the other hand, the synthesis time of the naive method increases almost linearly with the size of the input program.

Original Benchmarks. The results of evaluating AutoSync on the 18 original benchmarks from NVIDIA SDK are presented in Tab. II.

- Most of the barriers synthesized by AutoSync were placed at the same or equal-cost position as compared to the baseline version of the kernel.
- For five benchmarks AutoSync was able to generate a *more optimal placement* (with fewer barriers) than the baseline version. After a closer inspection, the reason was that AutoSync treats barrier placement as a global optimization problem instead of handling each barrier independently (as the programmer likely would).
- In practice, AutoSync requires very few iterations of the refinement loop (2–4), since the race locations are not nested very deeply; consequently the synthesis time does not necessarily grow with the size of the program.

Cost Model. We performed an experiment to evaluate the adequacy of the cost model we proposed Sec. III-C. We wrote multiple programs which were a combination of loops and conditionals and added barriers at different locations. We then measured the time taken by the kernels containing the barrier at different cost positions and generated the run time vs cost

Benchmark	LoC	V	O-B	AS-B	Time
convolutionColumnsKernel.cu	55	2	1	1	13.1
convolutionRowsKernel.cu	57	2	1	1	8.4
d_transpose.cu	26	2	1	1	1.1
imageDenoising_nlm2_kernel.cu	88	2	1	1	1.9
matrixMul.cu	72	4	2	2	15.5
mergeHistogram256Kernel.cu	25	3	1	1	3.2
mergeHistogram64Kernel.cu	26	3	1	1	3.3
reduce0.cu	26	3	2	1	3.3
reduce1.cu	28	4	2	2	6.1
reduce2.cu	30	4	2	1	5.6
reduce3.cu	32	3	2	1	3.2
reduce5.cu	94	3	4	3	10.1
reduce6.cu	104	4	4	3	15.8
sobol.cu	93	3	1	1	19.9
sum0.cu	25	3	2	2	4.7
sum1.cu	22	3	2	2	4.6
uniformUpdate.cu	17	3	1	1	2.6
uniform_add.cu	17	3	1	1	2.4

Loc: Lines of Code, V: Number of calls to GPUVERIFY, O-B: Number of Barriers in the original benchmark, AS-B: Number of Barriers in the synthesized program, Time: Synthesis Time (sec)

TABLE II: Evaluation of original Benchmarks

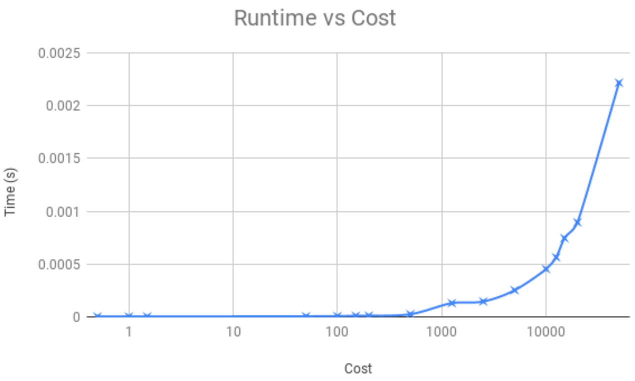


Fig. 10: The run-time overhead (sec) of placing barriers at different costs. The cost of barrier is computed as the cost model discussed above where LC=100 and IC=0.5 and every loop performs 100 iterations.

graph (Fig. 10). We can clearly see from the graph that the run time increases rapidly with the cost of the barrier placement. This graph suggests that the cost model described in Sec. III-C correlates well with the actual run-time overhead of barrier placement.

D. Threats to Validity

Out of the 18 benchmarks in our evaluation, 8 contained some user-provided invariants which were essential for GPUVERIFY to successfully verify the kernel. We believe it is fair to use these annotated programs because our tool is agnostic to the choice of oracle, and we hope that as invariant inference improves, our tool will become fully automatic. In addition, all our benchmarks are obtained by removing barriers from correctly synchronized kernels; hence a valid barrier placement always exists. In general, synchronization is not limited to placing barriers and might require more substantial changes to the code; such changes are out of the scope for our technique.

V. RELATED WORK

Synchronization synthesis for various concurrency models is a rich and active area of research. Prior work focused on traditional shared memory concurrency [9], [10], [11], [12], [13], [14] and network programs [15]. To our knowledge, AUTOSYNC is the first tool to perform synchronization synthesis for GPUs. GPUs are an interesting new domain for this line of work, because of the subtleties of the concurrency model, such as barrier divergence. Our technique shares similarities with [13], which also uses MaxSAT to find an optimal synchronization placement.

An important difference between AUTOSYNC and prior work in this area, is that we use an off-the-shelf verifier as a correctness oracle and define the minimal interface between the search engine and the oracle—data race locations—that still supports efficient synthesis. This design decision gives us soundness for free and allows AUTOSYNC to automatically leverage any future advances in GPU verification technology.

Code generation. A complementary approach to automatic synchronization is to compile a high-level language into GPU code [16], [17]. This approach works well when the high-level language matches the task at hand, but falls short if the programmer needs to hand-optimize the low-level GPU code.

Race detection for GPU kernels is also an extremely active research area [1], [6], [2], [3], [4], [5]. As mentioned in the introduction, these techniques can detect a missing barrier, but do not help the programmer find an optimal placement for the barrier. In this paper we show how to leverage these verification techniques as correctness oracles for synchronization synthesis. Even though our implementation uses GPUVERIFY [6], it can be adapted to work with any sound verification engine that uses predicated execution semantics and reports race locations and divergent barriers.

VI. CONCLUSIONS

We have presented a technique for automatically inserting barrier synchronization in GPU kernels. Our main contribution is two-fold. First, we show how to reuse an existing verifier as a correctness oracle and still achieve efficient synthesis by leveraging error information from failed verification attempts. Second, we show how to combine this error information with information about program structure to encode the search for an optimal barrier placement as a MaxSAT problem.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback, as well as Jeroen Ketema and Alastair Donaldson for their help with the GPUVERIFY benchmarks.

REFERENCES

[1] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," in *OOPSLA*, 2012.

[2] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *FSE*, 2010.

[3] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *PPoPP*, 2012.

[4] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *PLDI*, 2012.

[5] S. Blom, M. Huisman, and M. Mihelčić, "Specification and verification of gpgpu programs," *Sci. Comput. Program.*, vol. 95, no. P3, Dec. 2014.

[6] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson, "The design and implementation of a verification technique for gpu kernels," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 3, pp. 10:1–10:49, May 2015.

[7] N. Bjørner, A. Phan, and L. Fleckenstein, "νz - an optimizing SMT solver," in *TACAS*, 2015.

[8] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[9] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Program*, 1981.

[10] M. T. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL*, 2010.

[11] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations," in *CAV*, 2013.

[12] P. Cerny, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Regression-free synthesis for concurrency," in *CAV*, 2014.

[13] P. Cerný, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach, "From non-preemptive to preemptive scheduling using synchronization synthesis," in *CAV*, 2015.

[14] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct representation of concurrent trace sets," in *POPL*, 2015.

[15] J. McClurg, H. Hojjat, and P. Cerný, "Synchronization synthesis for network programs," in *CAV*, 2017.

[16] J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the gpu programming barrier with the auto-parallelising sac compiler," in *DAMP*, 2011.

[17] "TensorFlow documentation," https://www.tensorflow.org/programmers_guide/using_gpu, 2018.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Rely-Guarantee Reasoning for Automated Bound Analysis of Lock-Free Algorithms

Thomas Pani*†, Georg Weissenbacher*, Florian Zuleger*
*Formal Methods in Systems Engineering Group, TU Wien, Vienna, Austria
† Wolfgang Pauli Institute, Vienna, Austria
Email: {pani,weissenb,zuleger}@forsyte.at

***Abstract*—We present a thread-modular proof method for *complexity and resource bound analysis* of concurrent, shared-memory programs, lifting Jones' rely-guarantee reasoning to assumptions and commitments capable of expressing bounds. We automate reasoning in this logic by reducing bound analysis of concurrent programs to the sequential case. Our work is motivated by its application to *lock-free data structures*, fine-grained concurrent algorithms whose time complexity has to our knowledge not been inferred automatically before.**

I. INTRODUCTION

A. Program Complexity and Resource Bound Analysis

Program complexity and resource bounds analysis (bound analysis) aims to statically determine upper bounds on the resource usage of a program as expressions over its inputs. Despite the recent discovery of powerful bound analysis methods for *sequential* imperative programs (e.g., [1], [2], [3], [4], [5], [6], [7]), little work exists on bound analysis for *concurrent, shared-memory* imperative programs (cf. Section VII).

From a practical point of view, bound analysis is an important step towards proving functional correctness criteria of programs in resource-constrained environments: For example, in *real-time systems* intermediary results must be available within certain time bounds, or in *embedded systems* applications must not exceed hard constraints on CPU time, memory consumption, or network bandwidth.

B. Non-blocking Data Structures

We illustrate the necessity of extending bound analysis to concurrent, shared-memory programs on the example of *non-blocking data structures*: Devised to circumvent shortcomings of lock-based concurrency (like deadlocks or priority inversion), they have been adopted widely in engineering practice [8]. For example, the Michael-Scott non-blocking queue [9] is implemented in the Java standard library's `ConcurrentLinkedQueue` class.

Automated techniques have been introduced for proving both *correctness* (e.g., [10], [11], [12], [13]) and *progress* (e.g., [14], [15]) properties of non-blocking data structures. In this work, we focus on the progress property of *lock-freedom*, a liveness property that ensures absence of livelocks: Despite interleaved execution of multiple threads altering the data structure, some thread is guaranteed to complete its operation *eventually*.

From a practical, engineering point of view it is not enough to prove that a data structure operation completes *eventually*. Rather, it needs to make progress using a *bounded, measurable amount of resources*: Petrank et al. [16] formalize and study bounded lock-free progress as *bounded lock-freedom*, and discuss its relevance for practical applications. They describe its verification for a fixed number of threads and a given bound using model checking, but leave finding the bound to the user. Existing approaches for automatically proving progress properties like the ones presented in [14], [15] are limited to eventual progress. To our knowledge, bounded progress guarantees have not been inferred automatically before.

C. Overview

Reasoning about the resource consumption of non-blocking algorithms is an intricate and manually tedious problem. To illustrate this point, consider the following common design pattern for lock-free data structures: A thread aiming to manipulate the data structure starts by taking as many steps as possible without synchronization, preparing its intended update. Then, it attempts to alter the globally visible state by synchronizing on a single word in memory at a time. Interference from other threads may cause this synchronization to fail, and the thread to retry from the beginning. From the viewpoint of a single thread that accesses the data structure:

1) The *amount of interference* by other threads *directly affects its resource consumption*. In general, this means reasoning about an unbounded number of concurrent threads, even to infer resource bounds on a single thread.
2) The *point of interference* may occur at any point in the execution, due to the fine granularity of concurrency.

In this paper, we present an automated bound analysis for concurrent, shared-memory programs to remedy this situation: In particular, our method analyzes the *parameterized* system of N concurrent lock-free data structure client threads. To reason about this infinite family of systems and its interactions, we leverage and extend *rely-guarantee (RG) reasoning* [17]: RG reasoning considers each thread separately, modeling interleaved steps of other threads in an *environment assumption*. However, we will see that classic RG reasoning is too weak to obtain suitable bounds. Therefore, we extend RG reasoning to bound analysis. In the following we outline the major contributions of this paper.

ISBN: 978-0-9835678-8-2.

D. Contributions

1) We present the first extension of rely-guarantee specifications to bound analysis (Section III).
2) We formulate inference rules to reason about these extended specifications and instantiate them to derive our method for bound analysis of concurrent programs (Section IV).

   Apart from their specific use case in this work, we believe the inference rules are interesting in their own right, for example in comparison to the reasoning rules for liveness presented in [14] (cf. the discussion in Section VII).
3) We reduce bound analysis of concurrent programs to bound analysis of sequential programs, and obtain an algorithm for rely-guarantee bound analysis (Section V).
4) We implement our algorithm in the tool COACHMAN and apply it to lock-free data structures from the literature. To our knowledge, we are the first to automatically infer runtime complexity for widely studied lock-free data structures such as Treiber's stack [18] or the Michael-Scott queue [9] (Section VI).

II. MOTIVATING EXAMPLE

We start by giving an informal explanation of our method and of the paper's main contributions on a running example.

A. Running Example: Treiber's Stack

Fig. 1 shows the implementation of a lock-free concurrent stack known as *Treiber's stack* [18]. Our input programs are represented as control-flow graphs with edges labeled by guarded commands of the form $g \triangleright c$. We omit g if $g = \mathtt{true}$. We assume edges are executed atomically, and that programs execute in presence of a garbage collector; the latter prevents the so-called *ABA problem* and is a common assumption in the design of lock-free algorithms [8].

Values stored on the stack do not influence the number of times its operations are executed, thus we abstract them away for readability. The stack is represented by a null-terminated singly-linked list, with the shared variable `T` pointing to the top element. The `push` and `pop` methods may be called concurrently, with synchronization occurring at the guarded commands originating in ℓ_3 for `push` and ℓ_{13} for `pop`. These low-level atomic synchronization commands are usually implemented in hardware, through instructions like *compare-and-swap* (CAS) [8].

The stack operations are implemented as follows: Initially, `T` points to `NULL`. The `push` operation (Fig. 1a)

1) allocates a new list node `n` ($\ell_0 \to \ell_1$)
2) reads the global stack pointer `T` ($\ell_1 \to \ell_2$)
3) updates the newly allocated node's `next` field to the read value of `T` ($\ell_2 \to \ell_3$)
4) atomically: compares the value read in (2) to the actual value of `T`; if equal, `T` is updated to point to `n`, otherwise the operation restarts ($\ell_3 \to \ell_4$ and $\ell_3 \to \ell_1$ respectively).

The `pop` operation (Fig. 1b) proceeds similarly.

B. Problem Statement

Consider a general data structure client $P = \mathtt{op1}() \, [] \cdots [] \, \mathtt{opM}()$, where $\mathtt{op1}, \ldots, \mathtt{opM}$ are the data structure's operations, and $[]$ denotes non-deterministic choice. We compose N concurrent client threads P_1 to P_N accessing the data structure:

$$\|_N P \overset{\text{def}}{=} \underbrace{P}_{P_1} \| \cdots \| \underbrace{P}_{P_N}$$

Our goal is to design an automated procedure that automatically infers upper-bounds for all system sizes N on

1) the *thread-specific* resource usage caused by a control-flow edge of a single thread P_1 when executed concurrently with $P_2 \| \cdots \| P_N$, or
2) the *total* resource usage caused by a control-flow edge in total over all threads P_1 to P_N.

Remark (Cost model)**.** To measure the amount of resource usage, bound analyses are usually parameterized by a *cost model* that assigns each operation or instruction a *cost* amounting to the resources consumed. In this paper, we adopt a *uniform cost model* that assigns a constant cost to each control-flow edge. When we speak of the *complexity* of a program, we adopt a specific uniform cost model that assigns cost 1 to each control-flow back edge and cost 0 to all other edges; this reflects the asymptotic time complexity of the program.

Running example. Consider N concurrent copies $P_1 \| \cdots \| P_N$ of the Treiber stack's client program $\mathtt{push}() \, [] \, \mathtt{pop}()$, and the `push` operation's control-flow edge $\ell_1 \to \ell_2$. A manual analysis yields a *thread-specific* bound for P_1 telling us that this edge is executed at most N times by P_1: Each time that another thread successfully modifies stack pointer `T`, P_1's copy in `t` may become outdated, causing the test at ℓ_3 to fail ($\mathtt{t} \neq \mathtt{T}$), and P_1 to restart. After at most $N-1$ iterations, all other threads have finished their operations and returned, and P_1 executes $\ell_1 \to \ell_2 \to \ell_3 \to \ell_4$ without interference.

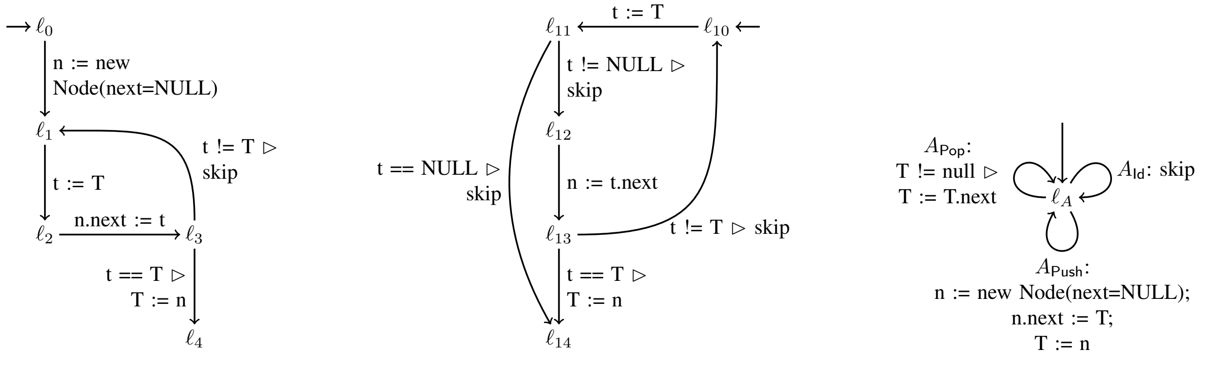
Similarly, a *total* bound for $P_1 \| \cdots \| P_N$ tells us that edge $\ell_1 \to \ell_2$ is executed at most $N(N+1)/2$ times by all threads P_1 to P_N in total: The first thread to successfully synchronize at ℓ_3 sees no interference and executes $\ell_1 \to \ell_2$ once. The second thread may need to restart once due to the first thread modifying `T`, and executes $\ell_1 \to \ell_2$ at most twice, etc. The last thread to synchronize has the worst-case bound we established as thread-specific bound for P_1: it executes $\ell_1 \to \ell_2$ N times. We obtain $N(N+1)/2$ as closed form for the total bound. In the following, we illustrate how to formalize and automate this reasoning.

C. Environment Abstraction

Client program $\|_N P$ from above is *parameterized* in the number of concurrent threads N. To reason about this infinite family of parallel client programs, we base our analysis on Jones' *rely-guarantee reasoning* [17]. For each thread, RG reasoning over-approximates the following as sets of binary relations over program states (*actions*):

- the thread's effect on the global state (its *guarantee*)

ISBN: 978-0-9835678-8-2.



(a) `push()` (b) `pop()` (c) Stateless program $P(\mathcal{A})$ for actions $\mathcal{A} = \{A_{\mathsf{Push}}, A_{\mathsf{Pop}}, A_{\mathsf{Id}}\}$ [19].

Fig. 1: Treiber's lock-free stack [18]. Stack pointer `T` is the sole global variable.

- the effect of all other threads (its *rely*) as the union of those threads' guarantees.

The effect of all other threads (the thread's *environment*) is thus effectively abstracted into a single relation. Crucially, this also abstracts away *how often* each environment action is performed, rendering Jones' RG reasoning unsuitable for concurrent bound analysis.

Running example. The program in Fig. 1c with actions $\mathcal{A} = \{A_{\mathsf{Push}}, A_{\mathsf{Pop}}, A_{\mathsf{Id}}\}$ summarizes the globally visible effect of P_1's environment $P_2 \parallel \cdots \parallel P_N$ for all $N > 0$. In particular, A_{Push} summarizes the effect of an environment thread executing edge $\ell_3 \to \ell_4$ from the point of view[1] of thread P_1, A_{Pop} that of $\ell_{13} \to \ell_{14}$, and A_{Id} that of all other edges. We discuss how to obtain $\mathcal{A}$ in Section V-A.

As is, the actions in $\mathcal{A}$ may be executed infinitely often. Our informal derivation of the bound in Section II-B however, had to determine *how often* other threads could interfere with the reference thread P_1 (altering pointer `T`) to bound its number of loop iterations.

Hence, we lift Jones' RG reasoning to concurrent bound analysis by enriching RG relations with bounds. We emphasize our focus on progress properties in this work: although our framework extends Jones' RG reasoning and can express safety properties, we only use it to reason about bounds; tighter integration is left for future work.

D. Rely-Guarantee Reasoning for Bound Analysis

In particular, relies and guarantees in our setting are maps $\{A_1 \mapsto b_1, \dots\}$ from actions A_i (which are binary relations over program states) to bound expressions b_i. Each relation describes an environment action, and the bound expression describes how often that action may occur on a run of the program.

We present a program logic for thread-modular [20] reasoning about bounds: A judgement in our logic takes the form

$$\mathcal{R}, \mathcal{G} \vdash \{S\}\ P\ \{S'\}$$

where $\{S\}\ P\ \{S'\}$ is a Hoare triple, and $\mathcal{R}, \mathcal{G}$ are a rely and guarantee. Its informal meaning is: For any execution of program P starting in a state from $\{S\}$, and environment interference described by the relations in $\mathcal{R}$ and occurring at most the number of times given by the respective bounds in $\mathcal{R}$, P changes the shared state according to the relations in $\mathcal{G}$ and at most the number of times described by the respective bounds in $\mathcal{G}$. In addition, the execution is safe (does not reach an error state) and if P terminates, its final state is in $\{S'\}$.

Running example. For readability, we focus on the analysis of Treiber's `push` method. The steps for `pop` are similar. To obtain one bound per edge, we split action A_{Id}: `skip` from Fig. 1c into several actions $A_{\mathsf{Id}}^{i,j}$: `skip`, one for each edge $\ell_i \to \ell_j$. For a rely or guarantee $\{A_{\mathsf{Id}}^{0,1} \mapsto b_1, A_{\mathsf{Id}}^{1,2} \mapsto b_2, A_{\mathsf{Id}}^{2,3} \mapsto b_3, A_{\mathsf{Id}}^{3,1} \mapsto b_4, A_{\mathsf{Push}} \mapsto b_5\}$, we fix the order of actions and write $(b_1, b_2, b_3, b_4, b_5)$ for short.

First, our method states the following RG quintuple:

$$\mathcal{R}, \mathcal{G} \vdash \{\mathit{Inv}\}\ P_1\ \{\mathsf{true}\}$$

where Inv is a data structure invariant stated over shared variables in a suitable assertion language (e.g., separation logic), $\mathcal{R} = (\infty, \infty, \infty, \infty, \infty)$, and $\mathcal{G} = (1, \infty, \infty, \infty, 1)$. Despite the unbounded environment $\mathcal{R}$ (which corresponds to Fig. 1c), we can already bound two edges, $\ell_0 \to \ell_1$ and $\ell_3 \to \ell_4$ of P_1, and thus the corresponding actions in $\mathcal{G}$: These edges are not part of a loop and – despite any interference from the environment – can be executed at most once.

We show how to automatically discharge (or rather, discover) such RG quintuples in Section V. Next, we use the bound information obtained in $\mathcal{G}$ to refine the environment $\mathcal{R}$ until a fixed point of the rely is reached.

Running example (continued)**.** We established that thread P_1 can perform actions $A_{\mathsf{Id}}^{0,1}$ and A_{Push} at most once. In our example, all threads are symmetric, thus each of the $N-1$ other threads can execute $A_{\mathsf{Id}}^{0,1}$ and A_{Push} at most once as well. The abstract environment representing these $N-1$ threads can

---

[1] Note that changes to local variables of $P_2, \dots, P_N$ are not visible to P_1.

thus execute each action $A^{0,1}_{\mathsf{Id}}$ and A_{Push} at most $N-1$ times. We obtain the *refined rely* $\mathcal{R}' = (N-1, \infty, \infty, \infty, N-1)$.

As we have reasoned in Section II-B, once the number of A_{Push} environment actions is bounded, P_1 loops only that number of times. We obtain the *refined guarantee*

$$\mathcal{G}' = (1, N, N, N-1, 1).$$

By the same reasoning as above, we multiply $\mathcal{G}'$ with $(N-1)$ (componentwise) and obtain the refined rely

$$\mathcal{R}'' = (N-1, N(N-1), N(N-1), (N-1)^2, N-1).$$

From $\mathcal{R}''$, we cannot obtain any tighter bounds, i.e., $\mathcal{G}'' = \mathcal{G}'$ is a fixed point, and we report $\mathcal{G}''$ and $\mathcal{G}'' + \mathcal{R}''$ as the thread-specific and total bounds of P_1 and $P_1 \parallel \cdots \parallel P_N$:

edge	thread-specific bound	total bound
$\ell_0 \to \ell_1$	1	N
$\ell_1 \to \ell_2$	N	N^2
$\ell_2 \to \ell_3$	N	N^2
$\ell_3 \to \ell_1$	$N-1$	$N(N-1)$
$\ell_3 \to \ell_4$	1	N

We demonstrate in Section VI that for more complex examples, more than two iterations of the rely-refinement are necessary to bound all edges. We formalize our reasoning in Sections III and IV, explain its automation in Section V, and describe further case studies in Section VI.

III. RG SPECIFICATIONS FOR BOUND ANALYSIS

In this section, we formalize the technique illustrated informally above. We start by stating our program model and formally define the kind of bounds we consider:

A. Program Model

Definition 1 (Program)**.** Let $LVar$ and $SVar$ be finite disjoint sets of typed *local* and *shared program variables*, and let $Var = LVar \cup SVar$. Let Val be a set of *values*. *Program states* $\Sigma\colon Var \to Val$ over Var map variables to values. We write $\sigma\downharpoonright_{Var'}$ where $Var' \subseteq Var$ for the projection of a state $\sigma \in \Sigma$ onto the variables in Var'. Let $GC = Guards \times Commands$ denote the set of *guarded commands* over Var and their effect be defined by $[\![\cdot]\!]\colon GC \to \Sigma \to 2^{\Sigma} \cup \{\bot\}$ where $\bot$ is a special error state. A *program* P over Var is a directed labeled graph $P = (L, T, \ell_0)$, where L is a finite set of *locations*, $\ell_0 \in L$ is the *initial location*, and $T \subseteq L \times GC \times L$ is a finite set of *transitions*. Let S be a predicate over Var that is evaluated over program states. We overload $[\![\cdot]\!]$ and write $[\![S]\!] \subseteq \Sigma$ for the set of states satisfying S. We represent executions of P as sequences of *steps* $r \in \Sigma \times T \times \Sigma$ and write $\sigma \xrightarrow{t} \sigma'$ for a step (σ, t, σ'). A *run* of P from S is a sequence of steps $\rho = \sigma_0 \xrightarrow{\ell_0, gc_0, \ell_1} \sigma_1 \xrightarrow{\ell_1, gc_1, \ell_2} \ldots$ such that $\sigma_0 \in [\![S]\!]$ and for all $i \geq 0$ we have $\sigma_{i+1} \in [\![gc_i]\!](\sigma_i)$.

Definition 2 (Interleaving of Programs)**.** Let $P_i = (L_i, T_i, \ell_{0,i})$ for $i \in \{1, 2\}$ be two programs over $Var_i = LVar_i \cup SVar_i$ such that $LVar_1 \cap LVar_2 = \emptyset$. Their *interleaving* $P_1 \parallel P_2$ over $Var_1 \cup Var_2$ is defined as the program

$$P_1 \parallel P_2 = (L_1 \times L_2, T, (\ell_{0,1}, \ell_{0,2}))$$

where T is given by $((\ell_1, \ell_2), gc, (\ell_1', \ell_2')) \in T$ iff $(\ell_1, gc, \ell_1') \in T_1$ and $\ell_2 = \ell_2'$ or $(\ell_2, gc, \ell_2') \in T_2$ and $\ell_1 = \ell_1'$.

Given a program P over local and shared variables $Var = LVar \cup SVar$, we write $\|_N P = P_1 \parallel \cdots \parallel P_N$ where $N \geq 1$ for the N-times interleaving of program P with itself, where P_i over Var_i is obtained from P by suitably renaming local variables such that $LVar_1 \cap \cdots \cap LVar_N = \emptyset$. Given a predicate S over Var, we write $\bigwedge_N S$ for the conjunction $S_1 \wedge \cdots \wedge S_N$ where S_i over Var_i is obtained by the same renaming.

Definition 3 (Expression)**.** Let Var be a set of integer program variables. We denote by $\mathrm{Expr}(Var)$ the set of arithmetic *expressions* over $Var \cup \mathbb{Z} \cup \{\infty\}$. The semantics function $[\![\cdot]\!]\colon \mathrm{Expr}(Var) \to \Sigma \to (\mathbb{Z} \cup \{\infty\})$ evaluates an expression in a given program state. We assume the usual expression semantics; in particular, $a \circ \infty = \infty$ and $a \leq \infty$ for all $a \in \mathbb{Z} \cup \{\infty\}$ and $\circ \in \{+, \times\}$.

Definition 4 (Bound)**.** Let $P = (L, T, \ell_0)$ be a program over variables Var, and let S over Var be a predicate describing P's initial states. Let $t \in T$ be a transition of P, and $\rho = \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots$ be a run of P from S. We use $\#(t, \rho) \in \mathbb{N}_0 \cup \{\infty\}$ to denote the number of times transition t appears on run ρ. An expression $b \in \mathrm{Expr}(Var_{\mathbb{Z}})$ over integer program variables $Var_{\mathbb{Z}} \subseteq Var$ is a *bound* for t on ρ iff $\#(t, \rho) \leq [\![b]\!](\sigma_0)$, i.e., if t appears at most b times on ρ.

Given a program $P = (L, T, \ell_0)$ and predicate S over local and shared variables $Var = LVar \cup SVar$, our goal is to compute a function $\mathrm{Bound}\colon T \to \mathrm{Expr}(SVar_{\mathbb{Z}} \cup \{N\})$, such that for all transitions $t \in T$ and all system sizes $N \geq 1$, $\mathrm{Bound}(t)$ is a bound for t of P_1 on all runs of $\|_N P = P_1 \parallel \cdots \parallel P_N$ from $\bigwedge_N S = S_1 \wedge \cdots \wedge S_N$. That is, Bound gives us the thread-specific bounds for transitions of P_1. In Section IV, we explain how to obtain total bounds on $\|_N P$ from that.

B. Extending Rely-Guarantee Reasoning for Bound Analysis

To analyze the infinite family of programs $\|_N P = P_1 \parallel \cdots \parallel P_N$, we abstract P_1's environment $P_2 \parallel \cdots \parallel P_N$: We define *actions*, which provide an abstract view of transitions by abstracting away local variables and program locations.

Definition 5 (Action, Environment Assertion)**.** Let Σ_S be a set of program states over shared variables $SVar$. An *action* $A \subseteq \Sigma_S \times \Sigma_S$ over $SVar$ is a binary relation over program states. Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be a finite set of actions. An *environment assertion* $\mathcal{E}_{\mathcal{A}}\colon \mathcal{A} \to \mathrm{Expr}(SVar)$ over $\mathcal{A}$ is a function that maps actions to bound expressions over $SVar$. We omit $\mathcal{A}$ from $\mathcal{E}_{\mathcal{A}}$ wherever it is clear from the context.

We use sequences a of actions to describe interference: Intuitively, the bound $\mathcal{E}_{\mathcal{A}}(A)$ describes how often action $A \in \mathcal{A}$ is permissible in such a sequence. This is captured by the $\models$ relation defined below. We also define operations and relations on environment assertions to compose and compare them.

Definition 6 (Operations and Relations on Environment Assertions)**.** Let $\mathcal{A}$ be a finite set of actions over shared variables $SVar$, let $A \in \mathcal{A}$ be an action, and let a be a finite or infinite word over actions $\mathcal{A}$. Let $\mathcal{E}_{\mathcal{A}}$ and $\mathcal{E}'_{\mathcal{A}}$ be environment assertions over $\mathcal{A}$. Let $\sigma \subseteq \Sigma_S$ be a program state over $SVar$. We overload $\#(A, a) \in \mathbb{N}_0 \cup \{\infty\}$ to denote the number of times A appears on a and define

$$a \models_\sigma \mathcal{E}_{\mathcal{A}} \text{ iff } \#(A, a) \leq [\![\mathcal{E}_{\mathcal{A}}(A)]\!](\sigma) \text{ for all } A \in \mathcal{A}.$$

Let $e \in \mathrm{Expr}(SVar)$ be an expression over $SVar$. For all actions $A \in \mathcal{A}$ we define

$$(e \times \mathcal{E}_{\mathcal{A}})(A) = e \times \mathcal{E}_{\mathcal{A}}(A), \text{and}$$
$$(\mathcal{E}_{\mathcal{A}} + \mathcal{E}'_{\mathcal{A}})(A) = \mathcal{E}_{\mathcal{A}}(A) + \mathcal{E}'_{\mathcal{A}}(A).$$

Further, let S be a predicate over $SVar$. We define

$$\mathcal{E}_{\mathcal{A}} \subseteq_S \mathcal{E}'_{\mathcal{A}} \text{ iff } [\![\mathcal{E}_{\mathcal{A}}(A)]\!](\sigma) \leq [\![\mathcal{E}'_{\mathcal{A}}(A)]\!](\sigma)$$

for all $A \in \mathcal{A}$ and all $\sigma \in [\![S]\!]$.

C. Trace Semantics of RG Quintuples

We abstract environment threads of interleaved programs with *RG quintuples* of either form

$$\mathcal{R}, \mathcal{G} \vdash \{S\}\, P\, \{S'\} \quad \text{or} \quad \mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$$

where P and $P_1 \parallel P_2$ are programs, S and S' are predicates such that $[\![S]\!] \subseteq \Sigma$ are *inital program states*, and $[\![S']\!] \subseteq \Sigma$ are *final program states*, and *rely* $\mathcal{R}$ and *guarantees* $\mathcal{G}$ and $\mathcal{G}_1, \mathcal{G}_2$ are environment assertions over a finite set of actions $\mathcal{A}$.

Remark (Notation of environment assertions)**.** Note that the relies and guarantees of a single RG quintuple are defined over the *same* set of actions $\mathcal{A}$; in Section V-A we show how to compute a set $\mathcal{A}$ that over-approximates P (or $P_1 \parallel P_2$) in a prelimiary analysis step. We choose to write relies and guarantees as functions over $\mathcal{A}$ as it simplifies notation throughout the paper. The reader may prefer to think of environment assertions $\{A_1 \mapsto b_1, \dots\}$ as sets of pairs of an action and a bound $\{(A_1, b_1), \dots\}$, in contrast to just sets of actions $\{A_1, \dots\}$ in Jones' RG reasoning.

In particular, $\mathcal{R}$ abstracts P's or $P_1 \parallel P_2$'s environment. The guarantees $\mathcal{G}$ and $(\mathcal{G}_1, \mathcal{G}_2)$ allow us to express both thread-specific and total bounds on interleaved programs: The guarantee $\mathcal{G}$ of quintuple $\mathcal{R}, \mathcal{G} \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$ contains total bounds for $P_1 \parallel P_2$, while the guarantees $\mathcal{G}_1, \mathcal{G}_2$ of $\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$ contain the respective thread-specific bounds of threads P_1 and P_2.

We model executions of RG quintuples as *traces*, which abstract runs of the concrete system. In particular, for each run of the concrete system, there exists a corresponding trace of the abstract system. This allows us to over-approximate bounds by considering the traces induced by RG quintuples.

$$\frac{\begin{array}{c}\mathcal{R} + \mathcal{G}_2, \mathcal{G}_1 \vdash \{S_1\}\, P_1\, \{S_1'\} \\ \mathcal{R} + \mathcal{G}_1, \mathcal{G}_2 \vdash \{S_2\}\, P_2\, \{S_2'\}\end{array}}{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S_1 \wedge S_2\}\, P_1 \parallel P_2\, \{S_1' \wedge S_2'\}}\ \textsc{Par}$$

$$\frac{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}}{\mathcal{R}, \mathcal{G}_1 + \mathcal{G}_2 \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}}\ \textsc{Par-Merge}$$

$$\frac{\begin{array}{c}\mathcal{R}_1, \vec{\mathcal{G}}_1 \vdash \{S_1\}\, P\, \{S_1'\} \\ S_2 \Rightarrow S_1 \quad \mathcal{R}_2 \subseteq_{S_2} \mathcal{R}_1 \quad \vec{\mathcal{G}}_1 \subseteq_{S_2} \vec{\mathcal{G}}_2 \quad S_1' \Rightarrow S_2'\end{array}}{\mathcal{R}_2, \vec{\mathcal{G}}_2 \vdash \{S_2\}\, P\, \{S_2'\}}\ \textsc{Conseq}$$

Fig. 2: Rely/guarantee proof rules for bound analysis. We write $\vec{\mathcal{G}}$ for either $\mathcal{G}$ or $(\mathcal{G}_1, \mathcal{G}_2)$. In the latter case, $\subseteq$ is applied componentwise.

Definition 7 (Trace)**.** Let $P = (L, T, \ell_0)$ be a program of form P_1 or $P_1 \parallel P_2$ and S be a predicate over local and shared variables $Var = LVar \cup SVar$. Let $\mathcal{A}$ be a finite set of actions over $SVar$. We represent executions of P interleaved with environment actions in $\mathcal{A}$ as sequences of *trace transitions* $\delta \in (L \times \Sigma) \times (L \times \Sigma \cup \{\bot\}) \times \{1, 2, \mathsf{e}\} \times \mathcal{A}$, where the first two components define the change in program location and state, the third component defines whether the transition was taken by program P_1 (1), P_2 (2), or the environment (e), and the last component defines which action summarizes the state change. For a trace transition $\delta = ((\ell, \sigma), (\ell', \sigma'), \alpha, A)$, we write $(\ell, \sigma) \xrightarrow{\alpha:A} (\ell', \sigma')$. A *trace* $\tau = (\ell_0, \sigma_0) \xrightarrow{\alpha_1:A_1} (\ell_1, \sigma_1) \xrightarrow{\alpha_2:A_2} \dots$ is a sequence of trace transitions. Let $|\tau| \in \mathbb{N}_0 \cup \{\infty\}$ denote the number of transitions of τ. We define the set $\mathrm{traces}(S, P)$ as the set of traces such that $\sigma_0 \in [\![S]\!]$ and for $0 < i \leq |\tau|$ we have either

- $\alpha_i = 1$, $(\ell_{i-1}, gc, \ell_i) \in T_1$ for some gc, $\sigma_i \in [\![gc]\!](\sigma_{i-1})$, and $(\sigma_{i-1}\downharpoonright_{SVar}, \sigma_i\downharpoonright_{SVar}) \in A_i$, or
- $\alpha_i = 2$, $(\ell_{i-1}, gc, \ell_i) \in T_2$ for some gc, $\sigma_i \in [\![gc]\!](\sigma_{i-1})$, and $(\sigma_{i-1}\downharpoonright_{SVar}, \sigma_i\downharpoonright_{SVar}) \in A_i$, or
- $\alpha_i = \mathsf{e}$, $\ell_{i-1} = \ell_i$, $(\sigma_{i-1}\downharpoonright_{SVar}, \sigma_i\downharpoonright_{SVar}) \in A_i$, and $\sigma_{i-1}\downharpoonright_{LVar} = \sigma_i\downharpoonright_{LVar}$.

The *projection* $\tau\downharpoonright_C$ of a trace $\tau \in \mathrm{traces}(S, P)$ to components $C \subseteq \{1, 2, \mathsf{e}\}$ is the sequence of actions defined as image of τ under the homomorphism that maps $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ to A if $\alpha \in C$, and otherwise to the empty word.

We now define the meaning of RG quintuples over traces:

Definition 8 (Validity)**.** We define $\mathcal{R}, \mathcal{G} \models \{S\}\, P\, \{S'\}$ iff for all traces $\tau \in \mathrm{traces}(S, P)$ such that τ starts in state $\sigma_0 \in [\![S]\!]$ and $\tau\downharpoonright_{\{\mathsf{e}\}} \models_{\sigma_0} \mathcal{R}$ (τ's environment transitions satisfy the rely):

- if τ is finite and ends in $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ for some $\ell, \ell', \sigma, \alpha, A$ then $\sigma' \neq \bot$ (the program is safe) and $\sigma' \in [\![S']\!]$ (the program is correct), and
- $\tau\downharpoonright_{\{1\}} \models_{\sigma_0} \mathcal{G}$ (τ's P-transitions satisfy the guarantee $\mathcal{G}$).

Similarly, $\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{S\}\, P_1 \parallel P_2\, \{S'\}$ iff for all $\tau \in \mathrm{traces}(S, P_1 \parallel P_2)$ s.t. τ starts in $\sigma_0 \in [\![S]\!]$ and $\tau\downharpoonright_{\{\mathsf{e}\}} \models_{\sigma_0} \mathcal{R}$:

- if τ is finite and ends in $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ for some $\ell, \ell', \sigma, \alpha, A$ then $\sigma' \neq \bot$ and $\sigma' \in [\![S']\!]$, and
- $\tau\downharpoonright_{\{1\}} \models_{\sigma_0} \mathcal{G}_1$ and $\tau\downharpoonright_{\{2\}} \models_{\sigma_0} \mathcal{G}_2$.

IV. RG REASONING FOR BOUND ANALYSIS

Similar to classic RG reasoning [17], [21], we propose inference rules to facilitate reasoning about our extended RG quintuples. Our inference rules are shown in Fig. 2:

- Par interleaves two threads P_1 and P_2 and expresses their thread-specific guarantees in $(\mathcal{G}_1, \mathcal{G}_2)$.
- Par-Merge combines thread-specific guarantees $(\mathcal{G}_1, \mathcal{G}_2)$ into a total guarantee $\mathcal{G}_1 + \mathcal{G}_2$.
- Conseq is similar to the consequence rule of Hoare logic or RG reasoning: it allows to strengthen precondition and rely, and to weaken postcondition and guarantee(s).

We instantiate these rules to derive the main underlying principle of our bound analysis in the proof of Theorem 2.

Theorem 1 (Soundness)**.** *The rules in Fig. 2 are sound.*

Proof sketch: By Definition 7 (trace semantics of RG quintuples) and induction on the trace length. ■

In the following, we assume existence of a procedure $\textsc{SynthG}(S, P, \mathcal{R})$ that takes a predicate S, a non-interleaved program P, and a rely $\mathcal{R}$ and computes a guarantee $\mathcal{G}$, such that $\mathcal{R}, \mathcal{G} \models \{S\}\, P\, \{\mathsf{true}\}$ holds. We present such a procedure in Section V.

Our main idea is to use SynthG to compute correct-by-construction guarantees for RG quintuple *fragments* of form $\mathcal{R}, ? \vdash \{\mathit{Inv}\}\ P_1\ \{\mathsf{true}\}$. From this, Theorem 2 stated below allows us to infer guarantees for P_1's environment $P_2 \parallel \cdots \parallel P_N$ and thus for $\|_N P = P_1 \parallel \cdots \parallel P_N$.

Theorem 2 (Generalization of Single-Thread Guarantees)**.** *Let P be a program over local and shared variables $\mathit{Var} = \mathit{LVar} \cup \mathit{SVar}$ and let $\|_N P = P_1 \parallel \cdots \parallel P_N$ be its N-times interleaving. Let S be a predicate over SVar. Let $\mathcal{A}$ over SVar be the set of actions summarizing the globally visible effect of $\|_N P$ started from S, and let $\mathcal{R}$ and $\mathcal{G}$ be environment assertions over $\mathcal{A}$. Let $\mathbf{0} = (0, \ldots, 0)$ denote the empty environment.*

If

$$(N-1) \times \mathcal{G} \subseteq_S \mathcal{R} \quad \textit{and} \quad \mathcal{R}, \mathcal{G} \models \{S\}\ P_1\ \{\mathsf{true}\}$$

then

$$\mathbf{0}, (\mathcal{G}, (N-1) \times \mathcal{G}) \models \{S\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\}.$$

I.e., if $(N-1) \times \mathcal{G}$ is smaller than $\mathcal{R}$, and if $\mathcal{R}, \mathcal{G} \models \{S\}\ P_1\ \{\mathsf{true}\}$ holds, then in an empty environment, P_1's environment $P_2 \parallel \cdots \parallel P_N$ executes actions $\mathcal{A}$ no more than $(N-1) \times \mathcal{G}$ times.

Proof sketch: By induction on the number of threads and repeated application of rules Conseq, Par-Merge, *and* Par. ■

Running example. Let us return to the task of computing bounds for N threads $\|_N P = P_1 \parallel \cdots \parallel P_N$ concurrently executing Treiber's `push` method. Our method starts from the RG quintuple fragment

$$\mathcal{R}, ? \vdash \{\mathit{Inv}\}\ P_1\ \{\mathsf{true}\} \tag{1}$$

for which it computes a correct-by-construction guarantee: It summarizes P_1's environment $P_2 \parallel \cdots \parallel P_N$ in the rely $\mathcal{R}$. At this point, it cannot safely assume any bounds on $P_2 \parallel \cdots \parallel P_N$, and thus on $\mathcal{R}$. Therefore, it lets $\mathcal{R} = (\infty, \infty, \infty, \infty, \infty)$. Next, our method runs RG bound analysis. As we have argued in Section II-D, this yields $\textsc{SynthG}(\mathit{Inv}, P_1, \mathcal{R}) = (1, \infty, \infty, \infty, 1)$, i.e., we have

$$(\infty, \infty, \infty, \infty, \infty), (1, \infty, \infty, \infty, 1) \models \{\mathit{Inv}\}\ P_1\ \{\mathsf{true}\}. \tag{2}$$

Remark (Role of Theorem 2)**.** At this point, our method cannot establish tighter bounds for P_1 unless it obtains tighter bounds for its environment $P_2 \parallel \cdots \parallel P_N$ and thus $\mathcal{R}$. In Section II-D, we informally argued that if $\mathcal{G} = (1, \infty, \infty, \infty, 1)$ is a guarantee for P_1, then $(N-1) \times \mathcal{G} = (N-1, \infty, \infty, \infty, N-1)$ must be a guarantee for the $N-1$ threads in P_1's environment $P_2 \parallel \cdots \parallel P_N$. Theorem 2 formalizes this principle: It allows us to switch the roles of reference thread and environment, i.e., to infer bounds on $P_2 \parallel \cdots \parallel P_N$ in an environment of P_1 from already computed bounds on P_1 in an environment of $P_2 \parallel \cdots \parallel P_N$.

Running example (continued)**.** Our method applies Theorem 2 to (2) and obtains

$$\begin{aligned}
&\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{\mathit{Inv}\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\} \text{ where}\\
&\quad \mathcal{R} = (0, 0, 0, 0, 0)\\
&\quad \mathcal{G}_1 = (1, \infty, \infty, \infty, 1)\\
&\quad \mathcal{G}_2 = (N-1, \infty, \infty, \infty, N-1)
\end{aligned}$$

From the above, we have that $(N-1, \infty, \infty, \infty, N-1)$ is a bound for P_1's environment $P_2 \parallel \cdots \parallel P_N$ when run in parallel with P_1. Going back to the RG quintuple fragment (1), our technique refines the rely $\mathcal{R}$, which models $P_2 \parallel \cdots \parallel P_N$, by letting $\mathcal{R} = (N-1, \infty, \infty, \infty, N-1)$. Again, it runs SynthG, which returns $(1, N, N, N-1, 1)$. Thus,

$$\begin{aligned}
&\mathcal{R}, \mathcal{G} \models \{\mathit{Inv}\}\ P_1\ \{\mathsf{true}\} \text{ where}\\
&\quad \mathcal{R} = (N-1, \infty, \infty, \infty, N-1)\\
&\quad \mathcal{G} = (1, N, N, N-1, 1)
\end{aligned}$$

Another refinement of $\mathcal{R}$ from $\mathcal{G}$ by Theorem 2 and another run of SynthG gives

$$\begin{aligned}
&\mathcal{R}, \mathcal{G} \models \{\mathit{Inv}\}\ P\ \{\mathsf{true}\} \text{ where}\\
&\quad \mathcal{R} = (N-1, N(N-1), N(N-1), (N-1)^2, N-1)\\
&\quad \mathcal{G} = (1, N, N, N-1, 1)
\end{aligned}$$

This time, the guarantee has not improved any further, i.e., our method has reached a fixed point and stops the iteration. Applying Theorem 2 gives

$$\begin{aligned}
&\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{\mathit{Inv}\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\} \text{ where}\\
&\quad \mathcal{R} = (0, 0, 0, 0, 0)\\
&\quad \mathcal{G}_1 = (1, N, N, N-1, 1)\\
&\quad \mathcal{G}_2 = (N-1, N(N-1), N(N-1), (N-1)^2, N-1)
\end{aligned}$$

To compute thread-specific bounds for the transitions of P_1, our method may stop here; the bounds can be read off $\mathcal{G}_1$. For example, the second component of $\mathcal{G}_1$ indicates that transition $\ell_1 \to \ell_2$ is executed at most N times. To compute total bounds

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

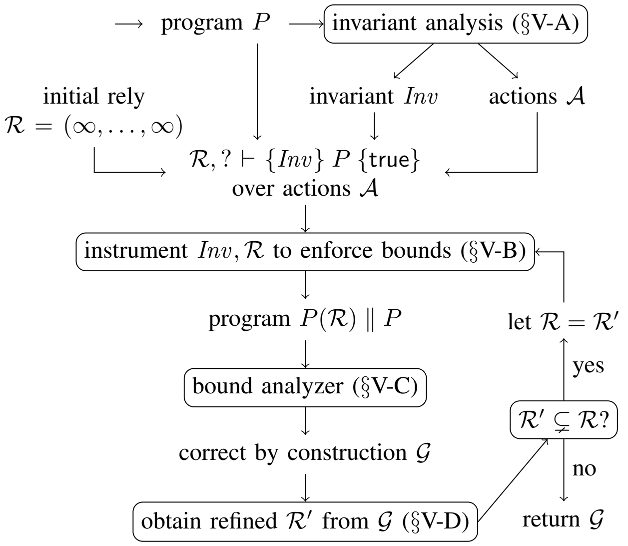


Fig. 3: Overview of our analysis.

for the transitions of the whole interleaved system $P_1 \parallel \cdots \parallel P_N$, our technique applies rule PAR-MERGE, which gives

$$\begin{aligned} &\mathcal{R}, \mathcal{G} \models \{Inv\}\ P_1 \parallel \cdots \parallel P_N\ \{\mathsf{true}\} \text{ where} \\ &\quad \mathcal{R} = (0, 0, 0, 0, 0) \\ &\quad \mathcal{G} = (N, N^2, N^2, (N-1)N, N) \end{aligned}$$

Again, bounds can be read off $\mathcal{G}$, for example the fourth component indicates that the back edge $\ell_3 \to \ell_1$ is executed at most $(N-1) \times N$ times by all N threads in total.

V. AUTOMATION

In this section, we describe the rely-guarantee bound algorithm previously presented on an example; Fig. 3 gives an overview. The algorithm builds on two main insights:

- We reduce RG bound analysis to sequential bound analysis. This allows us to implement procedure SYNTHG.
- We utilize Theorem 2 to iteratively refine bounds on environment assertions until a fixed point is reached.

A. Invariant Analysis

Given a program $P = (L, T, \ell_0)$, our algorithm starts with an invariant analysis to discover a data structure invariant Inv, a set of actions $\mathcal{A}$, and a map $\mathrm{EffectOf}\colon \mathcal{A} \to 2^T$ that indicates which transitions a given action abstracts. In our running example, each action corresponds to one transition, but in general coarser actions may be chosen. Many methods for obtaining these have been described in the literature (e.g., [22], [23], [24], [25], [19]). We use the tool TMREXP [19] as an off-the-shelf solver, which allows us to obtain $\mathcal{A}$ as a stateless program as shown in Fig. 1c.

This allows us to state the RG quintuple fragment

$$\mathcal{R}, ? \vdash \{Inv\}\ P_1\ \{\mathsf{true}\} \tag{3}$$

over $\mathcal{A}$ where $\mathcal{R} = (\infty, \ldots, \infty)$ and the guarantee is unknown. $\mathcal{R}$ soundly over-approximates P_1's environment $P_2 \parallel \cdots \parallel P_N$. We obtain a correct-by-construction guarantee from the thread-modular bound analysis described below.

B. Instrumentation

Given the RG quintuple fragment (3), our method first constructs the program $P(\mathcal{R})$: Let $\mathcal{A} = \{A_1, \ldots, A_m\}$. It starts from the stateless program

$$\textbf{while } (\mathtt{true}) \textbf{ do } A_1 \,[]\, \cdots \,[]\, A_m \textbf{ done}$$

and instruments it with counter variables ξ_A to enforce the bounds in $\mathcal{R}$:

Let $P(\mathcal{R}) = (\{\ell\}, T, \ell)$ be the program over variables $\{\xi_{A_1}, \ldots, \xi_{A_m}\}$ with initial states $[\![g_0]\!]$ where

$$\begin{aligned} T &= \{(\ell, gc_A, \ell) \mid A \in \mathcal{A}\} \\ gc_A &= \begin{cases} \xi_A > 0 \rhd \{A;\ \mathtt{\xi_A\ :=\ \xi_A - 1}\} & \text{if } \mathcal{R}(A) \neq \infty \\ \mathsf{true} \quad\ \rhd \{A\} & \text{otherwise} \end{cases} \\ g_0 &= \bigwedge_{A \in \mathcal{A}} \begin{cases} \xi_A = \mathcal{R}(A) & \text{if } \mathcal{R}(A) \neq \infty \\ \mathsf{true} & \text{otherwise} \end{cases} \end{aligned}$$

Proposition 1. *There exists an isomorphism between runs of $P_1 \parallel P(\mathcal{R})$ from $Inv \wedge g_0$, and traces $\{\tau \in \mathrm{traces}(Inv, P_1) \mid \tau$ starts in σ and $\tau|_{\{\mathtt{e}\}} \models_\sigma \mathcal{R}\}$, such that isomorphic runs and traces have the same length n, and for all positions $0 \leq i \leq n$ their location and state components are equal up to the instrumentation location and variables ℓ and ξ_A of $P(\mathcal{R})$.*

C. Bound Analysis

Our algorithm translates the interleaved heap-manipulating program $\hat{P} = P_1 \parallel P(\mathcal{R})$ and predicate $Inv \wedge g_0$ into an equivalent (bisimilar) integer program and predicate using the technique of [13] (alternatively one could directly compute bounds on the heap-manipulating program $\hat{P}$ using techniques such as described in [26], [27], [28]). From now on, let $\hat{P}$ and $Inv \wedge g_0$ refer to these translations.

Note that $\hat{P}$ is a sequential integer program that can be fed to an off-the-shelf sequential bound analyzer. Let $\hat{T}$ denote the transitions of $\hat{P}$. Our method runs the sequential bound analyzer on $\hat{P}$, which computes a function $\mathrm{SeqBound}\colon \hat{T} \to \mathrm{Expr}(Var_{\mathbb{Z}} \cup \{N\})$, such that for all $t \in \hat{T}$ and all $N \geq 1$, $\mathrm{SeqBound}(t)$ is a bound for t on all runs of $\hat{P}$ from $Inv \wedge g_0$.

Then, our technique maps bounds obtained on transitions of $\hat{P}$ back to the corresponding transitions of P_1 in $\hat{P} = P_1 \parallel P(\mathcal{R})$, which allows it to compute the desired guarantee for P_1: Letting

$$\mathcal{G}(A) = \sum_{t \in \mathrm{EffectOf}(A)} \mathrm{SeqBound}(t)$$

for all $A \in \mathcal{A}$ gives a correct-by-construction guarantee $\mathcal{G}$ for $\mathcal{R}, ? \vdash \{Inv\} P_1 \{\mathsf{true}\}$, i.e., we have $\mathcal{R}, \mathcal{G} \models \{Inv\} P_1 \{\mathsf{true}\}$.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

D. Bound Refinement

Our algorithm then uses Theorem 2 to refine the rely of P_1 and checks if the computation has reached a fixed point yet. Let $\mathcal{R}'(A) = (N-1) \times \mathcal{G}(A)$ for all $A \in \mathcal{A}$.

1) If $\mathcal{R}' \subsetneq \mathcal{R}$, by Theorem 2 $\mathcal{R}'$ is a valid bound for $P_2 \parallel \cdots \parallel P_N$. Our algorithm iterates the computation of bounds for $\mathcal{R}', ? \vdash \{\mathit{Inv}\}\ P_1\ \{\mathtt{true}\}$ starting from Section V-B.
2) If $\mathcal{R}' = \mathcal{R}$, the algorithm has reached a fixed point and reports the results of the analysis:
    a) For thread-specific bounds of P_1, return $\mathcal{G}$.
    b) For total bounds of $P_1 \parallel \cdots \parallel P_N$, apply Theorem 2 to get a guarantee for $P_2 \parallel \cdots \parallel P_N$, and use rule PAR-MERGE to sum up the guarantees of P_1 and $P_2 \parallel \cdots \parallel P_N$.
3) $\mathcal{R}' \not\subseteq \mathcal{R}$ can be avoided by implementing a sequential bound analyzer that is deterministic and monotonic in the sense that it always finds the same or smaller bounds on programs with further restricted transition relations.

VI. CASE STUDIES

We have implemented the algorithm of Section V in our tool COACHMAN [29] and tested it on three well-known lock-free data structures from the literature: Treiber's stack [18], the Michael-Scott queue [9], and the DGLM queue [30]. For the sequential bound analyzer, we have implemented an algorithm similar to the one described in [7]; its implementation is available online [29].

For each data structure, our tool constructs a general client program $P = \mathtt{op1}()\ [] \cdots []\ \mathtt{opM}()$, and analyzes its N-times interleaving $\|_N P = P_1 \parallel \cdots \parallel P_N$ for thread-specific bounds of a single thread P_i and total bounds of $P_1 \parallel \cdots \parallel P_N$ as described in Section II. For brevity, we only report complexity bounds here. All performance results were obtained on a single core of a 2.0GHz Intel Core i7 processor.

1) Treiber's stack [18]: We thoroughly discussed Treiber's stack in our running example (Section II). Our tool takes **2 iterations** to obtain the stack's thread-specific linear asymptotic complexity $\mathbf{O}(\mathbf{N})$ of a single thread P_i, and the total quadratic complexity $\mathbf{O}(\mathbf{N^2})$ of $P_1 \parallel \cdots \parallel P_N$ in **3 minutes**[2].

2) Michael-Scott queue [9]: This lock-free queue has, e.g., been implemented in the `ConcurrentLinkedQueue` class of the Java standard library. In contrast to Treiber's stack, the transitions of the Michael-Scott queue cannot be bounded with just a single refinement operation: It synchronizes via two CAS operations, the first one breaking/looping as in Treiber's stack, the second one located on a back edge of the main loop. Thus our algorithm cannot immediately bound the action corresponding to the second CAS. Rather, it first bounds the first CAS' action, refines and bounds the second CAS' action, and after a final refinement bounds all other edges. Our tool takes **3 iterations** to obtain the queue's thread-specific linear asymptotic complexity $\mathbf{O}(\mathbf{N})$, and the total quadratic complexity $\mathbf{O}(\mathbf{N^2})$ in **148 minutes**.

3) DGLM queue [30]: The DGLM queue is a recent, optimized version of the Michael-Scott queue. Similar to the Michael-Scott queue, our tool takes **3 iterations** to obtain the queue's thread-specific linear asymptotic complexity $\mathbf{O}(\mathbf{N})$, and the total quadratic complexity $\mathbf{O}(\mathbf{N^2})$ in **77 minutes**.

Remark (Discussion of algorithm runtime)**.** The increased runtime on the queue case studies compared to Treiber's stack is due to to their larger program LTS and doubled number of environment actions. In particular, the counter automaton produced by [13] for Treiber's stack has 531 vertices and 2,072 edges, while for the MS queue we obtain 6,165 vertices and 37,402 edges.

The runtime speedup on the DGLM queue compared to the MS queue is explained by its optimized `deq` method: Its LTS has 2 instead of 4 back edges, which drastically reduces the time spent in bound analysis.

VII. RELATED WORK

Albert et.al. [31] describe a RG bound analysis for actor-based concurrency. They use heuristics to guess an unsound guarantee and justify it by proving that all environment actions not captured by the guarantee occur only finitely often. We note that the approach of [31] leaves environment actions not captured by the guarantee completely unconstrained, i.e., they may change the program state arbitrarily, leading to coarser than necessary bounds. In contrast, our approach includes all environment actions, recognizes that actions occurring boundedly often already carry ranking information, and leaves their handling to the sequential bound analyzer.

More closely related to our work, Gotsman et al. [14] present a general framework for expressing liveness properties in RG specifications and apply it to prove termination/unbounded lock-freedom. They give rely and guarantee as words over actions, and instantiate it for properties stating that a set of actions does not occur infinitely often. They automatically discharge such properties in an iterative proof search over the powerset of actions. Our approach differs in various aspects: First, while our RG quintuples may be formulated as words over actions, the instantiation in [14] is suitable only for termination, but too weak for bound analysis. Second, the focus on liveness properties leads to more complicated proof rules in [14], which have to account for the fact that naive circular reasoning about liveness properties is unsound [32], [33], [14]. In contrast, all sequences of actions expressable by our environment assertions are safety-closed, allowing us to use the full power of RG-style circular arguments in the premises of our reasoning rules. Finally, we obtain bounds for all actions at once in a refinement step by reduction to sequential bound analysis, rather than iteratively querying a termination prover whether a particular action occurs only finitely often.

---

[2] A further optimization of the bound algorithm only applicable to this case study allows us to speed up the runtime to 47 seconds.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

VIII. CONCLUSION

We have presented the first extension of rely-guarantee reasoning to bound analysis, and automated bound analysis of concurrent programs by a reduction to sequential bound analysis. In addition, we have for the first time automatically inferred bounds for three widely-studied lock-free data structures.

IX. FUTURE WORK

While lock-freedom guarantees absence of live-locks, it does not guarantee starvation-freedom: If a thread's environment interferes infinitely often, the thread may loop forever. *Wait-freedom* is a stronger progress property that guarantees that each individual thread makes progress. Its implementation exposes global variables per thread; handling this is an interesting problem for the future.

While our framework extends Jones' RG reasoning, we have only given inference rules for parallel composition and a consequence rule and have left the concrete programming language and corresponding rules abstract. Our only requirement regarding safety is that the environment actions obtained in Section V-A over-approximate any thread's effect on the global state. Giving a full set of rules and exploring a tighter integretion between safety and (bounded) liveness properties is left for future work.

ACKNOWLEDGMENT

We would like to thank Erez Petrank and Ana Sokolova for comments and discussions on this work.

REFERENCES

[1] S. Gulwani and F. Zuleger, "The reachability-bound problem," in *PLDI*, 2010, pp. 292–304.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-form upper bounds in static cost analysis," *J. Autom. Reasoning*, vol. 46, no. 2, pp. 161–203, 2011.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord, "Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs," in *SAS*, ser. Lecture Notes in Computer Science, vol. 6337. Springer, 2010, pp. 117–133.

[4] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Analyzing runtime and size complexity of integer programs," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 4, pp. 13:1–13:50, 2016.

[5] A. Flores-Montoya and R. Hähnle, "Resource analysis of complex programs with cost equations," in *APLAS*, ser. Lecture Notes in Computer Science, vol. 8858. Springer, 2014, pp. 275–295.

[6] Q. Carbonneaux, J. Hoffmann, T. W. Reps, and Z. Shao, "Automated resource analysis with coq proof objects," in *CAV*, 2017, pp. 64–85.

[7] M. Sinn, F. Zuleger, and H. Veith, "Complexity and resource bound analysis of imperative programs using difference constraints," *J. Autom. Reasoning*, vol. 59, no. 1, pp. 3–45, 2017.

[8] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[9] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*. ACM, 1996, pp. 267–275.

[10] V. Vafeiadis, "Automatically proving linearizability," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 450–464.

[11] S. Chakraborty, T. A. Henzinger, A. Sezgin, and V. Vafeiadis, "Aspect-oriented linearizability proofs," *Logical Methods in Computer Science*, vol. 11, no. 1, 2015.

[12] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine, "An integrated specification and verification technique for highly concurrent data structures," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 7795. Springer, 2013, pp. 324–338.

[13] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar, "Programs with lists are counter automata," *Formal Methods in System Design*, vol. 38, no. 2, pp. 158–192, 2011.

[14] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis, "Proving that non-blocking algorithms don't block," in *POPL*. ACM, 2009, pp. 16–28.

[15] X. Jia, W. Li, and V. Vafeiadis, "Proving lock-freedom easily and automatically," in *CPP*. ACM, 2015, pp. 119–127.

[16] E. Petrank, M. Musuvathi, and B. Steensgaard, "Progress guarantee for parallel programs via bounded lock-freedom," in *PLDI*. ACM, 2009, pp. 144–154.

[17] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983, pp. 321–332.

[18] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, 1986.

[19] L. Holík, R. Meyer, T. Vojnar, and S. Wolff, "Effect summaries for thread-modular analysis - sound analysis despite an unsound heuristic," in *SAS*, ser. Lecture Notes in Computer Science, vol. 10422. Springer, 2017, pp. 169–191.

[20] C. Flanagan, S. N. Freund, and S. Qadeer, "Thread-modular verification for shared-memory programs," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 2305. Springer, 2002, pp. 262–277.

[21] Q. Xu, W. P. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *Formal Asp. Comput.*, vol. 9, no. 2, pp. 149–174, 1997.

[22] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv, "Thread-modular shape analysis," in *PLDI*. ACM, 2007, pp. 266–277.

[23] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv, "Thread quantification for concurrent shape analysis," in *CAV*, ser. Lecture Notes in Computer Science, vol. 5123. Springer, 2008, pp. 399–413.

[24] V. Vafeiadis, "Rgsep action inference," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 5944. Springer, 2010, pp. 345–361.

[25] A. Miné, "Static analysis of run-time errors in embedded critical parallel C programs," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 6602. Springer, 2011, pp. 398–418.

[26] T. Fiedor, L. Holík, A. Rogalewicz, M. Sinn, T. Vojnar, and F. Zuleger, "From shapes to amortized complexity," in *VMCAI*, 2018, pp. 205–225.

[27] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann, "Automatically proving termination and memory safety for programs with pointer arithmetic," *J. Autom. Reasoning*, vol. 58, no. 1, pp. 33–65, 2017.

[28] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla, "Automatic inference of resource consumption bounds," in *LPAR-18*, 2012, pp. 1–11.

[29] "Coachman," https://github.com/thpani/coachman.

[30] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *FORTE*, ser. Lecture Notes in Computer Science, vol. 3235. Springer, 2004, pp. 97–114.

[31] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin, "Rely-guarantee termination and cost analyses of loops with concurrent interleavings," *J. Autom. Reasoning*, vol. 59, no. 1, pp. 47–85, 2017.

[32] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 507–534, 1995.

[33] K. L. McMillan, "Circular compositional reasoning about liveness," in *CHARME*, ser. Lecture Notes in Computer Science, vol. 1703. Springer, 1999, pp. 342–345.

ISBN: 978-0-9835678-8-2.

Template-Based Verification of Heap-Manipulating Programs

Viktor Malík*[‡] Martin Hruska[‡] Peter Schrammel*[†] Tomáš Vojnar[‡]
*Diffblue Ltd, Oxford, UK [†]University of Sussex, Brighton, UK [‡]FIT BUT, IT4Innovations Centre of Excellence, CZ

***Abstract*—We propose a shape analysis suitable for analysis engines that perform automatic invariant inference using an SMT solver. The proposed solution includes an abstract template domain that encodes the shape of the program heap based on logical formulae over bit-vectors. It is based on computing a points-to relation between pointers and symbolic addresses of abstract memory objects. Our abstract heap domain can be combined with value domains in a straightforward manner, which particularly allows us to reason about shapes and contents of heap structures at the same time. The information obtained from the analysis can be used to prove memory safety and reachability properties, expressed by user assertions, of programs manipulating dynamic data structures, mainly linked lists. The solution has been implemented in the 2LS framework and compared against state-of-the-art tools that perform the best in heap-related categories of the well-known Software Verification Competition (SV-COMP). Results show that 2LS outperforms these tools on benchmarks requiring combined reasoning about unbounded data structures and their numerical contents.**

I. INTRODUCTION

Reasoning about dynamic data structures is one of the core problems in software verification. The techniques implemented in state-of-the-art verification tools for C programs such as those competing in the Software Verification Competition (SV-COMP) have shortcomings when it comes to combined reasoning about shape and content of data structures as our experiments revealed. We address this problem in this paper in the context of template-based program verification.

Template-based verification uses a logic-based synthesis approach to inferring the invariants required for proving program properties. It delegates semantic reasoning to SMT solvers and focusses on the design of appropriate template domains and efficient algorithms for finding the optimal template parameters (i.e. least fixed points in the abstract interpretation sense [14]). The use of such templates makes it straightforward to compute invariants describing both shape and value properties of data structures, which is more difficult when combining domains that are based on different principles.

Running example: To better illustrate the concepts and methods proposed in the paper, we use the program in Listing 1 as a running example. It creates a singly-linked list, each node containing a value between 10 and 20 (Lines 7–15). The list is afterwards traversed repeatedly and the value of each node is either incremented by 1 or halved (Lines 16–22). We add an assertion that, in every iteration, the value of each node stays between 10 and 20. The goal of the analysis is to prove that the assertion always holds. This requires an analysis capable of reasoning about unbounded linked data structures and numerical content of their nodes at the same time.

Listing 1: A running example

```
1  typedef struct node {
2    int val;
3    struct node *next;
4  } Node;
5
6  int main() {
7    Node *p, *list = malloc(sizeof(Node));
8    Node *tail = list;
9    *list = {.next = NULL, .val = 10};
10   while (__VERIFIER_nondet_int()) {
11     int x = __VERIFIER_nondet_int();
12     if (x < 10 || x > 20) continue;
13     p = malloc(sizeof(Node));
14     *p = {.next = NULL, .val = x};
15     tail→next = p; tail = p;
16   }
17   while (1) {
18     for (p = list; p!= NULL; p = p→next) {
19       assert(p→val <= 20 && p→val >= 10);
20       if (p→val < 20) p→val++;
21       else p→val /= 2;
22 } } }
```

To prove this property we have to infer that the value of the `val` field of the dynamic objects allocated in Line 7 and 13 is always in the range $[10, 20]$.

With the help of our technique, we will infer an invariant for the loop on Line 10 that states the following:

- `tail` may point to the sets of `Node` objects created in Line 7 and 13. We denote these sets ao_7 and ao_{13}, resp.
- The `next` field of ao_7 may point to ao_{13} or `null`. Its `val` field has a value in the interval [10,10].
- The `next` field of ao_{13} may point to ao_{13} or `null`. However, its `val` field has a value in the interval [10,20]. This means that ao_{13} abstracts a set of `Node` objects whose `val` fields have values in the interval [10,20].

For the loop in Line 18, we infer the invariant that the `val` fields of ao_7 and ao_{13} must both be in the interval [10,20], which implies that the property holds.

Contributions: The contributions of this paper, which form the contents of Sections III–VII, are as follows:

1) We propose a novel abstract template domain for reasoning over heap-allocated data structures such as singly and doubly linked lists using a template-based parameter synthesis engine.
2) We show how we can build product and power domain combinations of our heap domain with structural domains (e.g. trace partitioning) and value domains such as template polyhedra that capture the content of data structures.
3) We implement our abstract heap domain in the 2LS verification tool for C programs. We demonstrate the power

ISBN: 978-0-9835678-8-2.

of the proposed domain on benchmarks, which require combined reasoning about the shape and content of data structures, showing that other tools, which performed well in SV-COMP, cannot handle these examples.

II. TEMPLATE-BASED PROGRAM VERIFICATION

This section describes the approach to program verification using template-based synthesis of inductive invariants which the 2LS tool [35] is based upon and that underlies our approach too. The source program is first translated into single static assignment (SSA) form. Using this program representation, the verification task can then be expressed as a second-order logical formula. However, since suitable solvers for such formulae are not available, the verification problem is reduced to synthesising loop invariants using parametrised templates and an SMT solver to find suitable values of the parameters.

A. Program Verification Using Inductive Invariants

A state of a program is a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—states in the set are defined by models of the formula. Given a vector of variables $\vec{x}$, the predicate $\mathit{Init}(\vec{x})$ describes the initial states. A transition relation is described as a formula $\mathit{Trans}(\vec{x}, \vec{x}')$.

From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $\mathit{Init}(\vec{x})$. This is, however, difficult to compute, so instead, we use an *inductive invariant*. A verification task requires showing that the set of reachable states does not intersect with the set of error states $\mathit{Err}(\vec{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

$$\begin{aligned} \exists_2 \mathit{Inv}.\ \forall \vec{x}, \vec{x}'.\ &(\mathit{Init}(\vec{x}) \implies \mathit{Inv}(\vec{x})) \wedge \\ &(\mathit{Inv}(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}') \implies \mathit{Inv}(\vec{x}')) \wedge \\ &(\mathit{Inv}(\vec{x}) \Rightarrow \neg \mathit{Err}(\vec{x})) \end{aligned} \tag{1}$$

B. Invariant Inference via Templates

To directly handle Eq. (1) by a solver, it would require the capability to deal with second-order logic quantification. Since a suitably general and efficient second-order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant Inv to $\mathcal{T}(\vec{x}, \vec{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\vec{x}$ and template parameters $\vec{\delta}$. This restriction corresponds to the choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for the template parameters:

$$\begin{aligned} \exists \vec{\delta}.\ \forall \vec{x}, \vec{x}'.\ &(\mathit{Init}(\vec{x}) \implies \mathcal{T}(\vec{x}, \vec{\delta})) \wedge \\ &(\mathcal{T}(\vec{x}, \vec{\delta}) \wedge \mathit{Trans}(\vec{x}, \vec{x}') \implies \mathcal{T}(\vec{x}', \vec{\delta})) \end{aligned} \tag{2}$$

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\vec{d}$ as candidates for template parameters $\vec{\delta}$. For a value $\vec{d}$, the template formula $\mathcal{T}(\vec{x}, \vec{d})$ is an invariant if and only if Eq. (3) is unsatisfiable.

$$\begin{aligned} \exists \vec{x}, \vec{x}'.\ &\neg(\mathit{Init}(\vec{x}) \implies \mathcal{T}(\vec{x}, \vec{d})) \vee \\ &\neg(\mathcal{T}(\vec{x}, \vec{d}) \wedge \mathit{Trans}(\vec{x}, \vec{x}') \implies \mathcal{T}(\vec{x}', \vec{d})) \end{aligned} \tag{3}$$

From the abstract interpretation point of view, $\vec{d}$ is an abstract value, i.e. it represents (*concretises to*) the set of all program states $\vec{x}$ that satisfy the formula $\mathcal{T}(\vec{x}, \vec{d})$. The abstract values representing the infimum $\perp$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\vec{x}, \perp) \equiv \mathit{false}$ and $\mathcal{T}(\vec{x}, \top) \equiv \mathit{true}$ [8].

Formally, the concretisation function γ is: $\gamma(\vec{d}) = \{\vec{x} \mid \mathcal{T}(\vec{x}, \vec{d}) \equiv \mathit{true}\}$. In the abstraction function, to get the most precise abstract value representing the given concrete program state $\vec{x}$, we let $\alpha(\vec{x}) = \min(\vec{d})$ such that $\mathcal{T}(\vec{x}, \vec{d}) \equiv \mathit{true}$. Since the abstract domain forms a complete lattice, existence of such a minimal value $\vec{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\vec{d} = \perp$ and iteratively solves Eq. (3) using an SMT solver. If the formula is unsatisfiable, then an invariant has been found, otherwise a model of satisfiability is returned by the solver. The model represents a counterexample to the current instantiation of the template being an invariant. The value of the template parameter $\vec{d}$ is then updated by combining with the obtained model of satisfiability $\vec{d'}$ using a domain-specific join operator [8]. For example, assume we have a program with a loop that counts from 0 to 10 in variable x and we have a template $x \leq d$. Let's assume that the current value of the parameter d is 3 and we get a new model $d' = 4$. Then we update the parameter to 4 by computing $d \sqcup d' = \max(d, d')$, because $\max$ is the join operator for a domain that tracks numerical upper bounds.

C. Source Program Encoding

In this paper, we deal with non-recursive programs with all function calls inlined. As said above, we encode the program into a formula representing a specific *static single assignment form* (SSA). For acyclic programs, the SSA represents exactly the strongest postcondition of the program—as usual, with a fresh copy x_i of each variable x for each program location i where the value of x is modified. The effect of loops is over-approximated as described in [8]. In this encoding, special variables called *guards* are used to track the control flow of the program. In particular, for each program location i, a Boolean variable g_i is introduced, and its value encodes whether the program location is reachable.

To see how the over-approximation of program loops is achieved, note that, at the loop head, the program path coming from before the loop joins with the path coming from the end of the loop (assuming that all paths within the loop join before its end; and likewise for the paths coming from before the loop). To achieve acyclicity of the SSA, we cut the path coming from the end of the loop. We then represent the value

ISBN: 978-0-9835678-8-2.

of each variable x at the loop head using a *phi variable* x^{phi} whose value is defined by a non-deterministic choice between the value coming from before the loop, say x_0, and the value coming from the end of the loop. The latter value is represented by a newly introduced *loop-back* variable x^{lb}. In particular, we let $x^{phi} = g^{ls} \ ? \ x^{lb} : x_0$ where g^{ls} is a so-called *loop-select* Boolean guard that is unconstrained in order to model the non-deterministic choice. Moreover, to over-approximate the effect of the loop, the value of the loop-back variable x^{lb} is initially unconstrained too and later constrained by the derived candidate loop invariants.

Example. In Listing 1, the loop head at Line 10 joins two different values of variable *tail* coming from program locations 8 and 15. The value of *tail* coming from the end of the loop (denoted $tail_{15}$ in the SSA) is replaced by the loop-back variable $tail^{lb}_{16}$. The corresponding phi variable $list^{phi}_{10}$ then non-deterministically joins $tail^{lb}_{16}$ with the value of *tail* from before the loop via the loop-select variable g^{ls}_{16}:

$$list^{phi}_{10} = g^{ls}_{16} \ ? \ list^{lb}_{16} \ : \ list_8 \tag{4}$$

III. ABSTRACT MEMORY OPERATIONS IN THE SSA FORM

We now propose a representation of heap memory and operations over it, designed to be used within the approach laid out in Section II. The proposal respects the fact that the considered SSA form is an acyclic program representation, over-approximating reachable values of variables used in loops.

A. Abstract Memory Representation

Under our assumption of fully inlined, non-recursive programs, *static memory objects* correspond simply to a finite set *Var* of *program variables*: we do not need to consider the stack. We let $PVar, SVar \subseteq Var$, $PVar \cap SVar = \emptyset$, be the sets of variables of *pointer* and *structure type*, respectively. A linked data structure in C is typically defined using a `struct` type, which groups together named *fields* for the payload data and the link pointers (see Lines 1–4 in Listing 1). We use *Fld* to denote the finite set of fields used in the given program. Let $PFld \subseteq Fld$ be the set of all pointer-typed fields.

1) Abstract Dynamic Objects: We use *abstract dynamic objects* to represent *dynamic memory objects*, i.e. those that are allocated using `malloc` (or some of its variants) on the heap. An abstract dynamic object represents a set of concrete dynamic objects allocated at the same *allocation site* i, e.g. by the same `malloc` call located at Line i in Listing 1. However, a single abstract dynamic object is not sufficient to represent *all* concrete dynamic objects allocated by a given `malloc`. The reason for this is that the program may use several independent objects created at an allocation site at the same time. Typically, this issue is solved by the analysis algorithm materialising dynamic objects on-demand. We take a different approach and statically over-approximate the maximum number n_i of concrete objects required (see next section below). Hence, we use a *set* $AO_i = \{ao^k_i \mid 1 \leq k \leq n_i\}$ of abstract dynamic objects for that purpose. We let $AO = \cup_i AO_i$ and require $Var \cap AO = \emptyset$ and $AO_i \cap AO_j = \emptyset$ for $i \neq j$. The set of all objects of our program abstraction is then $Obj = AO \cup Var$.

Pairs consisting of an abstract dynamic object and a field, i.e. elements of the set $AO \times Fld$, represent an abstraction of the appropriate *fields* of all the represented concrete objects. We use the "dot" notation to represent such pairs: e.g. $ao_i.next$ denotes the abstraction of the $next$ field of all the concrete dynamic objects represented by ao_i.

We define $Ptr = PVar \cup ((SVar \cup AO) \times PFld)$ to be the set of all *pointers* of the given program abstraction. Pointers can be assigned addresses of objects. Since we currently do not support pointer arithmetic, the only addresses that we consider are *symbolic addresses* of static and dynamic objects together with the special address null. The symbolic address of an abstract dynamic object ao_i is an abstraction of the symbolic addresses of the concrete dynamic objects represented by ao_i. To get the address of both static and dynamic objects, we use the &-operator. Hence, the set *Addr* of addresses that we consider is defined as $Addr = \{\&o \mid o \in Obj\} \cup \{\mathsf{null}\}$.[1]

2) Pre-Materialisation: As mentioned above, instead of materialising dynamic objects on-demand, we pre-materialise a sufficient number n_i of them for each allocation site i and encode them into our SSA representation. In order for this abstraction to be sound, it is sufficient that the number n_i equals the maximal number of distinct concrete objects allocated at i that are simultaneously pointed to by some pointer at any location of the analysed program.

For each allocation site i, we compute the number n_i as follows. First, using a standard static may-alias analysis, we over-approximate, for each program location j, the set P^i_j of all pointer expressions of the source program that *may point* to some object allocated at i. These might be pointer variables from $PVar$, pointer-typed fields of static objects from $SVar \times PFld$, or pointer-typed fields of dynamic objects accessed through dereferences of pointers—i.e. elements of $PVar \times PFld$. For simplicity, we assume that all chained dereferences of the form $p \to f_1 \to f_2$ with $f_1, f_2 \in PFld$ are broken into two expressions using an intermediate variable. Overall, $P^i_j \subseteq PVar \cup ((SVar \cup PVar) \times PFld)$. Next, we compute the *must-alias relation* $\sim_j$. For each pair of pointers p and q and for each program location j, $p \sim_j q$ iff p and q must point to the same concrete dynamic object at j. Finally, we partition the set P^i_j into equivalence classes by $\sim_j$, and n_i is given by the maximal number of such classes at any j.

B. Operations over the Abstract Memory Representation

1) Dynamic Memory Allocation: We represent a call to `malloc` at program location i by a non-deterministic choice among the addresses of objects from the set AO_i. Hence, a statement $p = \mathtt{malloc}(\ldots)$ at i is translated to the formula $p_i = g^{os}_{i,1} ? \&ao^1_i : (g^{os}_{i,2} ? \&ao^2_i : (\ldots (g^{os}_{i,n_i-1} ? \&ao^{n_i-1}_i : \&ao^{n_i}_i)))$ where $g^{os}_{i,j}$, $1 \leq j < n_i$ are free Boolean variables, so-called *object-select guards*.

---

[1] We currently assume that addresses of newly allocated objects are fresh. Hence, we can miss behaviours where some memory space is recycled while some pointers are still pointing to it, which is undefined according to the C standard, but sometimes used in practice. If that was a problem, we could, e.g., extend our preliminary static analysis to detect objects that can possibly be in that form and add them among possible returns from the allocation.

Example. In Listing 1, two calls of `malloc` occur on Lines 7 and 13. For Line 7, a single abstract dynamic object ao_7 is created as there is just one concrete object allocated.[2] The `malloc` on Line 13 must be represented by two objects ao^1_{13} and ao^2_{13} as, e.g. on Line 14, variables `tail` and `p` may point to different concrete objects allocated by this `malloc` call. Specifically, the statement on Line 13 will be translated into the equality $p_{13} = g^{os}_{13}\,?\,\&ao^1_{13} : \&ao^2_{13}$. Abstract dynamic objects ao^1_{13} and ao^2_{13} then collectively represent all concrete dynamic objects allocated in the loop.

2) Reading through Dereferenced Pointers: We handle expressions of the form $p \to f$ for $p \in PVar$, $f \in Fld$ appearing on the right-hand side of assignments or in conditions as follows. We first perform a *may-points-to analysis*, which over-approximates for each pointer $p \in Ptr$ and each program location i the set of objects from Obj that p may point to at i. Using the result of the analysis, we can replace the pointer dereference $p \to f$ by a choice among the values of the field f of the objects possibly pointed to by p.

To facilitate the replacement, we introduce purely logical *dereference variables*. Assume that at program location i there appears an R-expression $p \to f$ and that the pointer p may point to a set of objects $O \subseteq Obj$ at i. We replace the use of $p \to f$ by using a fresh variable $drf(p).f_i$ whose value is defined by the formula $(\bigwedge_{o \in O} p_j = \&o \implies drf(p).f_i = o.f_k) \wedge ((\bigwedge_{o \in O} p_j \neq \&o) \implies drf(p).f_i = o_\perp)$ where p_j, $o.f_k$ are the relevant versions of the concerned variables at program location i and $o_\perp$ denotes a special "unknown object" (a result of a dereference of an unknown or invalid (null) address).[3]

Example. We give the translation of the assignment $p = p \to next$ from Line 18 in Listing 1. Since the assignment is executed at the end of each loop iteration, we define its program location to be Line 22. At this program location, p may point to the set of objects $\{ao_7, ao^1_{13}, ao^2_{13}\}$. Hence, the assignment will be represented by the following formula.

$$\begin{aligned}
&p_{22}{=}drf(p).next_{22} \wedge\\
&\left(p^{phi}_{18}{=}\&ao_7 \Rightarrow drf(p).next_{22}{=}ao_7.next^{phi}_{18}\right) \wedge\\
&\bigwedge_{l=1,2} \left(p^{phi}_{18}{=}\&ao^l_{13} \Rightarrow drf(p).next_{22}{=}ao^l_{13}.next^{phi}_{18}\right) \wedge\\
&\Big(\big(p^{phi}_{18}{\neq}\&ao_7 \wedge \bigwedge_{l=1,2} p^{phi}_{18}{\neq}\&ao^l_{13}\big) \Rightarrow drf(p).next_{22}{=}o_\perp\Big)
\end{aligned}$$

The first conjunct represents the transformed assignment, and the following conjuncts define the value of the dereference variable. The value of p entering program location 22 is the value from the loop head p^{phi}_{18}. If it equals the address of ao_7, ao^1_{13}, or ao^2_{13}, the value of $drf(p).next_{22}$ is $ao_7.next^{phi}_{18}$, $ao^1_{13}.next^{phi}_{18}$, or $ao^2_{13}.next^{phi}_{18}$, otherwise, it equals $o_\perp$.

As an optimisation, if the dereference variable is once created and the value of the concerned expression does not change, we reuse the existing dereference variable. Second, when dealing with a statement like $v = p \to f$, the use of the dereference variable may seem unnecessary as one can plug v_i instead of $drf(p).f_i$ into the formula defining the value of $drf(p).f_i$. This can be done, but, as explained below, the use of dereference variables can give us more precision when dealing with sequences of reading and writing operations.

3) Writing through a Dereference: When writing into an abstract dynamic object ao_i, we need to respect the fact that only one concrete object abstracted by ao_i is actually written to, and the others keep the original value. Hence, we need to make a join of the original and new value. We again use dereference variables to facilitate the transformation.

Assume that at program location i, we have an assignment $p \to f = v$, $p \in PVar$, $f \in Fld$, $v \in Var$, and that p may point to a set of objects $O \subseteq Obj$ at the entry to i.[4] We replace the L-expression $p \to f$ by a fresh variable $drf(p).f_i$ whose value is defined by the value of v, i.e. we assert that $drf(p).f_i = v_l$ where v_l is the version of v valid at program location i. We then use $drf(p).f_i$ to update the value of the field f of the referenced object, using the formula $\bigwedge_{o \in O} o.f_i = (p_j = \&o \wedge g^{os}_i)\,?\,drf(p).f_i : o.f_k$ where p_j, $o.f_k$ are the relevant versions of the variables p and $o.f$ at program location i.[5] The formula expresses the fact that $o.f_i$ gets updated if p equals the address of o, otherwise its value remains unchanged; k is the last program location before i where the value of $o.f$ was changed. The object-select guard g^{os}_i, which is a freshly introduced unconstrained Boolean variable, enforces that the value of field f is changed in only one of the concrete objects abstracted by o while it remains unchanged in the other objects abstracted by o. If o is not allocated in a loop (and hence representing a single instance), g^{os}_i may be omitted.

Example. For illustration, the assignment `tail->next=p` from Line 15 of Listing 1 will be translated into the formula:

$$\begin{aligned}
&(drf(list).next_{15} = p_{13}) \wedge\\
&\Big(ao_7.next_{15} = (list^{phi}_{10}{=}\&ao_7)\ ?\\
&\qquad\qquad drf(list).next_{15}\ :\ ao_7.next^{phi}_{10}\Big) \wedge\\
&\bigwedge_{l=1,2} \big(ao^l_{13}.next_{15} = (list^{phi}_{10}{=}\&ao^l_{13} \wedge g^{os}_{15})\ ?\\
&\qquad\qquad drf(list).next_{15}\ :\ ao^l_{13}.next^{phi}_{10}\big)
\end{aligned}$$

As mentioned above, the use of dereference variables may increase the precision of our analysis. This happens in particular when we write into an abstract object through some pointer and later read the written value back through the same pointer (or a pointer aliased with it) without any change of the pointers and the concerned value in between. Then, we get back exactly the value that we wrote, which would otherwise not happen due to the joins involved.

4) Memory Free: Since the `free` operation has no effect on the heap reachability itself, we defer its discussion to Section V devoted to checking memory safety.

---

[2] In fact, we should write ao^1_7, but we omit the superscript when a single abstract object suffices. Likewise for the object-select guards below.

[3] A dereference of the form $*p$ for a non-structured object can be handled analogously, just without the field f in the above formula.

[4] More complex assignments can be transformed into this form.

[5] A write to a dereference of the form $*p$ to a non-structured object can be handled analogously, omitting field f from the formula.

IV. AN ABSTRACT DOMAIN FOR HEAP ANALYSIS

We will now work towards our template-based abstract domain suitable for reasoning about properties of heap-manipulating programs, starting from a base shape domain and refining it. We will show that, due to the fact that all domains in the considered approach are based on templates, the new domain can be easily combined with other domains, e.g. for inferring properties about numerical data of data structures.

A. Base Abstract Shape Domain

In the considered approach, an abstract domain needs to have the form of a *template*—a fixed, parametrised, quantifier-free first-order logic formula describing the desired property of a program. As described in Section II, templates are used to efficiently compute *loop invariants* of the analysed program. These are used to constrain values of the *loop-back variables* that are used in the SSA-based program encoding to over-approximate values returning from the end of the loop to the loop head. Hence, a loop invariant describes a property that holds for some program variables at the end of the loop body after any iteration of the loop. Hence, we limit our shape domain to the set Ptr^{lb} of all *loop-back pointers*. Let L be the set of all loops in the program. Since there is one loop-back pointer variable for each pointer variable and each loop, we define $Ptr^{lb} = Ptr \times L$. We denote elements $(p, l) \in Ptr^{lb}$ by p_i^{lb} where i is the program location of the end of the loop l. Intuitively, the value of p_i^{lb} is an abstraction of the value of the pointer p coming from the end of the body of the loop l. The property that our base shape domain describes is the *may-point-to* relation between the set Ptr^{lb} and the set $Addr$.[6]

The template of our base shape domain has the form of the formula $\mathcal{T}^S \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$. It is a conjunction of so-called *template rows* $\mathcal{T}^S_{p_i^{lb}}$, each row corresponding to one loop-back pointer from the set Ptr^{lb}. A template row $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ describes the may-point-to relation for the loop-back pointer p_i^{lb}. The parameter $d_{p_i^{lb}} \subseteq Addr$ of the row (a so-called *abstract value of the row*) specifies the set of all addresses from the set $Addr$ that p may point to at the location i. The template row can thus be expressed as the quantifier-free formula $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv (\bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a)$.

Abstract values of template rows corresponding to pointer fields of abstract dynamic objects allow the domain to describe unbounded linked paths in the heap, such as list segments.

Example. In Listing 1, a list segment is created by the first loop. Objects in the segment are linked through the pointer field `next`, and they are represented by the abstract dynamic objects ao_{13}^1 and ao_{13}^2. In our base shape domain, the shape of this segment will be described by an invariant for the first loop, specifically by the two template rows for $ao_{13}^1.next_{16}^{lb}$ and $ao_{13}^2.next_{16}^{lb}$. They will give us the formula $\wedge_{l=1,2} \mathcal{T}^S_{ao_{13}^l.next_{16}^{lb}}(\{\&ao_{13}^1, \&ao_{13}^2, \mathsf{null}\})$ where the rows $\mathcal{T}^S_{ao_{13}^l.next_{16}^{lb}}$ are the formulae $ao_{13}^l.next_{16}^{lb} = \&ao_{13}^1 \vee ao_{13}^l.next_{16}^{lb} = \&ao_{13}^2 \vee ao_{13}^l.next_{16}^{lb} = \mathsf{null}$. These formulae say that the `next` fields of both ao_{13}^1 and ao_{13}^2 may either point to one of the objects themselves or to null. This describes an unbounded linked path in the heap composed of objects abstracted by ao_{13}^1 or ao_{13}^2 and terminated by null.

B. Guarded Shape Templates

In order to use the base shape domain in our approach, we have to augment it with information about the guard variables that encode the program's control flow in the SSA. The guards express when an appropriate loop-back control edge is executed and the loop-back pointer has a defined value[7]. A row of a *guarded shape template* is defined as a formula $\mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}}) \equiv G_{p_i^{lb}} \Rightarrow \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ where $G_{p_i^{lb}}$ is a conjunction of SSA guards associated with the definition of the variable p_i^{lb} and $\mathcal{T}^S_{p_i^{lb}}$ is as in the base shape domain. If $G_{p_i^{lb}}$ is true for a program run, the definition of p_i^{lb} was reached in the run. A shape template $\mathcal{T}^G$ with guards is then a conjunction $\mathcal{T}^G \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}})$.

Let p_i^{lb} be a loop-back pointer abstracting the value of a pointer $p \in Ptr$ coming from the end of a loop $l \in L$. The row guard $G_{p_i^{lb}}$ is a conjunction of the following guards:

- The guard g_j^{lh} linked with the head of the loop l located at program location j, encoding that the loop l is reachable.
- The guard g_i^{ls} linked with the use of p_i^{lb}. The value of g_i^{ls} is true if p_i^{lb} is chosen as the value of the corresponding *phi* variable at the head of l (see Section II-C).
- If p_i^{lb} describes a pointer field of some abstract dynamic object (i.e. it has the form $ao_j^k.f_i^{lb}$ for some $ao_j^k \in AO, f \in Fld$), we also use the guard $g^{ao_j^k}$ linked with the allocation of ao_j^k at program location j. This guard conjoins the guard expressing reachability of program location j with the object-select guards $g_{j,l}^{os}$ and their negations denoting allocation of the k-th materialisation ao_j^k of the object allocated at j.

Example. In Section IV-A, we presented a shape invariant describing the linked segment created by the first loop from Listing 1. The corresponding guards for the two template rows of that invariant are $G_{ao_{13}^1.next_{16}^{lb}} = g_{10} \wedge g_{16}^{ls} \wedge (g_{13} \wedge g_{13}^{os})$ and $G_{ao_{13}^2.next_{16}^{lb}} = g_{10} \wedge g_{16}^{ls} \wedge (g_{13} \wedge \neg g_{13}^{os})$. Here, the loop head guard is g_{10}, the loop-select guard is g_{16}^{ls}, and the allocation guard is given by the guard of the reachability of the allocation site g_{13} and by the appropriate object-select guards (g_{13}^{os} for ao_{13}^1 and $\neg g_{13}^{os}$ for ao_{13}^2, respectively).

C. Shape Domain with Symbolic Loop Paths

Unfortunately, guarded shape templates are not precise enough for many heap-manipulating programs. One often needs to allow the invariant of a loop to be able to distinguish which loops were or were not executed while reaching the given loop. This can, e.g. distinguish which objects were allocated and can hence be processed in the given loop.

---

[6] Note that unlike the previously mentioned point-to relations, this relation is computed not just syntactically but using the considered abstract semantics.

[7] Using the base domain without the guard variables would be sound. However, it would produce very imprecise results since the abstract value would need to cover even states in which the loop-back edge was not taken.

To deal with the above problem, we introduce the concept of *symbolic loop paths* and compute different invariants for different paths. Since we use loop-select guards to express the control flow through the loops (see Section II-C), a symbolic loop path is simply a conjunction of loop-select guards.[8] Let G^{ls} be the set of all loop-select guards of all loops in a program. A symbolic loop path π is then formally defined as $\pi = \bigwedge_{g \in G^{ls}} l_g$ where l_g is a literal of the variable g, i.e. either g or $\neg g$. We use Π to denote the set of all symbolic loop paths of a given program. A *shape template extended with symbolic loop paths* is then given by a formula $\mathcal{T}^L \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}^G_\pi$ where the $\mathcal{T}^G_\pi$ formulae are guarded shape templates as defined in Section IV-B. Here, $\pi_\perp$ a special path containing negative literals only. On that path no loop invariants are computed.

Example. We now show invariants for the pointer p for the second loop of the program in Listing 1. Using our (trace-insensitive) guarded shape domain, the corresponding template row would be $\mathcal{T}^G_{p^{lb}_{22}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\})$. In other words, p would be understood as possibly pointing to ao^1_{13} or ao^2_{13} even on paths where they were not allocated. However, symbolic loop paths allow us to obtain two different invariants depending on the execution of the first loop (for simplicity, we only provide the appropriate template row): namely, $g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\})$ for the case when the body of the first loop is executed and $\neg g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\mathsf{null}\})$ for the case when the body of the first loop is not executed.

D. *Combinations of Domains*

The true power of the template-based verification approach lies in the simplicity of domain combinations. Since templates are general logical formulae, they can be easily composed, forming abstract domains capable of describing more complex properties of programs while relying on the solver to do the heavy-lifting on the combination of the domain operations and the mutual reduction of their abstract values.

1) Power Templates: The definition of shape templates with symbolic loop paths shows one way how a complex template can be formed from a simpler one. In this case, the template parameter, i.e. the abstract value, maps particular symbolic loop paths to sets of parameters of the original shape template. In fact, the shape domain could be replaced by any other abstract domain. The symbolic paths template can hence be viewed as a *power template*—in the sense of power domains [15]—which assigns to each element of the base domain an element of the exponent domain.

2) Product Templates: From the perspective of program analysis, a very interesting possibility is the combination of the shape domain with an abstract domain capable of describing values of variables of non-pointer types, e.g. numerical variables (such as the well-known interval or octagon domains). The simplest way to achieve such a combination is to use a *Cartesian product template* that combines templates of different kinds to be used independently side-by-side. The proposed shape template with loop-back guards $\mathcal{T}^G$ from Section IV-C can be combined with a template for analysis of numerical values $\mathcal{T}^V$ by simply taking their conjunction, i.e. $\mathcal{T}^G \wedge \mathcal{T}^V$. This not only allows us to analyse programs that use pointer and numerical variables simultaneously, but also to reason about the contents of data structures on the heap. We achieve this by analysing numerical fields of abstract dynamic objects using the value part of the template.

In addition, we use this product template as the inner template of the template with symbolic loop paths, forming an even stronger abstract domain: $\mathcal{T}^{LV} \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}^G_\pi \wedge \mathcal{T}^V_\pi$. Using this domain for the running example allows us to analyse the shape and the contents of the linked list at the same time, obtaining the invariants described in Section I that enable us to prove the given property of interest.

V. MEMORY SAFETY ANALYSIS

Apart from checking user-defined assertions, we can also verify memory safety. This includes a number of properties: (1) pointer dereferencing safety, (2) `free` safety, and (3) absence of memory leaks.

A. *Dereferencing a* null *Pointer*

Since our invariants are over-approximating the reachable program states, we can soundly verify *may* (or better called *must-not*) properties. To check dereferences of null, for each expression $*p$ occurring in a program location i, we verify the assertion $p_j \neq \mathsf{null}$ where p_j is the version of p valid at i.

B. `Free` *Safety*

`Free` safety includes the absence of dereferencing a freed pointer and freeing an already freed pointer (a so-called "double free"). To prove absence from these errors, we introduce a new special variable *fr* initialised to null, which is then non-deterministically set to the address of the object to be freed in a `free` call. We replace each call of the form `free(p)` at program location i by a formula $fr_i = g^{fr}_i ? p_j : fr_k$, where p_j and fr_k are the versions of p and *fr*, respectively, valid in i, and g^{fr}_i is a free Boolean variable (a so-called *free guard*). Treating *fr* as a standard pointer-typed variable allows us to over-approximate the set of all freed addresses with the help of our shape domain. Then, in each program location i where either $*p$ or `free(p)` occurs, we can check for the assertion $p_j \neq fr_k$ to prove `free` safety (here, p_j and fr_k are again versions of p and *fr*, respectively, valid at i).

Even though this approach is sound, it is often too imprecise. Freeing one of the concrete objects does not mean that all objects were freed and that it is not safe any more to dereference/free the abstract object. To improve precision, we modify the representation of `malloc` calls. At each allocation site i, we add one more object ao^{co}_i to the set $\{ao^k_i\}$. The object can be chosen as the result of the allocation non-deterministically like any other ao^k_i, but it is guaranteed to be allocated only once (by an additional condition checking that, upon its allocation, no loop-back pointer can point to it). Hence, ao^{co}_i represents a concrete object. Then, for each

---

[8] The notion of symbolic loop paths can be easily generalised to program path sensitivity by including branches of conditional statements too.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

allocation site i, we only allow $\&ao_i^{co}$ to be assigned to fr. The checks for free safety described above are done on concrete objects only, avoiding possible imprecision stemming from dealing with multiple objects represented by a single abstract object which would join the possibly different values of these objects. Also, as ao_i^{co} represents an arbitrary concrete object allocated at i, if safety can be proven for it, it can be assumed to hold for any other object allocated at i.

C. Absence of Memory Leaks

Using fr, we then check whether some ao_i^{co} object may be not freed at the end of the program (if there is a leak, it must be possible to show it on some concrete object). Unfortunately, as we do not track the sequencing of abstract objects representing a set of objects allocated at an allocation site (even when they form a list segment), our analysis typically sees that ao_i^{co} may be skipped in the deallocation loops, and hence remains inconclusive on the memory leaks.

VI. IMPLEMENTATION

We implemented[9] the proposed shape domain within the 2LS framework [35] that uses the template-based verification method described in Section II. We extended the SSA form generated by the framework to handle dynamic memory allocation. 2LS is based on the CPROVER framework [13], which includes an SMT solver based on reduction to propositional logic. We used Glucose 4.0 as the back-end solver in our experiments. We let 2LS inline all functions before running our analysis. For combination with numerical domains described in Section IV-D, we use the template polyhedra domain that is already a part of 2LS. Our approach handles any sequential C program, however, invariants are not inferred for array contents and memory manipulation using pointer arithmetic.

VII. EXPERIMENTS

We performed the experiments to show how our approach improves the performance of 2LS and also how it compares to other state-of-the-art software verification tools.[10] We used BenchExec [4] to run the experiments with time limit set to 900 s and memory limit to 15 GB. The first comparison was done on the subcategories of the SV-COMP benchmarks [36] related to memory safety, particularly *ReachSafety-ControlFlow*, *ReachSafety-Heap*, *MemSafety-Heap*, *MemSafety-LinkedLists*, *MemSafety-Others*. Tasks in *ReachSafety* are checked for reachability of an error condition, tasks in *MemSafety* for absence of invalid pointer dereference, invalid free, and memory leaks. We compared our implementation to the version of 2LS from SV-COMP'17 without the proposed shape analysis.

The results are shown in Table I. The proposed method significantly improves the performance of the tool. Due to missing heap analysis support, the old version of 2LS often reported wrong results and therefore it had a negative score in three subcategories. 2LS with our analysis obtained a positive score in all subcategories and it is also faster in some of them.

Although the results show an improvement, we are still unable to compete with the best tools of SV-COMP'18 in the heap categories. This is mainly because our analysis does not yet support pointer arithmetic and is not yet expressive enough to handle various kinds of trees or nested lists.

However, the main purpose of our work was to extend possibilities of analysing combined shape and value properties of programs. To evaluate, we performed an experiment comparing our tool with the leaders of SV-COMP'18 in the heap-related categories, on tasks combining manipulation of unbounded data structures with a need to reason about the data stored in these structures. All these tasks[11] are correct programs created by our team, since no such programs are part of the SV-COMP benchmarks yet. For each task, we verify that no error state is reachable. The results of the evaluation are shown in Table II. Numbers in the table represent CPU time in seconds needed for the analysis of the example. The value *unknown* means that the tool was not able to analyse the task.

On these benchmarks, 2LS outperforms the other tools significantly. Even tools specialised in shape analysis, *Forester* [17] and *Predator* [16], often report unknown, timeout or even find a false error. This is probably caused by their inability to reason about the data stored in the lists. More general tools such as *Symbiotic* [9] or *Ultimate Automizer* [18] often time out since they probably lack an efficient abstraction for combination of shape and value properties. *CPAChecker* [3] (in the *CPA-Seq* configuration from SV-COMP'18) solved four tasks but times out on the rest.

VIII. RELATED WORK

There is a vast body of work on shape analysis. We can only give an overview of the main lines of research in this section. For a more complete survey, we refer to [25].

Many of the existing approaches to shape analysis are based on abstract interpretation [14], some of them dating back to 1980s [23]. In particular, the TVLA engine [34] came with a flexible approach based on abstract interpretation over a set of user-supplied predicates. In comparison, our approach can be viewed as using a set of parametrised predicates.

Several further approaches based on abstract interpretation and various underlying formalisms (logics, automata, graphs) are mentioned below. In general, our approach differs in that it uses inductive invariant synthesis based on gradually refining parameters of templates via SMT solving on the SSA form (with no iterative execution), instead of iteratively executing the program using abstract transformers and widening until a fixed point is reached. Hence, our approach does not use widening over gradually growing instances of dynamic data structures to capture unbounded sets of instances of such structures. Also, it does not use on-demand materialisation of a concrete memory node from an abstract representation of a set of such nodes followed by again abstracting the resulting

---

[9] Available at https://github.com/diffblue/2ls/releases/tag/2ls-0.7.

[10] All tools, benchmarks, and results are available here: https://pschrammel.bitbucket.io/schrammel-it/research/2ls/fmcad18_exp.tar.xz.

[11] See https://github.com/diffblue/2ls/tree/2ls-0.7/regression/heap-data.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE I: Comparison of 2LS using the proposed method with the previous version of the tool over the SV-COMP benchmark.

	RS-ControlFlow		**RS-Heap**		**MS-Heap**		**MS-LinkedLists**		**MS-Other**	
	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score
2LS	252	64	41	106	17.5	59	107	7	29	46
2LS-old	1400	45	53	-161	190	-194	96	-182	23	46

TABLE II: Comparison of 2LS with other tools on examples combining unbounded data structures and their stored data.

	2LS	**CPA-Seq**	**PredatorHP**	**Forester**	**Symbiotic**	**UAutomizer**
Calendar	2.88	timeout	false	unknown	timeout	timeout
Cart	23.70	timeout	false	unknown	timeout	timeout
Hash Function	3.65	8.51	unknown	unknown	unknown	timeout
MinMax	5.14	timeout	false	unknown	timeout	timeout
Packet Filter	431.00	timeout	timeout	unknown	unknown	timeout
Process Queue	6.62	7.68	timeout	unknown	timeout	timeout
Quick Sort	18.20	3.50	timeout	unknown	unknown	5.75
Running Example	1.24	timeout	timeout	unknown	timeout	unknown
SM1	0.53	timeout	0.31	false	timeout	timeout
SM2	0.55	5.41	false	false	timeout	14.50

memory configuration. These aspects are handled by our encoding into guarded templates and representing `malloc` calls by choosing abstract objects from a predefined pool.

Various extensions of Hoare logic have been developed to cope with heap-manipulating programs. E.g., [22] proposed a way to reason about lists using the Mona tool, which was then extended to more complex data structures [29] and their contents [27]. Another program logic is separation logic [32], which enables reasoning about local memory modifications, rather than looking at the memory as a whole. It has been used for deductive program verification based on user-provided annotations [11]. Fully automated approaches based on separation logic and abstract interpretation have also been proposed and used, e.g., in the Space Invader [37] and SLAyer [2] tools.

More recently, automation of separation logic using SMT solvers by reduction to effectively propositional logic has been proposed by [31], [20], [21]. A different approach [30] uses the Houdini algorithm to find inductive invariants over heap predicates generated from grammars. These works share the common approach with our method to use SMT solvers to reason about heap properties; however, each of them uses different techniques for synthesising the invariant predicates. For an overview on template-based analysis techniques for numerical properties, we refer to [8].

Other fully automated approaches based on abstract interpretation build on shape graphs [26], such as the Predator tool [16], or tree automata and regular tree model checking, such as [6] or the Forester tool [17]. These approaches primarily aim at handling unbounded heap structures. Their combination with reasoning about value properties is not easy as shown in the works [1], [19] that extended Forester with reasoning about finite data and a specialised support for handling ordered list segments. As our experiments showed, Forester and Predator could handle almost none of our examples.

Several further abstract domains have been proposed for combining shape and data domains (e.g. [10], [5]). Our approach has the advantage that such domain combinations need not be designed from scratch.

Beyond the mentioned tools, several participants in SV-COMP, such as CPAChecker [3], Symbiotic [9], Ultimate Automizer [18], or CBMC [13], provide support for dealing with dynamic data structures and their content. However, they cannot handle data structures of unbounded size.

All the above methods are *store-based*, i.e., they describe the heap explicitly by a graph encoded in different ways. Other approaches are inspired by *storeless semantics* [24] using pointer access paths [12], [33], [28], [7] to describe reachability properties on the heap. This idea proved most suitable for our purposes. A *pointer access path* does not concretely express the heap state, it only describes which dynamic objects are reachable from a pointer. Using a set of access paths for each pointer, one can efficiently describe the shape of the heap. Compared with our method, the above approaches, however, use abstract interpretation over CFGs, and their support of dealing with the data content is limited [28].

IX. Conclusions and Future Work

We present a verification approach for heap-manipulating programs based on template-based invariant synthesis. We propose an abstract template domain capable of expressing reachability in dynamic data structures. We show that the domain can easily be combined with other domains to form power and product domains that are able to express complex properties about the shape and the contents of data structures. We experimentally evaluate our approach by within the 2LS framework. We plan to extend the technique to support pointer arithmetic and to develop templates that can express more complex data structure shapes, such as trees, skip-lists, or nested lists. Moreover, we work on using our method to infer function summaries to enable a modular verification approach.

Acknowledgement: The Czech authors were supported by the Czech Science Foundation project 17-12465S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the FIT BUT internal project FIT-S-17-4014. Viktor Malík and Martin Hruška are holders of the Brno Ph.D. Talent Scholarship, funded by the Brno City Municipality.

REFERENCES

[1] Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 8172, pp. 224–239. Springer (2013)

[2] Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory Safety for Systems-Level Code. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 6806, pp. 178–183. Springer (2011)

[3] Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 6086, pp. 184–190. Springer (2011)

[4] Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: SPIN. Lecture Notes in Computer Science, vol. 9232, pp. 160–178. Springer (2015)

[5] Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: Programming Language Design and Implementation. pp. 578–589. ACM (2011)

[6] Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 4134, pp. 52–70. Springer (2006)

[7] Brain, M., David, C., Kroening, D., Schrammel, P.: Model and proof generation for heap-manipulating programs. In: Proceedings of the 23rd European Symposium on Programming. pp. 432–452. Springer (2014)

[8] Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety Verification and Refutation by k-Invariants and k-Induction. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 9291, pp. 145–161. Springer (2015)

[9] Chalupa, M., Vitovská, M., Strejcek, J.: SYMBIOTIC 5: Boosted instrumentation - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 10806, pp. 442–446. Springer (2018)

[10] Chang, B.E., Rival, X.: Relational inductive shape analysis. In: Principles of Programming Languages. pp. 247–260. ACM (2008)

[11] Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming 77(9), 1006–1036 (2012)

[12] Chong, S., , Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Proceedings of the 10th Static Analysis Symposium. pp. 463–482. Springer (2003)

[13] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)

[14] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. pp. 238–252 (1977)

[15] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages. pp. 269–282 (1979)

[16] Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Static Analysis Symposium. pp. 215–237. Springer (2013)

[17] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Computer-Aided Verification. pp. 424–440. Springer (2011)

[18] Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 447–451 (2018)

[19] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Vojnar, T.: Counterexample validation and interpolation-based refinement for forest automata. In: Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 10145, pp. 288–309. Springer (2017)

[20] Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A., Sagiv, M.: Modular reasoning about heap paths via effectively propositional formulas. In: Principles of Programming Languages. pp. 385–396. ACM (2014)

[21] Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 8559, pp. 35–51. Springer (2014)

[22] Jensen, J.L., Jørgensen, M.E., Klarlund, N., Schwartzbach, M.I.: Automatic verification of pointer programs using monadic second-order logic. In: Programming Language Design and Implementation. pp. 226–236. ACM (1997)

[23] Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Principles of Programming Languages. pp. 66–74. ACM (1982)

[24] Jonkers, H.B.M.: Abstract storage structures. In: Algorithmic Languages. pp. 321–343. IFIP (1981)

[25] Kanvar, V., Khedker, U.P.: Heap abstractions for static analysis. ACM Comput. Surv. 49(2), 29:1–29:47 (2016)

[26] Laviron, V., Chang, B.E., Rival, X.: Separating shape graphs. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 6012, pp. 387–406. Springer (2010)

[27] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Principles of Programming Languages. pp. 611–622. ACM (2011)

[28] Matosevic, I., Abdelrahman, T.S.: Efficient bottom-up heap analysis for symbolic path-based data access summaries. In: International Symposium on Code Generation and Optimisation. pp. 252–263. ACM (2012)

[29] Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Programming Language Design and Implementation. pp. 221–231. ACM (2001)

[30] Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Invariant synthesis for incomplete verification engines. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 10805, pp. 232–250. Springer (2018)

[31] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013)

[32] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science. pp. 55–74. IEEE Computer Society (2002)

[33] Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: Proceedings of the 32nd Principles of Programming Languages. pp. 296–309 (2005)

[34] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Principles of Programming Languages. pp. 105–118. ACM (1999)

[35] Schrammel, P., Kroening, D.: 2LS for Program Analysis - (Competition Contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9636, pp. 905–907. Springer (2016)

[36] Software Verification Competition: Benchmarks (2017), https://github.com/sosy-lab/sv-benchmarks/

[37] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 5123, pp. 385–398. Springer (2008)

ISBN: 978-0-9835678-8-2.

Using Loop Bound Analysis For Invariant Generation

Pavel Čadek*, Clemens Danninger*, Moritz Sinn†, Florian Zuleger*

*TU Wien
†St. Pölten UAS

***Abstract*—The problem of loop bound analysis can conceptually be seen as an instance of invariant generation. However, the methods used in loop bound analysis differ considerably from previous invariant generation techniques. Interestingly, there is almost no previous work comparing the two sets of techniques. In this paper, we show that loop bound analysis methods can efficiently produce invariants which are hard to prove for state-of-the-art invariant generation techniques (e.g., polynomial invariants or invariants relating many variables) and thus enrich the tool-set of invariant analysis.**

I. INTRODUCTION

In this paper we aim at connecting two fields of program analysis: invariant generation and loop bound analysis. Specifically, we suggest the use of loop bound analysis techniques for invariant generation. *Invariant generation* is a traditional discipline with a long history. Invariants are program properties (usually given as formulas over program variables) holding on a specific program location in each program run. A special case of interest are *loop invariants* which hold before and after each loop iteration. For example, in the program from Figure 1(a), we have the loop invariant $x + c \le n \ \wedge \ x \ge 0$. *Loop bound analysis* is a younger field, where most of the research was done in the last decade. Its goal is to find an upper bound on the number of iterations of a given loop inside a program. *Reachability bound analysis* [9] generalizes the problem to finding an upper (or lower) bound on the number of executions of a specific part of a program (e.g., a branching inside the loop). For example, in the program from Figure 1(a), n is a bound of the only loop, as well as a reachability bound of basic block B_1. We will use the term *bound analysis* to cover both, loop and reachability bound analysis.

Invariants and loop bounds are linked: 1) Invariants can be used to infer loop bounds. A straight-forward idea is to introduce a counter variable c for the loop of interest and to compute an invariant of shape $c \le \mathit{bound}$. While this idea only works for simple loops, more elaborated approaches have been proposed in the literature [8]. 2) Loop bounds can be used to infer invariants. We are not aware of any publication devoted to this point except for the brief discussions in [18] and [16]. In this paper, we address this gap and also show that invariant generation using loop bound analysis techniques can be more effective than state-of-the-art methods.

We illustrate the use of bound analysis for invariant generation on example (a) in Figure 1. The invariant $c \le n$ holding after the loop is hard to prove for state-of-the-art invariant analysis approaches, because of their need to derive suitable loop invariants. Here, we specifically need the loop invariant $x + c \le n \ \wedge \ x \ge 0$. Its problematic part is the relation $x + c \le n$. Since it does not syntactically appear in the program, it is hard to discover for template-based [15] or predicate-abstraction [12] approaches, because they need to rely on heuristics for template/predicate selection. In contrast, current abstract interpretation based approaches usually fix the expressible invariants in advance: the popular octagon abstract domain [14] cannot express the loop invariant (it can relate at most two variables); the polyhedra domain [4] can express it, but needs to be carefully controlled in order to scale to larger problems.

The central idea of using bound analysis for invariant generation is that variable values after a loop are determined by their values before the loop and the number of times they are increased or decreased inside the loop. In our example, we obtain the equation

$$\mathit{Post}^{\uparrow}(c) = \mathit{Pre}^{\uparrow}(c) + \mathit{Exec}^{\uparrow}(B_1) \cdot 1, \tag{1}$$

where $\mathit{Post}^{\uparrow}(c)$ (resp. $\mathit{Pre}^{\uparrow}(c)$) denotes an upper bound of c after (resp. before) the loop and $\mathit{Exec}^{\uparrow}(B_1)$ denotes an upper bound on the number of executions of the basic block B_1 (containing the instruction `c++`). We note that equation 1 is just a different representation of the postcondition $c \le \mathit{Pre}^{\uparrow}(c) + \mathit{Exec}^{\uparrow}(B_1) \cdot 1$. Hence, in order to prove the postcondition $c \le n$, it suffices to compute $\mathit{Pre}^{\uparrow}(c) = 0$ and $\mathit{Exec}^{\uparrow}(B_1) = n$. $\mathit{Pre}^{\uparrow}(c) = 0$ is determined from the precondition $c = 0$. The computation of $\mathit{Exec}^{\uparrow}(B_1)$ is where loop bound analysis comes into play, because the number of executions of block B_1 is the same as the number of loop iterations. The loop bound is inferred in the following way: Variable x is greater than 0 in the beginning of every loop iteration and it is decremented by 1 in every iteration, which means that the maximal value of x in the beginning (which is n) is an upper bound on the number of iterations. In this way, we get $\mathit{Exec}^{\uparrow}(B_1) = n$.

In this paper we make the following contributions:

1) We introduce a benchmark of challenging invariant generation tasks, which we took from previous invariant and loop bound analysis evaluations. We argue that these tasks are difficult for state-of-the-art invariant analysis techniques.

ISBN: 978-0-9835678-8-2.

	a	b	c	d	e	f	g
CPAchecker	true	unk	true	t/o	unk	unk	unk
Pagai	true	unk	unk	true	unk	unk	unk
Veriabs	true	t/o	true	t/o	t/o	t/o	true
Aligator	succ	succ	fail	fail	fail	fail	fail

TABLE I
Results of the evaluation on the examples from Figure 1.

2) We present the essence of the techniques underlying bound analysis by introducing a few simple concepts. We define the concepts such that they can be easily used for generating invariants and illustrate their usage on the tasks from our benchmark of challenging examples. The concepts are sufficient to solve all benchmark tasks. We believe that the concepts will enrich the tool set of invariant analysis.
3) We provide experimental evaluations on two large benchmark sets. Our first experiment is executed on part of the SV-COMP 2018 benchmark and demonstrates that the current invariant analysis techniques can be significantly improved by means of bound analysis. Our second experiment is executed on a large industrial benchmark, it shows that the class of invariants that can be verified by state-of-the-art invariant analysis tools is to a large extent different from the class of invariants that is found by bound analysis.

II. Challenges for State-of-the-Art Invariant Generation

In this section, we introduce our small benchmark of invariant generation tasks. The tasks are given in Figure 1. They model some of the invariant generation challenges, which we found in SV-COMP [22] - category ”Loops” (tasks (a), (b), (c), (e), (g)) or cBench [21] (tasks (d), (f)). They consist of a precondition, a while-loop written in a simple imperative C-like language, and the postcondition to be proven.

a) Challenges: In order to prove the postcondition, state-of-the-art invariant generation techniques typically need to infer loop invariants (properties holding before and after each iteration of a loop). We present three main challenges of our benchmark tasks [1]:

1) *Polynomial invariants:* Some part of the loop invariant is a polynomial inequation (resp. equation).
2) *Invariants with more than 2 variables:* Some part of the loop invariant is an inequation (resp. equation) relating more than two program variables.
3) *Disjunctive invariants:* The loop invariant requires a case distinction (e.g., $\mathbf{max}\{x, y\}$).

Next to each example in Figure 1, we state the loop invariant needed for proving the postcondition. Note that the loop invariants are often more complex than the postconditions.

b) Experimental Results: We have evaluated several state-of-the-art invariant generation tools on our benchmark of challenging examples. Pagai (git revision 16eed0f) [11] uses abstract interpretation with linear domains (interval, octagon, polyhedra) and path focusing. CPAchecker 1.6.12 combines several analysis in different modes. We used the predicate abstraction mode which worked the best on the benchmark. Veriabs [5], the winner of subcategory ReachSafety-Loops in SV-COMP 2018, abstracts loops by static value analysis with loop acceleration and k-induction and then uses bounded model checking to prove properties. Aligator [13] (git revision eb79fef) is a representative of polynomial loop invariant generation. The technique is built on recurrence equations.

The input C-programs for the tools were generated by introducing ”assume” resp. ”assert” statements representing the pre- resp. post-condition. E.g., for example (a), we generated the statements `assume(0<=m && m<=x && x<=n && c==0)` and `assert(m<=c && c<=n)`. For Aligator, we had to manually rewrite the examples into its input format and as Aligator only generates loop invariants, we could not include the precondition and postcondition.

Table I shows the results: ”unk” stands for an unknown result, ”true” for a successful proof of the assertion, and ”t/o” for timeout - 60 seconds. A special case is Aligator. Because it only generates the loop invariants, we distinguish two cases: ”succ” for successfully generating a loop invariant which is, together with the precondition, sufficient for proving the postcondition, and ”fail” otherwise. The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor using 1.5 GB memory. Regarding the timeouts, the tools did not finish computation even when we extended the limit to 5 minutes, so we consider such cases as failures.

The experimental results support our hypothesis that Figure 1 represents classes of problems which are difficult for state-of-the-art invariant generation techniques. In contrast, in Sections IV and V we will present simple concepts of bound analysis which suffice for analyzing the programs of Figure 1.

III. Basic Definitions

Program representation. Programs in our examples are while-loops without function calls (but with possible nesting) written in C, together with a precondition that holds before the loop. The conditions that we are not able to model or which are non-deterministic (e.g., depending on user input) are represented by the symbol $\star$.

Program Variables and States. By $\mathcal{V}$ we denote the finite set of *program variables* and by $\mathcal{C}$ its subset of *constant variables*, i.e. variables that are never altered in the program. For simplicity, we work only with integers.

A program *state* is a function $\sigma : \mathcal{V} \to \mathbb{Z}$ mapping program variables to their values. We denote the set of states by Σ.

Expressions and Conditions. *Expressions* are terms built with program variables, integers, and functions $+$, $-$, $\cdot$, $/$, $\mathbf{max}$, and $\mathbf{min}$. Division by default rounds down. We denote division with rounding up semantics by wrapping it with the brackets $\lceil . \rceil$. The set of expressions is denoted by Expr. A *constant expression* is an expression that does not contain any non-constant program variable. We denote the set of constant expressions by Expr^c.

---

[1] Although there is a variety of invariant generation techniques tackling with these challenges, they are either computationaly expensive or rely on heuristics. For a lack of space, we omit a detailed discussion about the techniques and their drawbacks.

	precondition	postcondition	loop invariant	concepts
a	$0 \le m \le x \le n \wedge c = 0$	$m \le c \le n$	$x + c \le n \wedge x + c \ge m \wedge x \ge 0$	RF (IV-A) MF (IV-B) VB1 (V-A)
b	$n \ge 0 \wedge m > 0 \wedge x = n \wedge c = 0$	$c = \lceil \frac{n}{m} \rceil$	$x + m \cdot c = n \wedge x \ge 1 - m$	RF (IV-A) MF (IV-B) VB1 (V-A)
c	$0 \le x \le n \wedge c = 0 \wedge y = 3$	$y \le \mathbf{max}\{3, n\}$	$x + c \le n \wedge x \ge 0 \wedge y \le \mathbf{max}\{3, n\}$	RF (IV-A) VB2 (V-B)
d	$n \ge 0 \wedge x = n \wedge c = 0 \wedge r = 0$	$c \le n$	$c + r + x \le n \wedge r \ge 0 \wedge x \ge 0$	LRF (IV-C) VB1 (V-A)
e	$0 \le x \le n \wedge m \ge 0 \wedge y = c = 0$	$c \le m \cdot n$	$c \le m \cdot (n - x) \wedge x \ge 0$	LRF (IV-C) VB1 (V-A)
f	$n \ge 0 \wedge r = y = n \wedge c = x = 0$	$c \le 2n$	$c + \mathbf{max}\{r, x\} + y \le 2n \wedge y \ge 0 \wedge x \ge 0$	LRF (IV-C) VB1 (V-A)
g	$x \ge 0 \wedge j = 0 \wedge i = 0$	$j \le \frac{(x-1) \cdot x}{2}$	$j \le \frac{(i-1) \cdot i}{2} \wedge i \le x$	RF (IV-A) VBRE (V-C)

a

```
    while(x>0){
B1:   x--; c++;
    }
```

b

```
    while(x>0){
B1:   x=x-m; c++;
    }
```

c

```
    while(x>0){
B1:   x--; c++;
      if(*)
B2:      y=c;
    }
```

d

```
    while(x>0){
B1:   x--;
      if(*)
B2:      r++;
      else
         while(r>0){
B3:         r--; c++;}
    }
```

e

```
    while(x>0){
B1:   x--; y=m;
      while(y>0){
B2:      y--; c++;
      }
    }
```

f

```
    while(y>0){
B1:   x=r;
      while(y>0 && *){
B2:      x++; y--; }
      while(x>0 && *){
B3:      x--; r--; c++;
      }
B4:   y--;
    }
```

g

```
    while(i<x){
B1:   i++; j=j+i;
    }
```

Fig. 1. Our small invariant generation benchmark. Each task consists of a precondition, a while-loop written in C, and the postcondition to be proven. The symbol $\star$ is used to abstract from some conditions in the programs, it represents a non-deterministic boolean value (e.g., dependent on a user input). Each label B_i denotes the basic block (sequence of assignments) associated with the respective line. For each program, we also state the loop invariant needed for state-of-the-art invariant generation techniques to prove the postcondition. The last column states combinations of concepts from Sections IV and V which are sufficient to prove the postcondition.

For expressions $e_1, e_2 \in \mathit{Expr}$ and a variable v, $e_1[v/e_2]$ is the expression e_1 where all occurrences of v are simultaneously replaced by e_2. Further, $e[v_i/e_i \mid i \in I]$ denotes multiple simultaneous replacements.

We can now extend the notion of a program state to whole expressions. For an expression $e \in \mathit{Expr}$ and a state $\sigma \in \Sigma$, we define $\sigma(e) = e[x/\sigma(x) \mid x \in \mathcal{V}]$. We say that $\sigma(e)$ is the *value of* e *in state* σ.

Conditions (except the non-deterministic condition $\star$) are formulas built from expressions and classical relational and logical operators. The set of conditions is denoted by Cond with Init being the precondition. We extend the concept of a simultaneous replacement from expressions to conditions. We say that a state σ *satisfies a condition* γ (denoted by $\sigma \models \gamma$) if $\gamma[x/\sigma(x) \mid x \in \mathcal{V}]$ is a tautology.

Basic Blocks. A program part consisting only of assignments is called a *basic block*. We denote the set of basic blocks in a program by $\mathcal{B}$. We assume a special *initial basic block* $B_b \in \mathcal{B}$ and *final basic block* $B_e \in \mathcal{B}$, which both consist of zero assignments. We also assume that each block either has

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

a single successor or exactly two successors connected by a branching condition.

We further require that B_b does not have any predecessor and that B_e does not have any successor. In our examples, each basic block is given as one line of code. For example, in the program from Figure 1(a), the basic block B_1 consists of two assignments, `x--` and `c++`. The blocks B_b and B_e are not explicitly marked in our examples.

Program semantics. The *effect of a basic block* is a function $\mathcal{E} : \mathcal{B} \to (\mathcal{V} \to \mathit{Expr})$ such that whenever a block B is executed in a program state σ resulting in a state σ', it holds that $\sigma'(v) = \sigma(\mathcal{E}(B)(v))$.

E.g., if $\mathcal{V} = \{x, y, c\}$, $\mathcal{C} = \{c\}$, and $B =$ `x++; y+=x;`, then $\mathcal{E}(B)(x) = x{+}1$, $\mathcal{E}(B)(y) = y{+}x{+}1$, and $\mathcal{E}(B)(c) = c$.

We assume that for each constant variable $c \in \mathcal{C}$, $\mathcal{E}(B)(c) = c$, i.e., constant variables never change their value.

We extend the effect of a basic block to expressions as follows: For $e \in \mathit{Expr}$, $\mathcal{E}(B)(e) = e[x/\mathcal{E}(B)(x) \mid x \in \mathcal{V}]$

A *program run* is a (possibly infinite) sequence $(\sigma_0, B_0), (\sigma_1, B_1), \ldots$, where each $(\sigma_i, B_i) \in \Sigma \times \mathcal{B}$, $B_0 = B_b$, $\sigma_0 \models \mathit{Init}$, each B_{i+1} is a successor of B_i, $\sigma_i \models \gamma$ for any branching condition γ between B_i and B_{i+1}, and for all $v \in \mathcal{V}$ we have $\sigma_{i+1}(v) = \sigma_i(\mathcal{E}(B_i)(v))$. Further, if the program run is finite with B_n being the last basic block, then $B_n = B_e$.

Execution bounds. Given a program run ρ, $\#(B, \rho)$ denotes the number of occurrences of the basic block B in ρ.

An *upper (resp. lower) execution bound for a basic block* B is a constant expression $b \in \mathit{Expr}^c$ such that for each program run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$, $\#(B, \rho) \le \sigma_0(b)$ (resp. $\#(B, \rho) \ge \sigma_0(b)$).

An execution upper (resp. lower) bound mapping is a function $\mathit{Exec}^\uparrow : \mathcal{B} \to \mathit{Expr}^c$ (resp. $\mathit{Exec}^\downarrow : \mathcal{B} \to \mathit{Expr}^c$) that maps an upper (resp. lower) bound to each basic block in the program. E.g., $\mathit{Exec}^\downarrow(B_1) = m$ and $\mathit{Exec}^\uparrow(B_1) = n$ in Figure 1(a). [2]

Invariants and expression/variable bounds. Let $B \in \mathcal{B}$ be a basic block. We say a condition $\gamma \in \mathit{Cond}$ is an *invariant before* B if for each program run $(\sigma_0, B_0), (\sigma_1, B_1) \ldots$ holds that if $B_i = B$ then $\sigma_i \models \gamma$. For example, $x > 0$ is an invariant before B_1 in Figure 1(a).

A *precondition* (resp. *postcondition*) is an invariant before B_b (resp. B_e). We use the term *universal invariant* to denote the invariant holding before each $B \in \mathcal{B}$.

An *initial, resp. final, resp. universal upper bound of an expression* e is a *constant* expression $b \in \mathit{Expr}^c$ such that $b \ge e$ is a precondition, resp. postcondition, resp. universal invariant.

An *initial, resp. final, resp. universal upper bound mapping* is a partial function $\mathit{Pre}^\uparrow$, resp. $\mathit{Post}^\uparrow$, resp. $\mathit{Univ}^\uparrow$ that maps an initial, resp. final, resp. universal upper bound to each expression.

We define the initial, resp. final, resp. universal *lower* bound analogically with the upper bounds (only the invariant is $b \le e$ instead of $b \ge e$) and the bound mappings are denoted as $\mathit{Pre}^\downarrow$, $\mathit{Post}^\downarrow$, and $\mathit{Univ}^\downarrow$.

For example, in Figure 1(a), we have $\mathit{Pre}^\downarrow(x) = m, \mathit{Pre}^\uparrow(x) = n, \mathit{Univ}^\downarrow(x) = 0, \mathit{Univ}^\uparrow(x) = n$, and $\mathit{Post}^\downarrow(x) = \mathit{Post}^\uparrow(x) = 0$.

For all the previous definitions, we use the term *variable bound* in case e is a variable.

Using Loop Bound Analysis For Invariant Generation. In this paper, we extract initial expression bounds from the precondition and use them to infer execution bounds. Based on the execution bounds and initial expression bounds, we compute universal and final variable bounds.

Note that computing final and universal *expression* bounds can usually be decomposed to several *variable* bound computations (whether we take an upper or a lower bound of each variable depends on the sign with which it appears in the expression). For example, we may compute $\mathit{Post}^\uparrow(2{\cdot}x{-}y{+}3)$ as $2 \cdot \mathit{Post}^\uparrow(x) - \mathit{Post}^\downarrow(y) + 3$.

We will describe concepts for execution bound computation in Section IV and concepts for variable bound generation in Section V.

IV. Computation of Execution Bounds

A. Ranking Functions

A lot of techniques for loop bound analysis use the concept of ranking functions (e.g., [3], [17], [18], [2]). Ranking functions are expressions that keep decreasing during an execution of a program and they are bounded from below, thereby proving that the program must eventually terminate.

Look at the program in Figure 1(a): x is a ranking function of block B_1, because (1) it is decreased by 1 with each execution of B_1, (2) it is never increased, and (3) it is bounded from below by 0. Note that in this way it measures the number of the remaining executions of B_1.

Definition 1: Let $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ be a program run and $e \in \mathit{Expr}$ an expression. We define $\triangledown(e, \rho) = |\{i \mid \sigma_i(e) > \sigma_{i+1}(e)\}|$, so $\triangledown(e, \rho)$ denotes the number of times e is decreased on ρ.

Definition 2: An expression e is called a *ranking function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \le \triangledown(e, \rho)$ (to each execution of B, we can assign at least one decrement of e)
2) $\forall i \ge 0.\ \sigma_i \models e \ge 0$ (e is bounded from below by 0)
3) $\forall i \ge 0.\ \sigma_i(e) \ge \sigma_{i+1}(e)$ (e is never increased)

The right ranking function can be found by simple heuristics. E.g., in [18], the expression e is a candidate if $e > 0$ appears in the looping condition.

---

[2] We note that the upper and lower bound mappings are not unique. We can infer different mappings depending on the used concept or ranking (resp. metering) function (see next section). However, for technical convenience and better readability we always use the same name $\mathit{Exec}^\uparrow$ and $\mathit{Exec}^\downarrow$ for the mappings.

Concept RF. Let B be a basic block and e its ranking function. Then $\mathit{Exec}^{\uparrow}(B) = \mathit{Pre}^{\uparrow}(e)$ is an upper execution bound for B.

Example 1: As we already mentioned, x is a ranking function of B_1 in Figure 1(a), so we get $\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x) = n$ by Concept RF. We summarize the results for all the examples in the following table:

	ranking fnct	execution bounds
a	$B_1 :\ x$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x) = n$
b	$B_1 :\ \lceil \frac{x}{m} \rceil$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(\lceil \frac{x}{m} \rceil) = \lceil \frac{n}{m} \rceil$
c	$B_1, B_2 :\ x$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(x) = n$
d	$B_1, B_2 :\ x$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(x) = n$
e	$B_1 :\ x$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x) = n$
f	$B_1, B_2,$ $B_4 :\ y$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Exec}^{\uparrow}(B_2) = \mathit{Exec}^{\uparrow}(B_4)$ $= \mathit{Pre}^{\uparrow}(y) = n$
g	$B_1 :\ x - i$	$\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x - i) = x$

Note that the expression representing the ranking function of some basic block does not have to decrease by executing the block itself. E.g., in Figure 1(d), x is a ranking function for block B_2 because each execution of B_2 is preceded by an execution of B_1 which decreases x. To find the ranking function of some basic block, it usually suffices to analyse all possible paths from the block back to itself and find expressions that decrease on these paths. Also note that the ranking function does not have to be just one variable (as can be seen in examples (b) and (g)).

We also want to remark that if we delete the initial condition $0 \leq n$ in Figure 1(a) (respectively in the other examples), the analysis does not fail. We would infer the ranking function $\mathbf{max}\{x, 0\}$. The conditions on non-negativity are added to the examples only for simplicity.

B. Metering Functions

Because the research in bound analysis so far focused mainly on computing upper execution bounds, there is no widely adopted term analogical to "ranking function" for computing lower execution bounds. For our paper, we have chosen to adapt (and redefine) the term *metering function* used in [7].

Definition 3: An expression e is called a *metering function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \geq \nabla(e, \rho)$ (to each decrement of e, we can assign at least one execution of B)
2) $\exists j.\ \sigma_j \models e \leq 0$ (e is eventually non-positive)
3) $\forall i \geq 0.\ \sigma_i(e) \leq \sigma_{i+1}(e) + 1$ (e is never decreased by more than 1 on one block)

Conditions (2) and (3) guarantee that the number of decrements of e is greater or equal to its lowest possible initial value. Because of condition (1), also the number of executions of B is greater or equal to e's lowest possible initial value. Note that the requirement that e is eventually less or equal to zero can be checked by a simple analysis. Usually, it follows from the negated looping condition of the loop. The candidates for metering functions can be chosen by the same heuristics as in the case of ranking functions.

Concept MF. Let B be a basic block and e its metering function. Then $\mathit{Exec}^{\downarrow}(B) = \mathit{Pre}^{\downarrow}(e)$ is a lower execution bound for B.

Example 2: As in the previous subsection, we summarise the metering functions and lower execution bounds in a table:

	metering function	execution bounds
a	$B_1 :\ x$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(x) = m$
b	$B_1 :\ \lceil \frac{x}{m} \rceil$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(\lceil \frac{x}{m} \rceil)$ $= \lceil \frac{n}{m} \rceil$
c	$B_1 :\ x$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(x) = n$
d	$B_1 :\ x$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(x) = n$
e	$B_1 :\ x$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(x) = 0$
f	$B_1, B_4 :\ y$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Exec}^{\downarrow}(B_4)$ $= \mathit{Pre}^{\downarrow}(y) = n$
g	$B_1 :\ x - i$	$\mathit{Exec}^{\downarrow}(B_1) = \mathit{Pre}^{\downarrow}(x - i) = 0$

Note that for block B_2 in example (c), x is a ranking function, but not a metering function. For all basic blocks that are not mentioned in the table, we may use the implicit metering function 0, which always satisfies all the conditions of a metering function and leads to the trivial lower execution bound 0.

C. Lexicographic Ranking Functions

We will now extend the concept of a ranking function. Let us look at example (d) in Figure 1: r would be a ranking function of B_3 if it was not incremented on B_2. However, we may notice that B_2 itself has the ranking function x, so B_2 is executed at most $\mathit{Pre}^{\uparrow}(x) = n$ times (by Concept RF) and thus r is increased altogether by at most n (by 1 with each execution of B_2). Hence B_3 cannot be executed more than $\mathit{Pre}^{\uparrow}(r) + n = n$ times. We will call r a local ranking function of B_3.

Definition 4: The expression e is called a *local ranking function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \leq \nabla(e, \rho)$
2) $\forall i \geq 0.\ \sigma_i \models e \geq 0$

Note that the only difference to Definition 2 is that a local ranking function may increase during a program run (condition (3) is missing).

Definition 5: Let $B_1, \ldots, B_n$ be basic blocks with local ranking functions $e_1, \ldots, e_n$ such that for each $1 \leq i < j \leq n$ it holds that $\mathcal{E}(B_j)(e_i) \leq e_i$ (i.e., we can order the basic blocks such that an execution of any of them does not increase local ranking functions of the blocks that are higher in the order). Then we say that $(e_1, \ldots, e_n)$ is a *lexicographic ranking function of blocks* $(B_1, \ldots, B_n)$.

Concept LRF. Let $(e_1, \ldots, e_n)$ be a lexicographic ranking function of blocks $(B_1, \ldots, B_n)$. Let $c_{i,j} \in \mathit{Expr}^c$ denote the maximal value by which one execution of B_i can increase e_j, i.e. $\mathcal{E}(B_i)(e_j) \leq e_j + c_{i,j}$. Then we set:

$$\mathit{Exec}^{\uparrow}(B_j) = \mathit{Pre}^{\uparrow}(e_j) + \sum_{k=1}^{j-1} \mathit{Exec}^{\uparrow}(B_k) \cdot c_{k,j}$$

More details about lexicographic ranking functions can be found in [17], [3], and [2]. For the computation of lower bounds, nothing like lexicographic metering functions has been published so far. It is possible to define such functions, but we omit the definition here for lack of space.

Example 3: In example (d) from Figure 1, we have the lexicographic ranking function (x, r) of blocks (B_2, B_3). We also have $c_{1,2} = 1$ (otherwise $c_{i,j} = 0$). Hence by applying Concept LRF, we get $\mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(x) = n$ and $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(r) + 1 \cdot \mathit{Exec}^{\uparrow}(B_2) = 0 + 1 \cdot n = n$.

On this example, we can see how lexicographic ranking functions can handle amortized complexity problems. Even though there are n iterations of the outer loop and B_3 can be executed up to $n - 1$ times during one iteration, it cannot be altogether executed more than n times.

For example (e), we infer the lexicographic ranking function (x, y) of blocks (B_1, B_2). We now need an auxiliary invariant $y \geq 0$ holding before B_1 (which is easily found by a simple invariant generator) to infer that the upper ranking function y of B_2 is increased by at most m with each execution of B_1. By applying Concept LRF, we get $\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x) = n$ and $\mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(y) + m \cdot \mathit{Exec}^{\uparrow}(B_1) = 0 + m \cdot n = m \cdot n$.

On example (f), we can see that the choice of the local ranking functions can have impact on the obtained bounds. When choosing x as a local ranking function of B_3, we get the lexicographic ranking function (y, y, x) of blocks (B_1, B_2, B_3) with $c_{1,3} = n$, $c_{2,3} = 1$, and $c_{1,2} = 0$. Thus $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(x) + n \cdot \mathit{Exec}^{\uparrow}(B_1) + 1 \cdot \mathit{Exec}^{\uparrow}(B_2) = 0 + n \cdot n + 1 \cdot n = n^2 + n$. However, this bound is unnecessarily coarse. When choosing $\mathbf{max}\{x, r\}$ as a local ranking function of B_3[3], the reset `x=r` on B_1 does not have any effect on the local ranking function and we get $c_{1,3} = 0$ and thus $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(\mathbf{max}\{x, r\}) + 0 \cdot \mathit{Exec}^{\uparrow}(B_1) + n \cdot \mathit{Exec}^{\uparrow}(B_2) = n + 0 + n = 2n$.

V. COMPUTATION OF VARIABLE BOUNDS

A. A Simple Variable Bound Computation

The simplest approach to variable bound computation was already suggested in the introduction and it follows from [3] and [17]. For a variable that is changed only at one basic block where it is incremented by a constant, we compute the final variable upper bound by multiplying the constant by the maximal number of executions of the block and adding the maximal initial value of the variable.

We now generalise this idea for a computation of lower variable bounds and we involve variables that are not only incremented, but also decremented on possibly more than one basic block. For each variable v, we define a set of tuples (B, d) where B is a basic block and d is the amount by which B increments or decrements v.

Definition 6: Let $v \in \mathcal{V}$. We define the increments and decrements of v in the following way:

$$\begin{array}{lcl} I(v) & = & \{(B,d) \in \mathcal{B} \times \mathit{Expr}^c \mid \mathcal{E}(B)(v) = v + d \wedge \\ & & \;\mathit{Init} \implies d > 0\} \\ D(v) & = & \{(B,d) \in \mathcal{B} \times \mathit{Expr}^c \mid \mathcal{E}(B)(v) = v - d \wedge \\ & & \;\mathit{Init} \implies d > 0\} \end{array}$$

In the following simple concept, we require that the variable, for which we are computing the bound, cannot be changed in any other way than incrementing and decrementing it by a constant. Note that if there is an assignment incrementing or decrementing the variable by a non-constant expression, we can replace the non-constant expression by its universal upper or lower bound (depending on whether we compute an upper or lower variable bound).

Concept VB1. Let $v \in \mathcal{V}$ and for each basic block B which does not appear in $I(v)$ or $D(v)$ it holds that $\mathcal{E}(B)(v) = v$ (i.e., it does not change v). Then we set

$$\begin{array}{rl} \mathit{Post}^{\uparrow}(v) = & \mathit{Pre}^{\uparrow}(v) + \sum_{(B,d) \in I(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ & \;- \sum_{(B,d) \in D(v)} \mathit{Exec}^{\downarrow}(B) \cdot d \\ \mathit{Post}^{\downarrow}(v) = & \mathit{Pre}^{\downarrow}(v) - \sum_{(B,d) \in D(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ & \;+ \sum_{(B,d) \in I(v)} \mathit{Exec}^{\downarrow}(B) \cdot d \\ \mathit{Univ}^{\uparrow}(v) = & \mathit{Pre}^{\uparrow}(v) + \sum_{(B,d) \in I(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ \mathit{Univ}^{\downarrow}(v) = & \mathit{Pre}^{\downarrow}(v) - \sum_{(B,d) \in D(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \end{array}$$

Example 4: For example (a) in Figure 1, we have $I(c) = \{(B_1, 1)\}$ and $D(c) = \emptyset$. Thus $\mathit{Post}^{\uparrow}(c) = \mathit{Pre}^{\uparrow}(c) + \mathit{Exec}^{\uparrow}(B_1) \cdot 1 = n$ and $\mathit{Post}^{\downarrow}(c) = \mathit{Pre}^{\downarrow}(c) + \mathit{Exec}^{\downarrow}(B_1) = m$. We may apply the same computation for examples (b), (d), (e), and (f) and use the execution bounds computed in the previous subsections.

We now demonstrate the computation of universal upper and lower bounds for variable r in example (f). We have $I(r) = \emptyset$ and $D(r) = \{(B_3, 1)\}$. We have computed $\mathit{Exec}^{\uparrow}(B_3) = 2n$ in the previous subsection and thus we get $\mathit{Univ}^{\uparrow}(r) = \mathit{Pre}^{\uparrow}(r) = n$ and $\mathit{Univ}^{\downarrow}(r) = \mathit{Pre}^{\downarrow}(r) - \mathit{Exec}^{\uparrow}(B_3) \cdot 1 = n - 2n = -n$.

B. Variable Bound Computation with Resets

We will now extend the previous simple variable bound concept such that it covers also basic blocks which *reset* the given variable, i.e. which set it to a new value independent of its previous value. Assume that we want to compute the final upper variable bound for a variable v. We do not analyse the order in which the blocks are executed, but we can safely

---

[3] Both x and r decrease on B_3 so $\mathbf{max}\{x, r\}$ is decremented by 1 with each execution of B_3 and x is always non-negative so also $\mathbf{max}\{x, r\}$ is non-negative.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

assume that all decrements of v happen before any reset (so they would not have any effect on the final value), then v is reset to the biggest possible value and afterwards incremented the maximum number of times. We proceed analogically with the lower bound. The concept is partially inspired by [18].

As in the previous subsection, we first define the set of resets for a given variable. For simplicity, we consider only resets of the form `v=a+expr`, where a is a variable and $expr$ is a constant expression.

Definition 7: Let $v \in \mathcal{V}$. We define the resets of v in the following way:

$$R(v) \quad = \quad \{(B, a, d) \in \mathcal{B} \times \mathcal{V} \times \mathit{Expr}^c \mid \mathcal{E}(B)(v) = a + d\}$$

To formulate the concept, we will yet need two auxiliary sets which contain the largest (resp. smallest) values (given as constant expressions) to which a given variable may be reset. For that purpose, we consider the initial value of a variable also as a reset.

Definition 8:

$$R_c^{\uparrow}(v) = \{\mathit{Pre}^{\uparrow}(v)\} \cup \textstyle\bigcup_{(B,a,d)\in R(v)} \{\mathit{Univ}^{\uparrow}(a) + d\}$$
$$R_c^{\downarrow}(v) = \{\mathit{Pre}^{\downarrow}(v)\} \cup \textstyle\bigcup_{(B,a,d)\in R(v)} \{\mathit{Univ}^{\downarrow}(a) + d\}$$

Now we can formulate the concept. Note that (as discussed earlier) decrements have no effect on the upper variable bound and increments have no effect on the lower variable bound. Thus, there is also no difference between final and universal variable bounds here.

Concept VB2. Let $v \in \mathcal{V}$ and for each basic block B that does not appear in $I(v)$, $D(v)$, or $R(v)$ it holds that $\mathcal{E}(B)(v) = v$ (i.e., it does not change v). Then we set

$$\mathit{Post}^{\uparrow}(v) = \mathit{Univ}^{\uparrow}(v) = \begin{array}{l}\mathbf{max}(R_c^{\uparrow}(v)) \\ \quad + \sum_{(B,d)\in I(v)} \mathit{Exec}^{\uparrow}(B) \cdot d\end{array}$$
$$\mathit{Post}^{\downarrow}(v) = \mathit{Univ}^{\downarrow}(v) = \begin{array}{l}\mathbf{min}(R_c^{\downarrow}(v)) \\ \quad - \sum_{(B,d)\in D(v)} \mathit{Exec}^{\uparrow}(B) \cdot d\end{array}$$

Example 5: In example (c) from Figure 1, we have $R(y) = \{(B_2, c, 0)\}, D(x) = \{(B_1, 1)\}, I(c) = \{(B_1, 1)\}$ and $I(y) = D(y) = I(x) = R(x) = D(c) = R(c) = \emptyset$. By Concept VB1, $\mathit{Univ}^{\uparrow}(c) = \mathit{Pre}^{\uparrow}(c) + \mathit{Exec}^{\uparrow}(B_1) = 0 + n = n$, and thus we can compute $R_c^{\uparrow}(y) = \{\mathit{Pre}^{\uparrow}(y), \mathit{Univ}^{\uparrow}(c) + 0\} = \{3, n\}$, and $\mathit{Post}^{\uparrow}(y) = \mathbf{max}\{3, n\}$ by Concept VB2.

C. Variable Bounds by Recurrence Equations

An alternative approach to Concept VB1 and Concept VB2 for variable bound computation is based on recurrence equations. It is similar to the technique [13] used by ALIGATOR.

If we have a non-nested loop, we can express variable values as functions over the loop counter (which represents the number of finished iterations). For example, the fact that a variable v is incremented by 1 in each iteration of a loop can be represented by a recurrence equation $[v](n) = [v](n-1)+1$ where $[v](n)$ denotes the value of v after n iterations (n is here the loop counter).

In comparison with [13], our generated invariants are inequalities instead of equalities and we incorporate the execution bounds, and thus take into account conditions in the loop. The advantage of using recurrence equations over the concepts VB1 and VB2 is that we can achieve more precise bounds (as demonstrated next). The disadvantage is that they are less general - in the way they are defined here, we may apply them only on non-nested loops without branching. However, the concept can be further extended to multi-path or nested loops as in [19].

Definition 9: For $v \in \mathcal{V}$, we introduce the functions $[v]^{\uparrow} : \mathbb{N} \rightarrow \mathit{Expr}^c$ and $[v]^{\downarrow} : \mathbb{N} \rightarrow \mathit{Expr}^c$ such that $[v]^{\downarrow}(n) \leq v$ and $v \leq [v]^{\uparrow}(n)$ holds after n iterations[4] of the main program loop.

For $n = 0$, we set $[v]^{\uparrow}(n) = \mathit{Pre}^{\uparrow}(v)$ and $[v]^{\downarrow}(n) = \mathit{Pre}^{\downarrow}(v)$. For $n > 0$, we define $[v]^{\uparrow}(n)$ and $[v]^{\downarrow}(n)$ recursively with the use of $[v]^{\uparrow}(n-1)$ and $[v]^{\downarrow}(n-1)$ by analysing the effect of one iteration (the biggest possible increase or decrease of v). Then we can infer the closed form solution from the recursive definitions by a syntactic pattern matching to the following well known case[5]:

$$f(n) = f(n-1)+c+d \cdot n \;\rightsquigarrow\; f(n) = f(0)+c \cdot n+d \cdot \frac{n \cdot (n+1)}{2}$$

where $n \in \mathbb{N}$ and $c, d \in \mathit{Expr}^c$.

Definition 10: We say a function $f : \mathbb{N} \rightarrow \mathit{Expr}^c$ is *increasing (resp. decreasing)* if for each $n \in \mathbb{N}$, $f(n) \leq f(n+1)$ (resp.$f(n) \geq f(n+1)$).

Concept VBRE. Let B be a basic block located immediately after the loop header. Let v be a variable for which we know the closed form of $[v]^{\uparrow}(n)$ (resp. $[v]^{\downarrow}(n)$). Then

$$\mathit{Post}^{\uparrow}(v) = \begin{cases} [v]^{\uparrow}(\mathit{Exec}^{\uparrow}(B)) & \text{if } [v]^{\uparrow}(n) \text{ is increasing;} \\ [v]^{\uparrow}(\mathit{Exec}^{\downarrow}(B)) & \text{if } [v]^{\uparrow}(n) \text{ is decreasing.} \end{cases}$$

Analogically for the lower expression bound:

$$\mathit{Post}^{\downarrow}(v) = \begin{cases} [v]^{\downarrow}\langle \mathit{Exec}^{\downarrow}(B)\rangle & \text{if } [v]^{\downarrow}(n) \text{ is increasing;} \\ [v]^{\downarrow}\langle \mathit{Exec}^{\uparrow}(B)\rangle & \text{if } [v]^{\downarrow}(n) \text{ is decreasing.} \end{cases}$$

Example 6: In Figure 1(g), we have $[i]^{\uparrow}(n) = [i]^{\uparrow}(n-1)+1$, hence $[i]^{\uparrow}(n) = [i]^{\uparrow}(0) + n = n$. $[j]^{\uparrow}(n) = [j]^{\uparrow}(n-1) + [i]^{\uparrow}(n-1) = [j]^{\uparrow}(n-1) + n - 1 = [j]^{\uparrow}(0) + (-1) \cdot n + 1 \cdot \frac{n \cdot (n+1)}{2} = \frac{n \cdot (n-1)}{2}$. By Concept RF, we have already inferred $\mathit{Exec}^{\uparrow}(B_1) = x$ in Subsection IV-A. Because $\frac{n \cdot (n-1)}{2}$ is increasing (the loop counter n is non-negative), we get $\mathit{Post}^{\uparrow}(j) = \frac{x \cdot (x-1)}{2}$ by replacing the counter with the upper execution bound.

Note that for the computation of upper variable bound for j with Concept VB1, we would have to replace the assignment `j=j+i` (incrementing j by a non-constant expression) by the

---

[4] We leave the notion of a loop iteration to the reader's intuition.

[5] Closed form computation of other types of recurrences is discussed, e.g., in [13].

assignment `j=j+x` (incrementing j by a constant expression). Thus, we would get $I(j) = \{(B_1, x)\}$ and $Post^{\uparrow}(j) = 0 + x \cdot Exec^{\uparrow}(B_1) = x \cdot x$ which is less precise than $\frac{x\cdot(x-1)}{2}$.

VI. Experimental Evaluation on Tasks From SV-COMP

We implemented the presented concepts into the tool Looperman [19]. We set up the following experiment on base of the SV-COMP 2018 benchmark in order to evaluate the contribution that bound analysis can make to solving invariant analysis challenges: We inserted the invariants that Looperman computes based on concepts RF, MF, and VB1 in form of "assume" statements into the benchmarks from SV-COMP's ReachSafety-Loops category. Specifically the invariants are added after the loop and immediately after the loop header, where they relate the current variable values to the values before the loop. For example, the code from Figure 1(a) looks as follows after applying the described pre-processing:

```
x_0 = x;
c_0 = c;
while(x>0){
  assume(0<x && x<=x_0);
  assume(0<=c && c<x_0);
  x--;
  c++;
}
assume(x==x_0-max(x_0,0));
assume(c==c_0+max(x_0,0));
```

We excluded the false-unreach cases (those with an invalid assertion) from the benchmark as we aim at proving program properties, not at finding counter-examples, which left us with 111 files with valid assertions. We ran the tools Veriabs, CPAchecker and Pagai on the 111 files with valid assertions that we enriched by our invariants, as well as on the original 111 files. We did not run Aligator because its inputs are restricted to a special format. For the evaluation, we used the same time limit of 900s as in SV-COMP. The files with generated invariants, Looperman, as well as a detailed table of results, is available at [1].

Table II compares the results the respective tool obtains with the help of the invariants inferred by Looperman (column 2) and without these invariants (column 1). CPAchecker was able to validate 16 (14.4%) additional assertions with the help of the invariants. Pagai improved its results by 9 cases (8.1%). Given that Veriabs already proves 103 of 111 assertions, it is hard to further improve its results, and in our experiment it did not not benefit from the additional "assume" statements. However, considering the results of our third experiment (see below), it seems that CPAchecker and Pagai are more reliable on real world code than Veriabs.

When comparing to other tools from SV-COMP 2018 on this set of programs, CPAchecker in predicate analysis mode would move from the 5th place to the 2nd place by using our additional invariants, preceded only by Veriabs with 103 proven files, and followed by UTaipan [6] with 82 proven files and UAutomizer [10] with 78 files.

	proven true (without invariants)	proven true (with invariants)
CPAchecker	68 (61.26%)	84 (75.68%)
Pagai	57 (51.35%)	66 (59.46%)
Veriabs	103 (92.79%)	103 (92.79%)

TABLE II
Results of the evaluation on 111 true-unreach programs of ReachSafety-Loops category from SV-COMP 2018 with and without our generated invariants.

	fail	oom	timeout	unknown	false	true	true (%)
CPAchecker	98	0	187	165	0	311	40.87%
Pagai	0	8	36	342	0	375	49.2%
Veriabs	183	0	265	262	10	41	5.4%
CBMC	0	8	474	0	0	279	36.66%

TABLE III
Results of the evaluation on all 761 loops in cBench for which Loopus inferred a bound over the stack variables.

We conclude that bound analysis techniques can considerably improve state-of-the-art approaches to invariant analysis.

VII. Experimental Evaluation on an Industrial Benchmark

By our third experiment on an industrial benchmark we evaluate to which extent invariant analysis tools can solve bound analysis problems. For this purpose we ran our bound analysis tool Loopus on the program and compiler optimisation benchmark *Collective Benchmark* [21] (cBench, 1027 different C files with 211.892 lines of code) and annotated the inferred bounds as assertions into the code: We instrumented a counter c for each loop and added the assertion $c \leq bound$, where $bound$ is the loop bound computed by Loopus. We then asked CPAchecker, Pagai and Veriabs to prove these assertions. Since it is to be expected that loop bounds formulated over heap values are difficult to verify, we only considered those bounds which are purely formulated over the stack variables. In this way, we generated 761 assertion tasks. We also ran the bounded model checker CBMC 5.3 [20] on our benchmark in order to check the correctness of the generated assertions (by loop unrolling CBMC can disprove wrong loop bounds in many cases). We chose a timeout of 60s for Loopus as well as for the verification tools because increasing the timeout did not improve results significantly, neither for Loopus nor the verifiers. The generated files with assertions, the version of Loopus which we used, as well as a detailed table of results, is available at [1].

Table III shows the overall results. The column "fail" states the number of loops (assertions about the loop bounds) for which the respective tool crashed, "oom" refers to the cases for which the tool ran out of memory, "timeout" are the cases where the computation exceeded 60 seconds, and "unknown", "false", and "true" are the cases where the tool terminated with results "unknown", "false", or "true" respectively. The result "false" indicates that the tool disproved the bound inferred by Loopus. We checked the 10 loops for which Veriabs refuted the asserted loop bound and it turned out that the bound is actually sound.

Interestingly, even though the assertions are usually of a simple form (we used an industrial, not an academic bench-

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

	fail	oom	timeout	unknown	false	true	true (%)
CPAchecker	61	0	186	160	0	309	43.16%
Pagai	0	8	33	300	0	375	52.37%
Veriabs	179	0	247	239	10	41	5.73%
CBMC	0	8	429	0	0	279	38.97%

TABLE IV

Results of the evaluation on 761 loops in cBench for which Loopus inferred a linear or constant bound over the stack variables.

mark), the best tool in this comparison, Pagai, succeeded to prove only 49% of the assertions. The second CPAchecker proved only 41%, CBMC 37%, and Veriabs 5.4%.

The low success rate is partially caused by the fact that some of the bounds (assertions) are polynomial, which is problematic for the provers, as discussed in Section II. Therefore we state the results restricted to the case of linear or constant bounds in Table IV. Nevertheless, the provers were not much more successful with 52% (Pagai), 43% (CPAchecker), 39% (CBMC), and 5.7% (Veriabs) proven assertions.

In conclusion, our experiment demonstrates that in many cases invariants computed by bound analysis cannot be computed by state-of-the-art invariant analysis techniques.

VIII. Conclusion

We have formulated simple bound analysis concepts for computing invariants which are challenging for state-of-the-art invariant generation techniques. On a set of tasks from the SV-COMP 2018 benchmark, we have demonstrated that current invariant analysis techniques can be significantly improved by means of bound analysis. Additionally, we have shown by an experimental evaluation on an industrial benchmark that the class of invariants which can be verified by state-of-the-art invariant analysis tools is to a large extent different from the class of invariants that is found by bound analysis. Our results show that using bound analysis techniques for invariant generation is very promising and we hope that they motivate further research in this area.

Acknowledgement: The first and the fourth author were supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF). We would also like to thank Marek Chalupa for his help with the experiments.

References

[1] `Experimental Evaluation`. http://forsyte.at/static/people/sinn/fmcad2018/.

[2] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

[3] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[5] P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla. Veriabs: Verification by abstraction and test generation - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 457–462, 2018.

[6] D. Dietsch, M. Greitschus, M. Heizmann, J. Hoenicke, A. Nutz, A. Podelski, C. Schilling, and T. Schindler. Ultimate taipan with dynamic block encoding - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 452–456, 2018.

[7] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. Lower runtime bounds for integer programs. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 550–567, 2016.

[8] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

[9] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304. ACM, 2010.

[10] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 447–451, 2018.

[11] J. Henry, D. Monniaux, and M. Moy. PAGAI: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[12] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[13] L. Kovács. Reasoning algebraically about p-solvable loops. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 249–264, 2008.

[14] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[15] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 of *LNCS*, pages 25–41. Springer, 2005.

[16] M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45.

[17] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, volume 8559 of *LNCS*, pages 745–761. Springer, 2014.

[18] M. Sinn, F. Zuleger, and H. Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *FMCAD*, pages 144–151. IEEE, 2015.

[19] P. Čadek, J. Strejček, and M. Trtík. Tighter loop bound analysis. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 512–527, 2016.

[20] `CBMC`. http://www.cprover.org/cbmc/.

[21] `CBench`. http://ctuning.org/wiki/index.php/CTools:CBench/.

[22] `SV-COMP`. https://sv-comp.sosy-lab.org/.

POST-VERIFICATION DEBUGGING AND RECTIFICATION OF FINITE FIELD ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA TECHNIQUES

Vikas Rao[1], Utkarsh Gupta[1], Irina Ilioaea[2], Arpitha Srinath[1], Priyank Kalla[1], and Florian Enescu[2]
[1]Electrical & Computer Engineering, University of Utah
[2]Mathematics & Statistics, Georgia State University

Abstract: *Formal verification of arithmetic circuits checks whether or not a gate-level circuit correctly implements a given specification model. In cases where this equivalence check fails – the presence of a bug is detected – it is required to: i) debug the circuit, ii) identify a set of nets (signals) where the circuit might be rectified, and iii) compute the corresponding rectification functions at those locations. This paper addresses the problem of post-verification debugging and correction (rectification) of finite field arithmetic circuits. The specification model and the circuit implementation may differ at any number of inputs. We present techniques that determine whether the circuit can be rectified at one particular net (gate output) – i.e. we address single-fix rectification.*

Starting from an equivalence checking setup modeled as a polynomial ideal membership test, we analyze the ideal membership residue to identify potential single-fix rectification locations. Subsequently, we use Nullstellensatz principles to ascertain if indeed a single-fix rectification can be applied at any of these locations. If a single-fix rectification exists, we derive a rectification function by modeling it as the synthesis of an unknown component problem. Our approach is based upon the Gröbner basis algorithm, which we use both as a decision procedure (for rectification test) as well as a quantification procedure (for computing a rectification function). Experiments are performed over various finite field arithmetic circuits that demonstrate the efficacy of our approach, whereas SAT-based approaches are infeasible.

I. INTRODUCTION

Past few years have seen extensive investigations into formal verification of arithmetic circuits. Circuits that implement polynomial computations over large bit-vector operands are hard to verify using methods such as SAT/SMT-solvers, decision diagrams, etc. Recent techniques have investigated the use of polynomial algebra and algebraic geometry techniques for their verification. These include verification of integer arithmetic circuits [1] [2] [3] and also finite field circuits [4] [5]. While these are successful in proving correctness or detecting the presence of bugs, the problem of debugging and correction of arithmetic circuits has only just begun to be addressed [6], [7].

In this paper, we address the problem of rectification of buggy finite field arithmetic circuits. A specification model (*Spec*) is given either as a polynomial description f over a finite field, or as a golden model of a finite field arithmetic circuit. The finite field considered is the field of 2^k elements (denoted by $\mathbb{F}_{2^k}$), where k corresponds to the operand-width (bit-vector word length). An implementation circuit C is also given. Equivalence checking is performed between the *Spec* and the circuit C, and the presence of a bug is detected. No restrictions on the number, type, or locations of the bugs are assumed.

We perform error-diagnosis and a subset $\mathcal{N}$ of the nets of the circuit is identified as *potential rectification locations.* Given the *Spec*, the buggy implementation circuit C, and the set $\mathcal{N}$ of potential rectifiable locations, our objective is to determine whether or not the buggy circuit can be rectified at *one particular net* $x_i \in \mathcal{N}$. This is called **single-fix rectification** in literature [8]. If a single-fix rectification does exist at net x_i in the buggy circuit, then our subsequent objective is to derive a polynomial function $U(X_{PI})$ in terms of the set of primary input variables X_{PI}. This polynomial can be translated (synthesized) into a logic subcircuit such that $x_i = U(X_{PI})$ acts as the rectification function for the buggy circuit C so that C matches the specification.

Our techniques and algorithms are based on symbolic computer algebra and algebraic geometry – particularly on the concepts of the Strong Nullstellensatz and Gröbner bases [9]. We show how to apply our techniques to rectify finite field arithmetic circuits, where conventional SAT-solver based rectification approaches are infeasible.

The paper is organized as follows: The following section reviews related previous work. Section III covers preliminary concepts. The formulation of the verification test is described in Section IV. Section V describes conditions for rectification at a particular net. Section VI describes how rectification function can be synthesized once single-fix rectification is deemed possible. Section VII describes our experimental results and Section VIII concludes the paper.

II. PREVIOUS WORK

Automated diagnosis and rectification of digital circuits has been addressed in [10], [11]. The paper [12] presents algorithms for synthesizing Engineering Change Order (ECO) patches - an analogous problem to rectification. The use of interpolation for ECO has been presented in [8], [13]. The single-fix rectification function approach in [13] has been extended in [8] to generate multiple partial-fix functions. As these approaches are SAT based, they are not efficient for arithmetic circuits. In contrast to these works, our work presents a word-level formulation for single-fix rectification. Computer algebra has been utilized for circuit debugging and rectification in [6], [7]. These approaches rely heavily on the structure of the arithmetic circuit for coefficient calculation.

ISBN: 978-0-9835678-8-2.

Moreover, if the arithmetic circuits contain redundancies, then we have shown that their approach is incomplete in that it cannot resolve the rectification question. We have uploaded an appendix at [14] for detailed discussion on these issues.

Once rectification is deemed feasible, the problem of finding the rectification function has been considered as a partial synthesis problem. The most recent and relevant approach [15], [16] resolves the unknown component problem using an incremental SAT formulation.

The approach used in [17] inserts logic corrector MUXs on the unknown sub-circuits and relies on SAT solvers to realize the functionality. The authors in [18] present a QBF formulation for answering whether a partial implementation can be extended to a complete design that models a given specification.

Despite using state-of-the-art SAT solvers, all the above approaches fail to verify large and complex finite field arithmetic circuits. We demonstrate the efficiency of our implementation by comparing the results against the most recent SAT based approach [15] showing improvement by several orders of magnitude.

III. PRELIMINARIES: NOTATION AND BACKGROUND

Let $\mathbb{F}_q$ denote the finite field of q elements, where $q = p^k$ is a prime power. To model functions over k-bit vector operands, we use $q = 2^k$, i.e. the finite field $\mathbb{F}_{2^k}$ of 2^k elements. The field $\mathbb{F}_{2^k}$ is constructed as $\mathbb{F}_{2^k} = \mathbb{F}_2[X] \pmod{P(X)}$, where $\mathbb{F}_2 = \{0, 1\}$ is the field of two elements, and $P(X)$ is a given irreducible polynomial of degree k with α as one of its root, i.e. $P(\alpha) = 0$.

Let $R = \mathbb{F}_q[x_1, \dots, x_n]$ be the polynomial ring in variables $x_1, \dots, x_n$ with coefficients in $\mathbb{F}_q$. A polynomial $f \in R$ is written as a finite sum of terms $f = c_1X_1 + c_2X_2 + \cdots + c_tX_t$. Here $c_1, \dots, c_t$ are coefficients and $X_1, \dots, X_t$ are monomials, i.e. power products of the type $x_1^{e_1} \cdot x_2^{e_2} \cdots x_n^{e_n}$, $e_j \in \mathbb{Z}_{\geq 0}$. To systematically manipulate the polynomials, a monomial order $>$ (also called a term order) is imposed on the polynomial ring. Subject to $>$, $X_1 > X_2 > \cdots > X_t$, and $lt(f) = c_1X_1$, $lm(f) = X_1$, $lc(f) = c_1$, are the *leading term*, *leading monomial* and *leading coefficient* of f, respectively. Also, for a polynomial f, $tail(f) = f - lt(f)$. In this work, we are mostly concerned with *lexicographic* (lex) term orders.

1) Polynomial Reduction via division: Let f, g be polynomials. If $lm(f)$ is divisible by $lm(g)$, then we say that *f is reducible to r* modulo g, denoted $f \xrightarrow{g} r$, where $r = f - \frac{lt(f)}{lt(g)} \cdot g$. Similarly, f can be *reduced w.r.t. a set of polynomials* $F = \{f_1, \dots, f_s\}$ to obtain a remainder r. This reduction is denoted as $f \xrightarrow{F}_+ r$, where the remainder r is said to be *reduced* – i.e. no term in r is divisible by the leading term of any polynomial f_j in F. Algorithm 1 (Alg. 1.5.1 from [9]) depicts a procedure for this reduction. Along with the remainder r, the algorithm also returns the set of quotients $\{u_1, \dots, u_s\}$ of division of f by $\{f_1, \dots, f_s\}$, respectively, such that $f = u_1 \cdot f_1 + \cdots + u_s \cdot f_s + r$.

Algorithm 1 Multivariate Reduction of f by $F = \{f_1, \dots, f_s\}$

```
procedure $multi\_var\_division(f, \{f_1, \dots, f_s\}, f_j \neq 0)$
    $u_j \leftarrow 0;\ r \leftarrow 0,\ h \leftarrow f$
    while $h \neq 0$ do
        if $\exists j$ s.t. $lm(f_j) \mid lm(h)$ then
            choose $j$ least s.t. $lm(f_j) \mid lm(h)$
            $u_j = u_j + \frac{lt(h)}{lt(f_j)}$
            $h = h - \frac{lt(h)}{lt(f_j)} f_j$
        else
            $r = r + lt(h)$
            $h = h - lt(h)$
    return $(\{u_1, \dots, u_s\}, r)$
```

2) Polynomial Ideals, Varieties and Gröbner Bases:

Definition III.1. Given a ring $R = \mathbb{F}_q[x_1, \dots, x_n]$ and a set of polynomials $F = \{f_1, \dots, f_s\}$ from R, the *ideal* generated by F is $J = \langle F \rangle \subseteq R$:

$$J = \langle f_1, \dots, f_s \rangle = \{h_1 \cdot f_1 + \cdots + h_s \cdot f_s : h_1, \dots, h_s \in R\}. \quad (1)$$

The polynomials $f_1, \dots, f_s$ form the basis of ideal J.

Let $\boldsymbol{a} = (a_1, \dots, a_n) \in \mathbb{F}_q^n$ be a point in the affine space, and f a polynomial in R. If $f(\boldsymbol{a}) = 0$, we say that f *vanishes* on $\boldsymbol{a}$. We have to analyze the *set of all common zeros* of the polynomials of F that lie within the field $\mathbb{F}_q$. This zero set is called the *variety*, which depends on the ideal generated by the polynomials. We denote it by $V(J)$, where: $V(J) = V_{\mathbb{F}_q}(J) = V_{\mathbb{F}_q}(f_1, \dots, f_s) = \{\boldsymbol{a} \in \mathbb{F}_q^n : \forall f \in J, f(\boldsymbol{a}) = 0\}$.

An ideal may have many different sets of generators, i.e. it is possible to have $J = \langle f_1, \dots, f_s \rangle = \langle g_1, \dots, g_t \rangle = \cdots = \langle h_1, \dots, h_r \rangle$, such that $V(f_1, \dots, f_s) = V(g_1, \dots, g_t) = \cdots = V(h_1 \dots, h_r)$. A Gröbner basis (GB) of an ideal is one such generating set $G = \{g_1, \dots, g_t\}$, which possesses many important properties that allow to solve many polynomial decision problems.

Definition III.2. [**Gröbner Basis**] [9]: For a monomial ordering $>$, a set of non-zero polynomials $G = \{g_1, g_2, \cdots, g_t\}$ contained in an ideal J, is called a Gröbner basis of J iff $\forall f \in J$, $f \neq 0$, there exists $g_i \in G$ such that $lm(g_i)$ divides $lm(f)$; i.e., $G = GB(J) \Leftrightarrow \forall f \in J : f \neq 0\ \exists g_i \in G : lm(g_i) \mid lm(f)$.

Then $J = \langle F \rangle = \langle G \rangle$ holds and $G = GB(J)$ forms a basis for J. The Gröbner basis for an ideal J can be computed using the Buchberger's algorithm [19]. It takes as input a set of polynomials $\{f_1, \dots, f_s\}$ and computes its GB $G = \{g_1, g_2, \cdots, g_t\}$. The reader may refer to Algorithm 1.7.1 in [9] for a detailed explanation.

Buchberger's algorithm can be easily extended to output not just the Gröbner basis $G = \{g_1, \dots, g_t\}$ but also a $t \times s$ matrix

M with polynomial entries such that:

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_t \end{bmatrix} = M \cdot \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} \tag{2}$$

An important property of Gröbner bases is that as a decision procedure, they allow for membership testing of a polynomial in an ideal.

Proposition III.1. (*Ideal Membership Testing*) Let $G = GB(J) = \{g_1, \dots, g_t\}$ be the Gröbner basis of ideal J, and f be any polynomial. Then $f \in J \iff f \xrightarrow{G}_+ 0$.

Therefore, if $f \in J$, f can be written as a linear combination (with polynomial coefficients) of the elements of the Gröbner basis:

$$f = u_1 g_1 + u_2 g_2 + \cdots + u_t g_t, \tag{3}$$

where u_i's correspond to the quotients of division $f \xrightarrow{g_1,\dots,g_t}_+ 0$. Subsequently, Eqns. (3) and (2) can be combined to give f as combination of the original polynomials $f_1, \dots, f_s$:

$$f = v_1 f_1 + \cdots + v_s f_s. \tag{4}$$

Given two ideals $J_1 = \langle f_1, \dots, f_s \rangle, J_2 = \langle h_1, \dots, h_r \rangle$, the sum $J_1 + J_2 = \langle f_1, \dots, f_s, h_1 \dots, h_r \rangle$, and their product $J_1 \cdot J_2 = \langle f_i \cdot h_j : 1 \le i \le s, 1 \le j \le r \rangle$. Ideals and varieties are dual concepts: $V(J_1 + J_2) = V(J_1) \cap V(J_2)$, and $V(J_1 \cdot J_2) = V(J_1) \cup V(J_2)$. Moreover, if $J_1 \subseteq J_2$ then $V(J_1) \supseteq V(J_2)$.

3) The Strong Nullstellensatz in Finite Fields: For any element $\alpha \in \mathbb{F}_q$, we have that $\alpha^q = \alpha$. Therefore, the polynomial $x^q - x$ vanishes everywhere in $\mathbb{F}_q$, and we call it a *vanishing polynomial*. Let $J_0 = \langle x_1^q - x_1, \dots, x_n^q - x_n \rangle$ be the ideal of all vanishing polynomials in the ring R. Then, $V_{\mathbb{F}_q}(J_0) = V_{\overline{\mathbb{F}_q}}(J_0) = \mathbb{F}_q^n$. Moreover, given any ideal J, $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J) \cap \mathbb{F}_q^n = V_{\overline{\mathbb{F}_q}}(J) \cap V_{\overline{\mathbb{F}_q}}(J_0) = V_{\overline{\mathbb{F}_q}}(J + J_0) = V_{\mathbb{F}_q}(J + J_0)$.

Definition III.3. Given an ideal $J \subset R$ and $V(J) \subseteq \mathbb{F}_q^n$, the *ideal of polynomials that vanish on* $V(J)$ is $I(V(J)) = \{f \in R : \forall \boldsymbol{a} \in V(J), f(\boldsymbol{a}) = 0\}$.

If f vanishes on $V(J)$, then $f \in I(V(J))$. The Strong Nullstellensatz, which has a special form over finite fields, characterizes the ideal $I(V(J))$.

Theorem III.1 (*The Strong Nullstellensatz over finite fields (Theorem 3.2 in [20])*)**.** For any ideal $J \subset \mathbb{F}_q[x_1, \dots, x_n]$, $I(V_{\mathbb{F}_q}(J)) = J + J_0$.

IV. THE VERIFICATION TEST

Given a specification polynomial f, and a circuit C, the verification test is formulated as presented in [4]. The circuit is modeled by a set of multivariate polynomials $F = \{f_1, \dots, f_s\}$ in the ring $R = \mathbb{F}_{2^k}[x_1, \dots, x_n]$ for the given data-path (operand) size k, where $x_1, \dots, x_n$ denote the nets (signals) in the circuit. As the circuit comprises Boolean logic gates, they are modeled as polynomials in $\mathbb{F}_2 \subset \mathbb{F}_{2^k}$:

$$\begin{aligned} z &= \neg a \rightarrow z + a + 1 \pmod 2 \\ z &= a \wedge b \rightarrow z + a \cdot b \pmod 2 \\ z &= a \vee b \rightarrow z + a + b + a \cdot b \pmod 2 \\ z &= a \oplus b \rightarrow z + a + b \pmod 2 \end{aligned} \tag{5}$$

The set of polynomials F generates an ideal, which we denote by $J = \langle F \rangle$. When C correctly implements f, then f agrees with every evaluation of all the nets in C. In other words, f vanishes on $V(J)$, or equivalently $f \in I(V(J))$. The Strong Nullstellensatz in finite fields (Thm. III.1) tells us that $I(V(J)) = J + J_0$, where $J_0 = \langle x_i^q - x_i : i = 1, \dots, n \rangle$. Thus, the verification test can be formulated as ideal membership testing of f in $J + J_0$ using Gröbner bases: to check if $f \xrightarrow{GB(J+J_0)}_+ 0$?

The Gröbner basis computation $GB(J + J_0)$ in $R = \mathbb{F}_q[x_1, \dots, x_n]$ exhibits high complexity, as it is shown to be bounded by $q^{O(n)}$ [20]. In [4], the authors showed that the expensive Gröbner basis computation can be avoided altogether for this verification test. It was shown that for any arbitrary combinational circuit, a specialized term order can be derived by analyzing the topology of the given circuit. Imposition of this term order on R renders the set of polynomials $F = \{f_1, \dots, f_s\}$ itself a Gröbner basis. Based on Buchberger's product criteria, their approach exploits the fact that *when the leading terms of all polynomials in F are relatively prime, then F already constitutes a Gröbner basis.*

Definition IV.1. Let C be an arbitrary combinational circuit described by a set of polynomials $F = \{f_1, \dots, f_s\}$ with variables $\{x_1, \dots, x_n\}$. Starting from the primary outputs, perform a reverse topological traversal of C and order the variables such that $x_i > x_j$ if x_i appears earlier in the reverse topological order. Impose a *lex* term order $>$ to represent each gate as a polynomial f_i, s.t. $f_i = x_i + tail(f_i)$. Then the set $F = \{f_1, \dots, f_s\}$ forms a Gröbner basis, as $lt(f_i) = x_i$ and $lt(f_j) = x_j$ for $i \neq j$ are relatively prime. This term order $>$ is called the **Reverse Topological Term Order (RTTO)**.

Our formulations also contain k-bit word-level variables corresponding to the input and output word-level operands. These variables can also be accommodated in RTTO $>$ by imposing a lex term order with the variable order *"Output word $>$ input words $>$ bit-level variables ordered reverse topologically"*. In [4], the authors analyzed the effect of such a term order further on ideal generators that include the vanishing polynomials. Let $X_{PI} \subset \{x_1, \dots, x_n\}$ be the primary input variables of the circuit. Let $F_0^{PI} = \{x_i^2 - x_i : x_i \in X_{PI}\}$ denote the set of bit-level vanishing polynomials in primary inputs. We utilize the following result from [4].

Proposition IV.1. (Corollary 6.1 in [4]) Using RTTO $>$ to represent the polynomials in R, the set $F \cup F_0^{PI}$ constitutes a Gröbner basis of $J + J_0$.

The benefit of using RTTO $>$ is that the verification test can be performed solely by way of polynomial division $f \xrightarrow{F,F_0^{PI}}_+ r$, and by checking whether or not $r = 0$? If $r = 0$, then C implements f. Otherwise when $r \neq 0$, there exists a bug in the design. Moreover, RTTO $>$ ensures that when $r \neq 0$, r comprises only primary input variables X_{PI}. Any assignment to X_{PI} that makes $r \neq 0$ generates a counter-example that can be used for debugging.

We use the verification setup under RTTO $>$ (i.e. Def. IV.1 and Prop. IV.1) to rectify the circuit. Our approach begins when the verification test detects the presence of a bug in the design, i.e. $f \xrightarrow{F,F_0^{PI}}_+ r$ with $r \neq 0$. In the sequel, we will use the circuit shown in Fig. 1 as a running example to demonstrate our approach to debugging and rectification. The circuit is a modified version of a Mastrovito multiplier [21], where extra redundant logic was first added in the circuit, and then a bug was introduced in the redundant logic.

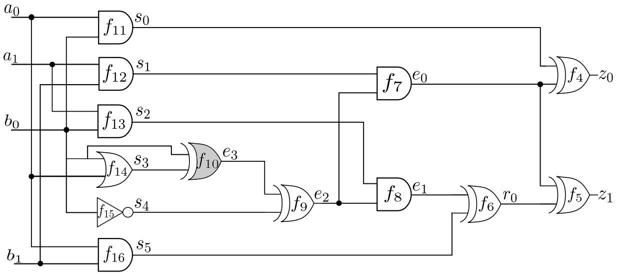


Fig. 1: Design verification of a 2-bit finite-field multiplier. The circuit is buggy, with the bug introduced at net e_3. A correct implementation includes an AND gate at e_3, which is replaced by an XOR gate to introduce a bug.

Example IV.1. We perform verification of the design of a 2-bit finite field multiplier in $\mathbb{F}_4$, where the output Z is to be computed as $A \cdot B$, where $Z = \{z_1, z_0\}, A = \{a_1, a_0\}, B = \{b_1, b_0\}$ are the given 2-bit operands. Assume further that $P(X) = X^2 + X + 1$ is the irreducible polynomial used to construct $\mathbb{F}_4 = \mathbb{F}_2[X] \pmod{P(X)}$, with $P(\alpha) = 0$.

The implemented circuit C is given as shown in Fig. 1. Denote polynomial $f : Z + A \cdot B$ as the design specification. For the verification test, we perform a reverse topological traversal of the circuit to derive RTTO $>$, i.e. a *lex* term order with variable order: $\{Z\} > \{A > B\} > \{z_0 > z_1\} > \{r_0\} > \{e_0 > e_1\} > \{e_2\} > \{e_3\} > \{s_0 > s_1 > s_2 > s_3 > s_4 > s_5\} > \{a_0 > a_1 > b_0 > b_1\}$.
The polynomials describing the circuit are given as:

$$\begin{array}{rl}
f_1 : Z + z_0 + \alpha z_1; & f_9 : e_2 + e_3 + s_4; \\
f_2 : A + a_0 + \alpha a_1; & f_{10} : e_3 + b_0 + s_3; \\
f_3 : B + b_0 + \alpha b_1; & f_{11} : s_0 + a_0 b_0; \\
f_4 : z_0 + s_0 + e_0; & f_{12} : s_1 + a_1 b_1; \\
f_5 : z_1 + e_0 + r_0; & f_{13} : s_2 + a_1 b_0; \\
f_6 : r_0 + e_1 + s_5; & f_{14} : s_3 + a_0 + b_0 + a_0 b_0; \\
f_7 : e_0 + s_1 e_2; & f_{15} : s_4 + b_0 + 1; \\
f_8 : e_1 + s_2 e_2; & f_{16} : s_5 + a_0 b_1;
\end{array}$$

Then $F = \{f_1, \dots, f_{16}\}$, $F_0^{PI} = \{a_0^2 - a_0, a_1^2 - a_1, b_0^2 - b_0, b_1^2 - b_1\}$, and $F \cup F_0^{PI}$ constitutes a Gröbner basis of $J + J_0$. Computing $f \xrightarrow{F,F_0^{PI}}_+ r$ gives $r = (\alpha+1)a_0a_1b_1b_0 + (\alpha+1)a_0a_1b_1 + (\alpha+1)a_1b_1b_0 + (\alpha)a_1b_0$. Since $r \neq 0$, the presence of a bug in the design is detected. Our objective now is to identify a net where rectification can be performed, and then to subsequently identify a rectification function.

V. Identification of the Rectification Target

After the presence of a bug is detected, we address the problem of single-fix rectification of C. In this section, we present an approach that ascertains whether or not a single-fix rectification can be applied at a given (target) net x_i in C. In principle, our approach can be applied at every net x_i in C to see if C at all admits single-fix rectification. However, it is possible to prune the search for these target nets x_i by analyzing the non-zero remainder obtained by the Gröbner basis reduction $f \xrightarrow{F,F_0^{PI}}_+ r$. We show how to construct a subset $\mathcal{N} \subseteq \{x_1, \dots, x_n\}$ as possible rectification targets. This rectification target pruning approach is inspired from [7]. Then we present our rectification theorem and the search for a rectification function.

1) Potential rectification target nets: The circuit C has k-bit operands, and the output is expressed as $Z = \sum_{i=0}^{k-1} z_i \alpha^i$. Then the non-zero remainder r can be partitioned based on the coefficients of the monomials in r and re-expressed as:

$$r = \alpha^0(r_0) + \alpha^1(r_1) + \cdots + \alpha^{k-1}(r_{k-1}) \qquad (6)$$

Non-zero terms r_i (with coefficient α^i) imply that the effect of the bug is observable at the bit-level output z_i. We consider the transitive fanin cones of logic of the output bits z_i. When a bug affects multiple outputs, a single-fix rectification might exist only at the nets that lie in the intersection of the respective fanin-cones of the affected outputs. In our experiments, we include these nets in $\mathcal{N}$ to check if any one of them admits a single-fix rectification.

Example V.1. As shown in Ex. IV.1, $f \xrightarrow{F,F_0^{PI}}_+ r = (\alpha+1)a_0a_1b_1b_0 + (\alpha+1)a_0a_1b_1 + (\alpha+1)a_1b_1b_0 + (\alpha)a_1b_0$. We re-write the remainder $r = \alpha^0 r_0 + \alpha^1 r_1 = \alpha \cdot (a_0a_1b_1b_0 + a_0a_1b_1 + a_1b_1b_0 + a_1b_0) + 1 \cdot (a_0a_1b_1b_0 + a_0a_1b_1 + a_1b_1b_0)$. Since both r_0 and r_1 are non-zero, the bug affects both primary outputs z_0, z_1. By identifying the nets that lie in the intersection of the fanin cones of z_0, z_1, we construct $\mathcal{N} = \{s_4, s_3, s_2, s_1, e_3, e_2, e_0\}$ as potential rectifiable locations.

2) Confirming a rectification target: After post-verification debugging is performed to identify a set of nets $\mathcal{N} \subseteq \{x_1, \dots, x_n\}$ that are potential rectification target nets, we now present an approach that *confirms* whether or not the circuit can indeed be single-fix-rectified at net x_i. *Single-fix-rectification at target net x_i means that there exists a polynomial function $U(X_{PI})$ which, when implemented at net x_i, ensures that the circuit C would correctly implement the specification f.* Note that $x_i = U(X_{PI})$ is a polynomial function of the type $\mathbb{F}_2^{|X_{PI}|} \rightarrow \mathbb{F}_2$ as it implements a subcircuit at net x_i.

In the set of polynomials F, we replace $f_i = x_i + U(X_{PI})$ as the polynomial for the rectification function at x_i, where $U(X_{PI})$ is a hitherto unknown/unresolved polynomial function component. In other words, F is updated to $F = \{f_1, \ldots, f_{i-1}, f_i = x_i + U(X_{PI}), f_{i+1}, \ldots, f_s\}$. We state and prove the *rectification theorem that checks for the existence of* $U(X_{PI})$ *as a single-fix rectification function at* x_i.

Theorem V.1 (Rectification Theorem)**.** Given the specification polynomial f, and the implementation circuit C, derive RTTO $>$ to represent the polynomials. Using RTTO $>$, construct two ideals:

- $J_L = \langle F_L \rangle$, where $F_L = \{f_1, \ldots, f_{i-1}, f_i = x_i + 1, f_{i+1}, \ldots, f_s\}$;
- $J_H = \langle F_H \rangle$, where $F_H = \{f_1, \ldots, f_{i-1}, f_i = x_i, f_{i+1}, \ldots, f_s\}$;

where the polynomials $f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_s$ are the same as in the generators of ideal J (representing the circuit), and f_i is replaced with $f_i = x_i + 1$ in J_L and $f_i = x_i$ in J_H, respectively. Perform the reductions:

- $f \xrightarrow{F_L, F_0^{PI}}_+ r_L$
- $f \xrightarrow{F_H, F_0^{PI}}_+ r_H$

Let $V_{\mathbb{F}_q}(r_L), V_{\mathbb{F}_q}(r_H)$ denote the varieties of r_L and r_H, respectively, over the given field $\mathbb{F}_q$. Then the buggy circuit C admits a single-fix rectification at the net (gate output) x_i **if and only if** $V_{\mathbb{F}_q}(r_L) \cup V_{\mathbb{F}_q}(r_H) = \mathbb{F}_q^{|X_{PI}|} = V(J_0^{PI})$.

Proof. As rectification at net x_i makes the circuit C match the specification f, f should vanish on $V(J)$. Thus, the rectification condition can be equivalently stated as: "f vanishes on $V_{\mathbb{F}_q}(J) \iff V_{\mathbb{F}_q}(r_L) \cup V_{\mathbb{F}_q}(r_H) = \mathbb{F}_q^{|X_{PI}|}$."

(i) To prove $\Rightarrow$: Let $x_{PI} \in \mathbb{F}_q^{|X_{PI}|}$ be an assignment to the primary input variables of C. For every point x_{PI}, there exists a corresponding assignment x_{int} to the rest of the variables of the circuit. For each primary input assignment, the target net x_i evaluates to either $x_i = 0$ or $x_i = 1$. When $x_i = 0$, then J_H vanishes on the point (x_{PI}, x_{int}). Likewise, when $x_i = 1$, J_L vanishes on (x_{PI}, x_{int}). Since $f \xrightarrow{J_H, J_0}_+ r_H$ and $f \xrightarrow{J_L, J_0}_+ r_L$, and f vanishes on the point (x_{PI}, x_{int}), we obtain that either $r_H(x_{PI}) = 0$ or $r_L(x_{PI}) = 0$. In other words, for every primary input assignment x_{PI}, either r_L or r_H vanishes. This implies that $V(r_L) \cup V(r_H) = \mathbb{F}_q^{|X_{PI}|} = V(J_0^{PI})$.

(ii) To prove "$\Leftarrow$": Say there exists an assignment to the primary inputs $x_{PI} \in \mathbb{F}_q^{|X_{PI}|}$ such that r_H vanishes on x_{PI}, i.e. $r_H(x_{PI}) = 0$. Corresponding to x_{PI}, there exists an assignment to the rest of the variables of the circuit x_{int}. As $f \xrightarrow{J_H, J_0}_+ r_H$, we have that f is a member of the ideal $J_H + J_0 + \langle r_H \rangle$. Therefore, when $r_H(x_{PI}) = 0$, the ideal J_H also vanishes on (x_{PI}, x_{int}), and J_0 by definition vanishes everywhere. This implies that $f(x_{PI}, x_{int}) = 0$. Similarly, the argument also holds that when $r_L(x_{PI}) = 0$, then $f(x_{PI}, x_{int}) = 0$. This proves that for all primary inputs if r_L or r_H vanishes, then f vanishes too; and that completes the proof.

□

Note that the check "Is $V_{\mathbb{F}_q}(r_L) \cup V_{\mathbb{F}_q}(r_H) = \mathbb{F}_q^{|X_{PI}|} = V(J_0^{PI})$?" can be performed as shown below, where the union of varieties corresponds to the product of ideals.

$$
\begin{aligned}
V_{\mathbb{F}_q}(r_L) \cup V_{\mathbb{F}_q}(r_H) = & V_{\mathbb{F}_q}(r_L \cdot r_H) = V_{\overline{\mathbb{F}_q}}(\langle r_L \cdot r_H \rangle + J_0^{PI}) \\
& = V_{\mathbb{F}_q}(\langle r_L \cdot r_H \rangle + J_0^{PI})
\end{aligned}
$$

Thus, to check for single-fix rectification at the net x_i, we need to compute the Gröbner basis $G = GB(\{r_L \cdot r_H\} \cup F_0^{PI})$ and see if G exactly equals F_0^{PI}.

Example V.2. Continuing with our running example, we demonstrate the rectification checks at nets e_3, s_1. As the bug was introduced at e_3, it is obvious that the circuit is rectifiable at e_3. For the rectification check at e_3, we mark the polynomial f_{10} for modification:

- $J_L = \langle F_L \rangle$, where $F_L = \{f_1, \ldots, f_{10} = e_3 + 1, \ldots, f_{16}\}$,
- $J_H = \langle F_H \rangle$, where $F_H = \{f_1, \ldots, f_{10} = e_3, \ldots, f_{16}\}$.

Reducing the specification $f : Z + A \cdot B$ modulo these ideals, we get:

- $r_L = f \xrightarrow{F_L, F_0^{PI}}_+ (\alpha + 1)a_1 b_1 b_0 + (\alpha + 1)a_1 b_1$
- $r_H = f \xrightarrow{F_H, F_0^{PI}}_+ (\alpha + 1)a_1 b_1 b_0 + (\alpha)a_1 b_0$

When we compute the Gröbner basis $G = GB(r_L \cdot r_H, F_0^{PI})$, we obtain $G = \{a_0^2 - a_0, a_1^2 - a_1, b_0^2 - b_0, b_1^2 - b_1\}$, corresponding to the ideal of all vanishing polynomials in primary inputs. This implies the existence of a rectification function at e_3.

In fact, the rectification test also passes for the net s_4; implying that the bug at e_3 can indeed be rectified at a different gate which does not lie in the fanin cone of e_3. However, the rectification test fails at net s_1. When the problem is formulated by modifying the polynomial f_{12} at net s_1, the corresponding computation for $G = GB(r_L \cdot r_H, F_0^{PI})$ results in $G = \{a_0^2 - a_0, b_0^2 - b_0, a_1^2 - a_1, b_1^2 - b_1,\ a_1 b_0,\ a_0 a_1 b_1 + (\alpha) a_0 a_1 b_0\}$. Due to the presence of the last 2 polynomials, $G \neq F_0^{PI}$, and rectification is not possible at net s_1. In our experiments, the rectification check is performed on subset $\mathcal{N}$ starting from the net closest to the primary inputs with the intent of reducing variables in computed rectification function.

VI. COMPUTING A RECTIFICATION FUNCTION

After the confirmation that the circuit indeed admits a rectification function at net x_i, our objective is to compute a rectification function $x_i = U(X_{PI})$. We call U the *unknown component* which has to be resolved. Due to the presence of internal *don't care* conditions, there may exist one or more polynomial functions U that may rectify the circuit. Our approach computes one of the candidate functions U, and proceeds as follows.

Once again, we use RTTO $>$ to represent the set of polynomials of the circuit. *The polynomial corresponding to the target net* x_i *is replaced by the polynomial* $f_i = x_i + U(X_{PI})$, *where $lm(f_i) = x_i$ and $tail(f_i) = U(X_{PI})$.* In other words, the set F is updated to $F = \{f_1, \dots, f_i = x_i + U, \dots, f_s\}$. Notice that due to RTTO $>$, the set F still constitutes a Gröbner basis, as all polynomials in F have leading terms that are relatively prime. Moreover, by virtue of Prop. IV.1, the set $F \cup F_0^{PI}$ also constitutes a Gröbner basis. Thus, for a correct implementation, the condition $f \xrightarrow{F \cup F_0^{PI}}_+ 0$ still holds. Using Prop. III.1 and Eqn. 3, we can rewrite f in terms of these generators as:

$$f = h_1 f_1 + h_2 f_2 + \cdots + h_i f_i + \cdots + h_s f_s + \sum_{x_l \in X_{PI}} H_l (x_l^2 - x_l) \tag{7}$$

where $h_1, \dots, h_s, H_l$ are arbitrary polynomials from the ring R. Substituting $f_i = x_i + U$ for the unknown component in Eqn. (7), we have:

$$\begin{aligned} f = {} & h_1 f_1 + \cdots + h_{i-1} f_{i-1} + \boldsymbol{h_i x_i} + \boldsymbol{h_i U} + \cdots + h_s f_s \\ & + \sum_{x_l \in X_{PI}} H_l \cdot (x_l^2 - x_l) \end{aligned} \tag{8}$$

$$\begin{aligned} f - h_1 f_1 - \cdots - h_{i-1} f_{i-1} - \boldsymbol{h_i x_i} = {} & \boldsymbol{h_i U} + h_{i+1} f_{i+1} + \\ \cdots + h_s f_s + \sum_{x_l \in X_{PI}} H_l \cdot (x_l^2 - x_l) & \end{aligned} \tag{9}$$

Notice that on the L.H.S. of Eqn. (9), the polynomials $f, f_1, \dots, f_{i-1}$ and the monomial x_i are *known quantities/expressions*. Therefore, f can be divided by $f_1, \dots, f_{i-1}$, and by x_i, to obtain the respective quotients of the division $h_1, \dots, h_i$ and a remainder r where $r = f - h_1 f_1 - \cdots - h_i x_i$. After h_i is computed (as the quotient of this division by x_i), the R.H.S. of Eqn. (9) consists of $h_i, f_{i+1}, \dots, f_s$ and all the vanishing polynomials $x_l^2 - x_l$ as known expressions. This implies that:

$$f - h_1 f_1 - \cdots - h_i x_i \quad \in \langle h_i, f_{i+1}, \dots, f_s, x_l^2 - x_l \rangle \tag{10}$$

$$r \quad \in \langle h_i, f_{i+1}, \dots, f_s, x_l^2 - x_l \rangle \tag{11}$$

This ideal membership implies that r can be written as some polynomial combination of the generators $h_i, f_{i+1}, \dots, f_s, x_l^q - x_l$. This combination can be identified by first computing the Gröbner basis G of the ideal $\langle h_i, f_{i+1}, \dots, f_s, x_l^q - x_l \rangle$, and then performing the ideal membership test $r \xrightarrow{G}_+ 0$, while utilizing Eqns. (3) and (4). As a result, we can write:

$$r = h_i' h_i + h_{i+1}' f_{i+1} + \cdots + h_s' f_s + \sum H_l (x_l^2 - x_l) \tag{12}$$

Then $U = h_i'$ is a polynomial function that forms the solution to the unknown component problem. Algorithmically, as $U = h_i'$ is computed as a quotient of division, U may contain any variables $X \subseteq \{x_1, \dots, x_n\}$ in its support. However, due to the imposition of RTTO $>$, U will contain only those variables x_j in its support set that are less than x_i in the reverse topological order. Once such a polynomial U is obtained, it can be easily expressed in terms of the primary input variables. To achieve such a normalization, U can be reduced modulo the set of polynomials $\{f_j = x_j + tail(f_j)\}$ such that x_j lies in the fanin cone of U. Performing this division also in a reverse topological fashion results in U being expressed in primary inputs only. In this fashion, the polynomial $f_i : x_i + U(X_{PI})$ can be identified to implement the function of a subcircuit at the net x_i so that C correctly implements f.

Note that in Eqn. (11), while $\{f_{i+1}, \dots, f_s\}$ constitutes a GB under RTTO, the set $\{h_i, f_{i+1}, \dots, f_s\}$ may not. So a GB computation is required. On the other hand, we may also encounter situations when h_i results as being a constant in the field $\mathbb{F}_q$. When a constant is a member of an ideal J, then $GB(J) = \{1\}$. To arrive at an implementable solution in this case, we multiply r by the inverse of h_i (h_i^{-1}) and reduce the result modulo the rest of the polynomials$\{f_{i+1}, \dots, f_s\}$.

$$r \cdot h_i^{-1} \xrightarrow{f_{i+1}} \xrightarrow{f_{i+2}} \dots \xrightarrow{f_s}_+ U. \tag{13}$$

We now demonstrate the application of this approach on our running example.

Example VI.1. In Ex. V.2, we showed that rectification is possible at the net e_3, i.e. there exists a polynomial $f_{10} : e_3 + U$ that can rectify the circuit. Using the same term order as in the previous examples, we mark $f_{10} = e_3 + U$ as the unknown component, and include it in the set $F = \{f_1, \dots, f_{10} = e_3 + U, \dots, f_{16}\}$. Based on Eqns. (9)-(11), we begin reducing the specification polynomial f modulo the set $\{f_1, \dots, f_9, e_3\} \cup F_0$. The reduction order for f based on RTTO $>$ is: $f \xrightarrow{f_1} \xrightarrow{f_2} \xrightarrow{f_3} \xrightarrow{f_4} \xrightarrow{f_5} \xrightarrow{f_6} \xrightarrow{f_7} \xrightarrow{f_8} \xrightarrow{f_9} \xrightarrow{lt(f_{10})} r$.

We will use the following notations to depict this reduction: '[]' to represent quotients of division h_j's, '()' to represent the divisors f_j's, and '{}' to represent the (partial) remainder fp_j obtained after every reduction step.

$f \xrightarrow{f_1} [1](Z + z_0 + \alpha z_1) + \{AB + z_0 + \alpha z_1\} \rightarrow fp_1$

$fp_1 \xrightarrow{f_2} [B](A + a_0 + \alpha a_1) + \{B a_0 + \alpha B a_1 + z_0 + \alpha z_1\} \rightarrow fp_2$

$fp_2 \xrightarrow{f_3} [a_0 + \alpha a_1](B + b_0 + \alpha b_1) + \{z_0 + \alpha z_1 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_3$

$fp_3 \xrightarrow{f_4} [1](z_0 + e_0 + s_0) + \{\alpha z_1 + e_0 + s_0 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_4$

$fp_4 \xrightarrow{f_5} [\alpha](z_1 + r_0 + e_0) + \{\alpha z_1 + e_0 + s_0 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_5$

$fp_5 \xrightarrow{f_6} [\alpha](r_0 + e_1 + s_5) + \{(\alpha + 1) e_0 + \alpha e_1 + s_0 + \alpha s_5 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_6$

$fp_6 \xrightarrow{f_7} [\alpha + 1](e_0 + e_2 * s_1) + \{\alpha e_1 + (\alpha + 1) e_2 s_1 + s_0 + \alpha s_5 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_7$

$fp_7 \xrightarrow{f_8} [\alpha](e_1 + e_2 * s_2) + \{(\alpha + 1) e_2 s_1 + \alpha e_2 s_2 + s_0 + \alpha s_5 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_8$

$fp_8 \xrightarrow{f_9} [(\alpha + 1) s_1 + \alpha s_2](e_2 + e_3 + s_4) + \{(\alpha + 1) e_3 s_1 + \alpha e_3 s_2 + s_0 + (\alpha + 1) s_1 s_4 + \alpha s_2 s_4 + \alpha s_5 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\} \rightarrow fp_9$

Finally, the obtained remainder fp_9 is reduced by $lt(f_{10}) = e_3$ to obtain the quotient h_{10} and the remainder r:

$$fp_9 \xrightarrow{lt(f_{10})} [\underbrace{(\alpha + 1) s_1 + \alpha s_2}_{h_{10}}](e_3) + \\ \underbrace{\{s_0 + (\alpha + 1) s_1 s_4 + \alpha s_2 s_4 + \alpha s_5 + \alpha a_0 b_1 + a_0 b_0 + (\alpha + 1) a_1 b_1 + \alpha a_1 b_0\}}_{r}$$

Now that we have $r, h_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}$ available as known expressions, the unknown component problem can be formulated as an ideal membership test using Eqn. (11) such that:

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

$$r \in \langle h_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16} \rangle + \langle F_0^{PI} \rangle.$$

The above ideal membership can be solved by first computing the Gröbner basis of the generators and then expressing r as a linear combination of the ideal members:

$$r = U \cdot h_{10} + h_{11} f_{11} + h_{12} f_{12} + h_{13} f_{13} + h_{14} f_{14} + h_{15} f_{15} + h_{16} f_{16}$$

In this case, the ideal membership test results in the polynomial r being expressed as:

$$r = [b_0]h_{10} + [1]f_{11} + [\alpha + 1]f_{12} + [\alpha s_4 + \alpha b_0]f_{13} + [0]f_{14} + [(\alpha + 1)s_1 + \alpha a_1 b_0]f_{15} + [\alpha]f_{16} + [0]f_{17} + [0]f_{18} + [0]f_{19} + [0]f_{20};$$

Thus, $U = b_0$ is a solution to the *unknown component* f_{10}, i.e. $f_{10} = e_3 + b_0$. This depicts that e_3 implements just the primary input net b_0, thus also identifying redundancy in the design.

VII. EXPERIMENTS

This section presents experimental results using our approach to debug the circuits and perform a single-fix rectification. We compare results of our implementation against the incremental SAT-based approach presented in [15] wherever it's relevant. The approach presented in [15] is implemented using PICOSAT [22]. The experiments were performed on a 3.5GHz Intel(R) Core$^{\text{TM}}$ i7-4770K Quad-Core CPU with 32 GB of RAM.

We have performed experiments for the cases when the bugs are present near the input, middle, or near the output of the circuit, represented using labels NI, NM, and NO respectively in the tables. All the algorithms were implemented in SINGULAR [23].

1) Verification between a word level specification v/s Mastrovito implementation: Table I presents the results of our approach when the bugs are placed in a Mastrovito multiplier implementation compared against a specification, which is given in terms of a word level polynomial f. A Mastrovito multiplier has word level specification $Z = A \times B \pmod{P(x)}$, where $P(x)$ is a given primitive polynomial for the datapath size k. Bugs in the circuit are introduced, and the presence of the bugs is detetced. Then we apply our approach to check for single-fix rectification interatively on the nets selected in $\mathcal{N}$. If rectification is feasible at x_i, the unknown component problem is solved to identify a rectification function.

We are able to verify and debug the circuits for upto 64-bits within our stipulated Time Out (TO) period.

2) Word level specification v/s Point addition implementation: Point addition is an operation required for the task of encryption, decryption and authentication in Elliptic Curve Cryptography (ECC). Modern approaches represent the points in projective coordinate systems, *e.g.*, the López-Dahab (LD) projective coordinate, due to which the point addition operation can be implemented as polynomials in the field.

Example VII.1. Given an elliptic curve: $Y^2 + XYZ = X^3 Z + aX^2 Z^2 + bZ^4$ over $\mathbb{F}_{2^k}$, where $X, Y, Z \in \mathbb{F}_{2^k}$ and similarly, a, b are constants from the field. Point addition over the elliptic curve is (X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, 1). Then X_3, Y_3, Z_3 can be computed as follows:

$$\begin{array}{ll} A = Y_2 \cdot Z_1^2 + Y_1 & B = X_2 \cdot Z_1 + X_1 \\ C = Z_1 \cdot B & D = B^2 \cdot (C + aZ_1^2) \\ Z_3 = C^2 & E = A \cdot C \\ X_3 = A^2 + D + E & F = X_3 + X_2 \cdot Z_3 \\ G = X_3 + Y_2 \cdot Z_3 & Y_3 = E \cdot F + Z_3 \cdot G \end{array}$$

Each of the polynomials in the above design are implemented as a (gate-level) logic block and are interconnected to obtain final outputs X_3, Y_3 and Z_3. Table II shows the comparison of the time required for debugging and rectification for the implementation of the block $D = B^2 \cdot (C + aZ_1^2)$.

TABLE II: Single fix rectification debug in Point Addition circuits against word level specification. Time is in seconds; *k* = Datapath Size, #Gates = No. of gates, K = 10^3, *a*=verification time, *b*=time for rectification check, *c*=time for component correction computation, *d*=total time

k	#Gates	Our implementation											
		NI				*NM*				*NO*			
		a	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
8	244	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	1.2K	1.3	3.9	1.5	6.7	1.2	3.7	2	6.9	1.2	3.7	1.8	6.7
32	3.9K	37	112	77	226	38	110	22	170	37	108	35	180

3) Word level specification v/s Barrett reduction implementation: Barrett reduction is the other widely used multiplier design method adopted in cryptography system designs. Based on Barrett reduction, a multiplier can be designed in two steps: multiplication $R = A \times B$ and a subsequent Barrett reduction G = R (mod P). Table III shows results for debugging and rectification of Barrett multipliers against a polynomial specification.

TABLE III: Single fix rectification debug in Barrett reduction circuits against word level specification. Time is in seconds; *k* = Datapath Size, #Gates = No. of gates, K = 10^3, *a*=verification time, *b*=time for rectification check, *c*=time for component correction computation, *d*=total time

k	#Gates	Our implementation											
		NI				*NM*				*NO*			
		a	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
8	134	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	427	0.1	0.0	0.0	0.1	0.0	0.2	0.1	0.3	0.0	0.1	0.3	0.4
32	1.4K	0.4	1.4	0.1	1.9	0.5	1.5	0.1	2.1	1.2	2.2	1.1	4.5
64	4.9K	19	58	5.4	82	21	60	1.7	83	63	104	141	308

Since the SAT-based approach cannot be applied against a word level specification polynomial, we perform experiments while using another multiplier implementation as the specification.

4) Verification between a specification and implementation given as gate level circuits: Mastrovito v/s Montgomery multipliers: Montgomery architectures [24] are considered more efficient than Mastrovito multipliers for exponentiation, as they do not require explicit reduction modulo $P(x)$ after each step.

Table IV presents the results of our approach to debug and rectification with the bugs placed in the Montgomery

TABLE I: Single fix rectification debug in Mastrovito circuit against word level specification. Time is in seconds; k = Datapath Size, #Gates = No. of gates, K = 10^3, a=verification time, b=time for rectification check, c=time for component correction computation, d=total time

k	#Gates	Our implementation											
		NI				*NM*				*NO*			
		a	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
9	0.23K	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	0.29K	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	0.35K	0.0	0.0	0.1	0.1	0.0	0.0	0.1	0.1	0.0	0.0	0.1	0.1
12	0.97K	0.1	0.5	0.4	0.9	0.2	0.5	0.4	1.1	0.5	0.8	0.4	1.7
13	0.82K	0.1	0.3	0.2	0.6	0.2	0.6	0.2	1.0	0.7	0.8	0.2	1.7
16	1.8K	0.9	2.6	1.0	4.5	1.1	3.5	1.0	5.6	2.8	5.3	1.0	9.1
32	5.4K	36	110	42	188	40	160	47	247	38	240	150	428
64	21.8K	2210	7100	2432	9532	2200	8000	2575	12775	2150	7840	10020	20010

TABLE IV: Rectification for Mastrovito circuit with Montgomery circuit as specification. Time is in seconds; k = Datapath Size, #Gates = No. of gates, (TO): Time-Out = 3 hrs, K = 10^3, a=verification time, b=time for rectification check, c=time for component correction computation, d=total time

k	#Gates	Incremental SAT **[15]**			Our Approach											
		NI	*NM*	*NO*	*NI*				*NM*				*NO*			
					a	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
9	0.6K	35	37	33	0.1	0.5	0.2	0.8	0.2	0.2	0.1	0.5	1.8	2.2	0.6	4.6
10	0.7K	231	215	214	0.3	1	0.5	1.8	0.3	1	0.8	2.1	4.7	5.4	0.2	10
11	0.9K	2090	1927	2000	0.6	2	1	3.6	0.8	2	32	35	9	10	0.4	19
12	1.6K	8676	23400	24085	3.2	9.6	3.5	16	3.2	9.3	12	24	155	160	1.6	316
13	1.7K	TO	TO	TO	3.3	10	4.5	18	3.5	10	22	35	170	177	1.6	349
16	3K	TO	TO	TO	27	81	35	143	28	83	48	159	210	176	2.5	389
32	9.8K	TO	TO	TO	2060	6595	1870	10525	2100	7320	1289	10709	2215	7870	1204	11289

multiplier with a Mastrovito multiplier circuit used as the specification. While the approach [15] finds a satisfying transformation assignment which can be mapped to a library gate, our approach debugs the circuit and finds a single fix rectification function. As shown in the table, our approach shows improvement by several orders of magnitude over [15].

It takes considerable amount of time for verification and rectification check when the bug is close to the output. We are working on further improving the experiments by employing better data structures like ZBDDs ([25]), and devising better heuristics to perform rectification check. Due to several limitations w.r.t the number of ring variables that can be declared in SINGULAR, we have had to restrict our experiments within 64-bit data-path size.

VIII. CONCLUSIONS

This paper has presented a fully automated debug approach for single fix rectification of finite field arithmetic circuits. Given a specification and its circuit implementation, we verify the circuit. If verification detects a bug, we identify all potential single-fix rectification target nets, and perform rectification check at each of these nets. If a net admits single-fix rectification, we compute a corresponding rectification function. The underlying theory and algorithms are based on Gröbner basis reductions, Nullstellensatz, and ideal membership test. The experimental results demonstrate the efficacy of our approach for finite field arithmetic circuits where we achieve several orders of magnitude improvement as compared to recent SAT-based approach. As part of our future work, we are working on improving the efficiency of our implementation to target higher bit-widths. We are also investigating how the current procedure can be extended to cover integer arithmetic circuits. Further research also includes exploring the current approach for the case of multi-fix rectification.

REFERENCES

[1] D. Ritirc, A. Biere, and M. Kauers, “Column-Wise Verification of Multipliers Using Computer Algebra,” in *Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[2] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, “Verification of Gate-level Arithmetic Circuits by Function Extraction,” in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[3] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, “Formal verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction,” in *Design, Automation Test in Europe Conference Exhibition*, 2016.

[4] J. Lv, P. Kalla, and F. Enescu, “Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits,” in *IEEE Trans. on CAD*, vol. 32, no. 9, 2013, pp. 1409–1420.

[5] A. Lvov, L. Lastras-Montano, B. Trager, V. Paruthi, R. Shadowen, and A. El-Zein, “Verification of Galois field based circuits by formal reasoning based on computational algebraic geometry,” *Formal Methods in System Design*, vol. 45, no. 2, pp. 189–212, Oct 2014.

[6] F. Farahmandi and P. Mishra, “Automated Debugging of Arithmetic Circuits Using Incremental Gröbner Basis Reduction,” in *IEEE International Conference on Computer Design (ICCD)*, 2017.

[7] F. Farahmandi and P. Mishra, “Automated Test Generation for Debugging Arithmetic Circuits,” in *Design, Automation Test in Europe Conference Exhibition(DATE*, 2016.

[8] K. F. Tang, C. A. Wu, P. K. Huang, and C. Y. Huang, “Interpolation-Based Incremental ECO Synthesis for Multi-Error Logic Rectification,” in *Proc. Design Automation Conf. (DAC)*, 2011, pp. 146–151.

[9] W. W. Adams and P. Loustaunau, *An Introduction to Gröbner Bases*. American Mathematical Society, 1994.

[10] J. C. Madre, O. Coudert, and J. P. Billon, “Automating the Diagnosis and the Rectification of Design Errors with PRIAM,” in *Proc. ICCAD*, 1989, pp. 30–33.

[11] H. T. Liaw, J. H. Tsaih, and C. S. Lin, “Efficient Automatic Diagnosis of Digital Circuits,” in *Proc. ICCAD*, 1990, pp. 464–467.

[12] C. C. Lin, K. C. Chen, S. C. Chang, and M. Marek-Sadowska, “Logic Synthesis for Engineering Change,” in *Proc. Design Automation Conf. (DAC)*, 1995, pp. 647–652.

[13] B. H. Wu, C. J. Yang, C. Y. Huang, and J. H. R. Jiang, “A Robust Functional ECO Engine by SAT Proof Minimization and Interpolation Techniques,” in *International Conference on Computer-Aided Design (ICCAD)*, 2010, pp. 729–734.

[14] Vikas Rao, ”Discussions on recent automatic debugging approaches,” available at http://eng.utah.edu/~utkarshg/cex.pdf, 2018.

[15] M. Fujita, “Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design,” *Proceedings of the IEEE*, 2015.

[16] S. Jo, T. Matsumoto, and M. Fujita, “SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions,” in *IEEE 21st Asian Test Symposium*, 2012.

[17] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, “Fault Diagnosis and Logic Debugging using Boolean Satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[18] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker, “Equivalence Checking of Partial Designs Using Dependency Quantified Boolean Formulae,” in *IEEE International Conference on Computer Design (ICCD)*, 2013.

[19] B. Buchberger, “Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal,” Ph.D. dissertation, University of Innsbruck, 1965.

[20] S. Gao, A. Platzer, and E. Clarke, “Quantifier Elimination over Finite Fields with Gröbner Bases,” in *Intl. Conf. Algebraic Informatics*, 2011.

[21] E. Mastrovito, “VLSI Designs for Multiplication Over Finite Fields GF(2^m),” *Lecture Notes in CS*, vol. 357, pp. 297–309, 1989.

[22] A. Biere, “Picosat Essentials,” *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2008.

[23] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, “SINGULAR 4-1-0 — A computer algebra system for polynomial computations,” 2016.

[24] C. Koc and T. Acar, “Montgomery Multiplication in GF(2^k),” *Designs, Codes and Cryptography*, 1998.

[25] S.-i. Minato, “Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems,” in *Proceedings of the 30th International Design Automation Conference*, 1993, pp. 272–277.

Automata Learning for Symbolic Execution

Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Tappler, Johannes Winter
Graz University of Technology, Austria

Abstract—**Black-box components conceal parts of software execution paths, which makes systematic testing, e. g., via symbolic execution, difficult. In this paper, we use automata learning to facilitate symbolic execution in the presence of black-box components. We substitute black-boxes in a software system with learned automata that model them, enabling us to symbolically execute program paths that run through black-boxes. We show that applying the approach on real-world software systems incorporating black-boxes increases code coverage when compared to standard techniques.**

I. INTRODUCTION

Symbolic execution is a method to analyze software systems. It has gained attention since its introduction in the 1970s [1, 2] and is used in testing, invariant detection, model checking, and proving software correctness [3, 4, 5, 6]. Symbolic execution achieves high test coverage in a setting where the source code is completely available.

In practice, many software systems incorporate black-box components for which the source code is not available (e. g., third-party software units, hardware peripherals). A thorough behavioral analysis in the presence of black-box components is challenging using methods like symbolic execution, because black-box components conceal parts of software execution. To symbolically execute such software systems, Cadar et al. [7] proposed to replace the calls to a black-box component with calls to manually written stubs that model the component's behavior. This is a very challenging and error-prone task because either one does not have access to documentations of black-box components or this labor-intensive effort is not worth it since the resulting model will only be used once. Consequently, we often use methods that are applicable in the presence of black-boxes; e. g., random testing, or model-based testing, which requires behavioral models. Alternatively, symbolic execution may be combined with concrete execution of black-box paths, such an approach is for instance used by concolic execution [4, 8, 9].

In this paper, we use automata learning to enable symbolic execution in the presence of black-box components. Figure 1 depicts the overall execution flow for our proposed setting. Given a System Under Test (SUT), divided into a white-box and a black-box component, we learn a finite-state machine (FSM) model of the black-box component, and compose it with the white-box to generate system-level test cases via symbolic execution; i. e., we replace the black-box with its model for test-case generation. Finally, we execute the generated test cases on the original system incorporating the black-box component itself to obtain high coverage on the white-box. Testing software units in isolation often results in a broad set of test cases that are not worth the effort of manual inspection. On the other hand, systematic testing of interactive software units results in a reduced number of system-level test cases. An advantage of the proposed approach is that it enables systematic testing in the presence of black-boxes.

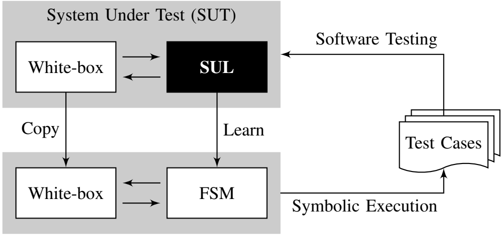


Fig. 1: System-level testing in the presence of a black-box. The system under test (SUT) comprises a white-box and a black-box component that is the system under learning (SUL).

Our approach is currently applicable to black-boxes that we can model as FSMs, and not applicable to other types of black-boxes like arithmetic functions. Moreover, there are concerns about the practicality of automata learning mostly due to the abstraction layer to counter state space explosion. Meanwhile, automata learning is successfully applied for systems with thousands of states [10] and there are techniques to support up to a million states [11]. Finally, learned FSMs of small abstract state space are shown to be sufficient for many interesting scenarios [12, 13] and this paper elaborates on one.

We built our method on top of KLEE [4], and LearnLib [14]. Applying our approach to a variety of real-world scenarios showed not only the test coverage increases, but the testing time also decreases, both compared to concolic execution. Results show coverage increase for units of interest in three real-world software systems that are dependent on an SPI controller (52.94%), an MQTT Broker (5.9% & 8.36%), and an SD-Card controller (75.36%).

Outline. This paper has the following structure. Section II summarizes automata learning and symbolic execution. Section III explains how to learn an automaton from a black-box and use it to execute the software system incorporating that black-box symbolically. Sections IV to VI provide case studies demonstrating the applicability of our approach in real-world scenarios. Section VII covers related work. Section VIII concludes and discusses future research directions.

This work was supported by the Graz University of Technology's LEAD project "Dependable Internet of Things in Adverse Environments". This work was partially supported by the ECSEL Joint Undertaking (ENABLE-S3, grant no. 692455), by the European Union (IMMORTAL, grant no. 644905).

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

II. PRELIMINARIES

A. Symbolic Execution

To infer what inputs cause which parts of a program to execute, symbolic execution assigns symbolic values to input variables and then explores the control-flow of the program [1, 2, 3, 4]. By keeping track of the program counter and constraints on symbolic input values, an execution engine discovers how inputs influence the execution path. Along each execution path, symbolic execution collects constraints from branch conditions and forms a conjunction of these constraints, called path condition. An execution path is feasible if its path condition is satisfiable; thus, a constraint solver can reveal with which input values an execution path is feasible and with which input values it is not. A feasible execution path represents multiple program runs whose concrete values satisfy the path condition; i. e., solutions to the path condition are concrete test cases.

Definition 1 (Execution State): An execution state is a triple $S = \langle \textsc{pc}, \sigma, \pi \rangle$, where $\textsc{pc}$ is the program counter, σ is a function from program variables to terms over concrete and symbolic values, and π is the path condition; i. e., a formula that imposes a set of constraints on the symbolic values.

To symbolically execute a program, symbolic execution evolves the execution state as soon as (1) an assignment is evaluated, (2) a conditional branch is evaluated, and (3) the program counter changes. Executing an assignment statement will update σ. Executing a conditional branch with the condition c duplicates the current execution state S into $S_{\textbf{true}}$ and $S_{\textbf{false}}$ and forks execution. Subsequently, the execution engine computes a symbolic formula ϑ_c from c by replacing program variables with the corresponding terms as determined by σ; next, it duplicates the path condition π for different branches and sets $\pi_{\textbf{true}} = \pi \land \vartheta_c$ and $\pi_{\textbf{false}} = \pi \land \neg\vartheta_c$. Finally, sometimes the program execution evolves through unconditional branches like `goto` statements, which affects the program counter $\textsc{pc}$ of the execution state.

To tackle the problem of symbolic execution in the presence of black-box components, one can combine symbolic and concrete execution such that whenever the program counter is leaving the program's scope symbolic values that flow through the black-box component are concretized and upon returning to the program's scope symbolic execution continues with concrete values. The approach is often called concolic execution, dynamic symbolic execution, or directed automated random testing [4, 8, 9]. In this paper, we propose an alternative approach via automata learning. For a thorough survey on symbolic execution please refer to [15].

B. Automata

Definition 2 (Finite-State Transducer): A finite-state transducer over input alphabet I and output alphabet O is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$, where Q is a nonempty set of states, q_0 is the initial state, $\delta \subseteq Q \times I \times Q$ is the transition relation, and $\lambda \subseteq Q \times I \times O$ is the output relation.

Definition 3 (Mealy Machine): A Mealy machine is a finite-state transducer $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ where its δ and λ are functions $\delta : Q \times I \to Q$ and $\lambda : Q \times I \to O$.

From this point forward, we write $q \xrightarrow{i/o} q'$ if $q' = \delta(q, i)$ and $o = \lambda(q, i)$ for Mealy machines, and if $(q, i, q') \in \delta$ and $(q, i, o) \in \lambda$ for finite-state transducers.

Definition 4 (Observation): An observation over input/output alphabet I and O is a pair $\langle \bar{\imath}, \bar{o} \rangle \in I^* \times O^*$ such that $|\bar{\imath}| = |\bar{o}|$. Given a Mealy machine $\mathcal{M}$, the set of observations of $\mathcal{M}$ from state q denoted by $obs_{\mathcal{M}}(q)$ are:

$$obs_{\mathcal{M}}(q) = \{\langle \bar{\imath}, \bar{o} \rangle \in I^* \times O^* \mid \exists q' : q \xrightarrow{\bar{\imath}/\bar{o}}{}^{*} q'\},$$

where $\xrightarrow{\bar{\imath}/\bar{o}}{}^{*}$ is the transitive and reflexive closure of the combined transition-and-output function to sequences which implies $|\bar{\imath}| = |\bar{o}|$. From this point forward, $obs_{\mathcal{M}} = obs_{\mathcal{M}}(q_0)$.

Definition 5 (Observation Equivalence): Given states $q, q' \in Q$, we define $q \approx q'$, that is q and q' are observation equivalent, only if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Given Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ over the same alphabet, $\mathcal{M}_1 \approx \mathcal{M}_2$ if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

C. Learning and Abstraction

Angluin [16] proposed an active automata learning algorithm called L^*. This algorithm learns a deterministic finite automaton accepting an unknown regular language L. It requires a *minimally adequate* teacher that needs to be able to answer two types of queries, *membership* and *equivalence* queries. First, the learner asks membership queries, checking inclusion of words in the language L. Once the learner has gained enough information to build a hypothesis automaton, it asks an equivalence query, checking whether the hypothesis accepts L. The teacher either responds with *yes*, signaling that learning was successful, or with a counterexample to equivalence. If provided with a counterexample, the learner integrates it into its knowledge and starts a new round of learning by issuing membership queries, which is concluded by an equivalence query. L^* was adapted to learn various forms of automata, including Mealy machines [17]. The basic principle remains the same, but *output queries* replace membership queries, which ask for outputs produced in response to input sequences.

To learn models of software systems, teachers are usually implemented via testing, as shown in Fig. 2 [18]. Output queries typically reset the System Under Learning (SUL), execute a sequence of inputs and collect the produced outputs. Equivalence queries can be approximated with model-based testing [19]. For that, a Conformance Testing (CT) component derives test queries from the hypothesis, which are executed to find discrepancies between SUL and hypothesis, i. e., counterexamples to observation equivalence (see Def. 4 and 5).

L^* is only affordable for small alphabets $I \cup O$; hence, Aarts et al. [20] suggested that we abstract away the concrete domain of the data, by forming equivalence classes in $I \cup O$. This is usually done by a mapper placed in between the learner and the SUL. For abstraction, the mapper maps concrete inputs I and outputs O to abstract inputs X and outputs Y.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

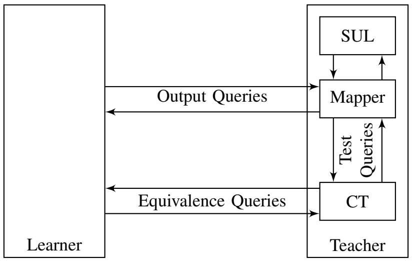


Fig. 2: Abstract automata learning through a mapper using L* (adapted from a figure in [18]).

Definition 6 (Mapper): A mapper for concrete inputs I, and concrete outputs O is a tuple $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$, where R is the set of mapper states, r_0 is the initial state, $\Delta : R \times (I \cup O) \to R$ is a transition function, X is a set of abstract inputs and Y is a set of abstract outputs, and $\nabla : (R \times I \to X) \cup (R \times O \to Y)$ is an abstraction function. From this point forward, we write $r \xrightarrow{a} r'$ if $\Delta(r, a) = r'$.

The mapper communicates with the SUL via the concrete alphabet, and with the teacher and learner via the abstract alphabet. In the setting shown in Fig. 2, the learner behaves the same as the original L* algorithm, but the teacher answers to the queries by indirectly interacting with the SUL through the mapper. Consequently, whenever the teacher receives a reset signal from the learner it resets the mapper along with the SUL to their initial states. Moreover, an individual step executing a single input and observing the output is performed as follows:

1) Given mapper's current state r, upon receiving abstract input $x \in X$, the mapper non-deterministically picks a concrete input symbol $i \in I$ such that $\nabla(r, i) = x$. If such $i \in I$ exists, then the mapper jumps to state $r' = \Delta(r, i)$ and forwards i to the SUL, otherwise it returns the output $\perp$ to the learner.
2) If the mapper has selected and forwarded an $i \in I$, then upon receiving a concrete output $o \in O$ from the SUL, the mapper forwards an abstract version $y = \nabla(r', o)$ to the learner and jumps to state $r'' = \Delta(r', o)$.

Learning an abstract Mealy machine is a slight generalization of L* [20]. From the learner's point of view nothing has changed; it learns a hypothesis $\mathcal{H}$ from observations; but it actually queries an abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ of a Mealy machine $\mathcal{M}$ induced by a mapper $\mathcal{A}$ as described by Def. 7. Meanwhile, the concretization of $\alpha_{\mathcal{A}}(\mathcal{M})$ induced by a mapper $\mathcal{A}$ is a finite-state transducer $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M}))$ defined by Def. 8.

Definition 7 (Abstraction): Let $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ be a Mealy machine, and let $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$ be a mapper. The abstraction of $\mathcal{M}$ via $\mathcal{A}$ is a finite-state transducer denoted as $\alpha_{\mathcal{A}}(\mathcal{M}) = \langle X, Y \cup \{\perp\}, Q \times R, \langle q_0, r_0 \rangle, \delta', \lambda' \rangle$, where δ' and λ' are given by the following rules:

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \nabla(r, i) = x,\ \nabla(r', o) = y}{(\langle q, r \rangle, x, \langle q', r'' \rangle) \in \delta' \wedge (\langle q, r \rangle, x, y) \in \lambda'}$$

$$\frac{\nexists i \in I : \nabla(r, i) = x}{(\langle q, r \rangle, x, \langle q, r \rangle) \in \delta' \wedge (\langle q, r \rangle, x, \perp) \in \lambda'}$$

Note that two issues may arise from abstraction. The abstraction function ∇ may be undefined for some inputs (second rule of Def. 7) and non-deterministic behavior may be introduced by the mapper. This non-determinism might occur if we have two pairs of concrete input/outputs pairs (i_1, o_1) and (i_2, o_2), observable in the same state, such that $\nabla(r, i_1) = \nabla(r, i_2)$ but $\nabla(r', o_1) \neq \nabla(r', o_2)$; i. e., the inputs map to the same abstract symbol, but the outputs map to different ones. While Aarts et al. [20] described a method to automatically refine the mapper, we manually refine it if we encounter such issues to ensure the learned model is an abstract Mealy machine.

Definition 8 (Concretization): Let $\alpha_{\mathcal{A}}(\mathcal{M}) = \langle X, Y \cup \{\perp\}, Q, q_0, \delta, \lambda \rangle$ be an abstract Mealy machine, and let $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$ be the mapper. The concretization of $\alpha_{\mathcal{A}}(\mathcal{M})$ via $\mathcal{A}$ is a finite-state transducer denoted as $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M})) = \langle I, O \cup \{\perp\}, Q \times R, \langle q_0, r_0 \rangle, \delta'', \lambda'' \rangle$ where δ'' and λ'' are given by the following rules:

$$\frac{q \xrightarrow{x/y} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \nabla(r, i) = x,\ \nabla(r', o) = y}{(\langle q, r \rangle, i, \langle q', r'' \rangle) \in \delta'' \wedge (\langle q, r \rangle, i, o) \in \lambda''}$$

$$\frac{q \xrightarrow{x/y} q',\ r \xrightarrow{i} r',\ \nabla(r, i) = x,\ \nexists o \in O : \nabla(r', o) = y}{(\langle q, r \rangle, i, \langle q, r \rangle) \in \delta'' \wedge (\langle q, r \rangle, i, \perp) \in \lambda''}$$

III. METHOD

In this section we describe our method as it is depicted in Fig. 1. First, we give an overview of the proposed configuration and then discuss the involved steps in detail. We start by learning an FSM of the black-box component with a manually defined mapper. Then, we compose this with the white-box. Finally, we execute the SUT symbolically, to generate test cases exercising as many execution paths as possible.

A. Model Learning

As described in Sect. II-C, we learn models by interacting with the SUL via a mapper performing abstraction. The concrete alphabet $I \cup O$ of the SUL generally contains input/output actions of the form $e(p_1, \ldots, p_n)$, i. e., we have input/output events $e \in E$ with n parameters. Mappers create equivalence classes of $I \cup O$ by defining constraints on parameters.

The state of the mapper comprises a fixed number of m variables recording the occurrence of events and storing action parameters. We can therefore view the mapper state as a tuple $r \in R \subseteq (E \cup \mathcal{X})^m$, where E is the set of events and $\mathcal{X}$ is a set of values relevant to the application domain, i. e., it includes the domains of the action parameters, as well as terms formed from parameter values. For the update Δ of the mapper state, we have l guarded update rules for each event e: $\Delta(\langle r_1, \ldots, r_m \rangle, e(p_1, \ldots, p_n)) = \langle r'_1, \ldots, r'_m \rangle$ *if* g_j, where the guard g_j is a quantifier-free formula over R and the parameters of e such that $\bigvee_{j=1}^{l} g_j = \top$ and $i \neq j \to g_i \wedge g_j = \perp$. Similarly, we have k guarded abstraction rules $\nabla(r, e(p_1, \ldots, p_n)) = z$ *if* g_z for each e, where z is a unique abstract symbol in $X \cup Y$.

```
Input: 1. $\mathcal{M} = \langle X, Y, Q, q_0, \delta, \lambda \rangle$,
       2. $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$
1:  function $i(p_1, \ldots, p_n)$
2:    switch $\nabla(r, i(p_1, \ldots, p_n))$ do
3:      case $x_1$
4:        $y \leftarrow \lambda(q, x_1)$
5:        $q \leftarrow \delta(q, x_1)$
6:        $r \leftarrow \Delta(r, i(p_1, \ldots, p_n))$
7:        $o(p'_1, \ldots, p'_j) \leftarrow \nabla^{-1}(r, y)$
8:        $r \leftarrow \Delta(r, o(p'_1, \ldots, p'_j))$
9:        return $o(p'_1, \ldots, p'_j)$
10:     case $x_2$
11:       $\vdots$
12: function $\nabla^{-1}(r, y)$
13:   $o(p'_1, \ldots, p'_j) \leftarrow$ $o_c$ s.t. $\nabla(r, o_c) = y$ if $g_y$
14:   for all $p' \in \{p'_1, \ldots, p'_j\}$ do
15:     makeSymbolic($p'$)
16:   assume($g_y$)
```

Fig. 3: Composition of learned model and mapper.

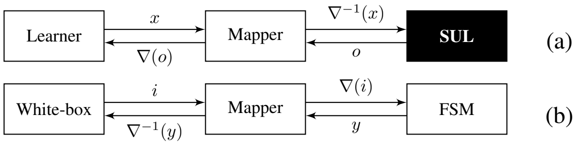


Fig. 4: Mapper in (a) learning vs. (b) symbolic execution

B. Symbolic Execution

Once an abstract model of the black-box component is learned, we compile it alongside the mapper to symbolically execute it. For that, we reverse the role of mapper as compared to learning; see Fig. 4. Therefore, we implement abstraction and concretization as described in Def. 7 and 8 via translation to source code. The interface to the translated composition of mapper and learned model consists of functions $i(p_1, \ldots, p_n)$, for each input event i, called by the white-box component. Figure 3 shows abstractly how these functions are implemented. First, we perform abstraction of inputs (Line 2). Consequently, if such a function is symbolically executed, execution initially forks to each **case**-branch and the execution engine adds the abstraction-rule guard g_x of each abstract input x to the respective path condition, thereby constraining symbolic parameters of i. After that, we update the model state (Line 5) and the mapper state (Lines 6 and 8). Finally, we return concretized outputs $o(p'_1, \ldots, p'_j)$ (Line 9). To update the mapper state, we actually need to check the guards of the update rules defining Δ. This detail is left implicit in Fig. 3.

Abstraction and updates of the state work as described by the ∇, Δ, and δ. For the concretization of an abstract output y, we need ∇^{-1}, but since ∇ performs abstraction, there is no immediate definition of ∇^{-1}. Instead, we retrieve the output event $o(p'_1, \ldots, p'_j)$ and the abstraction-rule guard g_y for y from the ∇ definition (Line 13). We declare the parameters of the output event to be symbolic values via makeSymbolic (Line 15) and through assume(g_y) we instruct the symbolic execution engine to add g_y to the path condition (Line 16). Hence, we let the execution engine find an instantiation of the output-event parameters satisfying g_y; i. e., it picks a value in O that is in the equivalence class corresponding to y.

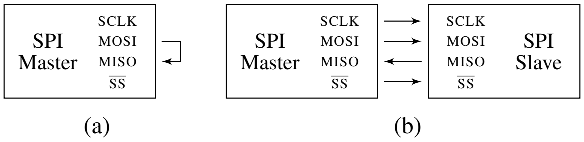


Fig. 5: Master-mode SPI peripheral in loopback along with SPI master-slave setup. SCLK is the serial clock, which is an output line of master, MOSI is the data output line from master to slave, MISO is the data output line from slave to master, and $\overline{\text{SS}}$ is the slave select and an output line of master.

```
byte tx_data = 0xdd, rx_data = 0;
LPC_SPI->CFG = SPI_CFG_MASTER | SPI_CFG_ENABLE;
while(~LPC_SPI->STAT & SPI_STAT_TXRDY);
LPC_SPI->TXDATCTL = SPI_TXDATCTL_SSEL_N(0xe)|
       SPI_TXDATCTL_FLEN(7)|SPI_TXDATCTL_EOT|tx_data;
while(~LPC_SPI->STAT & SPI_STAT_RXRDY);
rx_data = LPC_SPI->RXDAT;
if(rx_data != tx_data)
    abort();
while(~LPC_SPI->STAT & SPI_STAT_MSTIDLE);
```

Listing 1: Transmit and receive to/from slave [21, p. 349]

C. Testing

After translation, we generate system-level test cases via symbolic execution of the composition of the white-box, the learned model, and the mapper. We then run these test cases on the actual SUT, i. e., the white-box interacting with the black-box component, while profiling the observed behaviour, outputs, and executed code paths in the white-box. This step is necessary, because the learned model may not be equivalent to the black-box under abstraction. This is due to the fact that learning relies on conformance testing which is incomplete in general. Hence, running the generated test cases serves as a spuriousness check, i. e., we ensure that we will not report spurious errors, or spuriously covered code paths. Our method is therefore sound, but incomplete as it involves black-box conformance testing in the learning phase.

IV. Serial Peripheral Interface

In this section, we demonstrate how we can symbolically execute code that depends on a Serial Peripheral Interface (SPI). First, we study how to learn an SPI controller in its master-mode with a loopback setup to execute Listing 1 symbolically. Then, we show how we can extend our experiment to the whole master-slave setup of SPI.

A. Learning Master-Mode Controller of SPI

In this subsection, we symbolically execute Listing 1 that depends on the SPI bus of the NXP LPC810 micro-controller (MCU). Listing 1 drives an SPI controller in its master-mode with the purpose of sending a single byte to a slave-mode SPI controller and receiving a byte from it. The execution aborts when the received byte does not conform to the sent byte.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE I: Δ & ∇ functions of master-mode SPI mapper.

State (s)	Symbol (a)	$\Delta(s,a)$	$\nabla(s,a)$
r	`void`	r	ε
r	`read(STAT)`	r	ST
r	`read(RXDAT)`	r	RX
r	`write(TXDATCTL,`n`)`	n	TX
r	`STAT(`m`)`	r	
	if m = `0x01`		$\langle 0,0,1\rangle$
	if m = `0x03`		$\langle 0,1,1\rangle$
	if m = `0x102`		$\langle 1,1,0\rangle$
	if m = `0x103`		$\langle 1,1,1\rangle$
r	`RXDAT(n)`	r	
	if $n \neq r$		0
	if $n = r$		1

Learning. To simplify the learning, we learn the master-mode SPI controller in a loopback setup; that is, the same controller receives the transmitted byte; please see Fig. 5a. Primarily we need to know how to reset the SPI controller to its master-mode should the learner ask for a reset. In Line 2, we initialize the SPI controller to its master-mode by writing bit masks `SPI_CFG_MASTER` and `SPI_CFG_ENABLE` to `LPC_SPI->CFG` register.

Alphabet. To extract alphabets we ought to know a thing or two about the NXP LPC810 MCU. In Line 3, we read the `LPC_SPI->STAT` register and check if `SPI_STAT_TXRDY` bit is set to see if the transmission line is ready or not. The act of accessing and evaluating the value of `LPC_SPI->STAT` is an input symbol in I; accordingly, possible values for `SPI_STAT_TXRDY` (bit 0) represent outputs in O. The `STAT` register provides more SPI status flags whose possible values represent more outputs in O. Remaining SPI status flags are `SPI_STAT_RXRDY` (bit 1) and `SPI_STAT_MSTIDLE` (bit 8) [21, p. 239]. We extracted the following concrete alphabets from Listing 1:

$$I = \{\texttt{read(STAT)}, \texttt{read(RXDAT)}, \texttt{write(TXDATCTL,}n\texttt{)} \mid n \in \mathbb{N}\}\ ,$$

$$O = \{\texttt{void}, \texttt{STAT(0x01)}, \texttt{STAT(0x03)}, \texttt{STAT(0x102)}, \texttt{STAT(0x103)}, \texttt{RXDAT(}n\texttt{)} \mid n \in \mathbb{N}\}\ .$$

Mapper. We define a mapper over states $\mathbb{N} \cup \{\perp\}$ where $\perp$ is the initial state. We define the mapper's Δ and ∇ functions by Table I. The concrete values of the `STAT` register are mapped to triples $\langle$`SPI_STAT_MSTIDLE`, `SPI_STAT_RXRDY`, `SPI_STAT_TXRDY`$\rangle$.

Finally, the learning experiment results in the automaton that is depicted in Fig. 6, with which we were able to execute Listing 1 symbolically. An interesting observation that we made is according to the FSM depicted in Fig. 6, a data transmission in state s_0 triggers a state transition to state s_1 and an immediate data write to `TXDAT`, then a move to transmit holding register, and finally transmit to `RXDAT`. A subsequent data transmission results in a data write to `TXDAT`, then a move to transmit holding register. Since `RXDAT` register is occupied in s_1 the transmission to `RXDAT` never occurs, instead the Master Idle flag is cleared, indicating the transmit holding register is not empty, and current state changes to state s_2. If we do another data transmission in this state, current state changes to state s_3; where any data transmission rewrites `TXDAT` and clears Transmitter Ready flag.

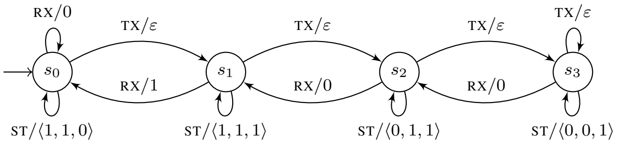


Fig. 6: Automaton of SPI controller as shown in Fig. 5a.

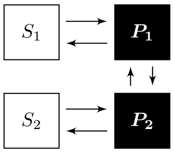


Fig. 7: Software units interacting via black-box SPI.

On the other hand, according to [21], when the transmit holding register is empty and the transmitter is not sending data the Master Idle flag (i. e., `SPI_STAT_MSTIDLE`) is set otherwise it is cleared. The Transmitter Ready flag (i. e., `SPI_STAT_TXRDY`) indicates whether data may be written to the transmit buffer or not. It is unset when writing data to `TXDAT` and set when the data is moved from the transmit buffer to the transmit shift register. The Receiver Ready flag (i. e., `SPI_STAT_RXRDY`) indicates if data is available to be read from the receiver buffer, and it is cleared after reading `RXDAT` or `RXDATSTAT`.

B. Learning Master-Slave Setup of SPI

In this subsection, we demonstrate how to generate system-level test cases for embedded software systems in the presence of a black-box communication channel.

Test Setup. Our embedded software system implements a padlock using two software units S_1 and S_2 that communicate through black-box peripherals P_1 and P_2; see Fig. 7. S_1 implements a user interface that unlocks a padlock with a 4-digit pin $p_0p_1p_2p_3$; see Listing 2. Meanwhile, S_2 implements a variant of a *combination lock automaton*; see Fig. 8. This automaton progresses on correct inputs $p_0p_1p_2p_3$ and resets otherwise. In each step, S_2 emits 1 in case of success and 0 otherwise. Finally, to check the pin, S_1 sends it to S_2.

We implemented our embedded software system on a pair of NXP LPC810 MCUs. Our port of S_1 to the primary MCU firmware (i. e., user interface) uses the SPI controller in its master-mode configuration to communicate with S_2 on the secondary MCU that uses the SPI controller in its slave-mode

```c
int main(int argc, char* argv[]) {
    do {
        int pin = getchar();
    } while(!S2.check(pin))
    grant_user_access();
    return 0;
}
```

Listing 2: S_1 runs user interface and access control.

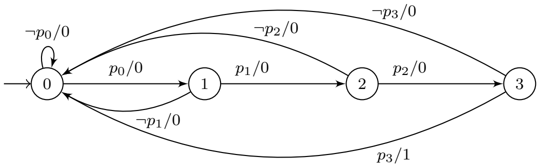


Fig. 8: S_2 runs a combination lock automaton for pin checks.

TABLE II: Code coverage in the presence of SPI controllers.

Coverage Metric	S_1		S_2	
	Concolic	Symbolic	Concolic	Symbolic
Line Coverage	84.78%	**86.96%**	38.24%	**91.18%**
Branch Coverage	57.14%	**71.43%**	25.00%	**85.00%**

configuration. Finally, due to the master/slave architecture of the SPI bus, communication is always initiated by S_1.

Alphabet. We extracted the learning alphabets as follows:

- Skimming the code from [21, p. 350] to work with an SPI in slave-mode, we extracted alphabets I_S and O_S.
- Ensuring $I_S \cap I_M = O_S \cap O_M = \emptyset$ by adding a distinguishing prefix to symbols, we fixed the alphabets I_M and O_M to work with an SPI in master-mode.
- Finally, we defined the concrete alphabets as $I = I_M \cup I_S$ and $O = O_M \cup O_S$ along with a mapper.

After roughly three hours, the experiment resulted in an automaton that models the interactive behaviour of the black-boxes shown in Fig. 7, with 348 states; i. e., $P_1 \times P_2$.

Symbolic Execution. The granting execution path in S_1 is unlikely to occur using concolic execution and random testing because it is very improbable to progress in S_2 not knowing the exact combination. On the other hand, unit-level symbolic execution of both S_1 and S_2 might reveal numerous execution paths; most of which, are not possible through interactive execution of S_1 and S_2; therefore, not worth the effort of manual inspection. Therefore, it is necessary to symbolically reason about how S_1 and S_2 restrict one another's behavior.

We symbolically executed S_1 along with S_2 interactively against the learned $P_1 \times P_2$ automaton. Symbolic execution resulted in five different execution paths almost immediately, while concolic execution through SPI communication channel only revealed one execution path after 22 hours. Table II summarizes the increase in test coverage gained by our proposed methodology against concolic execution.

V. MESSAGE QUEUING TELEMETRY TRANSPORT

Message Queuing Telemetry Transport (MQTT) is a publish-subscribe connectivity protocol for the Internet of Things. Whenever publishers publish a message to a topic, that message gets posted to a broker server. Subscribers register with the broker on a topic to receive messages published on it. Testing and verifying MQTT clients is difficult because they communicate through a black-box message broker.

Test Setup. Library implementations of the MQTT protocol specifications exist. We implemented our padlock software system using two MQTT libraries (i. e., libemqtt [22] and MQTT-C [23]) in C language. The goal is to execute the padlock software system along with the MQTT libraries against an MQTT broker symbolically; please see Fig. 9.

Test Driver. In our implementation, S_1 and S_2 agree on the MQTT Quality of Service level of 1 for a predefined topic to interact with each other. S_1 is the publisher providing the pin while S_2 is the subscriber implementing the combination lock automaton. Initially, both clients connect to the MQTT broker. Next, they exchange the pin and S_2 performs the pin check granting access to the user should the pin be correct. Finally, both disconnect from the MQTT broker.

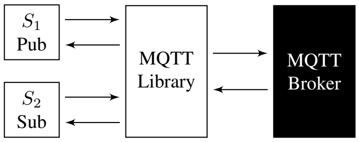


Fig. 9: Clients communicating via a broker using MQTT.

TABLE III: Δ & ∇ functions of MQTT mapper.

Source (s)	Symbol (a)	$\Delta(s,a)$	$\nabla(s,a)$
$\langle l,t,m\rangle$	$\texttt{Publish(}S_1,t',m'\texttt{)}$	$\langle l,t',m'\rangle$	Pub
$\langle l,t,m\rangle$	$\texttt{Subscribe(}S_2,t',\texttt{QoS1)}$	$\langle l\cup\{t'\},t,m\rangle$	SubQ1
$\langle l,t,m\rangle$	$\texttt{UnSubscribe(}S_2,t'\texttt{)}$	$\langle l\setminus\{t'\},t,m\rangle$	UnSub
$\langle l,t,m\rangle$	$\texttt{Receive(}S_2,t',m'\texttt{)}$	$\langle l,t,m\rangle$	
	if $(t'\in l\wedge m=m'\wedge t=t')$		Recv
	if $(t'\notin l\vee m\neq m'\vee t\neq t')$		ε
$\langle l,t,m\rangle$	everything else	$\langle l,t,m\rangle$	a

Learning. We used the learning setup configured by Tappler et al. [24] to learn the automaton of an MQTT broker. We extracted the concrete input alphabet for the learning experiment from the test driver as follows:

$$I = \{\texttt{Connect(}c\texttt{)},\texttt{Disconnect(}c\texttt{)},\texttt{Publish(}S_1,t,m\texttt{)},\\ \texttt{Subscribe(}S_2,t,\texttt{QoS1)},\texttt{UnSubscribe(}S_2,t\texttt{)}\}\ .$$

where $c \in \{S_1, S_2\}$ is the client, $t \in \mathbb{S}$ is a topic, $m \in \mathbb{S}$ is a message and $\mathbb{S}$ is the set of character strings. Moreover, in response to above input events, we observe following concrete output events; set O_{S_1} in S_1, and set O_{S_2} in S_2.

$$O_{S_1} = \{\texttt{ConnClosed(}S_1\texttt{)},\texttt{ConnAck(}S_1\texttt{)},\texttt{PubAck(}S_1\texttt{)},\texttt{void}\}\ ,\\ O_{S_2} = \{\texttt{ConnClosed(}S_2\texttt{)},\texttt{ConnAck(}S_2\texttt{)},\texttt{SubAck(}S_2\texttt{)},\\ \texttt{UnSubAck(}S_2\texttt{)},\texttt{Receive(}S_2\texttt{,Topic,Msg)},\texttt{void}\}\ .$$

Finally, since the broker triggers outputs in both clients, we define the concrete output alphabet for this experiment as

$$O = O_{S_1} \times O_{S_2}\ .$$

Mapper. The state space R of the mapper is $(2^{\mathbb{S}} \times \mathbb{S} \times \mathbb{S} \cup \{\langle\emptyset,\bot,\bot\rangle\})$ where $\langle\emptyset,\bot,\bot\rangle$ is the initial state. Each state is a triple $\langle l,t,m\rangle$ where l is the set of topics to which S_2 is subscribed, and m is the last message published to the last topic t. We define the mapper according to Table III. We learned an automaton of 10 states and 100 transitions from the EMQ broker (v. 2.3.6).

Symbolic Execution. Since KLEE does not support software sockets, we compare the coverage obtained by symbolically executing S_1 and S_2 against the learned broker automaton with that of random testing. For random testing, we generated the test data for the pin randomly and executed n^3 times as many tests as generated by symbolic execution. Table IV summarises the increase in test coverage for MQTT libraries and dismisses the coverage of S_1 and S_2 since their coverage were similar to that of the previous case study. The gap between coverage of libemqtt and MQTT-C is due to the fact that MQTT-C implements more of MQTT protocol.

ISBN: 978-0-9835678-8-2.

TABLE IV: Code coverage in the presence of a MQTT broker.

Coverage Metric	libemqtt		MQTT-C	
	Random Testing	Symbolic Execution	Random Testing	Symbolic Execution
Line Coverage	85.00%	**90.90%**	47.47%	**55.83%**
Branch Coverage	47.62%	**57.44%**	30.11%	**33.96%**

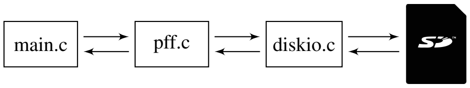


Fig. 10: Petit FAT File System.

VI. PETIT FAT FILE SYSTEM

The barrier in symbolic execution of software systems that are built on top of file systems is already addressed in KLEE [4]. KLEE models a basic file system that consists of a directory with n user-specified symbolic files. However, symbolic execution of file system implementations still remains an issue, because they are usually built using library level functionalities of disk controllers. In this section, we enable the symbolic execution of a file system that depends on a Secure Digital Card (SD-Card) controller. This helps to generate interesting test cases, which increase test coverage not only for the file system implementation, but also for the software system that is built on top of it.

Test Setup. Petit FAT File System (PFF) is an implementation of the FAT file system for 8-bit micro-controllers [25]. At the moment of writing this paper, the PFF consists of two main source files. The first source file, i. e., “diskio.c”, contains SD-Card specific code that is to be implemented based on the target MCU’s interface. The second, i. e., “pff.c”, is built on top of the first source file and implements the file system.

Test Driver. For learning and testing PFF, we used the setup shown in Fig. 10, i. e., we have diskio.c and pff.c communicating with an SD card. On top of that, we implemented a driver, i. e., “main.c”, that (1) mounts a partition, then (2) opens an arbitrary file, and eventually (3) reads 10 bytes of the file’s content and tests them against a predetermined value. Once we learned the SD-Card controller, we are able to symbolically execute not only our simple software, but also the PFF itself.

Learning. PFF uses the SD Memory Card protocol in SPI mode. The Physical Layer Simplified Specification [26] contains functional description of SD-Cards and the SD Memory Card protocol in SPI mode. By inspecting the functional description of SD-Card and PFF source code, we extracted following concrete alphabets with which the PFF can run the SD-Card communication protocol in SPI mode.

$$\begin{aligned} I = \{&\texttt{GO\_IDLE\_STATE(0x0)}, \texttt{SEND\_IF\_COND(0x1AA)}, \texttt{APP\_CMD(0x0)},\\ &\texttt{SD\_SEND\_OP\_COND(0x0)}, \texttt{SD\_SEND\_OP\_COND(1<<30)},\\ &\texttt{SEND\_STATUS(0x0)}, \texttt{READ\_SINGLE\_BLOCK(}n\texttt{)},\\ &\texttt{SD\_STATUS(0x0)}, \texttt{READ\_OCR(0x0)}\}\\ O = \{&\texttt{R1(}n\texttt{)}, \texttt{R2(}n\texttt{)}, \texttt{R3(}n\texttt{)}, \texttt{R7(}n\texttt{)}, \langle \texttt{R1(}n\texttt{)}, \texttt{DATA(}b\texttt{)}\rangle \mid n \in \mathbb{N}\}\,. \end{aligned}$$

In case of successful execution, input `READ_SINGLE_BLOCK` returns two outputs R1(n), and DATA(b) where b is a data block of size 512 bytes. Therefore, we define a compound symbol $\langle$R1(n), DATA(b)$\rangle$ in our output alphabet.

TABLE V: Abstraction for PFF.

State (s)	Symbol (a)	$\nabla(s,a)$
r_0, r_1	R1(n)	R1(n & 0x7F)
r_0, r_1	R2(n)	R2(n & 0x7FFF)
r_0	R3(n)	R3(n & 0x7FFFFFFFFF)
r_1	R3(n)	R3(n & 0x00F0000000)
r_0, r_1	R7(n)	R7(n & 0x7FF0000FFF)
r_0, r_1	$\langle$R1(n), DATA(b)$\rangle$	
	if n & 0x7F $\neq 0$	ε
	if n & 0x7F $= 0$	DATABLOCK
r_0, r_1	READ_SINGLE_BLOCK(n)	READBLOCK
r_0, r_1	everything else	a

TABLE VI: Coverage results for PFF & Certgate SD-Card.

Coverage Metric	pff.c		main.c	
	Concolic	Symbolic	Concolic	Symbolic
Line Coverage	8.53%	**83.89%**	28.57%	**100.0%**
Branch Coverage	4.41%	**55.15%**	10.00%	**100.0%**

Mapper. We define the state space R as $\{r_0, r_1\}$, and Δ by:

$$\Delta(r, a) = \begin{cases} r_1 & \text{if } a = \texttt{READ\_OCR(0x0)} \\ r_0 & \text{otherwise} \end{cases}$$

and we define the abstraction method by Table V. We ran the learning on three SD-Card controllers namely Certgate SDC, Kingston SDC, and Kingston SDHC. Although the abstract alphabet is very large, in practice we only observed 23, 51, and 44 abstract outputs for Certgate SDC, Kingston SDC, and SDHC respectively. The learned Mealy machines are of size 39, 68, and 41 states and 351, 612, and 369 transitions for Certgate SDC, Kingston SDC, and SDHC respectively.

Symbolic Execution. We ran the experiment for 24 hours using concolic execution and discovered one execution path. Meanwhile, symbolic execution increases the code coverage for both “pff.c” and “main.c” drastically; please see Table VI. Since “diskio.c” implements the interface to the black-box component we considered its code coverage to be irrelevant.

VII. RELATED WORK

Anand et al. [27] used type-dependence analysis to automatically pinpoint the variables to which the flow of symbolic values will cause a problem (e. g., parameters of black-box methods). They were able to automatically indicate problematic variables before performing symbolic execution along with contextual information that can help manual interventions. Although a first step towards coping with black-boxes in symbolic execution, a user had to implement models manually.

Cadar et al. [4] implemented 2500 lines of code to define simple models for roughly 40 system calls to model the execution environment. They also compiled and linked software systems that were built on top of the C standard library against a much more straightforward implementation (i. e., µClibc [28]) to facilitate symbolic execution of the whole software system. This manual effort is only worth for commonly used components. Moreover, since deployed software

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

systems often consist of more sophisticated implementations of components, this solution shifts system-level correctness to the correctness of handwritten models of the black-boxes.

Chipounov et al. [29] point out that manual modelling of black boxes is labor-intensive and that models are often inaccurate, especially when systems evolve. They present the S^2E platform, which avoids such problems by allowing symbolic execution of binaries, if source code is not available. In this paper, we proposed a method to symbolically execute codes that are dependent on black-boxes other than binaries.

Davidson et al. [30] encountered the same issue while extending symbolic execution to embedded platforms. They elaborated on scenarios in which the black-box is a hardware component. Not being aware of an architectural specification in the hardware component, the symbolic execution engine may follow an incorrect execution path. They manually modeled certain aspects of the hardware to facilitate symbolic execution. The problem is, architectural specifications are often abstruse, not well documented, or not published. Similarly, manual modeling of hardware components is often not practical, because it is both tedious and error-prone.

Jeon et al. [31] proposed to use program synthesis for modeling Java libraries to facilitate symbolic execution of software systems that are built on top of them. They instrumented the library source-code such that they can log simulations of tutorial programs exercising the library. Logs descriptively record either a call to or a return from a method that happened in a tutorial program discarding details of what happened inside the library after invocation. They successfully synthesized models that produced the same instantiations of design patterns as the library, should it run against the same tutorial program under the same inputs. This approach requests white-box access to the third-party components for instrumentation; moreover, it is based on instantiations of design patterns while we based our approach on finite-state machines; therefore, addressing a different and possibly broader set of components.

Godefroid et al. [8] showed that concolic execution might lead to divergence during system-level testing. Hence, the method with which concolic execution concretizes symbolic variables should be black-box specific. A program may induce exponentially many execution paths and concolic execution in a way prunes them unsystematically by replacing symbolic variables with random concrete values. This results in wandering through random execution paths pretty much like random testing; and like random testing, concolic execution also provides no sensible guarantees in terms of system-level coverage in presence of black-box components. Hence, concolic execution does not excel in presence of a black-box component whose behavior matters during path exploration.

Păsăreanu et al. [32] applied symbolic execution in unit-level testing while performing a system-level concrete execution to generate test cases that satisfy user-specified testing criteria. They outperformed random testing and manual test-case generation regarding both coverage and time. In a follow-up study, Davies et al. [33] used treatment learning to reduce number of system-level inputs that affect values of unit-level variables for a path condition of interest. Next, they applied function fitting to find a predictive relationship between the unit-level inputs and associated system-level inputs. Once they have calculated an approximation function for unit-level inputs with respect to system-level inputs, they form a higher-level path condition that also takes the approximation function (i. e., potentially interesting unit-level inputs) into account. They achieved higher coverage with fewer test cases compared to their previous study. The issue with this work is that approximated inputs of a software unit are not accurate enough to get a black-box, like a communication-protocol implementation, to run in practice.

VIII. Conclusion & Future Work

System-level test-case generation is complicated in the presence of black-box components; e. g., communication channels, communication protocols, locking mechanisms. This hardship arises from the fact that exact input values often trigger interesting behaviors of a software system, but the execution path affecting the system-level inputs is only partially visible. To cope with black-boxes, we propose to learn automata of them and instead execute software units against learned automata symbolically. Through this system-level symbolic execution, we can generate test cases for the actual software system under test. Using multiple case studies, we showed the applicability of our approach in generating test cases that cover corner cases and achieve higher coverage.

In this paper, we manually crafted mappers for our learning experiments using our own domain knowledge. Although labour intensive, mapper creation requires less effort compared to modeling systems manually, e. g., for model-based testing. Moreover, mappers are more easily reusable, e. g., [19] uses a single mapper for five different but similar systems. Additionally, we can avoid manual effort of crafting mappers for a certain class of systems through register automata learning [34], or through abstraction refinement [20, 35], which is our first direction for future research. For the second research direction, we speculate concolic execution might as well benefit from the additional information provided by the mappers; yet, we could not think of an easy way to enable that unless we assume the state space of black-box component is irrelevant. The third research direction would be to investigate how to embed the concept of time into our approach and a primary step can be extending our approach to the class of Mealy machines with timers [36]. Finally, we could extend the applicability of symbolic execution in system-level testing to a more comprehensive class of systems by investigating the possibility of approximating outputs of a black-box from its inputs using machine learning methods like treatment learning and function fitting as proposed in [33].

References

[1] J. C. King, “Symbolic Execution and Program Testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt, “SELECT—a formal system for testing and debugging programs by

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[3] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS'03*, 2003, pp. 553–568.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, 2008.

[5] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Invariant discovery guided by symbolic execution," in *Java PathFinder Workshop*, 2013.

[6] N. Tillmann and J. de Halleux, "Pex-white box test generation for .NET," in *TAP'08*, 2008, pp. 134–153.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI'05*, 2005.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272.

[10] W. Smeenk, J. Moerman, F. W. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *ICFEM'15*, 2015, pp. 67–83.

[11] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata learning with on-the-fly direct hypothesis construction," in *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, 2011, pp. 248–260.

[12] "Automata Learning benchmarks," accessed: August 24, 2018. [Online]. Available: http://automata.cs.ru.nl/

[13] B. K. Aichernig, W. Mostowski, M. R. Mousavi, M. Tappler, and M. Taromirad, "Model learning and model-based testing," in *Machine Learning for Dynamic Software Analysis: potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, 2018, pp. 74–100.

[14] M. Isberner, F. Howar, and B. Steffen, "The open-source LearnLib - a framework for active automata learning," in *CAV'15*, 2015, pp. 487–495.

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[16] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, 1987.

[17] M. Shahbaz and R. Groz, "Inferring Mealy machines," in *FM 2009*, 2009.

[18] F. W. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017.

[19] B. K. Aichernig and M. Tappler, "Learning from Faults: mutation testing in active automata learning," in *NASA Formal Methods - NFM 2017*, 2017, pp. 19–34.

[20] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012*, 2012.

[21] *LPC81x User manual*, NXP Semiconductors, 4 2014, rev. 1.6.

[22] "Libemqtt," https://github.com/menudoproblema/libemqtt, Accessed: May 1, 2018.

[23] "MQTT-C," https://github.com/LiamBindle/MQTT-C, Accessed: May 1, 2018.

[24] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing IoT communication via active automata learning," in *ICST'17, Tokyo, Japan*, 2017, pp. 276–287.

[25] "Petit FAT File System Module," http://elm-chan.org/fsw/ff/00index_p.html, accessed: May 1, 2018.

[26] *SD Specifications (Part 1) - Physical Layer Simplified Specification*, SD Association, 4 2017, ver. 6.00.

[27] S. Anand, A. Orso, and M. J. Harrold, "Type-dependence analysis and program transformation for symbolic execution," in *TACAS, Braga, Portugal, Proceedings*, 2007.

[28] "µClibc: a C library for embedded linux," https://www.uclibc.org/, accessed: May 1, 2018.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, 2012.

[30] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22th USENIX Security Symposium*, 2013, pp. 463–478.

[31] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama, "Synthesizing framework models for symbolic execution," in *ICSE'16*, 2016, pp. 156–167.

[32] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *ISSTA'08*, 2008, pp. 15–26.

[33] M. Davies, C. S. Păsăreanu, and V. Raman, "Symbolic execution enhanced system testing," in *VSTTE 2012, Philadelphia, PA, USA, Proceedings*, 2012, pp. 294–309.

[34] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Formal Asp. Comput.*, vol. 28, no. 2, pp. 233–263, 2016.

[35] F. Howar, B. Steffen, and M. Merten, "Automata learning with automated alphabet abstraction refinement," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds., 2011, pp. 263–277.

[36] B. Jonsson and F. Vaandrager, "Learning Mealy machines with timers," January 2018, preprint on webpage at www.sws.cs.ru.nl/publications/papers/fvaan/MMT/.

ISBN: 978-0-9835678-8-2.

Functional Synthesis via Input-Output Separation

Supratik Chakraborty
IIT Bombay
Mumbai, India
supratik@cse.iitb.ac.in

Dror Fried
Rice University
Houston, USA
dror.fried@rice.edu

Lucas M. Tabajara
Rice University
Houston, USA
lucasmt@rice.edu

Moshe Y. Vardi
Rice University
Houston, USA
vardi@cs.rice.edu

Abstract—**Boolean functional synthesis is the process of constructing a Boolean function from a Boolean specification that relates input and output variables. Despite significant recent developments in synthesis algorithms, Boolean functional synthesis remains a challenging problem even when state-of-the-art methods are used for decomposing the specification. In this work we bring a fresh decomposition approach, orthogonal to existing methods, that explores the decomposition of the specification into separate input and output components. We make use of an input-output decomposition of a given specification described as a CNF formula, by alternatingly analyzing the separate input and output components. We exploit well-defined properties of these components to ultimately synthesize a solution for the entire specification. We first provide a theoretical result that, for input components with specific structures, synthesis for CNF formulas via this framework can be performed more efficiently than in the general case. We then show by experimental evaluations that our algorithm performs well also in practice on instances which are challenging for existing state-of-the-art tools, serving as a good complement to modern synthesis techniques.**

I. INTRODUCTION

Boolean functional synthesis is the problem of constructing a Boolean function from a Boolean specification that describes a relation between input and output variables [2], [12], [19], [35]. This problem has been explored in a number of settings including circuit design [20], QBF solving [27], and reactive synthesis [36], and several tools have been developed for its solution. Nevertheless, scalability of Boolean functional synthesis methods remains a concern as the number of variables and size of the formula grows. This is not surprising since Boolean functional synthesis is in fact CO-NP$^{\text{NP}}$-hard.

A standard practice for handling the problem of scalability is based on decomposing the given formula into smaller sub-specifications and synthesizing each component separately [2], [19], [35]. The most common form of such decomposition, called *factorization*, is when the formula is represented as a conjunction of constraints, in which each conjunct can be seen as a sub-specification [19], [35]. The main challenge in this approach is that most factors cannot be synthesized entirely separately due to the dependencies created by shared input and output variables. The ways to meet this challenge are usually to either merge factors that share variables [35] or perform additional computations in order to combine the functions synthesized for different factors [19]. All these result in additional work that must be performed during the synthesis.

In this work, we propose an alternative decomposition framework, which follows naturally from the fact that variables in the specification are separated into input and output variables. This idea was originally inspired by [11], which explores the notion of *sequential relational decomposition*, in which a relation is decomposed into two by introducing an intermediate domain. Differently from factorization, this form of decomposition allows the two components to be synthesized completely independently. That work, however, shows that decomposition is hard in general, and if the relation is given as a Boolean circuit, decomposition is NEXPTIME-complete. Furthermore, there is no guarantee that synthesizing the two components independently would be easier than synthesizing the original specification, since the synthesis of one component might ignore useful information given by the other component.

We instead suggest a more relaxed notion of decomposition for specifications described as CNF formulas, in which every clause is split into an input and an output clause and the independent analyses of the input/output components "cooperate" to synthesize a function for the entire specification. Based on this concept, we describe a novel synthesis algorithm for CNF formulas called the "Back-and-Forth" algorithm, where rather than synthesizing the input and output components entirely independently we share information back and forth between the two components to guide the synthesis. More specifically, our algorithm alternates between SAT calls that follow the input-component structure analysis and MaxSAT calls that follow the output-component structure analysis. Thus, this approach builds on recent progress with SAT and MaxSAT solving [21], [30]. A notable consequence of our method is that, as the number of SAT calls is dependent on the structure of the input component, for specifications with some well-defined input structure we can perform synthesis in P$^{\text{NP}}$, compared to the generally mentioned CO-NP$^{\text{NP}}$-hardness. An additional advantage of our algorithm is that it constructs the synthesized function as a *decision list* [29]. Compared to other data structures for representing Boolean functions, such as ROBDDs or AIGs, decision lists have significant benefits in term of explainability, allowing domain specialists to validate and analyze their behavior (see discussion in Section VI for more details).

Work supported in part by NSF grants CCF-1319459 and IIS-1527668, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by a grant from MHRD, Govt of India, under the IMPRINT-1 scheme, and by the Brazilian agency CNPq through the Ciência Sem Fronteiras program. We thank Assaf Marron for useful discussions, and the anonymous reviewers for their suggestions. Extended version of the paper available on arXiv.

ISBN: 978-0-9835678-8-2.

We experimentally evaluate the "Back-and-Forth" algorithm on a suite of standard synthesis benchmarks, comparing its performance with that of state-of-the-art synthesis tools. Although these tools perform very well on many families of benchmarks, our results show that the "Back-and-Forth" algorithm is able to handle classes of benchmarks that these tools are unable to synthesize, indicating that it belongs in a portfolio of synthesis algorithms.

II. RELATED WORK

Constructing explicit representations of implicitly specified functions is a fundamental problem of interest to both theoreticians and practitioners. In the contexts of Boolean functional synthesis and certified QBF solving, such functions are also called *Skolem functions* [8], [14], [19]. Boole [9] and Lowenheim [22] studied variants of this problem when computing most general unifiers in resolution-based proofs. Unfortunately, their algorithms, though elegant in theory, do not scale well in practice [23]. The close relation between Skolem functions and proof objects in specialized QBF proof systems has been explored in [8], [14]. One of the earliest applications of Boolean functional synthesis has been logic synthesis - see [34] for a survey. More recently, Boolean functional synthesis has found applications in diverse areas such as temporal strategy synthesis [3], [16], [36], certified QBF solving [6], [7], [26], [28], automated program synthesis [31], [33], circuit repair and debugging [18], and the like. This has resulted in a new generation of Boolean functional synthesis tools, cf. [1], [2], [12], [14], [19], [27], [28], [35], that are able to synthesize functions from significantly larger relational specifications than what was possible a decade back.

Recent tools for Boolean functional synthesis can be broadly categorized based on the techniques employed by them. Given a specification $F(\vec{x}, \vec{y})$, where $\vec{x}$ denotes inputs and $\vec{y}$ denotes outputs, the work of [14] extracts Skolem functions for $\vec{y}$ in terms of $\vec{x}$ from a proof of validity of $\forall \vec{x}. \exists \vec{y}. F(\vec{x}, \vec{y})$ expressed in a specific format. The efficiency of this technique crucially depends on the existence and size of a proof in the required format. *Incremental determinization* [27] is a highly effective synthesis technique that accepts as input a CNF representation of a specification and builds on several successful heuristics used in modern conflict-driven clause-learning (CDCL) SAT solvers [30].

In [12], the composition-based synthesis approach of [17] is adapted and new heuristics are proposed for synthesizing Skolem functions from an ROBDD representation of the specification. The technique has been further improved in [35] to work with factored specifications represented as implicitly conjoined ROBDDs. CEGAR-based techniques that use modern SAT solvers as black boxes [1], [2], [19] have recently been shown to scale well on several classes of large benchmarks. The idea behind these techniques is to start with an efficiently computable initial estimate of Skolem functions, and use a SAT solver to test if the estimates are correct. A satisfying assignment returned by the solver provides a counterexample to the correctness of the function estimates, and can be used to iteratively refine the estimates. In [1], it is shown that transforming the representation of the specification to a special negation normal form allows one to efficiently synthesize Skolem functions.

Both ROBDD and CEGAR-based approaches make use of decomposition techniques to improve performance, the most common of which is *factorization* [19], [35]. In this method, every conjunct of a conjunctive specification is considered individually. The main drawback in this approach is that the dependencies between conjuncts limit how much each of them can be analyzed independently of the others, requiring either partially combining components, as in [35], or going through a process of refinement of the results [19]. This issue motivates the search for alternative notions of decomposition for synthesis problems. Our approach is loosely inspired by the idea of *sequential relational decomposition* explored in depth in [11]. A more direct application of this idea to synthesis might still be possible, but requires further exploration. In addition to the above techniques, templates or sketches have been used to synthesize functions when information about the possible functional forms is available a priori [32], [33].

As is clear from above, several orthogonal techniques have been found to be useful for the Boolean functional synthesis problem. In fact, there remain difficult corners, where the specification is stated simply, and yet finding Skolem functions that satisfy the specification has turned out to be hard for all state-of-the-art tools. Our goal in this paper is to present a new technique and algorithm for this problem, that does not necessarily outperform existing techniques on all benchmarks, but certainly outperforms them on instances in some of these difficult corners. We envisage our technique being added to the existing repertoire of techniques in a portfolio Skolem-function synthesizer, to expand the range of problems that can be solved.

III. PRELIMINARIES

A. Boolean Functional Synthesis

A specification for the Boolean functional synthesis problem is a (quantifier-free) Boolean formula $F(\vec{x}, \vec{y})$ over *input variables* $\vec{x} = (x_1, \ldots, x_m)$ and *output variables* $\vec{y} = (y_1, \ldots, y_n)$. Note that F can be interpreted as a relation $F \subseteq X \times Y$, where X is the set of all assignments $\hat{x}$ to $\vec{x}$ and Y is the set of all assignments $\hat{y}$ to $\vec{y}$. With that in mind, we denote by $Dom(F) = \{\hat{x} \mid \exists \hat{y}.(F(\hat{x}, \hat{y}) = 1)\}$ and $Img(F) = \{\hat{y} \mid \exists \hat{x}.(F(\hat{x}, \hat{y}) = 1)\}$ the domain and image of the relation represented by F. We also use $Img_{\hat{x}}(F) = \{\hat{y} \mid F(\hat{x}, \hat{y}) = 1\}$ to denote the image of a specific element $\hat{x} \in X$. If $Dom(F) = X$, then we say that F is *realizable*.

Two Boolean formulas $F(\vec{w})$ and $F'(\vec{w})$ are said to be *logically equivalent*, denoted by $F \equiv F'$, if they have the same solution space; that is, for every assignment $\hat{w}$ to $\vec{w}$, $F(\hat{w}) = 1$ iff $F'(\hat{w}) = 1$. Unless stated otherwise, all Boolean formulas mentioned in this work are quantifier free.

We say that a partial function $g : X \to Y$ *implements* a relation $F \subseteq X \times Y$ if for every $\hat{x} \in Dom(F)$ we have that $(\hat{x}, g(\hat{x})) \in F$. Such a g is also called a *Skolem function* of F.

Note that if F is realizable, then g is a total function. Finally, we define the *Boolean-synthesis problem* as follows:

Problem 1. *Given a specification $F(\vec{x},\vec{y})$, construct a partial function g that implements F.*

For more information on Boolean synthesis, see [12], [19].

B. Decision lists

Our choice of representation of Skolem functions in this work is inspired by the idea that we can represent an arbitrary Boolean function f by a *decision list* [29]. A decision list is an expression of the form `if` $f_1(\vec{x})$ `then` $\hat{y}_1$ `else if` $f_2(\vec{x})$ `then` $\hat{y}_2$ `else ... else` $\hat{y}_k$, where each f_i is a formula in terms of the input variables $\vec{x}$ and each $\hat{y}_i$ is an assignment to the output variables $\vec{y}$. The length k of the list corresponds to the number of decisions. Clearly, for a specification $F(\vec{x},\vec{y})$ with m input variables we can always synthesize as an implementation a decision list of length 2^m, where for every possible assignment of $\vec{x}$ we choose an assignment of $\vec{y}$ that satisfies the specification. Many specifications, however, can be implemented by significantly smaller decision lists, by taking advantage of the fact that multiple inputs can be mapped to the same output. Our analysis identifies and exploits these cases.

Despite being a natural representation, decision lists might not be appropriate for a physical implementation of the synthesized function as a circuit. In this case, it might make sense to collect the decisions into a more compact representation, such as an ROBDD.

C. Conjunctive Normal Form

A Boolean formula $F(\vec{w})$ is in *conjunctive normal form* (CNF) if F is a conjunction of clauses $C_1 \wedge \ldots \wedge C_k$, where every clause C_i is a disjunction of literals (a variable or its negation). A subset S of the clauses of a CNF formula F is *satisfiable* if there exists an assignment $\hat{w}$ to the variables $\vec{w}$ in F such that $C_i(\hat{w}) = 1$ for every clause $C_i \in S$. Similarly, a subset S of the clauses of F is *all-falsifiable* if there exists an assignment $\hat{w}$ such that $C_i(\hat{w}) = 0$ for every clause $C_i \in S$. A subset S of clauses is a *maximal satisfiable subset* (MSS) if S is satisfiable and every superset $S' \supset S$ is unsatisfiable. Similarly, S is a *maximal falsifiable subset* (MFS) if S is all-falsifiable and every superset $S' \supset S$ is not all-falsifiable. For more information on MSS and MFS, refer to [15].

IV. SYNTHESIS VIA INPUT-OUTPUT SEPARATION

In this section, we present a novel algorithm for Boolean functional synthesis from CNF specifications. Our approach is based on a separation of every clause into an input part and an output part. First, we describe how a decision list implementing the specification can be constructed by enumerating MFSs of the input clauses, or similarly by enumerating MSSs of the output clauses. Then, we show how we can benefit from alternating between the two: the MFSs can be used to avoid useless MSSs, while the MSSs can be used to cover multiple MFSs at the same time without enumerating all of them.

Given a CNF formula $F(\vec{x},\vec{y})$, assume $F(\vec{x},\vec{y}) = \bigwedge_{i=1}^{k} C_i$, where $C_1,\ldots,C_k$ are clauses over $\vec{x}$ and $\vec{y}$. Let $C_i|_{\vec{x}}$ denote the x-part of clause C_i, that is, the disjunction of all x literals in C_i. Similarly, let $C_i|_{\vec{y}}$ be the y-part of clause C_i, the disjunction of all y literals in C_i. We call $S_{\vec{x}} = \{C_i|_{\vec{x}} \mid C_i \text{ is a clause in } F\}$ and $S_{\vec{y}} = \{C_i|_{\vec{y}} \mid C_i \text{ is a clause in } F\}$ the set of input and output clauses of the specification, respectively.

In the following sections, we describe how to perform separate analyses of the input component $S_{\vec{x}}$ and the output component $S_{\vec{y}}$, and then how to combine these analyses into a single synthesis algorithm that alternates between the two components.

A. Analysis of the Input Component

In this subsection we assume that the specification F is realizable. First, consider a single assignment $\hat{x}$ to the input variables $\vec{x}$. Let $Fals(\hat{x}) = \{C_i|_{\vec{x}} \in S_{\vec{x}} \mid C_i|_{\vec{x}}(\hat{x}) = 0\}$ be the subset of input clauses that $\hat{x}$ falsifies. For a set $S'_{\vec{x}} \subseteq S_{\vec{x}}$ of input clauses, let $Co(S'_{\vec{x}}) = \{C_i|_{\vec{y}} \in S_{\vec{y}} \mid C_i|_{\vec{x}} \in S'_{\vec{x}}\}$ be the corresponding set of output clauses and let $MustSat(\hat{x}) = Co(Fals(\hat{x}))$. Note that $C_i \equiv (C_i|_{\vec{x}} \vee C_i|_{\vec{y}}) \equiv (\neg C_i|_{\vec{x}} \rightarrow C_i|_{\vec{y}})$ for every clause C_i. Therefore $MustSat(\hat{x})$ is the subset of output clauses that must be satisfied in order to satisfy F when $\hat{x}$ is the input assignment.

A key observation is that for two different input assignments $\hat{x}$ and $\hat{x}'$, if $Fals(\hat{x}') \subseteq Fals(\hat{x})$, then $MustSat(\hat{x}') \subseteq MustSat(\hat{x})$, and therefore every output assignment $\hat{y}$ that satisfies the specification for $\hat{x}$ also satisfies the specification for $\hat{x}'$. Hence, it is enough to consider only assignments for $\vec{x}$ that falsify a maximal number of input clauses. This leads to the following lemma:

Lemma 1. *Let $M_{\vec{x}}$ be an MFS of $S_{\vec{x}}$, and $\hat{y}$ be an assignment that satisfies $Co(M_{\vec{x}})$. Then: (1) For every assignment $\hat{x}$ such that $Fals(\hat{x}) \subseteq M_{\vec{x}}$, the assignment $(\hat{x},\hat{y})$ satisfies $F(\vec{x},\vec{y})$; and (2) There is no assignment $\hat{x}$ such that $Fals(\hat{x}) \supset M_{\vec{x}}$.*

Proof. (1) For every clause $C_i|_{\vec{x}} \in Fals(\hat{x})$, since $C_i|_{\vec{x}} \in M_{\vec{x}}$, we have that $C_i|_{\vec{y}}$ is in $Co(M_{\vec{x}})$ and therefore is satisfied by $\hat{y}$. Therefore, every clause C_i in $F(\vec{x},\vec{y})$ that is not satisfied by $\hat{x}$ is satisfied by $\hat{y}$. Note that (2) follows from $M_{\vec{x}}$ being maximal. ☐

From Lemma 1 and our assumption that $F(\vec{x},\vec{y})$ is realizable, we can conclude the following.

Corollary 1. *F can be implemented by a decision list of length equal to the number of MFS of $S_{\vec{x}}$, where each f_i in the decision list is of size linear in the size of the specification.*

Proof. Construct $f_i(\vec{x})$ by taking the conjunction of all input clauses $C|_{\vec{x}}$ not contained in the i-th MFS M_i. Then, $f_i(\vec{x})$ is satisfied exactly by those assignments $\hat{x}$ such that $Fals(\hat{x})$ is a subset of M_i. Then, set the corresponding output assignment $\hat{y}_i$ to an arbitrary satisfying assignment of $Co(M_i)$. ☐

Example 1. *Let $F(x_1,x_2,y_1,y_2) = (x_1 \vee \neg x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \neg y_1) \wedge (x_2 \vee y_1 \vee \neg y_2) \wedge (\neg x_1 \vee x_2 \vee y_2)$. We first construct*

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

input clauses $S_{\vec{x}} = \{(x_1 \vee \neg x_2), (x_1 \vee x_2), (x_2), (\neg x_1 \vee x_2)\}$ *and output clauses* $S_{\vec{y}} = \{(y_1), (\neg y_1), (y_1 \vee \neg y_2), (y_2)\}$. $S_{\vec{x}}$ *has three MFS:* $\{(x_1 \vee \neg x_2)\}$, $\{(x_1 \vee x_2), (x_2)\}$ *and* $\{(x_2), (\neg x_1 \vee x_2)\}$. *From these MFS we can construct a decision list implementing* F *in the way described above. Note that this decision list necessarily covers every possible input assignment:*

```
if $(x_1 \vee x_2) \wedge (x_2) \wedge (\neg x_1 \vee x_2)$ then $(y_1 := 1; y_2 := 0)$
else if $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ then $(y_1 := 0; y_2 := 0)$
else if $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ then $(y_1 := 1; y_2 := 1)$
```

Note that we require $F(\vec{x}, \vec{y})$ to be realizable because otherwise we cannot guarantee that $Co(M_{\vec{x}})$ will be satisfiable for every MFS $M_{\vec{x}}$ of the input clauses. If $Co(M_{\vec{x}})$ is unsatisfiable, however, it is not enough to simply remove the corresponding $f_i(\vec{x})$ from the decision list, because there might be a subset $M'_{\vec{x}} \subset M_{\vec{x}}$ for which $Co(M'_{\vec{x}})$ is satisfiable.

This is the first time to our knowledge that MFS are used for synthesis purposes. An advantage of enumerating MFS is that finding an MFS can be easily done, in a precise sense discussed below. One way to do this is through the *conflict graph* of the set of input clauses [13]. Given a set of clauses S, the conflict graph of S is the graph where every vertex corresponds to a clause in S, and there is an edge between two vertices iff the corresponding clauses have a complementary pair of literals between them (that is, the same variable appears in positive form in one clause and in negative form in the other). The complement of the conflict graph is called a *consensus graph* [13].

Since two clauses can be falsified at the same time iff there is no edge between them in the conflict graph, or alternatively there is an edge between them in the consensus graph, there is a one-to-one correspondence between MFS of the set of clauses, maximal independent sets (MIS) in the conflict graph, and maximal cliques in the consensus graph. Therefore, we can enumerate the MFS in a set of clauses by either enumerating MIS in the conflict graph or maximal cliques in the consensus graph. The benefit of this reduction is that maximal cliques display a so called *polynomial-time listability*, meaning that finding a maximal clique can be performed in polynomial time, and therefore enumeration takes polynomial time in the number of maximal cliques [15].

This relation between the set of MFS and maximal cliques implies that the size of the smallest decision list that implements a given specification is upper bounded by the number of maximal cliques in the consensus graph of the input clauses. Therefore we have the following result.

Theorem 1. *Synthesis can be performed in* P^{NP} *for specifications for which the consensus graph of* $S_{\vec{x}}$ *has a polynomial number of maximal cliques (such as planar or chordal graphs).*

Proof. Given a specification F, construct the consensus graph of the input component, enumerate the maximal cliques and for each one use a SAT solver to obtain a corresponding satisfying assignment for the output clauses. Since the number of maximal cliques is polynomial, only a polynomial number of SAT calls is required. $\square$

Theorem 1 demonstrates an improvement relative to the general CO-$\mathrm{NP}^{\mathrm{NP}}$-hardness of synthesis. Moreover, constructing the consensus graph of the input component is easy, as is testing for certain graph properties, such as planarity, that ensure a small number of maximal cliques. Therefore, Theorem 1 provides an elegant method of deciding whether synthesis can be performed efficiently in practice before even beginning the synthesis process.

To summarize this section, the analysis of the input component provides two insights. First, a decision list implementing the specification can be constructed from the list of MFS of the input clauses. Second, analyzing the graph structure of the input component allows us to identify classes of specifications for which synthesis can be performed more efficiently. Note that this analysis, however, does not take into account the properties of the output component, and as such the decision list produced by ignoring the output component may be longer than necessary. With that in mind, the next section presents a complementary analysis of the output component that can help to produce a smaller decision list.

B. Analysis of the Output Component

For the analysis of the output component, consider the set $MustSat(\hat{x})$, defined in the previous subsection, of output clauses that must be satisfied when $\hat{x}$ is the input assignment. Then for every two input assignments $\hat{x}$ and $\hat{x}'$, if $MustSat(\hat{x}') \subseteq MustSat(\hat{x})$, every output assignment $\hat{y}$ that satisfies the specification for $\hat{x}$ also satisfies the specification for $\hat{x}'$. Therefore, it is enough when constructing the decision list to consider only those satisfiable subsets of $S_{\vec{y}}$ that are of maximal size. Similarly to Lemma 1 in the previous section, this insight allows us to state the following lemma:

Lemma 2. *Let* $M_{\vec{y}}$ *be an MSS of* $S_{\vec{y}}$ *and* $\hat{y}$ *be an assignment that satisfies* $M_{\vec{y}}$. *Then: (1) for every assignment* $\hat{x}$ *such that* $MustSat(\hat{x}) \subseteq M_{\vec{y}}$, *the assignment* $(\hat{x}, \hat{y})$ *satisfies* $F(\vec{x}, \vec{y})$; *and (2) for every assignment* $\hat{x}$ *such that* $MustSat(\hat{x}) \supset M_{\vec{y}}$, *there is no* $\hat{y}'$ *such that the assignment* $(\hat{x}, \hat{y}')$ *satisfies* $F(\vec{x}, \vec{y})$.

Proof. (1) Since $\hat{y}$ satisfies every clause $C_i|_{\vec{y}}$ in $M_{\vec{y}}$, it must be that $\hat{y}$ also satisfies every clause in $MustSat(\hat{x})$. Therefore, for every clause C_i in F, either $C_i|_{\vec{x}}$ is satisfied by $\hat{x}$ (and therefore $C_i|_{\vec{y}} \notin MustSat(\hat{x})$) or $C_i|_{\vec{y}}$ is satisfied by $\hat{y}$. Therefore $(\hat{x}, \hat{y})$ satisfies $F(\vec{x}, \vec{y})$. (2) Since $M_{\vec{y}}$ is maximal, then in this case $MustSat(\hat{x})$ must be unsatisfiable. Therefore there is no $\hat{y}'$ that can satisfy all clauses that $\hat{x}$ does not already satisfy. $\square$

Therefore, similarly to the analysis of the input component, we have:

Corollary 2. F *can be implemented by a decision list of length equal to the number of MSS of* $S_{\vec{y}}$, *where each* f_i *in the decision list is of size linear in the size of the specification.*

Proof. Construct $f_i(\vec{x})$ by taking the conjunction of all input clauses $C|_{\vec{x}}$ such that $C|_{\vec{y}}$ is not contained in the i-th MSS M_i. Then, $f_i(\vec{x})$ is satisfied exactly by those assignments $\hat{x}$ such that $MustSat(\hat{x})$ is a subset of M_i. Then, set the corresponding output assignment $\hat{y}_i$ to an arbitrary satisfying assignment of M_i. $\square$

Example 2. *Let F, $S_{\vec{x}}$ and $S_{\vec{y}}$ be the same as in Example 1. $S_{\vec{y}}$ has three MSS: $\{(y_1), (y_1 \vee \neg y_2), (y_2)\}$, $\{(\neg y_1), (y_1 \vee \neg y_2)\}$ and $\{(\neg y_1), (y_2)\}$. From these MSS we can construct a decision list implementing F in the way described above. Note that some decisions in the list might be redundant:*

`if` $(x_1 \vee x_2)$ `then` $(y_1 := 1; y_2 := 1)$

`else if` $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ `then` $(y_1 := 0; y_2 := 0)$

`else if` $(x_1 \vee \neg x_2) \wedge (x_2)$ `then` $(y_1 := 0; y_2 := 1)$

Unlike the input component, the output analysis does not require the specification to be realizable to produce the correct answer: for every input $\hat{x}$ for which an output $\hat{y}$ exists, $MustSat(\hat{x})$ will be contained in some MSS, and therefore will be covered by the decision list. On the other hand, we do not care about the case where an input $\hat{x}$ has no corresponding output $\hat{y}$. Note, however, that unlike the input component, we do not have here a simple graph structure that can be exploited to obtain the list of MSSs, and finding an MSS is clearly NP-hard. Therefore, it is unlikely for us to be able to efficiently identify instances where the number of MSS is polynomial.

More importantly, however, is that taking into account only the output component and ignoring the input component may also lead to a large decision list that includes many MSSs that would never be activated by an input. This fact emphasizes the drawbacks of independent synthesis of the components, and motivates the development of an algorithm that combines the input and output analyses to produce a decision list that is smaller than either of the ones produced by each analysis individually.

C. Alternating between Input and Output Components

Our next goal is to combine the input and output analyses obtained so far into a synthesis procedure that constructs a decision list of length upper-bounded by the minimum among the number of MFS of the input clauses and the number of MSS of the output clauses. Due to the restrictions of the input analysis, if the specification is unrealizable the procedure terminates without producing a decision list. Extending the synthesis to unrealizable specifications is left for future work. We first state the following lemma:

Lemma 3. *If $F(\vec{x}, \vec{y})$ is realizable, then for every MFS $M_{\vec{x}}$ of $S_{\vec{x}}$, $Co(M_{\vec{x}}) \subseteq M_{\vec{y}}$ for some MSS $M_{\vec{y}}$ of $S_{\vec{y}}$.*

Proof. For every MFS $M_{\vec{x}}$, since $M_{\vec{x}}$ is all-falsifiable, there exists an input assignment $\hat{x}$ such that $Fals(\hat{x}) = M_{\vec{x}}$. Then, since F is realizable, $MustSat(\hat{x}) = Co(M_{\vec{x}})$ is satisfiable, and therefore is contained in some MSS. $\square$

Given an MFS $M_{\vec{x}}$ for the input clauses, we say that an MSS $M_{\vec{y}}$ for the output clauses *covers* $M_{\vec{x}}$ if $Co(M_{\vec{x}}) \subseteq M_{\vec{y}}$.

Algorithm 1 Back-and-Forth synthesis algorithm combining MFS and MSS analysis.

```
 1: initialize a list of MSSs $L$ to the empty list
 2: while there are still MFS left to generate do
 3:     $M_{\vec{x}} \leftarrow$ MFS of $S_{\vec{x}}$ not covered by any MSS in $L$
 4:     if MSS $M_{\vec{y}} \subseteq S_{\vec{y}}$ covering $M_{\vec{x}}$ exists then
 5:         add $M_{\vec{y}}$ to $L$
 6:     else
 7:         FAIL: specification is unrealizable
 8:     end if
 9: end while
10: construct decision list from $L$
```

Lemma 3 says that for every MFS $M_{\vec{x}}$, there exists at least one MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. Therefore, instead of producing a satisfying assignment for $Co(M_{\vec{x}})$, we can produce a satisfying assignment for $M_{\vec{y}}$. In fact, such satisfying assignment also takes care of every other MFS covered by $M_{\vec{y}}$, making it unnecessary to generate them.

The above insight gives rise to Algorithm 1, which we call the "Back-and-Forth" algorithm. In this algorithm, we maintain a list L of MSSs that is initially empty. At every iteration of the algorithm, we produce a new MFS that is not covered by the MSSs already in L. Then, we find an MSS that covers this new MFS. If no such MSS exists, it means the specification is unrealizable, and so the algorithm emits an error message and terminates. Otherwise, we add this MSS to L. After all the MFS have been covered, we construct a decision list from the obtained list L of MSS in the same way as described in Section IV-B: $f_i(\vec{x})$ is a formula that is satisfied exactly when $MustSat(\vec{x})$ is a subset of the i-th MSS, and the corresponding output assignment $\hat{y}_i$ is a satisfying assignment for that MSS.

Example 3. *Let F, $S_{\vec{x}}$ and $S_{\vec{y}}$ be the same as in Examples 1 and 2. In the first iteration, we generate the MFS $M^1_{\vec{x}} = \{(x_1 \vee \neg x_2)\}$. Then, we expand $Co(M^1_{\vec{x}}) = \{(y_1)\}$ into the MSS $M^1_{\vec{y}} = \{(y_1), (y_1 \vee \neg y_2), (y_2)\}$ and add $M^1_{\vec{y}}$ to L. Note that $M^1_{\vec{y}}$ also covers, besides $M^1_{\vec{x}}$, the MFS $\{(x_2), (\neg x_1 \vee x_2)\}$, and therefore this MFS will not need to be generated. The only remaining MFS is $M^2_{\vec{x}} = \{(x_1 \vee x_2), (x_2)\}$. $M^2_{\vec{y}} = Co(M^2_{\vec{x}}) = \{(\neg y_1), (y_1 \vee \neg y_2)\}$ is already an MSS, so we add it to L. Since all MFS have been covered, the procedure terminates. Note that we did not need to add the MSS $\{(\neg y_1), (y_2)\}$ to L, since no MFS is covered by this MSS. From L, we can now construct a decision list as described earlier:*

`if` $(x_1 \vee x_2)$ `then` $(y_1 := 1; y_2 := 1)$

`else if` $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ `then` $(y_1 := 0; y_2 := 0)$

a) Implementation details: The key steps of Algorithm 1 are the generation of the MFS $M_{\vec{x}}$ in line 3 and the MSS $M_{\vec{y}}$ in line 4. These steps are similar to the input and output analyses in Sections IV-A and IV-B. Since, however, we use communication between the input and output components, we have additional constraints on the MFS and MSS being generated. At each step the generated MFS must not be covered by the previously-generated MSSs, and the generated MSS must cover the most recently generated MFS.

While generating an arbitrary MFS can be done in polynomial time, we prove that adding the restriction that the MFS must not be covered by a previous MSS makes the MFS generation an NP-complete problem (see extended version of the paper for proper theorem and proof). Therefore, we implement the MFS generation in the following way. First, we use a SAT solver as an NP oracle to find an (not-necessarily maximal) all-falsifiable subset of $S_{\vec{x}}$ not covered by the previous MSSs. Then, we extend this subset to an MFS by iterating over the remaining input clauses and at each step adding to the growing set a clause that does not conflict with the clauses already present in that set. This process of obtaining an MFS from $S_{\vec{x}}$ is easier to implement when we use the conflict graph representation of $S_{\vec{x}}$. Given k previous MSSs $M_1, \ldots, M_k$ and the conflict graph $G = (V, E)$, we use the following SAT query to generate an all-falsifiable subset:

$$\varphi \equiv \bigwedge_{i=1}^{k} \left(\bigvee_{C_j|_{\vec{y}} \in S_{\vec{y}} \setminus M_i} z_j \right) \wedge \bigwedge_{(C_i|_{\vec{x}}, C_j|_{\vec{x}}) \in E} (\neg z_i \vee \neg z_j)$$

We use variable z_i to indicate whether clause $C_i|_{\vec{x}}$ is present in the all-falsifiable subset. The first conjunction encodes that for every previous MSS, the subset must include a clause $C_j|_{\vec{x}}$ not covered by that MSS. The second conjunction expresses that if two clauses conflict with each other, they cannot both be added to the subset. Note that whenever we generate a new MFS, we only need to add extra clauses of the first form to this query, allowing us to employ incremental capabilities of SAT solvers.

After extending the subset produced by the SAT solver to an MFS $M_{\vec{x}}$, we have to generate a new MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. For that we use a partial MaxSAT solver as an oracle. In a partial MaxSAT problem, some clauses are set as hard clauses and others are set as soft clauses [4]. The solver then returns an assignment that satisfies all hard clauses and the maximum possible number of soft clauses. We call the MaxSAT solver on the set of output clauses $S_{\vec{y}}$, where the clauses in $Co(M_{\vec{x}})$ are set as hard clauses, and all other clauses are set as soft clauses. This way, the MaxSAT solver is guaranteed to return a satisfiable set of clauses containing $Co(M_{\vec{x}})$ and of maximum size. Since a satisfiable subset of maximum size is necessarily maximal, the satisfied clauses returned by the MaxSAT solver is an MSS, as desired.

b) Analysis and Correctness: Since exactly one new MFS and one new MSS are generated at every iteration, the number of iterations in Algorithm 1 is upper bounded by $\min(\#MFS, \#MSS)$. Yet, since Algorithm 1 does not generate redundant MFS and MSS, the number of iterations, and thus the size of the decision list, can be much smaller.

We now formalize and prove the correctness of Algorithm 1.

Lemma 4. *For a realizable specification $F(\vec{x}, \vec{y})$, let $\langle (f_1, \hat{y}_1), \ldots, (f_k, \hat{y}_k) \rangle$ be the decision list produced by Algorithm 1. Then (1) For every $\hat{x}$ such that $f_i(\hat{x}) = 1$, $F(\hat{x}, \hat{y}_i) = 1$; (2) For every $\hat{x}$ there is at least one i such that $f_i(\hat{x}) = 1$.*

Proof. (1) Let $M_{\vec{y}}$ be the i-th MSS generated by the algorithm. Then, by construction, $f_i(\hat{x}) = 1$ iff $MustSat(\hat{x}) \subseteq M_{\vec{y}}$, and $\hat{y}_i$ is a satisfying assignment to $M_{\vec{y}}$. Therefore, if $f_i(\hat{x}) = 1$ then $\hat{y}_i$ satisfies $MustSat(\hat{x})$, and so $(\hat{x}, \hat{y}_i)$ satisfies F.

(2) For every $\hat{x}$, there exists an MFS $M_{\vec{x}}$ such that $Fals(\hat{x}) \subseteq M_{\vec{x}}$. If $M_{\vec{x}}$ was generated by the algorithm, then an MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$ was added to the MSS list. If $M_{\vec{x}}$ was not generated by the algorithm, it must be because there was already a previously generated MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. Either way, since $M_{\vec{y}}$ covers $M_{\vec{x}}$ and $Fals(\hat{x}) \subseteq M_{\vec{x}}$, $M_{\vec{y}}$ covers $Fals(\hat{x})$. Therefore, the corresponding f_i in the decision list is such that $f_i(\hat{x}) = 1$. □

From Lemma 4 we obtain the following corollary.

Corollary 3. *Given a realizable specification $F(\vec{x}, \vec{y})$, the decision list produced by Algorithm 1 implements F.*

It is worth noting that if the number of MFS is small as discussed in Section IV-A, then purely enumerating MFS, as in Section IV-A can be theoretically faster than using Algorithm 1. That is because finding an MFS can be done in polynomial time, while Algorithm 1 requires calls to a SAT and MaxSAT solvers. In practice, however, we observed that the Back-and-Forth algorithm often avoids a large number of redundant MFS, which makes up for the extra complexity in generating each MFS. Still, for specifications that are known to have a small number of MFS, restriction to the analysis of the input component as in Section IV-A can be sufficient.

D. Partitioning the Specification into Distinct Output Variables

Some of the cases in the back-and-forth analysis which cause the number of MFS or MSS to be exponential can be simplified by partitioning the specification into sets of clauses that do not share output variables. As an example, consider the specification for the identity function:

$$F(\vec{x}, \vec{y}) = (x_1 \leftrightarrow y_1) \wedge \ldots \wedge (x_k \leftrightarrow y_k)$$

or in a CNF form:

$$F(\vec{x}, \vec{y}) = (\neg x_1 \vee y_1) \wedge (x_1 \vee \neg y_1) \wedge \ldots \wedge (\neg x_k \vee y_k) \wedge (x_k \vee \neg y_k)$$

It is easy to see that both the number of MFS and MSS for this formula are 2^k. Each output variable, however, does not appear in the same clause with other output variables. Therefore, we can consider each pair $(\neg x_i \vee y_i) \wedge (x_i \vee \neg y_i)$ of clauses as a separate specification and synthesize it independently as a decision list of size 2. As such, the total number of MFS and MSS grow linearly with k.

Therefore we propose the following preprocessing step.

1) Given the specification F, construct a graph with a vertex for each clause and an edge between two vertices iff the corresponding clauses share an output variable.

2) Separate the graph into connected components $\mathbb{C}_1, \ldots, \mathbb{C}_k$. Note that the $\mathbb{C}_i$ are completely disjoint in terms of output variables.
3) For every $\mathbb{C}_i$, define a sub-specification F_i by taking only the clauses in F whose corresponding vertex is in $\mathbb{C}_i$.
4) Call Algorithm 1 for each specification F_i. This gives us a decision list D_i for F_i that decides on an assignment for only the output variables in F_i.

Since the F_i have disjoint sets of output variables, every D_i decides on an assignment for a different partition of output variables. Therefore, given an input $\hat{x}$ we can produce a corresponding output $\hat{y}$ by simply evaluating each D_i independently on $\hat{x}$ and combining the results.

V. EXPERIMENTAL EVALUATION

In order to evaluate the performance of the Back-and-Forth synthesis algorithm, we ran the algorithm on benchmarks from the 2QBF track of the QBFEVAL'16 QBF-solving competition [25]. This track is composed of QBF benchmarks of the form $\forall \vec{x}. \exists \vec{y}. F(\vec{x}, \vec{y})$, where F is a CNF formula. We can see these benchmarks as synthesis problems asking if we can synthesize a Skolem function for the existential variables in terms of the universal variables such that the formula F is satisfied. For this experimental evaluation we used only those benchmarks that are realizable, since adjusting the Back-and-Forth algorithm to handle unrealizable benchmarks is future work. The benchmarks can be classified into seven families: MUTEXP (7 instances), QSHIFTER (6 instances), RANKINGFUNCTIONS (49 instances), REDUCTIONFINDING (34 instances), SORTINGNETWORKS (22 instances), TREE (5 instances) and FIXPOINTDETECTION (93 instances). Because benchmarks in the same family tend to have similar properties, it makes sense to evaluate performance over each family, rather than over specific instances.

We compared the running time of the Back-and-Forth algorithm on these benchmarks with three state-of-the-art tools that employ different synthesis approaches: the CDCL-based CADET [27], the ROBDD-based RSynth [35], and the CEGAR-based BFSS [1]. Since the Back-and-Forth algorithm, CADET and RSynth are all sequential algorithms, to ensure fair comparison of computational effort, the version of BFSS used was compiled with the MiniSAT SAT solver [10] instead of the parallelized UniGen sampler used in [1]. We leave for future work the exploration of performance of the different tools in a parallel scenario.

Our implementation of the Back-and-Forth algorithm used the Glucose SAT solver [5], based on MiniSAT, and the Open-WBO MaxSAT solver [24]. The implementation also used the partitioning described in Section IV-D. All experiments were executed in the DAVinCI cluster at Rice University, consisting of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8 GB of RAM per core. Our algorithm has not been parallelized, so the cluster was solely used to run multiple experiments simultaneously. Each instance had a timeout of 8 hours.

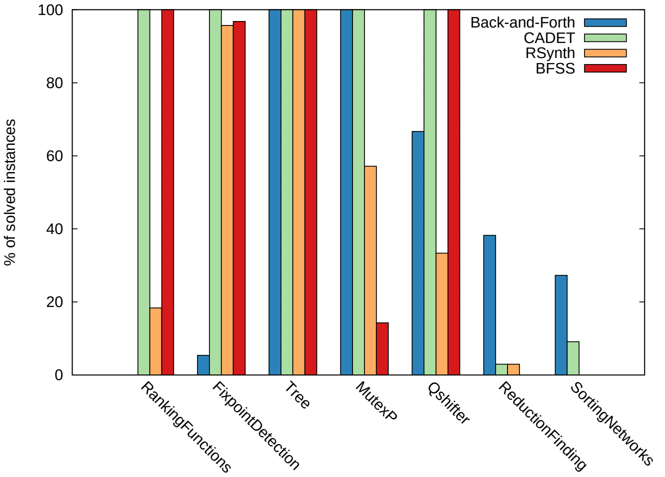


Fig. 1. Percentage of instances solved by each synthesis algorithm for each of the benchmark families.

Figure 1 shows for each family the percentage of instances each tool was able to solve in the time limit. We can divide the results into three parts:

In the RANKINGFUNCTIONS and FIXPOINTDETECTION families the Back-and-Forth algorithm timed out on almost all instances, only being able to solve the easiest instances of FIXPOINTDETECTION. CADET, on the other hand, performed very well, being able to solve all instances. RSynth and BFSS also outperformed the Back-and-Forth algorithm, although they did not perform as well as CADET.

The TREE, MUTEXP, and QSHIFTER families had almost all instances solved by the Back-and-Forth algorithm in under 45 seconds (except for the two hardest instances of QSHIFTER, which timed out), in many cases outperforming RSynth or BFSS. Even so, CADET still performed the best in these classes, solving all instances faster than our algorithm.

Lastly, REDUCTIONFINDING and SORTINGNETWORKS seem to be the most challenging families for existing tools, with CADET only being able to solve two instances in total, RSynth one, and BFSS none. In contrast, our Back-and-Forth algorithm solved 13 cases in REDUCTIONFINDING and 6 in SORTINGNETWORKS. Furthermore, as can be seen in Figure 2, every instance that was solved by other tools was also solved by the Back-and-Forth algorithm, which was faster by over an order of magnitude.

In summary, the Back-and-Forth algorithm performed competitively in 5 out of 7 families, and was strictly superior in 2 out of 7 families. Due to the difficulty of analyzing CNF formulas, the exact reason why the algorithm performs well in these particular families and not in others remains an open question, to be explored in future work. Still, the results suggest that the Back-and-Forth algorithm can serve as a good complement to modern synthesis tools, performing well exactly in the cases in which these tools struggle the most, and therefore it would be a good candidate for membership in

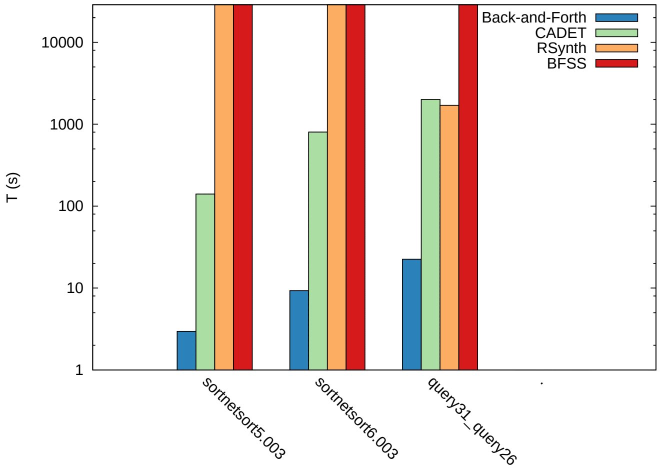


Fig. 2. Running time, in seconds, of each synthesis algorithm on instances of the ReductionFinding and SortingNetworks families that were solved by at least one algorithm besides Back-and-Forth. Bars of maximum height indicate the algorithm timed out on the benchmark.

a portfolio of synthesis algorithms.

VI. DISCUSSION

A recurrent observation in recent evaluations [1], [2], [19], [35] of Boolean functional synthesis tools has been that no single tool or algorithm dominates the others in all classes of benchmarks. To build industry-strength Boolean functional solvers, it is therefore inevitable that a portfolio approach be adopted. Since decomposition-based techniques (beyond factored specifications) have not been used in existing tools so far, our original motivation was to develop a decomposition-centric framework for Boolean functional synthesis that complements (rather than dominates) the strengths of existing tools. As our experiments with the Back-and-Forth algorithm show, we have been able to take the first few steps in this direction by successfully solving some classes of benchmarks that state-of-the-art tools choke on. While we have tried to understand features of these benchmarks that make them particularly amenable to our technique, a lot more work remains to be done to elucidate this relation clearly.

Yet another motivation for exploring a decomposition-centric synthesis approach was to be able to generate Skolem functions in a format that lends itself to easy independent validation by domain experts. Interestingly, despite the singular importance of this aspect, it has been largely ignored by existing Boolean functional synthesis tools, most of which construct a circuit representation of the function using an acyclic-graph data structure such as an ROBDD or an And-Inverter Graph. While these are known to be efficient representations of Boolean functions, they are not amenable to easy validation by a domain expert, especially when their sizes are large, often requiring a satisfiability solver to check that the generated Skolem functions indeed satisfy the specifications. Synthesizing functions as decision lists is a natural and well-studied choice for meeting this objective. Along with each decision in the decision list, we can also identify the clauses that contribute to the generation of the outputs (these are clauses whose input components are falsified by the decision), thereby providing clues about which part of the specification is responsible for the outputs generated in a particular branch of the decision-list representation. Our work shows that decomposition-based techniques lend themselves easily to such representations.

In order to be consistent with performance comparison experiments reported in the literature, all specifications used in our evaluation were prenex CNF (PCNF) formulas taken from the QBFEVAL'16 benchmark suite. While this certainly presents challenging instances of Boolean functional synthesis, PCNF is not a natural choice of representing specifications in several important application areas. For example, the industry standard (IEC 1131-3) for reactive programs for programmable logic controllers (PLC) includes a set of languages that allow the user to specify combinations of outputs based on different combinations of input conditions. The same is also true in the specification of several bus protocols like the VME Bus or AMBA Bus. Scenario-based specifications such as these are much more amenable to our decomposition-based approach, since there is a natural separation of input and output components of the specification. In addition, with such specifications, it is meaningful to analyze the structure of dependence between the input and output components, and exploit structural properties (viz. the size of the MIS in the conflict graph as explained in Section IV) in synthesis. We believe that as we look beyond PCNF representations of specifications, techniques like those presented in this paper will be even more useful in a portfolio approach to synthesis.

In our experimental evaluation, we chose CADET as a representative of the state-of-the-art on Boolean synthesis stemming from the QBF community. This is due to its focus on 2QBF (which suffices for Boolean synthesis of realizable specifications) and its performance on recent QBFEVAL competitions. Another certifying QBF solver, CAQE [28], uses techniques that are similar to the clause splitting used in our algorithm. But CAQE targets QBF instances with arbitrary quantifier alternation, requiring additional mechanisms for handling these cases, and furthermore does not perform the same analysis as here, based on MFS and MSS. Due to their similarities, it would be interesting to perform a comparison between the two algorithms in the future.

Finally, the techniques presented in this work are clearly not the only ways to achieve synthesis via decomposition, and there exists scope for significant innovation and creativity, both in the manner in which a specification is decomposed, and in the way the decomposition is exploited to arrive at an efficient synthesis algorithm. One example lies in identifying algorithms for sequential decomposition, as presented in [11], which are applicable to a synthesis context. In summary, synthesis based on input-output decomposition presents uncharted territory that deserves systematic exploration in order to complement the strengths of existing synthesis tools.

REFERENCES

[1] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah. What's Hard About Boolean Functional Synthesis? In *Computer Aided Verification - 30th International Conference, CAV 2018*, pages 251–269, 2018.

[2] S. Akshay, S. Chakraborty, A. K. John, and S. Shah. Towards Parallel Boolean Functional Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, pages 337–353, 2017.

[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic Computational Techniques for Solving Games. *STTT*, 7(2):118–128, 2005.

[4] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, pages 427–440, 2009.

[5] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 399–404, 2009.

[6] V. Balabanov and J.-H. R. Jiang. Unified QBF Certification and Its Applications. *Form. Methods Syst. Des.*, 41(1):45–65, Aug. 2012.

[7] V. Balabanov and J. R. Jiang. Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In *Computer Aided Verification - 23rd International Conference, CAV 2011*, pages 149–164, 2011.

[8] V. Balabanov, M. Widl, and J. R. Jiang. QBF Resolution Systems and Their Proof Complexities. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 154–169, 2014.

[9] G. Boole. *The Mathematical Analysis of Logic*. Philosophical Library, 1847.

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing - SAT 2003 - 6th International Conference*, pages 502–518, 2003.

[11] D. Fried, A. Legay, J. Ouaknine, and M. Y. Vardi. Sequential Relational Decomposition. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 432–441, 2018.

[12] D. Fried, L. M. Tabajara, and M. Y. Vardi. BDD-Based Boolean Functional Synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016*, pages 402–421, 2016.

[13] R. Ganian and S. Szeider. New Width Parameters for Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference*, pages 38–52, 2017.

[14] M. Heule, M. Seidl, and A. Biere. Efficient Extraction of Skolem Functions from QRAT Proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2014*, pages 107–114, 2014.

[15] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal Falsifiability - Definitions, Algorithms, and Applications. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19*, pages 439–456, 2013.

[16] S. Jacobs, R. Bloem, R. Brenguier, R. Könighofer, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker. The Second Reactive Synthesis Competition (SYNTCOMP 2015). In *Proceedings Fourth Workshop on Synthesis, SYNT 2015*, pages 27–57, 2015.

[17] J. R. Jiang. Quantifier Elimination via Functional Composition. In *Computer Aided Verification, 21st International Conference, CAV 2009*, pages 383–397, 2009.

[18] S. Jo, T. Matsumoto, and M. Fujita. SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions. In *Proceedings of the 2012 IEEE 21st Asian Test Symposium*, ATS '12, pages 19–24. IEEE Computer Society, 2012.

[19] A. K. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay. Skolem Functions for Factored Formulas. In *Formal Methods in Computer-Aided Design, FMCAD 2015*, pages 73–80, 2015.

[20] J. H. Kukula and T. R. Shiple. Building Circuits from Relations. In *Computer Aided Verification, 12th International Conference, CAV 2000*, pages 113–123, 2000.

[21] C. M. Li and F. Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 613–631. 2009.

[22] L. Lowenheim. Über die Auflösung von Gleichungen in Logischen Gebietkalkul. *Math. Ann.*, 68:169–207, 1910.

[23] E. Macii, G. Odasso, and M. Poncino. Comparing Different Boolean Unification Algorithms. In *Proceedings of 32nd Asilomar Conference on Signals, Systems and Computers*, pages 17–29, 2006.

[24] R. Martins, V. M. Manquinho, and I. Lynce. Open-WBO: A Modular MaxSAT Solver. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 438–445, 2014.

[25] M. Narizzano, L. Pulina, and A. Tacchella. The QBFEVAL web portal. In *Logics in Artificial Intelligence*, pages 494–497. Springer Berlin Heidelberg, 2006.

[26] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Bierre. Resolution-Based Certificate Extraction for QBF - (Tool Presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, pages 430–435, 2012.

[27] M. N. Rabe and S. A. Seshia. Incremental Determinization. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference*, pages 375–392, 2016.

[28] M. N. Rabe and L. Tentrup. CAQE: A Certifying QBF Solver. In *Formal Methods in Computer-Aided Design, FMCAD 2015*, pages 136–143, 2015.

[29] R. L. Rivest. Learning Decision Lists. *Machine Learning*, 2(3):229–246, 1987.

[30] J. P. M. Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, pages 131–153. 2009.

[31] A. Solar-Lezama. Program Sketching. *STTT*, 15(5-6):475–495, 2013.

[32] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 281–294, 2005.

[33] S. Srivastava, S. Gulwani, and J. S. Foster. Template-Based Program Verification and Program Synthesis. *STTT*, 15(5-6):497–518, 2013.

[34] L. M. Tabajara. BDD-Based Boolean Synthesis. Master's thesis, Rice University, 2018.

[35] L. M. Tabajara and M. Y. Vardi. Factored Boolean Functional Synthesis. In *Formal Methods in Computer Aided Design, FMCAD 2017*, pages 124–131, 2017.

[36] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi. Symbolic LTLf Synthesis. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*, pages 1362–1369, 2017.

ISBN: 978-0-9835678-8-2.

Learning Linear Temporal Properties

Daniel Neider
Max Planck Institute for Software Systems
67663 Kaiserslautern, Germany
Email: neider@mpi-sws.org

Ivan Gavran
Max Planck Institute for Software Systems
67663 Kaiserslautern, Germany
Email: gavran@mpi-sws.org

***Abstract*—We present two novel algorithms for learning formulas in Linear Temporal Logic (LTL) from examples. The first learning algorithm reduces the learning task to a series of satisfiability problems in propositional Boolean logic and produces a smallest LTL formula (in terms of the number of subformulas) that is consistent with the given data. Our second learning algorithm, on the other hand, combines the SAT-based learning algorithm with classical algorithms for learning decision trees. The result is a learning algorithm that scales to real-world scenarios with hundreds of examples, but can no longer guarantee to produce minimal consistent LTL formulas. We compare both learning algorithms and demonstrate their performance on a wide range of synthetic benchmarks. Additionally, we illustrate their usefulness on the task of understanding executions of a leader election protocol.**

I. INTRODUCTION

Making sense of the observed behavior of complex systems is an important problem in practice. It arises, for instance, in debugging (especially in the context of distributed systems), reverse engineering (e.g., of malware and viruses), specification mining for formal verification, and modernization of legacy systems, to name but a few examples. However, understanding a system based on examples of its execution is clearly a challenging task that can quickly become overwhelming without proper tool support.

In this paper, we address this problem and develop learning-based techniques to help engineers understand the dynamic (i.e., temporal) behavior of complex systems. More precisely, we solve the problem of learning formulas in Linear Temporal Logic (LTL) [1], which are meant to distinguish between desirable and undesirable executions of a system (e.g., to explain the root-cause of a bug). The particular choice of LTL in this work is motivated by two observations: first, logical formulas often provide concise descriptions of the observed behavior and are relatively easy for humans to comprehend; second, LTL—together with Computational Tree Logic (CTL) [2]—is widely considered to be the de facto standard for specifying temporal properties and, hence, many engineers are familiar with its use.

The precise problem we are aiming at is the following: given a sample $\mathcal{S}$ consisting of two finite sets of positive and negative examples, learn an LTL formula φ that is consistent with $\mathcal{S}$ in the sense that all positive examples satisfy φ, whereas all negative examples violate φ.[1] To be as general and succinct as possible, we here consider examples to be infinite, ultimately periodic words (e.g., traces of a non-terminating system) and assume the standard syntax of LTL. However, our techniques can easily be adapted to the case of finite words and extend smoothly to arbitrary future-time temporal operators, such as "release", "weak until", and so on. We fix all necessary definitions and notations in Section II.

The main contribution of this work are **two novel learning algorithms for LTL formulas from data**, one based on SAT solving, the other on learning decision trees.

SAT-based learning algorithm: The idea of our first algorithm, presented in Section III, is to reduce the problem of learning an LTL formula to a series of satisfiability problems in propositional Boolean logic and to use highly-optimized SAT solvers to search for solutions. Inspired by ideas from bounded model checking [10], our learning algorithm produces a series of propositional formulas $\Phi_n^{\mathcal{S}}$ for increasing values of $n \in \mathbb{N} \setminus \{0\}$ that depend on the sample $\mathcal{S}$ and have the following two properties: (1) $\Phi_n^{\mathcal{S}}$ is satisfiable if and only if there exists an LTL formula of size n (i.e., with n subformulas) that classifies the examples correctly, and (2) a model of $\Phi_n^{\mathcal{S}}$ contains sufficient information to construct such an LTL formula. By increasing the value of n until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, we obtain an effective algorithm that learns an LTL formula that is guaranteed to classify the examples correctly (given that the sample is non-contradictory).

By design, our SAT-based learning algorithm has three distinguished features, which we believe are essential in practice. First, our algorithm learns LTL formulas of minimal size (i.e., with the minimal number of subformulas). As we seek to learn formulas to be read by humans, the size of the learned formula is a crucial metric since larger formulas are generally harder to understand than smaller ones. Second, once an LTL formula has been learned, our algorithm can be queried for further, distinct formulas that are consistent with the sample. We believe that this feature is important in practice as it allows generating multiple explanations for the observed data. Third, our algorithm does not rely on an a priori given

---

[1] Note that, in contrast to classical computational learning theory [3] and modern statistical machine learning [4], [5], we seek to learn a formula that does not make mistakes on the examples. In fact, separation problems of this sort are of great interest in automata and formal language theory. Prominent examples in this area are the minimization of incompletely-specified state machines [6], [7] and Regular Model Checking [8], [9].

ISBN: 978-0-9835678-8-2.

set of templates, which is in stark contrast to existing work on learning temporal properties (e.g., Bombara et al. [11]). To the best of our knowledge, our SAT-based algorithm is in fact the first learning algorithm that is not restricted to a fixed class of templates. However, restrictions to the shape of LTL formulas (e.g., to the popular GR(1)-fragment of LTL [12]) can easily be encoded if desired.

Learning algorithm based on decision trees: Our second learning algorithm, which we present in Section IV, trades in the guarantee of finding minimal solutions in order to attain better scalability. The key idea is to perform the learning in two phases. In the first phase, we run the SAT-based learning algorithm described above on various subsets of the examples. This results in a (small) number of LTL formulas, named "LTL primitives", that classify at least these subsets correctly. In the second phase we use a standard learning algorithm for decision trees [13] to learn a Boolean combination of these LTL primitives that classifies the whole set of examples correctly, though it might not be minimal. Note, however, that we need to carefully choose the subsets of examples such that the resulting LTL primitives (a) separate all pairs of positive and negative examples and (b) are general enough to permit "small" decision trees. We have experimented with numerous strategies to select subsets, but in this paper we present only the two that performed best. A well known advantage of decision trees is that they are simple to comprehend due to their rule-based structure.

In Section V, we evaluate the performance of both learning algorithms on a wide range of synthetic benchmarks that reflect typical patterns of LTL formulas used in practice. Additionally, we illustrate their usefulness for understanding causes of inconsistencies in the leader election used by Zookeeper's atomic broadcast protocol [14].

Details and proofs omitted due to space constraints can be found in an extended version of this paper [15].

Related Work

Learning of temporal properties from examples has recently attracted increasing interest, especially in the area of *Signal Temporal Logic (STL)* [16] and *parametric STL* [17]. Examples include the work by Asarin et al. [17], Kong et al. [18], [19], Vaidyanathan et al. [20], and Bartocci, Bortolussi, and Sanguinetti [21]. In contrast to our SAT-based learning algorithm, however, all of these techniques either rely on user-given templates or can only learn formulas from very restricted syntactic fragments. Various techniques for mining LTL specifications [22], [23] and CTL specifications [24] exist as well, but these also rely on templates or restrict the class of formulas severely. To the best of our knowledge, our SAT-based algorithm is in fact the first that is capable of learning unrestricted LTL formulas without relying on user-given templates. Nonetheless, expert knowledge in form of constraints on the syntax can easily be encoded if desired.

Our SAT-based learning algorithm is inspired by bounded model checking [10] and earlier work of the first author on learning (minimal) automata over finite words [7], [9]. However, since regular languages are strictly more expressive than LTL (the former being equivalent to monadic second-order logic [25], while the latter being equivalent to fist-order logic [26]), automata learning techniques—including active learning algorithms [27], [28] that operate in Angluin's active learning framework [29]—are not immediately applicable. However, lifting the methods developed in this work to an active learning setup, without a detour via automata, is part of our plans for future work.

Using decision trees to learn Signal Temporal Logic (STL) formulas has been explored by Bombara et al. [11], whose main contribution is an adaptation of the classical impurity measure to account for STL formulas. However, this work still requires user-defined STL primitives to be provided, which serve as the features for the decision tree learning algorithm. By contrast, our technique uses the SAT-based learning algorithm to infer LTL primitives fully automatically.

Learning of logical formulas has also been studied in the context of *probably approximately correct learning (PAC)* [3]. Grohe and Ritzert [30], for instance, considered learning of first-order definable concepts over structures of small degree. Subsequently, Grohe, Löding, and Ritzert [31] studied the learning of hypotheses definable using monadic second order logic on strings. Due to the fundamental differences between PAC learning and the learning model considered here (one being approximate and the other being exact), their techniques cannot easily be applied.

II. PRELIMINARIES

In this section, we set up definitions and notations used throughout the paper.

Finite and Infinite Words: An *alphabet* Σ is a nonempty, finite set. The elements of this set are called *symbols*.

A *finite word* over an alphabet Σ is a sequence $u = a_0 \dots a_n$ of symbols $a_i \in \Sigma$, $i \in \{0, \dots, n\}$. The empty sequence is called *empty word* and written as ε. The length of a finite word u is denoted by $|u|$, where $|\varepsilon| = 0$. Moreover, Σ^* denotes the set of all finite words over the alphabet Σ, while $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of all non-empty words.

An *infinite word* over Σ is an infinite sequence $\alpha = a_0 a_1 \dots$ of symbols $a_i \in \Sigma$, $i \in \mathbb{N}$. We denote the i-th symbol of an infinite word α by $\alpha(i)$ and the infinite suffix starting at position j by $\alpha[j, \infty)$. Given $u \in \Sigma^+$, the infinite word $u^\omega = uu \dots \in \Sigma^\omega$ is the infinite repetition of u. An infinite word α is called *ultimately periodic* if it is of the form $\alpha = uv^\omega$ for a $u \in \Sigma^*$ and $v \in \Sigma^+$. Finally, Σ^ω denotes the set of all infinite words over the alphabet Σ.

Propositional Boolean Logic: Let *Var* be a set of propositional variables, which take Boolean values from $\mathbb{B} = \{0, 1\}$ (0 representing *false* and 1 representing *true*). Formulas in *propositional (Boolean) logic*—which we denote by capital Greek letters—are inductively constructed as follows:

- each $x \in \mathit{Var}$ is a propositional formula; and
- if Ψ and Φ are propositional formulas, so are $\neg\Psi$ and $\Psi \vee \Phi$.

ISBN: 978-0-9835678-8-2.

Moreover, we add syntactic sugar and allow the formulas true, false, $\Psi \land \Phi$, $\Psi \Rightarrow \Phi$, and $\Psi \Leftrightarrow \Phi$, which are defined as usual.

A *propositional valuation* is a mapping $v \colon \mathit{Var} \to \mathbb{B}$, which maps propositional variables to Boolean values. The semantics of propositional logic is given by a satisfaction relation $\models$ that is inductively defined as follows: $v \models x$ if and only if $v(x) = 1$, $v \models \neg\Psi$ if and only if $v \not\models \Psi$, and $v \models \Psi \lor \Phi$ if and only if $v \models \Psi$ or $v \models \Phi$. In the case that $v \models \Phi$, we say that v *satisfies* Φ and call it a *model* of Φ. A propositional formula Φ is *satisfiable* if there exists a model v of Φ. The *size* of a formula is the number of its subformulas (as defined in the usual way).

The satisfiability problem of propositional logic is the problem to decide whether a given formula is satisfiable. Although this problem is well-known to be NP-complete [32], modern SAT solvers implement optimized decision procedures that can check satisfiability of formulas with millions of variables [33]. Moreover, SAT solvers also return a model if the input-formula is satisfiable.

Linear Temporal Logic: Linear Temporal Logic (LTL) [1] is an extension of propositional Boolean logic with modalities that allow expressing temporal properties. Starting with a finite, nonempty set $\mathcal{P}$ of *atomic propositions*, formulas in LTL—which we denote by small Greek letters—are inductively defined as follows:

- each atomic proposition $p \in \mathcal{P}$ is an LTL formula;
- if ψ and φ are LTL formulas, so are $\neg\psi$, $\psi \lor \varphi$, $\mathsf{X}\,\psi$ ("next"), and $\psi \,\mathsf{U}\, \varphi$ ("until").

Again, we add syntactic sugar and allow the formulas $\mathit{true} := p \lor \neg p$ for some $p \in \mathcal{P}$, $\mathit{false} := \neg\mathit{true}$, as well as $\psi \land \varphi$ and $\psi \to \varphi$, which are defined as usual. Moreover, we allow the additional temporal formulas $\mathsf{F}\,\psi := \mathit{true} \,\mathsf{U}\, \psi$ ("finally") and $\mathsf{G}\,\psi := \neg\,\mathsf{F}\,\neg\psi$ ("globally"). The *size* of an LTL formula φ, which we denote by $|\varphi|$, is the number of its subformulas. Finally, let $\mathcal{C} = \{\land, \lor, \neg, \to, \mathsf{F}, \mathsf{G}, \mathsf{U}, \mathsf{X}\}$ be the set of LTL operators.

LTL formulas are interpreted over infinite words $\alpha \in (2^{\mathcal{P}})^\omega$, though there exist various semantics for LTL over finite words and our techniques smoothly extend to these situations. For the sake of a simpler presentation, we define the semantics of LTL in a slightly non-standard way by means of a *valuation function* V. This functions maps pairs of LTL formulas and infinite words to Boolean values and is inductively defined as follows: $V(p, \alpha) = 1$ if and only if $p \in \alpha(0)$, $V(\neg\varphi, \alpha) = 1 - V(\varphi, \alpha)$, $V(\varphi \lor \psi, \alpha) = \max\{V(\varphi, \alpha), V(\psi, \alpha)\}$, $V(\mathsf{X}\,\varphi, \alpha) = V(\varphi, \alpha[1, \infty))$, and $V(\varphi \,\mathsf{U}\, \psi, \alpha) = \max_{i \geq 0}\{\min\{V(\psi, \alpha[i, \infty)), \min_{0 \leq j < i}\{V(\varphi, \alpha[j, \infty))\}\}\}$. We call $V(\varphi, \alpha)$ the *valuation of φ on α* and say that α *satisfies* φ if $V(\varphi, \alpha) = 1$.

Our SAT-Based learning algorithm relies on a canonical syntactic representation of LTL formulas, which we call *syntax DAGs*. A syntax DAG is essentially a syntax tree (i.e., the unique tree labeled with atomic propositions as well as Boolean and temporal operators that is derived from the inductive definition of an LTL formula) in which common subformulas are shared. This sharing turns the syntax tree into a directed, acyclic graph (DAG), whose number of nodes coincides with the number of subformulas of the represented LTL formula. As an example, Figure 1b (on Page 4) depicts the (unique) syntax DAG of the formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \lor (\mathsf{F}\,\mathsf{G}\, q)$, in which the subformula $\mathsf{G}\, q$ is shared; the corresponding syntax tree is depicted in Figure 1a. Note that syntactically distinct formulas have different (i.e., non-isomorphic) syntax DAGs.

Samples and Consistency: Throughout this paper, we assume that the data we learn from is given as two (potentially empty) finite, disjoint sets $P, N \subset (2^{\mathcal{P}})^\omega$ of ultimately periodic words. The words in P are interpreted as *positive examples*, while the words in N are interpreted as *negative examples*. We call the pair $\mathcal{S} = (P, N)$ a *sample*. Since we want to work with the ultimately periodic words in a sample algorithmically, we assume that they are stored as pairs (u, v) of finite words $u \in (2^{\mathcal{P}})^*$ and $v \in (2^{\mathcal{P}})^+$, which can be accessed individually. To measure the complexity of a sample, we define its *size* to be $|\mathcal{S}| = \sum_{uv^\omega \in P \cup N} |u| + |v|$.

Given an LTL formula φ and a sample $\mathcal{S} = (P, N)$, both over a set $\mathcal{P}$ of atomic propositions, we call φ *consistent* with $\mathcal{S}$ if $V(\varphi, uv^\omega) = 1$ for each $uv^\omega \in P$ (i.e., all positive examples satisfy φ) and $V(\varphi, uv^\omega) = 0$ for each $uv^\omega \in N$ (i.e., all negative examples do not satisfy φ); in this case, we also say that φ *separates* P and N. We call φ *minimally consistent with* $\mathcal{S}$ if φ is consistent with $\mathcal{S}$ and no consistent LTL formula of smaller size exists.

III. A SAT-BASED LEARNING ALGORITHM

The fundamental task we solve in this section is:

> *"given a sample $\mathcal{S}$, compute an LTL formula of minimal size that is consistent with $\mathcal{S}$".*

We call this task *passive learning of LTL formulas*—as opposed to active learning [29] where the learning algorithm is permitted to actively query for additional data. Note that this problem can have more than one solution as there can be multiple, non-equivalent LTL formulas that are minimally consistent with a given sample.

Before we explain our learning algorithm in detail, let us briefly comment on the minimality requirement in the definition above. On the one hand, we observe that the problem becomes simple if no restriction on the size is imposed: for $\alpha \in P$ and $\beta \in N$, construct a formula $\varphi_{\alpha,\beta}$ with $V(\varphi_{\alpha,\beta}, \alpha) = 1$ and $V(\varphi_{\alpha,\beta}, \beta) = 0$ that describes the first symbol where α and β differ using a sequence of X-operators and an appropriate propositional formula; then, $\bigvee_{\alpha \in P} \bigwedge_{\beta \in N} \varphi_{\alpha,\beta}$ is consistent with $\mathcal{S}$ since we assume P and N to be disjoint. However, simply characterizing all differences between positive and negative examples is clearly overfitting the sample and, hence, arguably of little help in practice. On the other hand, we believe that small formulas are easier for humans to comprehend than large ones, which justifies spending effort on learning a smallest formula. However, we do not impose any preference amongst minimal consistent formulas (which is an interesting topic for future work).

Let us now turn to describing our learning algorithm. Its underlying idea is to reduce the construction of a minimally consistent LTL formula to a satisfiability problem in propositional logic and use a highly-optimized SAT solver to search for solutions. More precisely, given a sample $\mathcal{S}$ and a natural number $n \in \mathbb{N} \setminus \{0\}$, we construct a propositional formula $\Phi_n^{\mathcal{S}}$ of size polynomial in n and $|\mathcal{S}|$ that has the following two properties:

1) $\Phi_n^{\mathcal{S}}$ is satisfiable if and only if there exists an LTL formula of size n that is consistent with $\mathcal{S}$; and
2) if v is a model of $\Phi_n^{\mathcal{S}}$, then v contains sufficient information to construct an LTL formula ψ_v of size n that is consistent with $\mathcal{S}$.

By increasing the value of n by one and extracting an LTL formula ψ_v from a model v of $\Phi_n^{\mathcal{S}}$ as soon as it becomes satisfiable (indeed, any model is sufficient), we obtain an effective algorithm that learns an LTL formula of minimal size that is consistent with $\mathcal{S}$. This idea is shown in pseudo code as Algorithm 1. In fact, the existence of a trivial solution for the passive LTL learning task (as sketched at the beginning of this section) shows that Algorithm 1 is guaranteed to terminate, and the size of this solution provides an upper bound on the value of n.

Algorithm 1: SAT-based learning algorithm

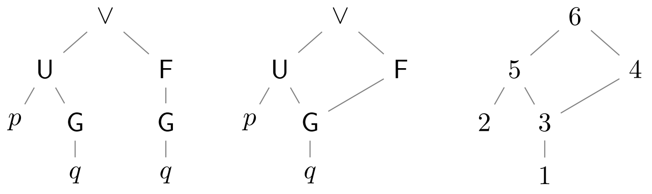
```
Input: a sample $\mathcal{S}$
$n \leftarrow 0$;
repeat
    $n \leftarrow n + 1$;
    Construct and solve $\Phi_n^{\mathcal{S}}$;
until $\Phi_n^{\mathcal{S}}$ is satisfiable, say with model $v$;
Construct and return $\psi_v$;
```

The key idea of the formula $\Phi_n^{\mathcal{S}}$ is to encode the syntax DAG of an (unknown) LTL formula $\varphi^\star$ with n subformulas and then constrain the variables of $\Phi_n^{\mathcal{S}}$ such that $\varphi^\star$ is consistent with the sample $\mathcal{S}$. To simplify our encoding, we assign to each node of this syntax DAG a unique *identifier* $i \in \{1, \ldots, n\}$ such that (a) the identifier of the root is n and (b) if the identifier of an inner node is i, then the identifiers of its children are less than i. Note that such a numbering scheme is not unique for a given syntax DAG, but it entails that the root always has identifier n and the node with identifier 1 is always labeled with an atomic proposition. We refer the reader to Figures 1b and 1c for an example.

We encode a syntax DAG using three types of propositional variables:

- $x_{i,\lambda}$ where $i \in \{1, \ldots, n\}$ and $\lambda \in \mathcal{P} \cup \mathcal{C}$;
- $l_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$; and
- $r_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$.

Intuitively, the variables $x_{i,\lambda}$ encode a labeling of the syntax DAG in the sense that if a variable $x_{i,\lambda}$ is set to $true$, then node i is labeled with λ (recall that each node is labeled with either an atomic proposition from $\mathcal{P}$ or an operator from $\mathcal{C}$). The variables $l_{i,j}$ and $r_{i,j}$, on the other hand, encode the structure of the syntax DAG (i.e., the left and/or right child of inner nodes): if variable $l_{i,j}$ ($r_{i,j}$) is set to $true$, then j is the identifier of the left (right) child of node i. By convention, we ignore the variables $r_{i,j}$ if node i of the syntax DAG is labeled with an unary operator; similarly, we ignore both $l_{i,j}$ and $r_{i,j}$ if node i is labeled with an atomic proposition. Note that in the case of $l_{i,j}$ and $r_{i,j}$, the identifier i ranges from 2 to n because node 1 is always labeled with an atomic proposition and, hence, cannot have children. Moreover, j ranges from 1 to $i-1$ to reflect the fact that identifier of children have to be smaller than the identifier of the current node.



(a) Syntax tree (b) Syntax DAG (c) Identifiers

Fig. 1: Syntax tree, syntax DAG, and identifiers of the syntax DAG for the LTL formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \vee (\mathsf{F}\,\mathsf{G}\, q)$

TABLE I: Constraints enforcing that the variables $x_{i,\lambda}$ encode a syntax DAG

$$\Big[\bigwedge_{1\leq i\leq n}\ \bigvee_{\lambda\in\mathcal{P}\cup\mathcal{C}} x_{i,\lambda}\Big] \wedge \Big[\bigwedge_{1\leq i\leq n}\ \bigwedge_{\lambda\neq\lambda'\in\mathcal{P}\cup\mathcal{C}} \neg x_{i,\lambda} \vee \neg x_{i,\lambda'}\Big] \tag{1}$$

$$\Big[\bigwedge_{2\leq i\leq n}\ \bigvee_{1\leq j<i} l_{i,j}\Big] \wedge \Big[\bigwedge_{2\leq i\leq n}\ \bigwedge_{1\leq j<j'<i} \neg l_{i,j} \vee \neg l_{i,j'}\Big] \tag{2}$$

$$\Big[\bigwedge_{2\leq i\leq n}\ \bigvee_{1\leq j<i} r_{i,j}\Big] \wedge \Big[\bigwedge_{2\leq i\leq n}\ \bigwedge_{1\leq j<j'<i} \neg r_{i,j} \vee \neg r_{i,j'}\Big] \tag{3}$$

$$\bigvee_{p\in\mathcal{P}} x_{1,p} \tag{4}$$

To enforce that the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ in fact encode a syntax DAG, we impose the constraints listed in Table I. Formula (1) ensures that each node is labeled with exactly one label. Similarly, Formulas (2) and (3) enforce that each node (except for node 1) has exactly one left and exactly one right child (although we ignore certain children if the node represents an unary operator or an atomic predicate). Finally, Formula (4) makes sure that node 1 is labeled with an atomic proposition.

Let Φ_n^{DAG} now be the conjunction of Formulas (1) to (4). Then, one can construct a syntax DAG from a model v of Φ_n^{DAG} in a straightforward manner: simply label node i with the unique label λ such that $v(x_{i,\lambda}) = 1$, designate node n as the root, and arrange the nodes of the DAG as uniquely described by $v(l_{i,j})$ and $v(r_{i,j})$. Moreover, we can easily derive an LTL formula from this syntax DAG, which we denote by ψ_v. Note, however, that ψ_v is not yet related to the sample $\mathcal{S}$ and, thus, might or might not be consistent with it.

To enforce that ψ_v is indeed consistent with $\mathcal{S}$, we now constrain the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ further. More precisely, we add for each ultimately periodic word uv^ω in $\mathcal{S}$ a propositional formula $\Phi_n^{u,v}$ that tracks the valuation of the LTL formula encoded by Φ_n^{DAG} (and all its subformulas) on uv^ω. The observation that enables us to do this is the following.

Observation 1: Let $uv^\omega \in (2^{\mathcal{P}})^\omega$, ψ be an LTL formula over $\mathcal{P}$, and $k \in \mathbb{N}$. Then, $uv^\omega[|u|+k, \infty) = uv^\omega[|u|+m, \infty)$ with $m \equiv k \mod |v|$. In addition, $V(\varphi, uv^\omega[|u|+k, \infty)) = V(\varphi, uv^\omega[|u|+m, \infty))$ holds for every LTL formula φ.

Intuitively, Observation 1 states that the suffixes of a word uv^ω eventually repeat periodically. As a consequence, the valuation of an LTL formula on a word uv^ω can be determined based only on the finite prefix uv (recall that the semantics of temporal operators only depend on the suffixes of a word). To illustrate this claim, consider the LTL formula $\mathsf{X}\,\varphi$ and assume that we want to determine the valuation $V(\mathsf{X}\,\varphi, uv^\omega[|uv|-1, \infty))$ (i.e., $\mathsf{X}\,\varphi$ is evaluated at the end of the prefix uv). Then, Observation 1 permits us to compute this valuation based on $V(\varphi, uv^\omega[|u|, \infty))$, as opposed to the original semantics of the X-operator, which recurs to $V(\varphi, uv^\omega[|uv|, \infty))$ (i.e., the valuation at the next position). Note that similar, though more involved ideas can be applied to all other temporal operators.

Each formula $\Phi_n^{u,v}$ is built over an auxiliary set of propositional variables $y_{i,t}^{u,v}$ where $i \in \{1, \ldots, n\}$ is a node in the syntax DAG and $t \in \{0, \ldots, |uv|-1\}$ is a position in the finite word uv. The meaning of these variables is that the value of $y_{i,t}^{u,v}$ corresponds to the valuation $V(\varphi_i, uv^\omega[t, \infty))$ of the LTL subformula φ_i that is rooted at node i. Note that the set of variables for two distinct words from the sample must be disjoint.

To obtain this desired meaning of the variables $y_{i,t}^{u,v}$, we impose the constraints listed in Table II, which are inspired by bounded model checking [10]. Formula (5) implements the LTL semantics of atomic propositions and ensures that if node i is labeled with $p \in \mathcal{P}$, then $y_{i,t}^{u,v}$ is set to 1 if and only if $p \in uv(t)$. Next, Formulas (6) and (7) implement the semantics of negation and disjunction, respectively: if node i is labeled with $\neg$ and node j is its left child, then $y_{i,t}^{u,v}$ is the negation of $y_{j,t}^{u,v}$; on the other hand, if node i is labeled with $\vee$, node j is its left child, and node j' is its right child, then $y_{i,t}^{u,v}$ is the disjunction of $y_{j,t}^{u,v}$ and $y_{j',t}^{u,v}$. Moreover, Formula (8) implements the semantics of the X-operator, following the idea of "returning to the beginning of the periodic part v" as sketched above. Finally, Formula (9) implements the semantics of the U-operator. More precisely, the first conjunction in the consequent covers the positions $t \in \{0, \ldots, |u|-1\}$ in the initial part u, while the second conjunct covers the positions $t \in \{|u|, \ldots, |uv|-1\}$ in the periodic part v. Thereby, the second conjunct relies on an auxiliary set $t \looparrowright_{u,v} t'$ defined by

$$t \looparrowright_{u,v} t' := \begin{cases} \{t, \ldots, t'-1\} & \text{if } t < t'; \\ \{|u|, \ldots, t'-1, t, \ldots, |uv|-1\} & \text{if } t \geq t', \end{cases}$$

which contains all positions in v "between t and t'". To avoid cluttering this section too much, we have omitted the description of the missing operators $\wedge$, $\rightarrow$, F, G and the constants *true* and *false*, which are implemented analogously. Moreover, our SAT encoding is extensible, and additional LTL operators such as weak until or weak and strong release can easily be added.

TABLE II: Constraints enforcing that the variables $y_{i,t}^{u,v}$ track the valuation of the prospective LTL formula on ultimately periodic words

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{p \in \mathcal{P}} x_{i,p} \rightarrow \left[\bigwedge_{0 \leq t < |uv|} \begin{cases} y_{i,t}^{u,v} & \text{if } p \in uv(t) \\ \neg y_{i,t}^{u,v} & \text{if } p \notin uv(t) \end{cases} \right] \tag{5}$$

$$\bigwedge_{\substack{1 < i \leq n \\ 1 \leq j < i}} (x_{i,\neg} \wedge l_{i,j}) \rightarrow \bigwedge_{0 \leq t < |uv|} \left[y_{i,t}^{u,v} \leftrightarrow \neg y_{j,t}^{u,v} \right] \tag{6}$$

$$\bigwedge_{\substack{1 < i \leq n \\ 1 \leq j, j' < i}} (x_{i,\vee} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \bigwedge_{0 \leq t < |uv|} \left[y_{i,t}^{u,v} \leftrightarrow (y_{j,t}^{u,v} \vee y_{j',t}^{u,v}) \right] \tag{7}$$

$$\begin{gathered} \bigwedge_{\substack{1 < i \leq n \\ 1 \leq j < i}} (x_{i,\mathsf{X}} \wedge l_{i,j}) \rightarrow \\ \left[\left[\bigwedge_{0 \leq t < |uv|-1} y_{i,t}^{u,v} \leftrightarrow y_{j,t+1}^{u,v} \right] \wedge \left[y_{i,|uv|-1}^{u,v} \leftrightarrow y_{j,|u|}^{u,v} \right] \right] \end{gathered} \tag{8}$$

$$\begin{gathered} \bigwedge_{\substack{1 < i \leq n \\ 1 \leq j, j' < i}} (x_{i,\mathsf{U}} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \\ \left[\left[\bigwedge_{0 \leq t < |u|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{t \leq t' < |uv|} \left[y_{j',t'}^{u,v} \wedge \bigwedge_{t \leq t'' < t'} y_{j,t''}^{u,v} \right] \right] \wedge \right. \\ \left. \left[\bigwedge_{|u| \leq t < |uv|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{|u| \leq t' < |uv|} \left[y_{j',t'}^{u,v} \wedge \bigwedge_{t'' \in t \looparrowright_{u,v} t'} y_{j,t''}^{u,v} \right] \right] \right] \end{gathered} \tag{9}$$

For each $uv^\omega \in P \cup N$, let $\Phi_n^{u,v}$ now be the conjunction of Formulas (5) to (9). Then, we define

$$\Phi_n^{\mathcal{S}} := \Phi_n^{DAG} \wedge \left[\bigwedge_{uv^\omega \in P} \Phi_n^{u,v} \wedge y_{n,0}^{u,v} \right] \wedge \left[\bigwedge_{uv^\omega \in N} \Phi_n^{u,v} \wedge \neg y_{n,0}^{u,v} \right].$$

Note that the subformula $\Phi_n^{u,v} \wedge y_{n,0}^{u,v}$ makes sure that $uv^\omega \in P$ satisfies the prospective LTL formula (more concretely, uv^ω starting from position 0 satisfies the LTL formula at the root of the syntax DAG), while $\Phi_n^{u,v} \wedge \neg y_{n,0}^{u,v}$ ensures that $uv^\omega \in N$ does not satisfy it.

To prove the correctness of our learning algorithm, we first establish that the formula $\Phi_n^{\mathcal{S}}$ has in fact the desired properties.

Lemma 1: Let $\mathcal{S} = (P, N)$ be a sample, $n \in \mathbb{N} \setminus \{0\}$, and $\Phi_n^{\mathcal{S}}$ the propositional formula defined above. Then, the following holds:

1) If an LTL formula of size n that is consistent with $\mathcal{S}$ exists, then the propositional formula $\Phi_n^{\mathcal{S}}$ is satisfiable.
2) If $v \models \Phi_n^{\mathcal{S}}$, then ψ_v is an LTL formula of size n that is consistent with $\mathcal{S}$.

Termination and correctness of Algorithm 1 then follow from Lemma 1.

Theorem 1: Given a sample $\mathcal{S}$, Algorithm 1 terminates eventually and outputs an LTL formula of minimal size that is consistent with $\mathcal{S}$.

Proof: Since there exists a consistent LTL formula for every non-contradictory sample, Part 1 of Lemma 1 guarantees that Algorithm 1 terminates. Moreover, Part 2 ensures that the output is indeed an LTL formula that is consistent with $\mathcal{S}$. Since n is increased by one in every iteration of the loop until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, the output of Algorithm 1 is a consistent LTL formula of minimal size. □

It is important to emphasize that the size of $\Phi_n^{\mathcal{S}}$ and, hence, the performance of Algorithm 1 depends on the size of a sample $\mathcal{S} = (P, N)$, as summarized next.

Remark 1: The formula $\Phi_n^{\mathcal{S}}$ ranges over $\mathcal{O}(n^2 + n|\mathcal{S}|)$ variables and is of size $\mathcal{O}(n^2 + n^3 \sum_{uv^\omega \in P \cup N} |uv|^3)$.

Finally. we conclude this section with a remark on incorporating expert knowledge into the learning process.

Remark 2: By adding constraints to the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$, one can easily incorporate expert knowledge (e.g., syntactic templates) into the learning process.

IV. A DECISION TREE BASED LEARNING ALGORITHM

The SAT-based algorithm described in Section III is an elegant, out-of-the-box way to discover minimal LTL formulas describing a sample. Even though it scales well beyond toy examples, its running time seems too prohibitive for real-world examples (as discussed in Section V). That is why we now present a learning algorithm based on a combination of SAT solving and decision tree learning.

Our second algorithm proceeds in two phases, outlined in Algorithm 2. In the first phase, we run Algorithm 1 on small subsets of P and N. This is repeated until we obtain a set Π of LTL formulas (we call them *LTL primitives*) that separate all pairs of words from P and N. In the second phase, formulas from Π are used as features for a standard decision tree learning algorithm [13]. The resulting decision tree is a Boolean combination of LTL formulas $\varphi_i \in \Pi$ that is consistent with the sample.

Algorithm 2: Learning algorithm based on decision trees

```
Input: a sample $\mathcal{S}$
1 Run Algorithm 1 on small subsets of $P$ and $N$ to
    construct a set $\Pi = \{\varphi_1, \dots, \varphi_n\}$ of LTL formulas
    such that for each pair $u_1 v_1^\omega \in P$ and $u_2 v_2^\omega \in N$ there
    exists a $\varphi_i \in \Pi$ with $V(\varphi_i, u_1 v_1^\omega) = 1$ and
    $V(\varphi_i, u_2 v_2^\omega) = 0$;
2 Learn a decision tree $t$ with LTL primitives from $\Pi$ as
    features and return the resulting Boolean combination
    $\psi_t$ of LTL primitives (which is consistent with $\mathcal{S}$);
```

Note that this relaxes the problem addressed in Section III: we can no longer guarantee finding a formula of minimal size. However, decision trees are among the structures that are the easiest to interpret by end-users. That makes them suitable for our use-case, and the minimality of formulas is replaced by structural simplicity of decision trees.

Learning Decision Trees: We assume familiarity with decision tree learning and refer the reader to a standard textbook for further details [5]. As illustrated in Figure 3, the decision trees we seek to learn are tree-shaped structures whose inner nodes are labeled with LTL formulas from Π and whose leaves are labeled with either *true* or *false*. The LTL formula represented by such a tree t is given by $\psi_t \coloneqq \bigvee_{\rho \in \mathfrak{P}} \bigwedge_{\varphi \in \rho} \varphi$ where $\mathfrak{P}$ is the set of all paths from the root to a leaf labeled with *true* and $\varphi \in \rho$ denotes that φ occurs on ρ (negated if the path follows a dashed edge).

To learn a decision tree over LTL primitives, we perform a preprocessing step and modify the sample as follows. For each word $uv^\omega \in P \cup N$, we use the LTL primitives as features and create a Boolean vector of size $|\Pi|$ with the i-th entry set to $V(\varphi_i, uv^\omega)$; this vector is then labeled with *true* if $uv^\omega \in P$ or with *false* if $uv^\omega \in N$. In the second step, we apply a standard learning algorithm for decision trees to this modified sample (we used Gini impurity [34] as split heuristic in our experiments). Since we are interested in a tree that classifies our sample correctly, we disable heuristics such as pruning.

Obtaining LTL Primitives: Meaningful features are essential for a successful classification using decision trees. In our algorithm, features are generated from the set of LTL primitives Π. We used two different strategies, called Strategy α and Strategy β, for obtaining Π.

Strategy α iteratively chooses subsets $P' \subset P$ and $N' \subset N$ of size k according to probability distributions prob_P and prob_N on P and N, respectively. After a formula φ separating P' and N' is found using Algorithm 1 and added to Π, prob_P and prob_N are updated to increase the likelihood of any word that is not yet classified correctly by any of the $\varphi \in \Pi$ to be selected. This process is repeated until all pairs of positive and negative examples are separated by some LTL primitive or restarted after a user-given number of iterations. Although this strategy is, in general, not guaranteed to terminate due to its probabilistic nature, it always did in our experiments.

Strategy β computes LTL primitives in a more aggressive way. Starting with the set $S = P \times N$, it uniformly at random selects k pairs from S and uses Algorithm 1 to compute an LTL primitive φ that separates those pairs. Then, it removes all pairs separated by φ from S and repeats the process until S becomes empty (i.e., all pairs of examples are separated).

We refer to the extended version of this paper [35] for a detailed explanation of both strategies.

Correctness: The correctness of Algorithm 2 is formalized below.

Theorem 2: Given a sample $\mathcal{S}$, Algorithm 2 learns a (not necessarily minimal) formula ψ_t that is consistent with $\mathcal{S}$.

Theorem 2 follows from the fact that Step 1 of Algorithm 2 constructs a set of LTL primitives that allows separating any pair of positive and negative examples. Once such a set is constructed, any decision tree learner produces a decision tree t that is guaranteed to classify the examples correctly. The resulting LTL formula ψ_t, hence, is consistent with $\mathcal{S}$.

V. EVALUATION

In this section, we answer questions that arise naturally: how performant is Algorithm 1 and what is the performance gain of Algorithm 2. Furthermore, what is the complexity of

TABLE III: Common LTL patterns used in practice [37]

Absence	Existence	Universality
$\mathsf{G}(\neg p_0)$	$\mathsf{F}(p_0)$	$\mathsf{G}(p_0)$
$\mathsf{F}(p_1) \rightarrow (\neg p_0 \,\mathsf{U}\, p_1)$	$\mathsf{G}(\neg p_0 \vee \mathsf{F}(p_0 \wedge \mathsf{F}(p_1)))$	$\mathsf{F}(p_1) \rightarrow (p_0 \,\mathsf{U}\, p_1)$
$\mathsf{G}(p_1 \rightarrow \mathsf{G}(\neg p_0))$	$\mathsf{G}(p_0 \wedge (\neg p_1 \rightarrow (\neg p_1 \,\mathsf{U}(p_2 \wedge \neg p_1))))$	$\mathsf{G}(p_1 \rightarrow \mathsf{G}(p_0))$

the learned decision trees in terms of the number of decision nodes, and, finally, how do different parameters influence the performance of Algorithm 2. After answering these questions with experiments performed on synthetic data, we demonstrate the usefulness of our algorithms for understanding executions of a leader-election algorithm.

We implemented both learning algorithms in a Python tool[2] using Microsoft Z3 [36]. All experiments were conducted on Debian machines with Intel Xeon E7-8857 CPUs at 3 GHz, using up to 5 GB of RAM.

Performance on Synthetic Data: To simulate real-world use-cases, we generated samples based on common LTL patterns [37], which are shown in Table III. Starting from a pattern formula ψ, we generated sets of random words and separated them into P and N depending on whether they are a model of ψ or not. Thereby, we fixed $|u| + |v| = 10$ for all words in the sample and added noise in form of one additional atomic proposition that is not constrained by the pattern formula. The size of the generated samples ranges between 50 and 5000. In total, we generated 192 samples.

Figure 2 compares the running times of Algorithm 1 and Algorithm 2 (using Strategy α and $k = 3$) on samples of varying sizes. (So as not to clutter the presentation too much, we selected four LTL patterns that showed a typical behavior of our learning algorithms. The complete results are available in the technical report [35].) Overall, Algorithm 1 produces minimal formulas consistent with a sample. It does so even for samples of considerable size, but if the sample size grows beyond 2000 (varies over samples), the SAT-based learner (Algorithm 1) frequently times out. When Algorithm 2 (using decision tree learning) is applied to these samples—as shown on the right-hand-side of Figure 2—none of the computations timed out and the running times significantly improved.

What kind of trees does Algorithm 2 produce? An example output of the algorithm is shown in Figure 3. Moreover, as Table IV illustrates, Algorithm 2 learns small trees, often with less than five inner nodes. Upon closer inspection, we noticed that it often happens that one of the LTL primitives was the specified formula itself. This suggests that small subsets already characterize our samples completely.

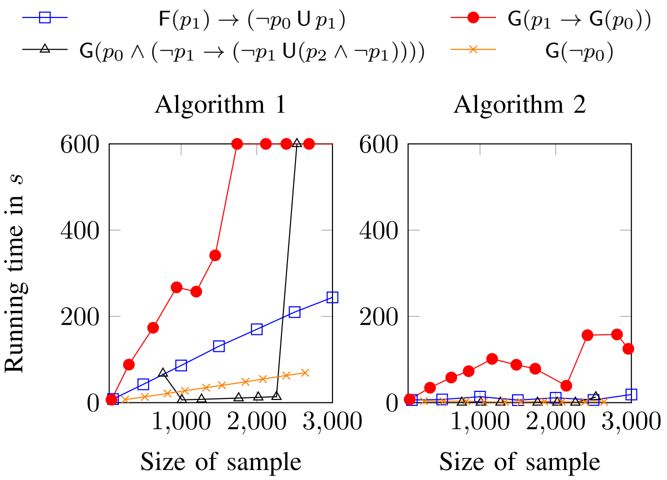
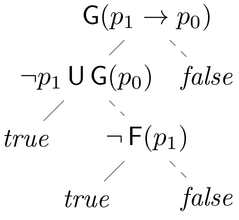


Fig. 2: Comparison of Algorithm 1 and Algorithm 2



If a node evaluates to true, the outgoing full edge should be followed (and the dashed edge otherwise). The tree on the left represents the formula $\psi_t = [\mathsf{G}(p_1 \rightarrow p_0) \wedge (\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)] \vee [\mathsf{G}(p_1 \rightarrow p_0) \wedge \neg(\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)) \wedge \neg\, \mathsf{F}(p_1)]$.

Fig. 3: A decision tree obtained from a sample generated from the LTL pattern $\mathsf{G}(p_1 \rightarrow \mathsf{G}(p_0))$

To be able to compare decision trees to the formulas learned by Algorithm 2, we measure the *size* of a tree t in terms of the size of the formula ψ_t this tree encodes. In our experiments, the formulas learned by Algorithm 2 were on average 1.41 times larger than those learned by Algorithm 1. However, there are outlier trees that are four times bigger than the one learned by Algorithm 1. Nonetheless, about 70 % are of the same size. Even for the outliers, as emphasized previously, the readability does not degrade completely because the rule-based structure of decision trees is known to be easily understandable by humans. Note that the runtime and size of decision trees depends on the parameters of Algorithm 2, which we discuss next.

Tuning the Decision Tree-Based Algorithm: As described in Section IV, Algorithm 2 can be tuned by various parameters (sampling strategy for obtaining LTL primitives, size of sample subsets, probability increase rate, and number of repetitions inside a single sampling). In this subsection, we explore how those parameters affect the performance of the algorithm.

TABLE IV: Different parameters used for Algorithm 2

Sampling strategy	Subset size k	Number of timeouts	Avg. running time in s	Avg. number of nodes in a tree
α	3	0 / 192	21.00	3.05
α	6	4 / 192	35.28	1.47
α	10	8 / 192	42.72	1.2
β	3	4 / 192	30.92	1.37
β	6	12 / 192	48.46	1.19
β	10	21 / 192	48.11	1.06

Table IV shows the performance of Algorithm 2 for different parameters, averaged over all 192 benchmarks. As the table indicates, the less aggressive method of separating sets, Strategy α, performs better. It seems that if the subset sizes are increased, or Strategy β is used, the sampled subsets already describe the specified formula completely. Finally, we chose Strategy α and $k = 3$ to be our default parameters. Varying the probability decrease rate and the number of repetitions inside

---

[2] Our tool is publicly available at https://github.com/gergia/samples2LTL.

ISBN: 978-0-9835678-8-2.

a single sampling did not influence the performance much.

Explaining Executions of a Leader Election Protocol: A number of methods exist for finding errors or reproducing certain behavior in distributed systems through systematic testing [38], [39]. However, finding an execution and a corresponding schedule is only a first step towards understanding an issue. In the following, we demonstrate how to apply our technique in order to obtain a minimal LTL description of a specific inconsistency in a leader election protocol.

The leader election protocol we consider is the *Fast Leader Election* algorithm [14], [40] used by Apache Zookeeper. In this protocol, every node has a unique ID and initially tries to become the leader. To this end, every node sends messages to all other nodes proclaiming its leadership. Upon receiving a message by an aspirant leader with a higher ID, a node gives up its claim and acknowledges its support for the aspirant. If a node learns that an aspirant node has a support of a majority of all nodes, it commits (after waiting for a constant time for new messages) to the aspirant as the leader. Once committed, the node never again changes its decision and informs any other node of its commitment (one example is the message depicted by the dotted arrow in Figure 5). If a node has not committed and learns about another node that has committed, it commits to the same leader.

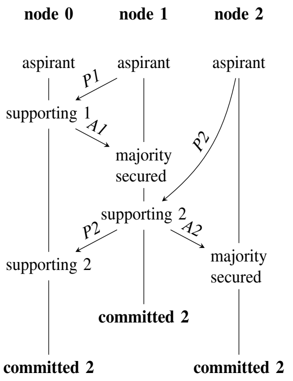


Fig. 4: Consistent schedule for an execution of the leader election protocol

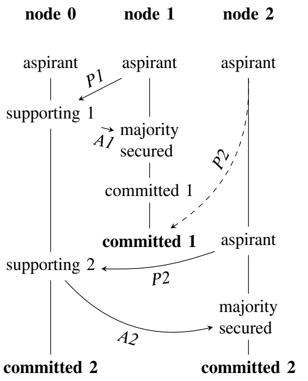


Fig. 5: Inconsistent schedule for an execution of the leader election protocol

Figure 4 shows an example of a successful leader election with three nodes in an UML-style message sequence chart. The messages exchanged between nodes are proposing the leader i (P_i) and node j acknowledging the claim of a leader (A_j). The arrows indicate exchanged messages and imply a precedence of events. Note that not all messages are shown in the figures, but only the ones important for understanding the protocol.

In Figure 4 all the nodes have committed to the same leader. On the other hand, Figure 5 shows a schedule that ends up in an inconsistent state where nodes committed to different leaders. This schedule was discovered by the PCTCP algorithm [41], which systematically explores the space of possible executions of distributed algorithms. The situation in Figure 5 is caused by the asynchronous communication: for performance reasons, nodes commit as quickly as possible and then discard any messages, which otherwise would have changed their commitment (indicated as a dashed line in Figure 5). Note, however, that this is not a bug in Zookeeper's broadcast algorithm, as a leader without a quorum will not be allowed to perform any action in the later phase.

To better understand how this inconsistent state arises, our goal is to generate an LTL formula that describes the difference between the schedules in Figures 4 and 5. To this end, we constructed a sample by generating 20 linearizations of the schedule from Figure 4 and 20 linearizations of the schedule from Figure 5. Since we seek an explanation for the inconsistent behavior, the former (with consistent outcomes) correspond to negative examples (set N), and the latter (with inconsistent outcomes) correspond to positive examples (set P). The set of atomic propositions used to construct the examples contains twelve elements: $\mathit{recv}(i, j)$ for $i, j \in \{1, 2, 3\}$ (meaning that node j received a message from node i) and $\mathit{comm}(i)$ for $i \in \{1, 2, 3\}$ (meaning that node i committed to a leader).[3]

Finally, we ran Algorithm 1 on this sample. The result was the formula $\neg \mathit{recv}(2, 1) \mathbin{\mathsf{U}} \mathit{comm}(1)$. Intuitively, node 1 did not receive a message from node 2 before it committed to a leader. That is exactly the difference between the schedules in Figures 4 and 5. Also, it hints at a specific reason for the inconsistency in Figure 5, thus potentially helping the engineers improve the system. Note, however, that this experiment still required a significant amount of manual effort. In order to apply the technique in practice, more automation is needed.

Summary: Algorithm 2 significantly improves upon the performance of Algorithm 1, though with a small increase in the size of the formula. The original motivation of getting readable explanations for the behavior of a system is preserved due to the fact that decision-trees are easy to comprehend. Algorithm 2 works the best using Strategy α and subsets of size $k = 3$. Finally, our techniques are able to give interesting insight into real-world systems.

VI. CONCLUSION

We have presented two novel algorithms for learning LTL formulas from examples. Our first algorithm is based on SAT solving, while the second algorithm extends the first with techniques for learning decision trees. We have shown that both algorithms are able to learn LTL formulas for a comprehensive set of benchmarks that we have derived from common LTL patterns. Moreover, we have demonstrated how our methods can help understand distributed algorithms.

Interesting directions of future work include the integration of LTL past-time operators, lifting our techniques to an active learning setup [29], as well as the development of similar learning algorithms for CTL. Furthermore, we plan to investigate the use of maximum-margin classifiers, such as support vector machines. To this end, one needs to develop a notion of distance between temporal formulas and words, which is clearly of independent, theoretical interest as well.

---

[3] While we could have included more information into propositions, we had to obscure some in order to avoid "stating the obvious" of the form "node 1 committed to node 1 as a leader, while node 2 committed to node 2".

REFERENCES

[1] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[2] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs*, ser. Lecture Notes in Computer Science, vol. 131. Springer, 1981, pp. 52–71.

[3] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984. [Online]. Available: http://doi.acm.org/10.1145/1968.1972

[4] A. Blum, J. Hopcroft, and R. Kannan, *Foundations of Data Science*, January 2018. [Online]. Available: https://www.cs.cornell.edu/jeh/book.pdf

[5] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997. [Online]. Available: http://www.worldcat.org/oclc/61321007

[6] C. P. Pfleeger, "State reduction in incompletely specified finite-state machines," *IEEE Trans. Computers*, vol. 22, no. 12, pp. 1099–1102, 1973. [Online]. Available: https://doi.org/10.1109/T-C.1973.223655

[7] D. Neider, "Computing minimal separating dfas and regular invariants using SAT and SMT solvers," in *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7561. Springer, 2012, pp. 354–369.

[8] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, "Regular model checking," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 403–418. [Online]. Available: https://doi.org/10.1007/10722167_31

[9] D. Neider and N. Jansen, "Regular model checking using solver technologies and automata learning," in *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7871. Springer, 2013, pp. 16–31. [Online]. Available: https://doi.org/10.1007/978-3-642-38088-4_2

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003. [Online]. Available: https://doi.org/10.1016/S0065-2458(03)58003-2

[11] G. Bombara, C. I. Vasile, F. Penedo, H. Yasuoka, and C. Belta, "A decision tree approach to data classification using signal temporal logic," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. ACM, 2016, pp. 1–10.

[12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012. [Online]. Available: https://doi.org/10.1016/j.jcss.2011.08.007

[13] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, 2011, pp. 245–256. [Online]. Available: https://doi.org/10.1109/DSN.2011.5958223

[15] D. Neider and I. Gavran, "Learning linear temporal properties," *CoRR*, vol. abs/1806.03953, 2018. [Online]. Available: http://arxiv.org/abs/1806.03953

[16] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3253. Springer, 2004, pp. 152–166. [Online]. Available: https://doi.org/10.1007/978-3-540-30206-3_12

[17] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric identification of temporal properties," in *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7186. Springer, 2011, pp. 147–160. [Online]. Available: https://doi.org/10.1007/978-3-642-29860-8_12

[18] Z. Kong, A. Jones, A. M. Ayala, E. A. Gol, and C. Belta, "Temporal logic inference for classification and prediction from data," in *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*. ACM, 2014, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2562059.2562146

[19] Z. Kong, A. Jones, and C. Belta, "Temporal logics for learning and detection of anomalous behavior," *IEEE Trans. Automat. Contr.*, vol. 62, no. 3, pp. 1210–1222, 2017. [Online]. Available: https://doi.org/10.1109/TAC.2016.2585083

[20] P. Vaidyanathan, R. Ivison, G. Bombara, N. A. DeLateur, R. Weiss, D. Densmore, and C. Belta, "Grid-based temporal logic inference," in *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*, 2017, pp. 5354–5359. [Online]. Available: https://doi.org/10.1109/CDC.2017.8264452

[21] E. Bartocci, L. Bortolussi, and G. Sanguinetti, "Learning temporal logical properties discriminating ECG models of cardiac arrhytmias," *CoRR*, vol. abs/1312.7523, 2013. [Online]. Available: http://arxiv.org/abs/1312.7523

[22] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. IEEE, 2011, pp. 43–50. [Online]. Available: https://doi.org/10.1109/MEMCOD.2011.5970509

[23] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 81–92. [Online]. Available: https://doi.org/10.1109/ASE.2015.71

[24] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Autom. Softw. Eng.*, vol. 18, no. 3-4, pp. 263–292, 2011. [Online]. Available: https://doi.org/10.1007/s10515-011-0084-1

[25] J. R. Büchi, "On a decision method in restricted second-order arithmetic," in *Int. Congr. for Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, 1962, pp. 1–11.

[26] A. W. Kamp, "Tense logic and the theory of linear order," Ph.D. dissertation, University of California, Los Angeles, 1968.

[27] A. Farzan, Y. Chen, E. M. Clarke, Y. Tsay, and B. Wang, "Extending automated compositional verification to the full class of omega-regular languages," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 2–17.

[28] D. Angluin and D. Fisman, "Learning regular omega languages," *Theor. Comput. Sci.*, vol. 650, pp. 57–72, 2016.

[29] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[30] M. Grohe and M. Ritzert, "Learning first-order definable concepts over structures of small degree," in *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. [Online]. Available: https://doi.org/10.1109/LICS.2017.8005080

[31] M. Grohe, C. Löding, and M. Ritzert, "Learning mso-definable hypotheses on strings," in *International Conference on Algorithmic Learning Theory, ALT 2017, 15-17 October 2017, Kyoto University, Kyoto, Japan*, ser. Proceedings of Machine Learning Research, vol. 76. PMLR, 2017, pp. 434–451. [Online]. Available: http://proceedings.mlr.press/v76/grohe17a.html

[32] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[33] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017, pp. 5061–5063.

[34] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees. Wadsworth, 1984*. Routledge, 1993.

[35] D. Neider and I. Gavran, "Learning linear temporal properties," *CoRR*, vol. abs/1806.03953, 2018. [Online]. Available: http://arxiv.org/abs/1806.03953

[36] 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, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[37] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the Second Workshop on Formal Methods in Software Practice*, ser. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7–15. [Online]. Available: http://doi.acm.org/10.1145/298595.298598

[38] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, 2010, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736040

[39] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?" *PACMPL*, vol. 2, no. POPL, pp. 46:1–46:24, 2018. [Online]. Available: http://doi.acm.org/10.1145/3158134

[40] A. Medeiros, "Zookeeper's atomic broadcast protocol: Theory and practice," 2012.

[41] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Berfrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees," in *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2018, to appear.

ISBN: 978-0-9835678-8-2.

The ELDARICA Horn Solver

Hossein Hojjat
Rochester Institute of Technology, University of Tehran
hh@cs.rit.edu

Philipp Rümmer
Uppsala University
philipp.ruemmer@it.uu.se

***Abstract*—This paper presents the ELDARICA version 2 model checker. Over the last years we have been developing and maintaining ELDARICA as a state-of-the-art solver for Horn clauses over integer arithmetic. In the version 2, we have extended the solver to support also algebraic data types and bit-vectors, theories that are commonly applied in verification, but currently unsupported by most Horn solvers. This paper describes the high-level structure of the tool and the interface that it provides to other applications. We also report on an evaluation of the tool. While some of the techniques in ELDARICA have been documented in research papers over the last years, this is the first tool paper describing ELDARICA in its entirety.**

I. INTRODUCTION

In recent years, the computer-aided verification community has been advocating Horn clause solving as a uniform framework for reasoning about different aspects of software safety [7], [20], [32], [25]. Horn clauses form a fragment of first-order logic, modulo various background theories, in which models can be constructed effectively with the help of model checking algorithms. Horn clauses can be used as an intermediate verification language that elegantly captures various classes of systems (e.g., sequential code, programs with functions and procedures, concurrent programs, or networks of timed automata) and various verification methodologies (e.g., the use of state invariants, verification with the help of contracts, Owicki-Gries-style invariants, or rely-guarantee methods). Horn solvers can be used as *off-the-shelf backends* in verifiers, and thus enable construction of verification systems in a modular way.

ELDARICA first appeared as a solver for Horn clauses over Presburger arithmetic in 2013 [32].[1] It combines Predicate Abstraction [19] with Counterexample-Guided Abstraction Refinement (CEGAR) [12] to automatically check whether a given set of Horn clauses is satisfiable. The tool has been significantly improved since then and can now solve problems over the theories of integers, algebraic data-types [24], and bit-vectors. It can process Horn clauses and programs in a variety of formats, implements sophisticated algorithms to solve tricky systems of clauses without diverging, and offers an elegant API for programmatic use.

A. An Initial Example

To verify systems using Horn clauses, we first need to fix a set R of uninterpreted fixed-arity relation symbols, which represent the *unknowns* in the Horn clauses. A *constrained Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where

- C is a constraint over some background theory;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in R$ to first-order terms, usually including first-order variables;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in R$ to first-order terms, or false.

H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = \mathit{true}$, we usually leave out C and just write $H \leftarrow B_1 \wedge \cdots \wedge B_n$. First-order variables in a clause are implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in S$.

A solution to a set of Horn clauses assigns a formula to each relation symbol in such a way that all Horn clauses become valid formulas, considering first-order variables as implicitly universally quantified. When no solution exists, a derivation of *false* can be constructed as a counterexample.

Figure 1 shows a simple C program, together with a control-flow graph illustrating the program structure. The verification task consists of proving that the assertion in the program can never fail, i.e., showing program safety. In order to extract a set of Horn clauses that encode program safety, relation symbols $R = \{r_1, r_2\}$ representing state invariants of the program are introduced. The arguments of the relation symbols correspond to the values of program variables that are in scope at a particular location; in this case, to the value of n. The Horn clauses in Figure 1c represent the program transitions, and include a clause with empty body for the function entry point, two clauses corresponding to the assignments in the body of the loop, and an assertion clause with head false for the program assertion.

The clauses are constructed in such a way that safety of the program is equivalent to satisfiability of the Horn clauses. Solvers search for solutions of the Horn clauses with the help of techniques like CEGAR (e.g., in HSF [20] or ELDARICA) or IC3/PDR (e.g., in Z3 [21]). Beyond just sequential programs, Horn clauses can elegantly represent also concurrent programs, programs with functions and procedures, or timed and parameterized systems (e.g., [20], [25]).

In a verification system based on Horn clauses, Horn solvers are typically interfaced either using a textual format, most often just a Horn dialect of SMT-LIB [6], or programmatically. Figure 2 shows the Horn clauses from Figure 1 in SMT-LIB, assuming that the program variable n ranges over mathematical integers. The corresponding clauses in signed bit-vector

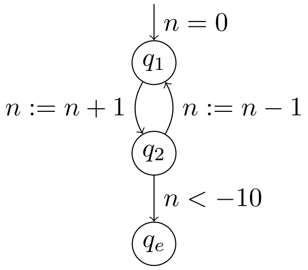
---

[1] https://github.com/uuverifiers/eldarica/

ISBN: 978-0-9835678-8-2.

```
int n = 0;
while (true) {
  n = n + 1;
  assert(n >= -10);
  n = n - 1;
}
```

(a) C program



(b) Control-flow graph

$$
\begin{aligned}
r_1(0) &\leftarrow \\
r_2(n+1) &\leftarrow r_1(n) \\
r_1(n-1) &\leftarrow r_2(n) \\
\mathit{false} &\leftarrow \neg(n \geq -10) \wedge r_2(n)
\end{aligned}
$$

(c) Horn Clause Representation

Fig. 1: Sample code with corresponding Control-Flow-Graph and Horn clauses. The clauses are satisfied by setting $r_1(n) \equiv (n = 0)$ and $r_2(n) \equiv (n = 1)$.

```
(set-logic HORN)

(declare-fun r1 (Int) Bool)
(declare-fun r2 (Int) Bool)

(assert (r1 0))
(assert (forall ((n Int))
         (=> (r1 n) (r2 (+ n 1)))))
(assert (forall ((n Int))
         (=> (r2 n) (r1 (- n 1)))))
(assert (forall ((n Int))
         (=> (and (r2 n) (not (>= n (- 10)))) false)))

(check-sat)
```

Fig. 2: The example from Figure 1 in SMT-LIB notation, with mathematical integer semantics.

```
(set-logic HORN)

(declare-fun r1 ((_ BitVec 32)) Bool)
(declare-fun r2 ((_ BitVec 32)) Bool)

(assert (r1 (_ bv0 32)))
(assert (forall ((n (_ BitVec 32)))
         (=> (r1 n) (r2 (bvadd n (_ bv1 32))))))
(assert (forall ((n (_ BitVec 32)))
         (=> (r2 n) (r1 (bvsub n (_ bv1 32))))))
(assert (forall ((n (_ BitVec 32)))
         (=> (and (r2 n)
                 (not (bvsge n (bvneg (_ bv10 32)))))
             false)))

(check-sat)
```

Fig. 3: The example from Figure 1 in SMT-LIB notation, with bit-vector semantics.

arithmetic of width 32 is shown in Figure 3. Both sets of Horn clauses can easily be proven satisfiable by ELDARICA and other tools.

B. Related Work.

Horn solvers have been implemented using a variety of algorithms, often by extending methods from hardware or software model checking to the more general case of solving sets of Horn clauses. Existing state-of-the-art tools can be classified according to their underlying solving algorithm as the following:

- **CEGAR** and predicate abstraction, such as HSF [20], Duality [30], and ELDARICA;
- **IC3/PDR**, such as the PDR engine in Z3 [21]. The algorithm implemented in SPACER [28] extends IC3/PDR by maintaining both under- and over-approximations during analysis;
- **Transformation** of Horn clauses, such as VeriMAP [13] and Rahft [26];
- **Machine learning**, such as SynthHorn [33], FreqHorn [17] and HoIce [11], which progressively drive concrete invariant samples and use machine learning classification techniques to find the inductive invariant.

Many of the solvers in addition use techniques like abstract interpretation to synthesise invariants, and this way support the main algorithm.

Compared to other Horn solvers, distinguishing features of ELDARICA are the set of convergence heuristics implemented (Section II-C), which enable ELDARICA to solve particularly tricky Horn problems, the range of supported theories (including algebraic data types and bit-vectors), and the provided API.

II. AN OVERVIEW OF ELDARICA

We start by describing the ELDARICA design and implementation. ELDARICA is open source, entirely implemented in Scala, and only depends on Java or Scala libraries,[2] which implies that ELDARICA can be used on any platform with a JVM. ELDARICA can be used as a standalone tool, but can also easily be integrated as a library into other systems implemented in Scala or Java. To reduce the JVM start-up/warm-up delay in standalone use, ELDARICA can also be run in a daemon mode.

ELDARICA uses PRINCESS [31] as SMT solver for satisfiability and implication checks, and as interpolation procedure for Presburger arithmetic [9], algebraic data-types [24], and bit-vectors [3]. The CEGAR engine of ELDARICA also loads PRINCESS as a library that provides the data-structures

---

[2] With the exception of the FLATA library optionally used for acceleration, as described below, which depends on Yices [16].

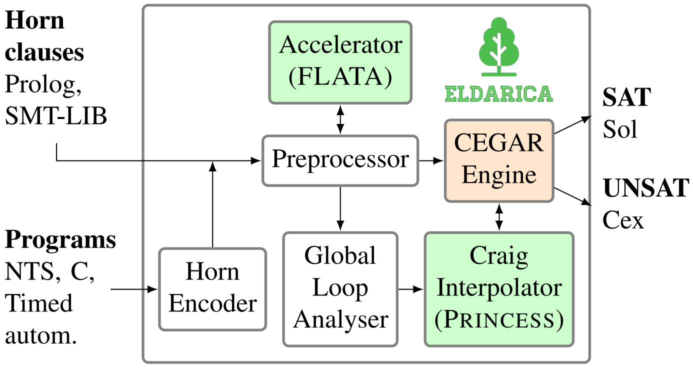


Fig. 4: The Main Architectural Components of ELDARICA

to represent terms, formulas, and background theories. This approach initially reduced the implementation effort, but also helps to speed up SMT queries because copying and conversion of expressions largely becomes unnecessary.

A. *Available Front-ends and Formats*

Figure 4 presents an overview of the ELDARICA's architecture. ELDARICA accepts input in a range of formats: the main input format for Horn clauses is (standard-compliant) SMT-LIB v2 [6], writing each clause as an explicitly quantified disjunction or implication. Support for the SMT-LIB rule dialect offered by Z3 is considered for the future. ELDARICA also supports Prolog-style input of Horn clauses over integer arithmetic.

ELDARICA is also able to parse programs in two (simple) formats, and handle the clause encoding internally. ELDARICA can read and verify input in the Numerical Transition Systems (NTS) format [1], [23], a format handled and produced by several verification tools. ELDARICA can also parse programs in a fragment of the C language (currently excluding pointers, arrays, and heap), as well as networks of timed automata in a C-like language with support for unbounded parallelism, clocks, binary communication channels, and time invariants [25].

B. *The Main Algorithms Used in* ELDARICA

To check the satisfiability of Horn clauses, ELDARICA applies lazy Cartesian predicate abstraction [19], [5], in combination with a variant of Counterexample-Guided Abstraction Refinement (CEGAR) [12], [4]. Horn clauses are first sent through a number of preprocessing stages, applying transformations such as (forward) slicing, (forward and backward) reachability analysis to eliminate dead relation symbols, clause inlining, splitting of clauses with complicated constraints or long bodies, constant propagation, abstract interpretation over an interval domain to infer basic information about variable ranges, as well as interval constraint propagation to further narrow down variable ranges.

The main CEGAR engine of ELDARICA then attempts to construct an abstract reachability (hyper)graph (ARG) that would witness satisfiability of Horn clauses, starting from a (user-provided, and often empty) set of predicates for each relation symbol. Implication properties are checked with the help of the SMT solver PRINCESS. If ARG construction fails, the obtained abstract counterexample DAG is checked for spuriousness by PRINCESS, resulting in either a concrete counterexample, or additional predicates computed through Craig interpolation. By default, ELDARICA maps the counterexample DAG to a tree interpolation problem for this purpose, but also disjunctive interpolation [32] can be switched on using the command-line option `-disj`.

C. *Convergence Heuristics*

Beyond basic CEGAR and Craig interpolation, ELDARICA applies two methods to minimise the likelihood of divergence, i.e., of the phenomenon that a model checker can sometimes fail to discover the right predicates, and continue refining the constructed abstraction indefinitely. The first method is based on *acceleration:* if during preprocessing cycles consisting of only linear clauses (with only conjunctive, numeric constraints) are detected, then precise static acceleration [10], [22] is applied to replace the cycle with a single clause with the same effect. ELDARICA loads the FLATA tool[3] as a Java library for this purpose. Acceleration later helps Craig interpolation to discover sufficiently general predicates, and has been shown to significantly extend the reach of CEGAR for tricky verification tasks [22]. Since this optimisation can sometimes slow down the model checker, and it is only applicable for cycles with linear clauses, it is optional and can be switched on with the command-line option `-stac`.

As a second method, ELDARICA uses *interpolation abstraction* [29] to control the predicates computed through Craig interpolation. Interpolation abstraction is driven by the results of a global analysis of the cycles (corresponding to program loops) present in a set of Horn clauses, including information about modified loop variables and strides of loop counters, derived during preprocessing. Among others, interpolation abstraction helps to analyse loops modifying multiple variables, e.g. the Horn clauses corresponding to the following program:

```
int x = 0, y = 0;
while (x < N) {
  x++; y++;
}
assert(N < 0 || y == N);
```

In this case, loop analysis will identify the term `x - y` as a useful expression (or *interpolation template*) in invariants, and interpolation abstraction will guide the interpolation process towards expressions that avoid the variables `x`, `y`, unless they occur in the context `x - y`. This approach enables ELDARICA to rank interpolants according to their expected generality, and has been shown to speed up the solving process, as well as to significantly reduce the possibility of divergence [29], [14]. Interpolation templates can also be specified manually by the user to control the derived predicates.

Interpolation abstraction is enabled by default, but can optionally be switched off with the option `-abstract:off`.

---

[3] http://nts.imag.fr/index.php/Flata

There is also an option `-abstractPO` for running a portfolio of two solvers, one with interpolation abstraction enabled, and one without interpolation abstraction.

III. Status of Theory Support

A. *Unbounded Integers*

The development of Eldarica initially focused on the theory of unbounded linear integer arithmetic (LIA, quantifier-free Presburger arithmetic, but also including Booleans), for which efficient Craig interpolation is well understood. Among the supported theories, linear integer arithmetic in Eldarica is at this point the most refined and mature, and has been evaluated extensively in previous work [22], [29], [14].

Based on the interpolation procedure presented in [3], we have recently also added support for non-linear integer arithmetic (NIA) to Eldarica. The handling of NIA is best-effort though: procedures for NIA are necessarily incomplete, and quantifier-free interpolants do not exist in all cases. We have not yet collected a lot of experience with NIA problems.

B. *Arrays*

Eldarica can also handle problems with arrays, and can compute quantified solutions for such problems using the transformation approach from [8]. Eldarica accepts an extended Horn fragment for problems with arrays, with additional universal quantifiers allowed in front of each occurrence of a relation symbol specifying the intended quantifier structure of solutions. As an example, we consider a program filling an array with consecutive numbers:

```
int n, int ar[];
assume(n > 0);

int i = 0;
while (i < n) {
  ar[i] = i;
  i++;
}

assert(forall int j; 0 <= j && j < n => ar[j] >= 0);
```

A simple Horn representation of this verification task, using a single relation symbol inv representing the required loop invariant, is given in Figure 5. The encoding specifies that solutions are supposed to be of the form $inv(n, i, ar) = \forall ind.\ invM(n, i, ind, ar[ind])$, where the matrix $invM$ is the actual unknown to be determined by the Horn solver.

Instead of providing the quantifier pattern explicitly in the SMT-LIB input, it is also possible to leave the introduction of quantifiers to Eldarica, and simply declare inv to be a symbol with an array argument:

```
(declare-fun inv (Int Int (Array Int Int)) Bool)
```

The number of quantifiers to be introduced can be controlled using the command-line option `-arrayQuans:`n.

C. *Algebraic Data-Types*

Moving towards version 2, we have recently added support for algebraic data-types (ADTs) with fully-free constructors to Eldarica. This makes it possible to analyse Horn clauses with common data-types like enumerations, unions, tuples, lists, or trees. Clauses can also contain *size constraints,* i.e., reason about the number of occurrences of constructor symbols in a term.[4] This can be used to talk about the length of lists or the size of trees. ADTs are handled with the help of the decision and interpolation procedure presented in [24].

```
(set-logic HORN)

(declare-fun invM (Int Int Int Int) Bool)

(define-fun inv ((n Int) (i Int)
                 (ar (Array Int Int))) Bool
  (forall ((ind Int)) (invM n i ind (select ar ind))))

(assert (forall ((n Int) (ar (Array Int Int)))
         (=> (> n 0) (inv n 0 ar))))

(assert (forall ((n Int) (i Int) (ar (Array Int Int)))
         (=> (and (inv n i ar) (< i n))
             (inv n (+ i 1) (store ar i i)))))

(assert (forall ((n Int) (i Int) (ar (Array Int Int)))
         (=> (and (inv n i ar) (>= i n) )
             (forall ((j Int))
              (=> (and (<= 0 j) (< j n))
                  (>= (select ar j) 0))))))

(check-sat)
```

Fig. 5: An array example in SMT-LIB. To solve the example using Eldarica, the option `-splitClauses` is needed.

```
 1  (set-logic HORN)
 2
 3  (declare-datatype list ((nil)
 4                          (cons (hd Int) (tl list))))
 5
 6  (declare-fun C (list list list) Bool)
 7
 8  (define-fun len ((l list)) Int (- (_size l) 1))
 9
10  (assert (forall ((y list)) (C nil y y)))
11
12  (assert (forall ((x list) (y list) (r list) (i Int))
13           (=> (C x y r)
14               (C (cons i x) y (cons i r)))))
15
16  (assert (forall ((x list) (y list) (r list))
17           (=> (and (not (= r nil)) (C x y r))
18               (or (= (hd r) (hd x))
19                   (= (hd r) (hd y))))))
20
21  (assert (forall ((x list) (y list) (r list))
22             (=> (C x y r)
23                 (= (len r) (+ (len x) (len y))))))
24
25  (check-sat)
```

Fig. 6: A list example in SMT-LIB.

Figure 6 shows a Horn problem over the data-type of lists of integers. The data-type is defined with constructors `nil`, `cons`, and selectors `hd`, `tl`. The size of a list, in terms of the number of constructor symbols, can be accessed using the built-in operator `_size`; since `_size` also counts the `nil` operator, in line 8 we define a function `len` that computes

---

[4] SMT-LIB does currently not define a size operator for ADTs, so that resulting input is not SMT-LIB compliant.

standard list length. The relation symbol `C` is then defined to compute list concatenation, and in lines 16–23 two properties of concatenation are verified. A programmatic version of the example is provided in the next section.

At this point, Eldarica is only able to compute quantifier-free (and recursion-free) solutions of Horn clauses over ADTs, which restricts the class of systems and properties that can meaningfully be analysed. For instance, Eldarica cannot derive solutions that state sortedness of an unbounded list, or the property that all list elements are positive.

D. Bit-Vectors

Eldarica version 2 also supports Horn clauses over bit-vectors, using a lazy encoding approach to map bit-vector constraints to quantifier-free Presburger constraints, which can then be solved and interpolated using the existing procedures in Princess. The details of the interpolation procedure are described in a companion paper at FMCAD 2018 [3]. Eldarica supports almost the full SMT-LIB bit-vector theory, although the interpolation procedure used for bit-vectors is optimised mainly for arithmetic constraints (as opposed to bit-wise operators) in Horn clauses. An SMT-LIB example with bit-vectors is given in Figure 3, and a programmatic example in the next section.

IV. Programmatic Use of Eldarica

A. Algebraic Data Types

Since Eldarica is implemented in Scala, it offers a convenient embedded domain-specific language for writing formulas and clauses, and can easily be integrated into other Scala applications. Integration into Java applications takes a similar form, but lacks the syntactic sugar provided through Scala, and at the moment requires the programmer to go through the slightly cumbersome process of calling Scala methods from Java. Formulas and data-types are constructed using the API of the underlying SMT solver Princess.[5]

A complete runnable example is shown in Figure 7. In line 11, debugging assertions are switched off. In lines 13–17, again the ADT of lists over integers with sort name `list`, constructors `nil`, `cons`, and selectors `hd`, `tl` is defined (mutually recursive data-types can be created similarly). Lines 26–29 declare variables of sort integer and list, respectively, and line 31 a ternary relation symbol `C` over lists. The clauses in lines 34–35 are written in Prolog-like notation, and axiomatise `C` to represent concatenation. In line 39, a property about the head of a list resulting from concatenation is stated as a third clause. In line 41 the satisfiability of the three clauses is checked, with solution $C(x, y, r) \equiv y = r \vee hd(r) = hd(x)$.

To run the example, it is only necessary to have the Scala build tool `sbt` installed, which is included in many Linux distributions. Further dependencies, such as the Scala compiler and Eldarica itself, will be downloaded automatically by the command `sbt run`.

```
 1  // List-example.scala
 2
 3  import ap.SimpleAPI
 4  import ap.theories.ADT
 5  import lazabs.horn.bottomup._
 6  import ADT._
 7  import HornClauses._
 8  import ap.parser.IExpression._
 9
10  object ListExample extends App {
11    lazabs.GlobalParameters.get.assertions = false
12
13    val listADT = new ADT (Seq(”list”),
14      Seq((”nil”,  CtorSignature(Seq(), ADTSort(0))),
15          (”cons”, CtorSignature(Seq(
16                      (”hd”, OtherSort(Sort.Integer)),
17                      (”tl”, ADTSort(0))),
18                    ADTSort(0)))))
19
20    val Seq(list)            = listADT.sorts
21    val Seq(nil, cons)       = listADT.constructors
22    val Seq(_, Seq(hd, tl)) = listADT.selectors
23
24    SimpleAPI.withProver { p =>
25      import p._
26
27      val n = createConstant(”n”, Sort.Integer)
28      val x = createConstant(”x”, list)
29      val y = createConstant(”y”, list)
30      val r = createConstant(”r”, list)
31
32      val C = createRelation(”C”, Seq(list, list, list))
33
34      val defClauses = List(
35        C(nil(), y, y)                 :- true,
36        C(cons(n, x), y, cons(n, r)) :- C(x, y, r)
37      )
38
39      val prop =
40        (hd(x) === hd(r) | hd(y) === hd(r)) :- (
41           C(x, y, r), r =/= nil())
42
43      SimpleWrapper.solve(prop :: defClauses) match {
44        case Left(sol) =>
45          println(”sat”); println(sol mapValues (pp(_)))
46        case Right(cex) =>
47          println(”unsat”); println(cex)
48      }
49    }
50  }
```

```
name := ”list−example”
scalaVersion := ”2.11.8”
resolvers += ”uuverifiers” at ”http://logicrunch.↩
     research.it.uu.se/maven/”
libraryDependencies += ”uuverifiers” %% ”eldarica” % ”↩
     2.0−alpha2”
```

```
// Output of the program
> sbt run
[...]
sat
Map(C/3 -> _1 = _2 | hd(_2) = hd(_0))
[...]
```

Fig. 7: Runnable Eldarica example, analysing Horn clauses over the data-type of lists. The program can be compiled and run with the command `sbt run`, which takes care of downloading all dependencies (including Eldarica itself), compilation, and execution.

---

[5] http://www.philipp.ruemmer.org/princess/doc/

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Benchmarks	#	Int		BV	
		ELDARICA sat/unsat	Z3 sat/unsat	ELDARICA sat/unsat	Z3 sat/unsat
Consistency	56	27/27	28/27	5/16	0/0
HOLA [15]	46	45/0	36/0	29/4	1/0
IntDualyzer	6	5/1	5/1	3/3	1/0
SLayer (chain.)	68	0/6	17/34	0/2	10/28
SLayer (fan.)	66	0/6	20/31	0/0	15/24
qarmc	13	9/1	11/1	5/1	0/0
ssh-simplified	23	13/8	9/9	13/6	1/0

Fig. 8: Results for ELDARICA 2.0-alpha3 and Z3 4.7.1 on integer and bit-vector benchmarks, an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime was limited to 30min wall clock time, and heap space to 2GB. The table shows total number of benchmarks and the number of the benchmarks that each solver could solve.

B. *Bit-vectors*

We show an example of Horn clauses over bit-vectors in Figure 9. The overall structure of the program is similar as in the previous section. Bit-vector expressions are again constructed using the corresponding PRINCESS API, with the bit-vector operators provided in class `ModuloArithmetic`. The expression `bv(32, n)` generates the literal 32-bit constant n, while `bvadd` represents bit-vector addition. More generally, the bit-vector API offers access to the complete SMT-LIB bit-vector theory. The option `useTemplates` of the `SimpleWrapper` enables interpolation abstraction, which is in the API disabled by default.

V. EXPERIMENTAL RESULTS

Extensive experimental evaluations of ELDARICA have been published in multiple recent research papers [29], [14], we only report some experiments on some of the new features of ELDARICA version 2. Figure 8 shows a comparison of ELDARICA 2.0-alpha3[6] and Z3 4.7.1 on integer and bit-vector benchmarks. ELDARICA was run with the option `-abstractPO`, and Z3 with default options.

We use a collection of benchmarks in linear integer arithmetic from various sources.[7] C programs from HOLA [15] were first translated to NTS using Frama-C, and then to Horn clauses by ELDARICA. Since there are not many benchmarks for Horn clauses in bit-vector arithmetic, we wrote a script to convert all the operations in linear integer arithmetic to their equivalent bit-vector operations (32 bit signed). Using the script we transformed the original linear integer arithmetic benchmarks to bit-vector benchmarks. Of course, this can potentially change the satisfiability of the original benchmark, but it is useful for making a library of benchmarks of Horn clauses in bit-vector arithmetic.

The experiments show that ELDARICA performs well on most benchmark families. This might be due to the effective convergence heuristics in ELDARICA (Section II-C). An

```
// BV-example.scala

import ap.SimpleAPI
import ap.theories.ModuloArithmetic._
import lazabs.horn.bottomup._
import HornClauses._
import ap.parser.IExpression._

object BVExample extends App {
  lazabs.GlobalParameters.get.assertions = false

  SimpleAPI.withProver { p =>
    import p._

    val x = createConstant(”x”, UnsignedBVSort(32))
    val y = createConstant(”y”, UnsignedBVSort(32))

    val C = createRelation(”C”,
              Seq(UnsignedBVSort(32),
                  UnsignedBVSort(32)))
    val D = createRelation(”D”,
              Seq(UnsignedBVSort(32),
                  UnsignedBVSort(32)))

    val defClauses = List(
      C(bv(32, 1), bv(32, 1)) :- true,
      C(bvadd(x, bv(32, 1)),
        bvadd(bv(32, 1), y))  :- C(x, y),
      D(x, y)                 :- (C(x, y),
                                   x === bv(32, 0))
    )

    val prop =
      (y === bv(32, 0)) :- D(x, y)

    SimpleWrapper.solve(prop :: defClauses,
                        useTemplates = true) match {
      case Left(sol) =>
        println(”sat”); println(sol mapValues (pp(_)))
      case Right(cex) =>
        println(”unsat”); println(cex)
    }
  }
}
```

```
name := ”eldarica-example”
scalaVersion := ”2.11.8”
resolvers += ”uuverifiers” at ”http://logicrunch.↩
     research.it.uu.se/maven/”
libraryDependencies += ”uuverifiers” %% ”eldarica” % ”↩
     2.0-alpha2”
```

```
// Output of the program
> sbt run
[...]
sat
Map(C/2 -> _0 = _1, D/2 -> _1 = 0 & _0 = 0)
[...]
>
```

Fig. 9: Runnable ELDARICA example, analysing Horn clauses over bit-vectors. As in Figure 7, the program can be compiled and run with the command `sbt run`.

---

[6] https://github.com/uuverifiers/eldarica/releases

[7] Benchmarks available at https://github.com/chc-comp/eldarica-misc and https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses/

exception are the benchmarks in the SLayer families, which are solved more efficiently by Z3, possibly due to a large number of Boolean relation symbols arguments. Converting the problems to bit-vector semantics tends to produce harder benchmarks for both solvers. On many families ELDARICA can still solve a comparable number of problems, but generally fewer than with integer semantics.

VI. ADOPTION

ELDARICA has been used in a variety of applications, we list some examples. CoCoSim [2] is an analysis and code generation framework for Simulink that uses ELDARICA as one possible back-end. Similarly, JayHorn [27], a software model checking tool for Java supports ELDARICA as one of its back-ends. VAC [18] (Verifier of Access Control) an automatic tool for the analysis of Administrative Role Based Access Control (ARBAC) policies also relies on ELDARICA for solving Horn clauses. ELDARICA has also been used for the analysis of business processes expressed as Petri nets [29].

VII. CONCLUSIONS

ELDARICA is an efficient open source Horn solver supporting integer arithmetic, arrays, algebraic data types, and bit-vectors. It supports various input formats including SMT-LIB, Prolog, and numerical transition systems, and provides a Scala API. Future work includes (i) integration of further background theories, (ii) further improved heuristics to solve Horn clauses while avoiding divergence, (iii) generation of *quantified* solutions for problems with algebraic data types, and (iv) optimisation.

Acknowledgements: We thank Viktor Kuncak, Radu Iosif, Filip Konečný, and Pavle Subotic for their contributions to the ELDARICA project. We thank the reviewers for helpful comments. This work was supported by the Swedish Research Council (VR) under grant 2014-5484, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

REFERENCES

[1] http://nts.imag.fr.
[2] https://coco-team.github.io/cocosim/.
[3] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljić. Bit-vector interpolation and quantifier elimination by lazy reduction. In *FMCAD*, 2018. To appear.
[4] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN PLDI*, 2001.
[5] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283. Springer, 2001.
[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.
[7] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.
[8] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.
[9] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.
[10] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *TACAS*, pages 428–442, 2008.
[11] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. ICE-based refinement type discovery for higher-order functional programs. In *TACAS*, pages 365–384, 2018.
[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[13] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. VeriMAP: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
[14] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In *NFM*, pages 265–281, 2017.
[15] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456, 2013.
[16] B. Dutertre and L. de Moura. The YICES SMT solver. http://yices.csl.sri.com/.
[17] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *FMCAD*, pages 100–107, Austin, TX, 2017.
[18] Anna Lisa Ferrara, P. Madhusudan, Truc L. Nguyen, and Gennaro Parlato. VAC — verifier of administrative role-based access control policies. In *CAV*, pages 184–191, 2014.
[19] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
[20] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
[21] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.
[22] Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *ATVA*, pages 187–202, 2012.
[23] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM 2012*, pages 247–251, 2012.
[24] Hossein Hojjat and Philipp Rümmer. Deciding and interpolating algebraic data types by reduction. In *SYNASC*, 2017.
[25] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In *HCVS*, pages 39–52, 2014.
[26] Bishoksan Kafle, John P. Gallagher, and José F. Morales. Rahft: A tool for verifying horn clauses using abstract interpretation and finite tree automata. In *CAV*, pages 261–268, 2016.
[27] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. JayHorn: A framework for verifying Java programs. In *CAV*, pages 352–358, 2016.
[28] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.
[29] Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. Guiding Craig interpolation with domain-specific abstractions. *Acta Inf.*, 53(4):387–424, 2016.
[30] Kenneth McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical report, January 2013. MSR-TR-2013-6.
[31] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
[32] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.
[33] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *ACM SIGPLAN PLDI*, pages 707–721, 2018.

ISBN: 978-0-9835678-8-2.

TRAU : SMT solver for string constraints

Parosh Aziz Abdulla
Uppsala University, Sweden
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica, Taiwan
yfc@iis.sinica.edu.tw

Bui Phi Diep
Uppsala University, Sweden
bui.phi-diep@it.uu.se

Lukáš Holík
Brno University of Technology, Czech Republic
holik@fit.vutbr.cz

Ahmed Rezine
Linköping University, Sweden
ahmed.rezine@liu.se

Philipp Rümmer
Uppsala University, Sweden
philipp.ruemmer@it.uu.se

Abstract—**We introduce TRAU, an SMT solver for an expressive constraint language, including word equations, length constraints, context-free membership queries, and transducer constraints. The satisfiability problem for such a class of constraints is in general undecidable. The key idea behind TRAU is a technique called flattening, which searches for satisfying assignments that follow simple patterns. TRAU implements a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. The approximations are refined in an automatic manner by information flow between the two modules. The technique implemented by TRAU can handle a rich class of string constraints and has better performance than state-of-the-art string solvers.**

I. INTRODUCTION

The recent years have seen a wealth of research on *string constraints*, in particular in the form of SMT solvers that can efficiently check satisfiability of quantifier-free formulas over a background theory of strings and regular expressions (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]). String solvers can be applied in a variety of verification approaches, for instance in software model checking to take care of implication and path feasibility checks; the most widespread adoption has occurred in the area of *security analysis* for languages like JavaScript and PHP, for instance to discover information leaks or vulnerability to injection attacks (e.g., [12], [13], [14]). To process constraints from those domains, it is necessary for string solvers to handle a delicate combination of (theoretically and practically) highly challenging operations: *concatenation* in word equations, to model assignments in programs; *context-free grammar,* to model properties or attack patterns; *string length,* to express string manipulation in programs; and *transduction,* to express sanitisation, escape operations, and replacement operations in strings. Since the full combination of those theories is known to be undecidable, many SMT solvers are complete only for certain fragments of the full logic.

In this paper, we present TRAU, an SMT solver for string constraints, that can handle all of the above mentioned operations. TRAU implements the framework of Counter-Example Guided Abstraction Refinement (CEGAR) proposed in [8]. This framework contains both an under- and an over-approximation module. The key idea behind TRAU is a technique called flattening [8]. It is based on the observation that both satisfiability and unsatisfiability of common constraints can be shown through witnesses of simple patterns that can be captured by flat languages (i.e., a language consisting of the set of words in $w_1^* w_2^* \cdots w_n^*$ where $w_1, w_2, \ldots, w_n$ are finite words). Compared to [8], TRAU implements several optimizations that are keys to its current efficiency (namely, a precise and efficient over-approximation module and a better strategy for splitting equalities). Furthermore, TRAU can handle efficiently the case of *transduction,* which is the string operation that is currently least well supported in existing string solvers, albeit extremely important for security analysis, and often a bottleneck in applications. (Observe that the tool in [8] does not support transducer constraints.) We show that transduction can elegantly be reduced to context-free membership constrains. In fact, the technique implemented by TRAU can handle a rich class of string constraints and has better performance than state-of-the-art string solvers.

Related Work. During the last years, several SMT solvers for strings and related logics have been introduced. A number of tools handle string constraints, including context-free membership, by fixing an upper bound on the length of the possible solutions (e.g., [1], [12], [13], [15], [16]). In contrast, the under-approximation module of TRAU does not impose any bound on the length of solutions but rather limits the search only for solutions that belong to flat languages in a similar manner to [8]. More recently, DPLL(T)-based string solvers lift the restriction of strings of bounded length; this generation of solvers includes Z3-str [3], CVC4 [5], S3 [4], Norn [17], and Sloth [11]. Most of those solvers are more restrictive than TRAU in their support for language constraints. To the best of our knowledge, TRAU and Hampi [1] are the only string solvers which can handle context-free membership constraints. Observe that TRAU does not impose any bound on the length of the solutions while Hampi does. Furthermore, TRAU implements a DPLL(T)-style proof procedure for strings in a similar manner to [17] in order to gain in efficiency. Another related technique are automata-based solvers for analyzing string-manipulated programs (e.g., [2], [6], [18]). However, many kinds of constraints, including length constraints, word equations, and context-free grammars, cannot be handled by such automata-based solvers in a complete manner. Compared

ISBN: 978-0-9835678-8-2.

to [8], TRAU implements several optimizations, including a DPLL(T)-style proof procedure, that are keys to its current efficiency. Furthermore, TRAU supports transducer constraints which is not the case of [8].

II. PRELIMINARIES

Let Σ be a finite alphabet. We use Σ^* to denote the set of finite words over Σ, and use ϵ to denote the empty word. For a word $w \in \Sigma^*$, we use $\texttt{length}(w)$ to denote the length of w. We denote by w^R the reverse image of w. A language $L \subseteq \Sigma^*$ is said to be $(\mathbb{p}, \mathbb{q})$-*flat*, for some $\mathbb{p}, \mathbb{q} \in \mathbb{N}$, if there are words $w_1, w_2, \ldots, w_{\mathbb{q}} \in \Sigma^*$ such that $\texttt{length}(w_i) \leq \mathbb{p}$ for all $i : 1 \leq i \leq \mathbb{q}$, and $L = (w_1)^* \cdot (w_2)^* \cdots (w_{\mathbb{q}})^*$.

A *Context-Free Grammar* (CFG) is defined by a quadruple $\mathcal{G} = \langle N, T, P, S \rangle$ where N is a finite set of *non-terminals*, T is a finite set of *terminals*, P is a finite set of *productions*, and $S \in N$ is the *start symbol*. The language $\mathcal{L}(\mathcal{G})$ of the grammar $\mathcal{G}$ is defined in the standard manner.

A *Pushdown Automaton* (PDA) is defined by $\mathcal{P} = \langle Q, \Sigma, \Gamma, \Delta, q_{init}, q_{acc} \rangle$ where Q is a finite set of *states*, Σ is a finite input alphabet, Γ is a stack alphabet, $\Delta \subseteq (Q \times \Gamma^* \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \times Q)$ is a finite set of transitions, $q_{init} \in Q$ is the *initial state*, and $q_{acc} \in Q$ is the *accepting state*. The language $\mathcal{L}(\mathcal{P})$ of the pushdown automaton $\mathcal{P}$ is defined in the standard manner (where the stack content is empty at the initial and final configurations). It is well-known that the class of languages accepted by pushdown automata and the one accepted by context free grammars coincide (i.e., given a pushdown automaton $\mathcal{P}$ (resp. a context-free grammar $\mathcal{G}$), one can construct a context-free grammar $\mathcal{G}$ (resp. a pushdown automaton $\mathcal{P}$) such that $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{G})$).

A *Finite-State Transducer* is $\mathcal{T} = \langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$, where Q is a finite set of *states*, Σ is a finite alphabet, $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $q_{init} \in Q$ is the *initial state*, and $q_{acc} \in Q$ is the *accepting state*. For words $w_1, w_2 \in \Sigma^*$, we write $w_2 \in \mathcal{T}(w_1)$ to denote that there is a sequence $q_0 \langle a_1, b_1 \rangle q_1 \langle a_2, b_2 \rangle \cdots \langle a_n, b_n \rangle q_n$ such that $q_0 = q_{init}$, $q_n = q_{acc}$, $\langle q_i, \langle a_{i+1}, b_{i+1} \rangle, q_{i+1} \rangle \in \Delta$ for all $i : 0 \leq i < n$, $w_1 = a_1 a_2 \cdots a_n$, and $w_2 = b_1 b_2 \cdots b_n$.

III. THE STRING CONSTRAINT LANGUAGE

In this section, we define string constraints over a finite alphabet Σ and a finite set of variables $\mathbb{X}$ ranging over Σ^*. The syntax of a formula ψ is given in Figure 1. ψ is given in the conjunctive normal form where each literal clause can be either a string (dis-)equality ϕ_s, a context-free membership ϕ_g, a transducer constraint ϕ_t or an arithmetic constraint ϕ_i. A string equality (resp. disequality) is of the form $tr_s = tr_s$ (resp. $(tr_s \neq tr_s)$) where tr_s is a (string) term. Each string term tr_s is a sequence composed of variables in $\mathbb{X}$ and symbols from Σ.

Formally, a string term is either a word $w \in \Sigma^*$, a string variable $x \in \mathbb{X}$ or a concatenation of two string terms. A transducer constraint is of the form $tr_s \in \mathcal{T}(tr_s)$ where $\mathcal{T}$ is a transducer and tr_s is a string term. A context-free grammar membership constraint is of the form $tr_s \in \mathcal{G}$ where $\mathcal{G}$ is a context-free grammar and tr_s is a string term. An arithmetic constraint ϕ_i is a relational expression between two integer terms tr_i where an integer term is either the length of a string term $\texttt{length}(tr_s)$ or an integer k.

$$
\begin{array}{rcl}
\psi & ::= & \phi \mid \psi \wedge \psi \\
\phi & ::= & \phi_s \mid \phi_i \mid \phi_t \mid \phi_g \\
\phi_s & ::= & tr_s = tr_s \mid tr_s \neq tr_s \\
\phi_t & ::= & tr_s \in \mathcal{T}(tr_s) \\
\phi_g & ::= & tr_s \in \mathcal{G} \\
\phi_i & ::= & tr_i \geq tr_i \\
tr_s & ::= & w \mid x \mid tr_s \bullet tr_s \\
tr_i & ::= & \texttt{length}(tr_s) \mid k
\end{array}
$$

Fig. 1: Constraint Syntax

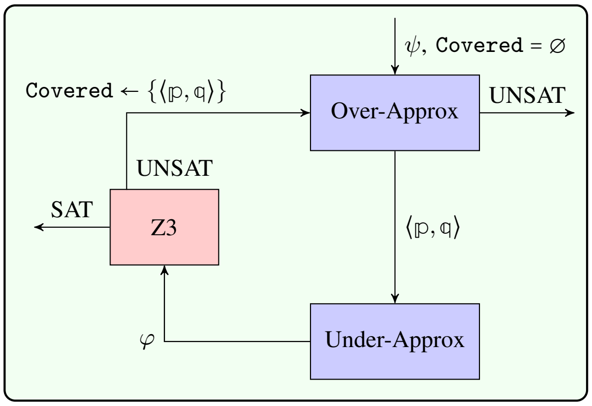


Fig. 2: Architecture of TRAU

The formula ψ is said to be *satisfiable* iff there is an interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$ such that η satisfies ψ. Otherwise, it is said to be *unsatisfiable*.

IV. ARCHITECTURE OVERVIEW

In this section, we present the architecture of our tool TRAU which checks the satisfiability of string constraint formulae (as defined in Section III). The architecture of TRAU is shown in Figure 2. TRAU consists of two main modules, namely the *Over-Approx* module and the *Under-Approx* module. It uses the SMT solver Z3 to handle arithmetic constraints.

The Over-Approx module takes as input a formula ψ and a finite set $\texttt{Covered} \subseteq \mathbb{N}^2$ of (abstract) parameters. The set $\texttt{Covered}$ is empty at the beginning. This set stores abstract parameters used by the Under-Approx module to check the satisfiability in previous iterations. The Over-Approx then constructs an over-approximation ψ' of ψ. The formula ψ' is constructed such that it falls in the decidable fragment of the theory of strings with regular membership constraints and length constraints [7]. Thus, we are able to apply similar techniques as the ones used in Norn [7] to check the satisfiability of ψ'. If ψ' is unsatisfiable, then ψ is unsatisfiable, and TRAU terminates. If ψ' is satisfiable, a satisfying assignment for ψ' is returned. Then we extract an abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle \in \mathbb{N}^2$ from the satisfying interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$

as follows: α is one of minimal pairs such that for any variable $x \in \mathbb{X}$, the word $\eta(x)$ belongs to an *α-flat language* [8].

The Under-Approx module takes as input the abstract parameter α and the set of constraints ψ. It limits the search only for solutions of ψ that belong to an *α-flat language*. By [8], checking the existence of a solution ψ that belongs to an *α-flat language* can be reduced to the satisfiability problem of an existential Presburger formula. Therefore, the Under-Approx module produces as output an existential Presburger formula φ such that φ is satisfiable iff there is an interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$ such that η satisfies ψ and for every variable $x \in \mathbb{X}$, we have that $\eta(x)$ belongs to an *α-flat language*.

Then, Z3 checks the satisfiability of the existential Presburger formula φ. If Z3 returns that φ is satisfiable, then we deduce that ψ is also satisfiable. In that case, we can even construct an interpretation η that satisfies ψ, and TRAU terminates. In the case Z3 returns that φ is unsatisfiable, we are unable to find a solution of ψ that is accepted by an *α-flat language*. Thus, α is added to the set `Covered` and the control is given back to the *Over-Approx* module to produce a new pair α which is not in `Covered` (by requiring that the solutions do not belong to an *α-flat language*).

V. EFFICIENT HANDLING OF TRANSDUCER CONSTRAINTS

TRAU handles transducer constraints differently from the method presented in [8]. Rather than extending the Under-Approx module to transducers, we transform transducer constraints to context-free membership constraints. Let ψ be a string constraint and let ϕ_t be a transducer constraint appearing in ψ. Let us assume that ϕ_t is of the form $t' \in \mathcal{T}(t)$ where $\mathcal{T} = \langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$ is a transducer and t and t' are string terms. In order to construct the context-free membership constraints, we first construct a pushdown automaton $\mathcal{P}$ such that a word w is accepted by $\mathcal{P}$ iff there are two words u and v such that $u \in \mathcal{T}(v)$ and $w = v \cdot \sharp \cdot u^R$ where $\sharp$ is a fresh symbol (not in Σ). The pushdown automaton $\mathcal{P} = \langle Q \cup \{q_{final}\}, \Sigma \cup \{\sharp\}, \Sigma, \Delta', q_{init}, q_{final} \rangle$ has the same set of states as $\mathcal{T}$ plus one extra accepting state $q_{final} \notin Q$. Any accepting run of $\mathcal{P}$ can be split into two phases. In the first phase, the pushdown automaton simulates the transducer by: (i) performing the same changes on the state, (ii) reading the same input letter, and (iii) pushing into the stack the output letter read by the transducer. Formally, for each transition $\langle q, \langle a, b \rangle, q' \rangle$ of $\mathcal{T}$, the pushdown automaton $\mathcal{P}$ has a transition of the form $\langle q, \epsilon, a, b, q' \rangle$. At the end of this phase, the pushdown automaton reaches the same state as the transducer, reads the same input word, and stores the output word read by the transducer into its stack. The second phase of the pushdown automaton $\mathcal{P}$ starts, in non-deterministic manner, when its current state is q_{acc}. First, the pushdown moves its state from q_{acc} to q_{final} while reading the special $\sharp$ (i.e., the pushdown automaton $\mathcal{P}$ has the following transition $\langle q_{acc}, \epsilon, \sharp, \epsilon, q_{final} \rangle$). From the state q_{final}, the pushdown automaton $\mathcal{P}$ starts emptying its stack while reading each popped symbol (i.e., the pushdown automaton $\mathcal{P}$ has a transition of the form $\langle q_{final}, a, a, \epsilon, q_{final} \rangle$ for each letter $a \in \Sigma$). It is easy to see that a word w is in $\mathcal{L}(\mathcal{P})$ iff there are two words u and v such that $u \in \mathcal{T}(v)$ and $w = v \cdot \sharp \cdot u^R$.

Let $\mathcal{G}$ be a context-free grammar that accepts the same language as the pushdown automaton $\mathcal{P}$ (i.e., $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{P})$). Let $\mathcal{G}_1$ (resp. $\mathcal{G}_2$) be the context-free grammar that accepts exactly the following set of words $\{w \cdot \sharp \cdot w^R \,|\, w \in \Sigma^*\}$ (resp. Σ^*).

Now, we can replace the transducer constraint ϕ_t by the conjunction of the following context-free membership constraints: $t \cdot \sharp \cdot y \in \mathcal{G}$, $y \cdot \sharp \cdot t' \in \mathcal{G}_1$ and $t \cdot y \cdot t' \in \mathcal{G}_2$ where y is a fresh variable. Observe that we need the constraint $t \cdot y \cdot t' \in \mathcal{G}_2$ to enforce that the interpretations $\eta(y)$, $\eta(t)$, and $\eta(t')$ are over the alphabet Σ (since the alphabet of the newly constructed formulas is $\{\Sigma \cup \sharp\}$). Let us assume that ψ' is the string constraint obtained from ψ by replacing any transducer constraint by the conjunction of the three context-free membership constraints (constructed as described above). Then, it is easy to see that ψ is satisfiable iff ψ' is satisfiable.

VI. OPTIMIZING THE OVER-APPROXIMATION MODULE

Suppose that we have a constraint formula ψ together with a set $\texttt{Covered} \subseteq \mathbb{N}^2$ of parameter values. We assume w.l.o.g. that ψ does not contain any transducer constraints (see Section V). The over-approximation module in [8] proceeds as follows: First, it replaces any context-free membership constraint of the form $tr_s \in \mathcal{G}$ in ψ by a constraint of the form $tr_s \in L$ where L is a regular language accepting the upward closure of $\mathcal{L}(\mathcal{G})$ [19], [20]. Then, it limits the search only for solutions that do not belong to any *α-flat language* with $\alpha \in \texttt{Covered}$. Finally, it replaces any occurrence of a variable x by a fresh copy of x that satisfies the same word equation, membership and length constraints as x. The resulting string constraints falls in the decidable fragment of the theory of strings [7], [17]. In contrast, TRAU adopts a lazy approach in the replacement of variables. More precisely, TRAU starts by choosing an occurrence of a variable x to replace by a fresh copy that satisfies the same membership and length constraints. Then, TRAU checks if the resulting string constraint satisfies the *acyclicity* condition of [7], [17]. If it is the case then the replacement procedure terminates. Otherwise, TRAU chooses another occurrence of a variable to replace by a fresh copy.

VII. OPTIMIZING THE UNDER-APPROXIMATION MODULE

We present one important optimization that TRAU implements. This optimization significantly improves the Under-Approx module (implemented in [8]) when applied to equality constraints. In practice, after flattening an equality constraint (i.e., computing a finite-state automaton that characterizes the intersection of flat languages), the size of the constructed automaton A could become fairly large. Consequently, the arithmetic SMT solver may have poor performance when checking the satisfiability of the constructed existential Presburger formula characterizing the Parikh image [21], [22] of A. We found that problem can be improved by combining the flattening technique proposed in [8] with the DPLL(T)-style proof procedure and the length-guided splitting of equalities

		CVC4	Z3-str3	S3P	Trau-pre	Trau
Kaluza suite	sat	35235	34495	**35264**	35202	**35264**
	unsat	**12014**	11799	**12014**	12019	**12014**
	timeout	35	350	6	63	6
	error/unknown	0	640	0	0	0
PISA suite	sat	7	**8**	6	-	**8**
	unsat	4	**4**	1	-	**4**
	timeout	0	0	5	-	0
	error/unknown	1	0	0	-	0
AppScan suite	sat	7	**8**	6	-	**8**
	unsat	0	0	0	-	0
	timeout	1	0	1	-	0
	error/unknown	0	0	1	-	0
Transducer suite	sat	-	-	3	-	**11**
	unsat	-	-	10	-	2
	timeout	-	-	0	-	4
	error/unknown	-	-	4	-	0
StringFuzz suite	sat	618	605	-	-	**723**
	unsat	160	190	-	-	**261**
	timeout	247	207	-	-	0
	error/unknown	0	23	-	-	41

TABLE I: Experimental results. All *satisfying* results of Trau are cross-checked by S3P to guarantee correct solutions. Runtime was limited to 20s for the Kaluza, PISA, AppScan, StringFuzz suites and to 100s for the Transducer suite. The row "(*un*)*sat*" indicates the number of benchmarks for which the solvers report (un)satisfiable.

procedure used in [7]. This is mainly due to the fact that we limit the search for solutions that belong to α-flat languages.

Fix a set of constraints ψ, a finite set of variables $\mathbb{X}$, and an abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. To handle the equality constraints efficiently, we proceed as follows: First, we construct the string constraint ψ' by replacing any occurrence of a variable x in ψ, that belongs to an $(\mathbb{p}, \mathbb{q})$-flat language, by $x_1 \cdot x_2 \cdots x_{\mathbb{q}}$ where $x_1, x_2, \ldots, x_{\mathbb{q}}$ are fresh variables that belong to $(\mathbb{p}, 1)$-flat languages. Assume w.l.o.g that ψ' contains an equality constraint ϕ_s of the form $x_1 \cdots x_m = y_1 \cdot y_2 \cdots y_n$. Observe that $x_1, \ldots, x_m, y_1, \ldots, y_n$ belong to $(\mathbb{p}, 1)$-flat languages. Then, for every $j : 1 \leq j \leq m$ (resp. $i : 1 \leq i \leq n$), we construct a string constraint φ (resp. φ') from ψ' by: (1) deleting the equality constraint ϕ_s from ψ', (2) replacing any occurrence of the variable y_1 (resp. x_1) by $x_1 \cdot x_2 \cdots x_j$ (resp. $y_1 \cdot y_2 \cdots y_i$), and (3) adding the equality constraint $x_{j+1} \cdots x_m = y_2 \cdots y_n$ (resp. $x_2 \cdots x_m = y_{i+1} \cdots y_n$). For each string constraint φ (resp. ψ'), we repeat the procedure of splitting of the equality constraints until the obtained string constraint does not contain equality constraints. Finally, we declare the string constraint ψ to be satisfiable if one of the constructed string constraints is satisfiable; otherwise we add the abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$ to the set `Covered`.

Observe that such a splitting strategy will limit the search space for solutions to a subset of $(\mathbb{p}, \mathbb{q})$-flat languages. However, this is not a restriction since if ψ is satisfiable then for the abstract parameter $\alpha = \langle 1, \mathbb{q} \rangle$, with $\mathbb{q}$ is the maximal length of the strings appearing in a satisfying assignment of ψ, the splitting strategy will lead to a satisfiable string constraint.

This splitting strategy is also significantly improved by using a DPLL(T)-Style proof procedure and a length-guided splitting procedure as in [7].

VIII. Experimental Results

In this section, we describe the experimental evaluation of the Trau solver to validate the effectiveness of the techniques presented in the paper. We have implemented Trau as an open source solver and used Z3 [23] as the SMT solver to handle generated arithmetic constraints from the Under-Approx module. Trau takes inputs in SMTLIB format. Trau does not run any parts concurrently to boost the performance. We compare Trau against four other state-of-the-art string solvers, namely Z3-str3 [10], CVC4 [5], [24] (the newest version), S3P [25], and Trau-pre [26]. We do not compare with Sloth [11] since it does not support length constraints which disqualifies it in a majority of our test cases. For our comparison with Z3-str3, we use the version that is part of Z3 4.6. Each benchmark suite draws from real world applications with diverse characteristics. The summary of the results is given in Table I. All experiments were performed on an Intel Core i7 2.7Ghz with 8 GB of RAM. In most experiments, the time limit is 20s since it is widely used in the evaluation of other string solvers.

Kaluza suite. The Kaluza suite [12] is generated by a JavaScript symbolic execution engine. It consists of 47284 test cases, including length, regular and (dis)equality constraints. For this suite, CVC4 times out on 35 cases while Trau-pre times out on 63 cases. Z3-str3 times out on 350 cases and

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

cannot answer on 640 cases. TRAU and S3P have the same performance, which is better than the other solvers as they time out only in 6 cases. When increasing the timeout to 40s, TRAU can solve all the remaining cases (they all are *sat* cases) while other solvers cannot.

PISA and AppScan suite. The PISA suite includes constraints from real-world Java sanitizer methods that were used in the evaluation of the PISA system [27]. The suite has 12 tests, including transducer constraints such as Substring, IndexOf, and Replace operations. The AppScan suite is derived from security warnings output by IBM Security AppScan Source Edition [28]. The suite has 8 tests, including transducer constraints and (dis)equality constraints. In both suites, the performance of TRAU is comparable to Z3-str3 (they are able to solve all test cases). CVC4 cannot give an answer for 1 test case in each suite. TRAU-PRE cannot run these suites since it does not support transducer constraints.

Transducer suite. The Transducer suite is inspired by the Google closure library [29], which supports sanitizing strings to protect websites from vulnerabilities. The suite has 17 tests, including transducer constraints such as Replace and ReplaceAll. Since only S3P and TRAU support ReplaceAll constraints, we do not include Z3-str3, CVC4, and TRAU-PRE in this comparison. Within the time limit, TRAU showed the satisfiability of 11 tests while S3P did it only for 3 tests.

StringFuzz suite. StringFuzz [30] is a fuzzer for automatically generating SMT-LIB string constraints. StringFuzz can help in exposing bugs and performance issues for string solvers. We use StringFuzz to generate 1025 tests including word (dis)equalities and regular membership constraints. These generated tests consist of a combination of small and large examples (in terms of the number of used variables and expected lengths of satisfying string assignments). TRAU can solve 984 tests (of them 723 tests are *sat* and 261 tests are *unsat*) in the suite. CVC4 and Z3-str3 can determine the satisfiability of only 778 and 795 tests, respectively. We do not run S3P and TRAU-PRE because they do not support some constraints in the suite. TRAU gives up in 41 tests containing non-membership constraints that are currently not supported.

ACKNOWLEDGEMENTS

This research has been partially supported by the Swedish Research Council (VR) under grant 2014-5484, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), the Czech Science Foundation project 16-24707Y, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), the FIT BUT internal project FIT-S-17-4014, and the Ministry of Science and Technology of Taiwan (project 106- 2221-E-001-009-MY3).

REFERENCES

[1] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A Solver for String Constraints," in *ISTA'09*. ACM, 2009, pp. 105–116.

[2] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *TACAS'10*, ser. LNCS, vol. 6015. Springer, 2010, pp. 154–157.

[3] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A Z3-based string solver for web application analysis," in *ESEC/FSE'13*. ACM, 2013, pp. 114–124.

[4] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *CCS'14*. ACM, 2014, pp. 1232–1243.

[5] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A DPLL(T) theory solver for a theory of strings and regular expressions," in *CAV'14*, ser. LNCS, vol. 8559. Springer, 2014, pp. 646–662.

[6] S. Kausler and E. Sherman, "Evaluation of string constraint solvers in the context of symbolic execution," in *ASE '14*. ACM, 2014, pp. 259–270.

[7] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *CAV'14*, ser. LNCS, vol. 8559. Springer, 2014, pp. 150–166.

[8] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Flatten and conquer: a framework for efficient analysis of string constraints," in *PLDI*. ACM, 2017, pp. 602–617.

[9] A. W. Lin and P. Barceló, "String solving with word equations and transducers: towards a logic for analysing mutation XSS," in *POPL'16*. ACM, 2016, pp. 123–136.

[10] M. Berzish, Y. Zheng, and V. Ganesh, "Z3str3: A string solver with theory-aware branching," *CoRR*, vol. abs/1704.07935, 2017.

[11] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar, "String constraints with concatenation and transducers solved efficiently," *PACMPL*, vol. 2, no. POPL, pp. 4:1–4:32, 2018.

[12] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 513–528.

[13] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *NDSS*. The Internet Society, 2010.

[14] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *TACAS*, ser. LNCS, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 154–157.

[15] J. D. Scott, P. Flener, J. Pearson, and C. Schulte, "Design and implementation of bounded-length sequence variables," in *CPAIOR*, ser. LNCS, D. Salvagnin and M. Lombardi, Eds., vol. 10335. Springer, 2017, pp. 51–67.

[16] J. D. Scott, P. Flener, and J. Pearson, "Constraint solving on bounded string variables," in *CPAIOR*, ser. LNCS, vol. 9075. Springer, 2015, pp. 375–392.

[17] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An SMT solver for string constraints," in *CAV'15*, ser. LNCS, vol. 9206. Springer, 2015, pp. 462–469.

[18] H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang, "String analysis via automata manipulation with logic circuit representation," in *CAV'16*, ser. LNCS, vol. 9779. Springer, 2016, pp. 241–260.

[19] J. van Leeuwen, "Effective constructions in well-partially- ordered free monoids," *Discrete Mathematics*, vol. 21, no. 3, pp. 237 – 252, 1978.

[20] M. F. Atig, A. Bouajjani, and T. Touili, "On the reachability analysis of acyclic networks of pushdown systems," in *CONCUR'08*, ser. LNCS, vol. 5201. Springer, 2008, pp. 356–371.

[21] R. Parikh, "On context-free languages," *J. ACM*, vol. 13, no. 4, 1966.

[22] J. Esparza, P. Ganty, S. Kiefer, and M. LuttenbergSer, "Parikh's theorem: A simple and direct automaton construction," *Inf. Process. Lett.*, vol. 111, no. 12, pp. 614–619, 2011.

[23] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[24] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "CVC4," 2016. [Online]. Available: http://cvc4.cs.nyu.edu/papers/CAV2014-strings/

[25] M. Trinh, D. Chu, and J. Jaffar, "Progressive reasoning over recursively-defined strings," in *CAV'16*, ser. LNCS, vol. 9779. Springer, 2016, pp. 218–240.

[26] "Trau Solver." [Online]. Available: https://github.com/diepbp/Trau

[27] T. Tateishi, M. Pistoia, and O. Tripp, "Path- and index-sensitive string analysis based on monadic second-order logic," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, pp. 1–33, 2013.

[28] "IBM Security AppScan Tool and Source." [Online]. Available: https://www.ibm.com/us-en/marketplace/ibm-appscan-source.

[29] "Google Closure Library." [Online]. Available: https://github.com/google/closure-library/.

[30] "StringFuzz." [Online]. Available: https://github.com/dblotsky/stringfuzz

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Solving Constrained Horn Clauses Using Syntax and Data

Grigory Fedyukovich*, Sumanth Prabhu†, Kumar Madhukar†, Aarti Gupta*
*Princeton University, Princeton, USA {grigoryf, aartig}@cs.princeton.edu
†TCS Research, Pune, India {sumanth.prabhu, kumar.madhukar}@tcs.com

***Abstract*—A Constrained Horn Clause (CHC) is a logical implication involving unknown predicates. Systems of CHCs are widely used to verify programs with arbitrary loop structures: interpretations of unknown predicates, which make every CHC in the system true, represent the program's inductive invariants. In order to find such solutions, we propose an algorithm based on Syntax-Guided Synthesis. For each unknown predicate, it generates a formal grammar from all relevant parts of the CHC system (i.e., *using syntax*). Grammars are further enriched by predicates and constants guessed from models of various unrollings of the CHC system (i.e., *using data*). We propose an iterative approach to *guess and check* candidates for multiple unknown predicates. At each iteration, only a candidate for one unknown predicate is sampled from its grammar, but then it gets propagated to candidates of the remaining unknowns through implications in the CHC system. Finally, an SMT solver is used to decide if the system of candidates contributes towards a solution or not. We present an evaluation of the algorithm on a range of benchmarks originating from program verification tasks and show that it is competitive with state-of-the-art in CHC solving.**

I. INTRODUCTION

To formally prove that a program meets a given safety specification, one needs to discover inductive invariants for every loop that appears in the program. Each loop invariant safely approximates the set of program states reachable before and after the corresponding loop. However, it is hard to synthesize them in isolation: if there is a program path through two loops, then invariants for these loops are likely related. For existing approaches to invariant synthesis, the increase in complexity of loop structure enlarges the search space drastically and lowers the chances of finding a suitable system of invariants.

We view the task of program verification as an instance of a more general problem of Constrained Horn Solving (e.g., [1], [2], [3], [4], [5], [6]). It takes as input a set of logical implications, called Constrained Horn Clauses (CHCs), over a set of unknown predicates, and aims at either finding a suitable interpretation for all predicates, that makes all implications true or showing that no such interpretation exists. Therefore, a conventional formulation of the invariant synthesis task for a transition system is an instance of the CHC task itself, which involves only one unknown predicate.

In this work, we present an algorithm for solving CHC tasks of arbitrary structure. It is based on a recently proposed solution for the CHC task for transition systems [7], [8], [9]; and it relies on a paradigm of Syntax-Guided Synthesis (SyGuS) [10]. In our context, each unknown predicate of the CHC system gets its own formal grammar that encodes the search space for a solution. Then, candidate formulas are sampled from the corresponding grammars and substituted in the CHC system, and the resulting formulas are checked by a Satisfiability Modulo Theories (SMT) solver for validity.

Our central idea behind the grammar construction is to use both syntax and data. In particular, this process relies on 1) pre-computed predicates obtained by parsing the interpreted parts of the CHC system, and 2) pre-computed predicates and constants synthesized from various traces (i.e., models of unrollings) of the CHC system. With these ingredients at hand, a single grammar per unknown predicate is created. By construction, it describes all the pre-computed predicates and possibly more. The use of syntax and data to obtain grammars are complementary to one another. Using syntax makes a number of useful candidates readily available that may be computationally expensive to derive from data. Whereas using data provides meaningful semantic candidates that the CHC system may be syntactically oblivious to.

However, the need to synthesize interpretations for multiple unknowns from multiple grammars produces a bottleneck: all candidates should be consistent with each other. That is, each pair of candidates for two unknowns that might appear in one CHC should make the CHC true. It is hard to enforce this requirement in practice: usually, either one or both candidates would be withdrawn and re-synthesized – this would make our algorithm inefficient. Instead, our algorithm exploits a more accurate approach to sampling: it generates a candidate for one unknown predicate at a time, and then propagates it to candidates of the remaining unknowns through all possible implications in the CHC system.

In comparison to existing approaches to CHC solving, our approach has several unique features. First, to the best of our knowledge, it exploits data more extensively than any other tool: it allows generating candidates on the fly, for which it gets models from various formulas obtained from CHCs. Furthermore, our algorithm does not necessarily consider candidates of a fixed predetermined shape: due to the use of grammars to learn candidates, the shape of pre-computed predicates (using syntax and data) is modified during the run of the algorithm. Compared to the algorithm of generating data candidates for transition systems [9], our algorithm explores unrollings modularly (i.e., for each loop in isolation), and thus it avoids SMT solving for potentially large formulas.

Finally, our approach does not involve a potentially expen-

ISBN: 978-0-9835678-8-2.

sive fixed-point computation. Although our propagation routine is algorithmically similar to that in Generalized Property Directed Reachability [1], [4], we do not apply it recursively. Thus, our algorithm can never diverge while unwinding loops. The tradeoff is that our approach is not guaranteed to find an invariant, but it often does due to the rich grammars we generate, as shown in our experimental evaluation.

Our algorithm has been implemented on top of FREQHORN, a SyGuS-based CHC solver [7]. We have evaluated its effectiveness on a range of benchmarks originated from the verification tasks (i.e., programs with two or more loops and their safety specifications). Compared to state-of-the-art, our prototype exhibits a competitive performance and delivers results for most of the examples where the competing tools diverge. Our tool is particularly effective while discovering complex invariants over non-linear arithmetic.

The rest of the paper is structured as follows. Sect. II gives definitions, notation, and useful lemmas. Then, Sect. III presents our algorithm for a SyGuS-based CHC solver, driven by syntax, data and the candidate propagation. Finally, Sect. IV summarizes the evaluation, Sect. V outlines the related work, and Sect. VI concludes the paper.

II. PRELIMINARIES

For a given formula φ in a first-order theory $\mathcal{T}$, the Satisfiability Modulo Theories (SMT) task is to decide whether there is an assignment m of values to variables in φ that makes φ true. If every satisfying assignment to φ is also a satisfying assignment to some formula ψ, we write $\varphi \implies \psi$. By $\top$ and $\bot$ we denote constants true and false, respectively. By $Expr$ we denote a space of all possible quantifier-free formulas in $\mathcal{T}$ and by $Vars$ a range of possible variables in $\mathcal{T}$.

A. Constrained Horn Clauses

Definition 1. *A* linear constrained Horn clause *(CHC) over a set of uninterpreted relation symbols $\mathcal{R}$ is a formula in first-order logic that has the form of one of three implications (called respectively a* fact*, an* inductive clause*, and a* query*):*

$$\varphi(\vec{x_1}) \implies \boldsymbol{inv}_1(\vec{x_1})$$
$$\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}, \vec{x_2}) \implies \boldsymbol{inv}_2(\vec{x_2})$$
$$\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}) \implies \bot$$

where $\boldsymbol{inv}_1, \boldsymbol{inv}_2 \in \mathcal{R}$ are uninterpreted symbols, $\vec{x_1}, \vec{x_2}$ are vectors of variables, and φ, called a body*, is a fully interpreted formula (i.e., φ does not have applications of $\boldsymbol{inv}_1$ or $\boldsymbol{inv}_2$).*

For a CHC C, by $src(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the premise of C (if C is a fact, we write $src(C) \overset{\text{def}}{=} \top$). Similarly, by $dst(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the conclusion of C (if C is a query, we write $dst(C) \overset{\text{def}}{=} \bot$). We define functions rel and $args$, such that for each $\boldsymbol{inv}(\vec{x})$, $rel(\boldsymbol{inv}(\vec{x})) \overset{\text{def}}{=} \boldsymbol{inv}$ and $args(\boldsymbol{inv}(\vec{x})) \overset{\text{def}}{=} \vec{x}$. For a CHC C, by $body(C)$ we denote the body (i.e., φ) of C.

Example 1. *Fig. 1 shows a small C-like program*[1] *with three loops and its CHC-encoding. Each loop corresponds to one of the uninterpreted relation symbols $\mathcal{R} = \{\boldsymbol{inv}_1, \boldsymbol{inv}_2, \boldsymbol{inv}_3\}$. CHC* ***A*** *encodes the initial assignments to variables (including a nondeterministic choice for m and n) and assumptions over values of m and n. CHCs* ***B****,* ***D****, and* ***F*** *encode bodies of the first, the second, and the third loops, respectively. In order to represent a nondeterministic conditional in the first loop, CHC* ***B*** *contains the disjunction of encodings of both branches. CHCs* ***C*** *and* ***E*** *encode the fragments of the program between loops. Importantly, they include negations of the guards of preceding loops. Finally, CHC* ***G*** *encodes the negation of the assertion and the negation of the guard of the last loop.*

Linear CHCs can encode programs with nested loops, but cannot encode programs with non-inlined function calls[2]. For simplicity of presentation, the paper considers systems of CHCs that have only one query.

Definition 2. *Given a set of uninterpreted relation symbols $\mathcal{R}$ and a set S of CHCs over $\mathcal{R}$ we say that S is satisfiable if there exists an interpretation for each $\boldsymbol{inv} \in \mathcal{R}$ that makes all implications in S valid.*

Strictly speaking, an *interpretation* assigns to each symbol $\boldsymbol{inv} \in \mathcal{R}$ with arity n a relation over n-tuples. This relation can be represented by a formula φ over (at most) n free variables, denoted $fv(\varphi) \subseteq Vars$. In a specific application of $\boldsymbol{inv}$ to arguments $\vec{x}$, the free variables of φ are substituted by $\vec{x}$.

Example 2. *The system of CHCs in Fig. 1 is satisfiable (which means the program is safe), and a possible solution maps uninterpreted symbols to their interpretations as follows: $\boldsymbol{inv}_1 \mapsto x + y + n = m$, $\boldsymbol{inv}_2 \mapsto (x + y + n = m \wedge n = 0)$, and $\boldsymbol{inv}_3 \mapsto (x + y + n = m \wedge n = 0 \wedge x = 0)$.* □

B. Unrolling of CHCs

The following is built on ideas from Bounded Model Checking (BMC) [11] which aims at exploring finite length traces of programs.

Definition 3. *Given a system S of CHCs over $\mathcal{R}$, an unrolling of S of length k is a conjunction $\pi_{\langle C_0, \ldots, C_k \rangle} \overset{\text{def}}{=} \bigwedge_{0 \le i \le k} body(C_i)(\vec{x_i}, \vec{x_{i+1}})$, such that 1) each $C_i \in S$, 2) for each pair C_i and C_{i+1}, $rel(dst(C_i)) = rel(src(C_{i+1}))$, and variables of each $\vec{x_i}$ are shared only between $body(C_{i-1})(\vec{x}_{i-1}, \vec{x_i})$ and $body(C_i)(\vec{x_i}, \vec{x_{i+1}})$.*

Note that Def. 3 gives a more general notion of unrolling than it is customary for BMC. In particular, it allows the first step C_0 to be taken from an arbitrary place of the CHC system, i.e., C_0 is not necessarily a fact. We can consider unrollings, search for their models, and generate so called *behavioral*

---

[1] Because the presentation of our approach in terms of CHCs could be difficult to comprehend (e.g., notation is heavyweight in parts), here and throughout the paper we bring the analogy with program verification.

[2] We elaborate on the case with *nonlinear* CHCs in Sect. III-F.

```
int x = 0, y = 0;
int m = n = nondet();
assume (m >= 0);
while (n != 0) {
  n--;
  if (nondet()) x++;
  else y++;
}
while (x != 0) { m--; x--; }
while (y != 0) { m--; y--; }
assert (m == 0);
```

(*A*) $x' = 0 \wedge y' = 0 \wedge m' = n' \wedge m' \geq 0 \Longrightarrow \boldsymbol{inv}_1(x',y',m',n')$

(*B*) $\boldsymbol{inv}_1(x,y,m,n) \wedge \neg(n = 0) \wedge n' = n - 1 \wedge m' = m \wedge \Big((x' = x+1 \wedge y' = y) \vee (x' = x \wedge y' = y+1)\Big) \Longrightarrow \boldsymbol{inv}_1(x',y,'m',n')$

(*C*) $\boldsymbol{inv}_1(x,y,m,n) \wedge (n = 0) \wedge n' = n \wedge m' = m \wedge x' = x \wedge y' = y \Longrightarrow \boldsymbol{inv}_2(x',y,'m',n')$

(*D*) $\boldsymbol{inv}_2(x,y,m,n) \wedge \neg(x = 0) \wedge n' = n \wedge m' = m - 1 \wedge x' = x - 1 \wedge y' = y \Longrightarrow \boldsymbol{inv}_2(x',y,'m',n')$

(*E*) $\boldsymbol{inv}_2(x,y,m,n) \wedge x = 0 \wedge n' = n \wedge m' = m \wedge x' = x \wedge y' = y \Longrightarrow \boldsymbol{inv}_3(x',y,'m',n')$

(*F*) $\boldsymbol{inv}_3(x,y,m,n) \wedge \neg(y = 0) \wedge n' = n \wedge m' = m - 1 \wedge x' = x \wedge y' = y - 1 \Longrightarrow \boldsymbol{inv}_3(x',y,'m',n')$

(*G*) $\boldsymbol{inv}_3(x,y,m,n) \wedge y = 0 \wedge \neg(m = 0) \Longrightarrow \bot$

Fig. 1: Example program: (**left**) source code, and (**right**) its CHC encoding.

candidates for interpretations of unknown symbols that appear in the unrollings. We elaborate on this in Sect. III-C.

The following lemma provides yet another use of unrollings (for which C_0 is required to be a fact, and C_k – the query). We can enumerate various such unrollings and check satisfiability of the resulting formulas. Once a satisfiable formula is found, it does not make any sense to search for interpretations of any symbols in $\mathcal{R}$.

Lemma 1. *Given a system of CHCs S, let $\pi_{\langle C_0,\ldots,C_k\rangle}$ be one of its unrollings, such that C_0 is a fact, and C_k is the query. Then if $\pi_{\langle C_0,\ldots,C_k\rangle}$ is satisfiable then S is unsatisfiable.*

C. Polynomial behavioral candidates

We recall a few basic definitions from linear algebra that are needed for the generation of behavioral candidates. Given a vector space $\mathbf{V}$ over a field $\mathbf{F}$, its *basis* $\mathbf{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ is a minimal subset of $\mathbf{V}$ satisfying:

1) $\forall a_1, \ldots, a_n \in \mathbf{F}$, if $\sum\limits_{1 \leq i \leq n} a_i \cdot \mathbf{v}_i = 0$, then $\bigwedge\limits_{1 \leq i \leq n} a_i = 0$.
2) $\forall \mathbf{v} \in \mathbf{V}, \exists a_1, \ldots, a_n \in \mathbf{F}$ such that $\mathbf{v} = \sum\limits_{1 \leq i \leq n} a_i \cdot \mathbf{v}_i$.

Consider the following fixed-degree polynomial equation:

$$c_1 \cdot \alpha_1 + c_2 \cdot \alpha_2 + \cdots + c_n \cdot \alpha_n = 0 \tag{1}$$

where $\alpha_i = x_1^{k_1} \cdots x_l^{k_l}$ are *monomials*, $c_i \in \mathbb{Q}$ are *coefficients*, and $x_1, \ldots, x_n$ are the variables from *Vars*. The *degree* of a monomial is the sum $\sum\limits_{1 \leq i \leq n} k_i$, and the degree of a polynomial equation is the highest degree among its monomials.

Given the values of variables from *Vars*, let a *data matrix* contain values of monomials for *Vars* up to degree d. We rely on [12] to obtain equations of form (1) over *Vars* using a data matrix. When these values are substituted for monomials, we get a system of linear equations over $c_1, \ldots, c_n$. Solutions to these equations form a vector space, and the basis of this vector space, computed by the well-known Gauss-Jordan elimination algorithm, gives coefficients of polynomial equations.

III. CHC SOLVING AS ENUMERATIVE SEARCH

In this section, we first give a general idea of our setup, then proceed to describe details that make the search procedure effective in practice and finally summarize everything in one algorithm.

A. Basic idea

A solution for a system of CHCs S with uninterpreted symbols $\mathcal{R}$ is a mapping ℓ from each symbol to a formula (written as $\ell : \mathcal{R} \to \mathit{Expr}$) that makes each CHC in S true. For a synthesis of ℓ, suppose that every $\boldsymbol{inv} \in \mathcal{R}$ has its grammar $G(\boldsymbol{inv})$ that describes a set of possible candidate formulas for $\boldsymbol{inv}$. In a naive scenario, in each iteration of a synthesis loop, a candidate formula for each $\boldsymbol{inv}$ gets sampled from $G(\boldsymbol{inv})$. All candidates are substituted in S, and if at least one of the implications is invalid then the entire system of candidates is *failing* and the synthesis loop iterates.

Clearly, this naive approach has a large search space. For example, if for the system of CHCs in Fig. 1, the candidate for all three uninterpreted symbols $\boldsymbol{inv}_1$, $\boldsymbol{inv}_2$, and $\boldsymbol{inv}_3$ is $x + y + n = m$, then all of them will be rejected because the candidate for $\boldsymbol{inv}_3$ is too coarse to prove the query (i.e., it needs to be conjoined with $x = 0 \wedge n = 0$). However, following [7] and [8], we can optimize the search by synthesizing conjunction-free lemmas for each $\boldsymbol{inv}_i$ separately and then by conjoining them together.

Definition 4. *For a system of CHCs S over $\mathcal{R}$ and a mapping $\ell : \mathcal{R} \to \mathit{Expr}$, we say that ℓ is a set of* lemmas *for S if it makes every CHC in S (except the query) valid.*

Example 3. *For the system of CHCs in Fig. 1, a mapping from all $\boldsymbol{inv}_1$, $\boldsymbol{inv}_2$, and $\boldsymbol{inv}_3$ to $x + y + n = m$ is one set of lemmas. A mapping $\boldsymbol{inv}_1 \mapsto \top$, $\boldsymbol{inv}_2 \mapsto n = 0$, and $\boldsymbol{inv}_3 \mapsto n = 0$ is another set of lemmas.* □

Lemma 2. *Given a system of CHCs S over $\mathcal{R}$ and two sets of lemmas ℓ_1 and ℓ_2, let a mapping $\ell_3 : \mathcal{R} \to \mathit{Expr}$ be such that for each $\boldsymbol{inv} \in \mathcal{R} \,.\, \ell_3(\boldsymbol{inv}) \overset{\text{def}}{=} \ell_1(\boldsymbol{inv}) \wedge \ell_2(\boldsymbol{inv})$. Then ℓ_3 is a set of lemmas for S.*

Our algorithm generates grammars based on a set of formulas, called *seeds* [8]. By construction, grammars should be able to describe all seeds and, as a side effect, also formulas which are syntactically close to seeds (called *mutants*). In the next two subsections, we outline the process of determining seeds automatically.

B. Collecting seeds from syntax

Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$ be an uninterpreted symbol for which we wish to generate a formal grammar. Perhaps, the most obvious sources of seeds are the bodies of CHCs in S that have applications of $\boldsymbol{inv}$. First, the body of a CHC C that has applications of $\boldsymbol{inv}$ is parsed, and clauses that contain only variables in $args(src(C))$ or only variables in $args(dst(C))$ are extracted. Then, the obtained formulas are rewritten in terms of variables $\vec{x} \subseteq Vars$ (practically, it is convenient to specify $\vec{x} \overset{\text{def}}{=} args(src(C'))$ of some CHC C' with $\boldsymbol{inv} = rel(src(C'))$).

Formally, for a formula φ in Conjunctive Normal Form, let $Cnjs(\varphi)$ be a set of its clauses. For sets of variables $\vec{x}$ and $\vec{y}$, let a set $F_{\vec{x},\vec{y}}(\varphi)$ be defined as $F_{\vec{x},\vec{y}}(\varphi) \overset{\text{def}}{=} \{\psi \mid \exists \phi \in Cnjs(\varphi) \,.\, \psi = \phi[\vec{x}/\vec{y}] \land fv(\phi) \subseteq \vec{x}\}$, where $\phi[\vec{x}/\vec{y}]$ denotes the result of substitutions of variables $\vec{x}$ in ϕ by variables $\vec{y}$. Thus, a set of seeds obtained from bodies of CHCs can be defined as follows.

Definition 5. *Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$. Then*

$$SyntSeeds(\boldsymbol{inv})(\vec{x}) \overset{\text{def}}{=} \bigcup_{C \in S \text{ s.t. } rel(src(C)) = \boldsymbol{inv}} F_{args(src(C)),\vec{x}}(body(C)) \cup \bigcup_{C \in S \text{ s.t. } rel(dst(C)) = \boldsymbol{inv}} F_{args(dst(C)),\vec{x}}(body(C))$$

Example 4. *For the system of CHCs in Fig. 1, all four conjuncts of $body(\boldsymbol{A})$ give seeds $\{x = 0, y = 0, m = n, m \geq 0\}$ for $\boldsymbol{inv}_1$ and $\vec{x} = \langle x, y, m, n\rangle$. Furthermore, seeds $\neg(n = 0)$ and $n = 0$ are obtained from $body(\boldsymbol{B})$ and $body(\boldsymbol{C})$ respectively.* ☐

C. Collecting seeds from data

We bootstrap the grammar generation by seeds that are learned from the concrete values of variables produced while checking satisfiability of various unrollings of CHCs. If a CHC system S encodes some program, then an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ would correspond to a program trace whose sequentially executed statements are encoded by bodies of each C_i. If such an unrolling is unsatisfiable, then the corresponding program trace is infeasible. Otherwise, a model of the unrolling gives the concrete values of program variables at each execution step. We follow the ideas of the generation of behavioral seeds from models of program unrollings recently presented in [9].

The CHC task makes our setting different from [9], which considers CHCs with one uninterpreted relation symbol only. First, the presence of multiple symbols (and consequently, multiple loops) drastically complicates the creation of unrollings: the resulting formulas become too large and might become difficult for SMT solving. Second, it might be difficult to find a satisfiable unrolling since an unwinding number suitable for one loop might not be suitable for another loop. For example in Fig. 1, if the first and the second loops are unrolled n times, then to get a satisfiable unrolling, the third loop should be unrolled only zero times.

To overcome these two challenges, we propose to explore unrollings modularly: for each cycle in isolation. Recall that Def. 3 allows an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ to start from the body of some CHC C_0, where C_0 is not a fact. Thus, when determining behavioral seeds for some $\boldsymbol{inv}$ (e.g., when there is no fact in S with an application of $\boldsymbol{inv}$), we are free to consider any unrolling that starts from an arbitrary C_0, as long as $rel(dst(C_0)) = \boldsymbol{inv}$. In addition, we must ensure that $\boldsymbol{inv}$ is visited often enough, and the cycle has been terminated after C_k; otherwise, the collected data would not be sufficient for generating meaningful seeds. Def. 6 reflects these conditions formally.

Definition 6. *Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$. If an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ is such that 1) $rel(src(C_0)) \neq \boldsymbol{inv}$, 2) $rel(dst(C_0)) = \boldsymbol{inv}$, 3) $rel(src(C_k)) = \boldsymbol{inv}$, and 4) $rel(dst(C_k)) \neq \boldsymbol{inv}$, and $\big|\{C_i \in \langle C_0,\ldots,C_k\rangle \text{ s.t. } rel(dst(C_i)) = \boldsymbol{inv}\}\big| = n$, we call it* modular *for $\boldsymbol{inv}$ and denote it $\pi^n_{\boldsymbol{inv}}$.*

For practical reasons, we are interested in minimal unrollings $\pi^n_{\boldsymbol{inv}}$ satisfying Def. 6 for some n and $\boldsymbol{inv} \in \mathcal{R}$. Then we obtain a model $m_{\boldsymbol{inv}}$ of $\pi^n_{\boldsymbol{inv}}$ and compute the data matrix using the values in $m^n_{\boldsymbol{inv}}$ for every $args(dst(C_i)) \in \langle C_0,\ldots,C_k\rangle$, such that $rel(dst(C_i)) = \boldsymbol{inv}$. This data matrix is then used to discover behavioral seeds for $\boldsymbol{inv}$, denoted $BehavSeeds(\boldsymbol{inv})$, that have the fixed-degree polynomial form (1) (recall Sect. II-C).

Example 5. *For CHCs in Fig. 1, $\pi^3_{\boldsymbol{inv}_1} \overset{\text{def}}{=} body(\boldsymbol{A})(\vec{x}_0) \land body(\boldsymbol{B})(\vec{x}_0, \vec{x}_1) \land body(\boldsymbol{B})(\vec{x}_1, \vec{x}_2) \land body(\boldsymbol{C})(\vec{x}_2, \vec{x}_3)$. We are interested in values of variables in $\vec{x}_0$, $\vec{x}_1$ and $\vec{x}_2$ (which correspond to program variables $\langle x, y, m, n\rangle$ at the beginning of each loop iteration) that make $\pi^3_{\boldsymbol{inv}_1}$ true. For instance:*

x	*y*	*m*	*n*
0	*0*	*2*	*2*
0	*1*	*2*	*1*
1	*1*	*2*	*0*

Using this data matrix, we can generate a set $BehavSeeds(\boldsymbol{inv})(\langle x, y, m, n\rangle) = \{x + y - m + n = 0\}$. It is easy to see that this equality holds for every row of the data matrix. ☐

D. Candidate propagation

In practice, seeds obtained using methods from Sect. III-B and Sect. III-C are often insufficient for generating rich enough formal grammars. Consequently, candidate formulas that are sampled from these grammars, are often insufficient for the discovery of useful lemmas. Recall a solution of the system of CHCs in Fig. 1, as shown in Ex. 2. It requires a set of lemmas that have conjunct $n = 0$ in interpretations of $\boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$. However, the set of formulas shown in Ex. 4, can offer $n = 0$ only for $\boldsymbol{inv}_1$. Our main idea, described formally in the rest of this subsection, is to exploit that every CHC C with $rel(dst(C)) = \boldsymbol{inv}_2$ or $rel(dst(C)) = \boldsymbol{inv}_3$ has a clause $n' = n$ in its body (i.e., it merely reuses an old value of n), and thus the candidate $n = 0$ of $\boldsymbol{inv}_1$ can be pushed forward to become a candidate of $\boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$.

Before propagating candidates, we need to ensure that they are *self-consistent* in the following sense.

Definition 7. *Given a system of CHCs S over $\mathcal{R}$ and a subset $\mathcal{R}' \subseteq \mathcal{R}$. A mapping $Cand : \mathcal{R}' \to Expr$ is called* self-consistent *if it makes every CHC in $S' \stackrel{\text{def}}{=} \{C \in S \mid (src(C) = \top \vee rel(src(C)) \in \mathcal{R}') \wedge rel(dst(C)) \in \mathcal{R}'\}$ valid.*

Clearly, if the candidates are not self-consistent, they cannot be extended to a set of lemmas. Alg. 1 gives a simple routine to check the self-consistency of candidates with respect to CHCs S' that have applications of symbols from $\mathcal{R}'$ only. If the algorithm finds an invalid CHC C, then it weakens the candidate for $rel(dst(C))$ and repeats the self-consistency check. Intuitively, if C has the form (2), then (3) is invalid.

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies \boldsymbol{inv}_j(\vec{x_j}) \tag{2}$$

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies Cand(\boldsymbol{inv}_j)(\vec{x_j}) \tag{3}$$

Alg. 1 weakens $Cand(\boldsymbol{inv}_j)$ to $\top$, and thus (3) becomes trivially valid. Continuing such operation for other CHCs from S' guarantees discovering a self-consistent set of candidates. Note that Alg. 1 takes as additional input a set of formulas which are already proved to be lemmas (recall Def. 4).

Further reasoning of the candidate propagation, given self-consistent formulas $Cand$ for some $\mathcal{R}' \subseteq \mathcal{R}$, boils down to recursive post- and precondition inference: for any CHC in S that has the form (2), where $\boldsymbol{inv}_i \in \mathcal{R}'$ and $\boldsymbol{inv}_j \notin \mathcal{R}'$, we wish to identify a formula $Cand(\boldsymbol{inv}_j)$, such that (3) holds. Symmetrically, if $\boldsymbol{inv}_i \notin \mathcal{R}'$ and $\boldsymbol{inv}_j \in \mathcal{R}'$, we wish to identify a formula $Cand(\boldsymbol{inv}_i)$, such that again (3) holds.

The method of candidate propagation is based on quantifier elimination.

Definition 8. *Given a formula that has the form* (4)*.*

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies \boldsymbol{inv}_j(\vec{x_j}) \tag{4}$$

Forward propagation *of $Cand(\boldsymbol{inv}_i)$ gives a formula $Cand(\boldsymbol{inv}_j)$, such that:*

$$Cand(\boldsymbol{inv}_j)(\vec{x_j}) \stackrel{\text{def}}{=} \exists \vec{x_i} \,.\, Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \tag{5}$$

Intuitively, if $\varphi(\vec{x_i}, \vec{x_j})$ encodes a transition from a program state $\vec{x_i}$ to a program state $\vec{x_j}$, then $Cand(\boldsymbol{inv}_j)(\vec{x_j})$ encodes a set of all possible states that are reachable from $Cand(\boldsymbol{inv}_i)(\vec{x_i})$ by making the $\varphi(\vec{x_i}, \vec{x_j})$ step. Note that in case $Cand(\boldsymbol{inv}_i)(\vec{x_i}) = \top$, propagating $\top$ can still give meaningful candidates, if e.g., the dst-arguments do not depend on the src-arguments. On the other hand, if $Cand(\boldsymbol{inv}_i)(\vec{x_i}) = \bot$, propagating $\bot$ ends up with $\bot$ again.

Note that the result of forward propagation (5) can be substituted back to implication (4) and make it true. Interestingly, the operation of *backward propagation* (defined below) does not have such property; and to enforce it, we should apply an additional weakening of the propagated formula.

Definition 9. *Given a formula that has the form* (6)*.*

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies Cand(\boldsymbol{inv}_j)(\vec{x_j}) \tag{6}$$

Backward propagation *of $Cand(\boldsymbol{inv}_j)$ gives a formula $Cand(\boldsymbol{inv}_i)$, such that:*

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \stackrel{\text{def}}{=} \exists \vec{x_j} \,.\, Cand(\boldsymbol{inv}_j)(\vec{x_j}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \tag{7}$$

Algorithm 1: WEAKEN: establishing self-consistency.

```
Input: CHCs $S'$ over $\mathcal{R}'$, set of candidates
       $Cand : \mathcal{R}' \to Expr$; learned $Lemmas : \mathcal{R} \to 2^{Expr}$
Output: weakened $Cand$
$allGood \leftarrow \top$;
for all $C \in S'$ do
    if $\bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \wedge$
        $Cand(rel(src(C)))(args(src(C))) \wedge body(C) \not\Longrightarrow$
            $Cand(rel(dst(C)))(args(dst(C)))$ then
        $Cand(rel(dst(C))) \leftarrow \top$;
        $allGood \leftarrow \bot$;
        break;
if $allGood$ then return $Cand$;
else return WEAKEN($Cand, \mathcal{R}', S', Lemmas$);
```

Algorithm 2: EXTEND: recursive propagation.

```
Input: CHCs $S$ over $\mathcal{R}$; $\mathcal{R}' \subseteq \mathcal{R}$, set of candidates
       $Cand : \mathcal{R}' \to Expr$; learned $Lemmas : \mathcal{R} \to 2^{Expr}$
Output: $res \in \{\top, \bot\}$, extended $Cand$
1  $Cand \leftarrow$ WEAKEN($Cand, \mathcal{R}', S', Lemmas$);
2  if $\forall \boldsymbol{inv} \in \mathcal{R}' \,.\, Cand(\boldsymbol{inv}) = \top$ then return $\langle \bot, \_ \rangle$;
3  for all $C \in S$ s.t. $rel(src(C)) \in \mathcal{R}'$ and $rel(dst(C)) \notin \mathcal{R}'$
    do
4      $Cand(rel(dst(C))) \leftarrow$
         PROPAGATEFORWARD($C$, $Cand$);
5      $\langle positive, Cand \rangle \leftarrow$
             EXTEND($S, \mathcal{R}' \cup \{rel(dst(C))\}, Cand, Lemmas$);
6      if $\neg positive$ then return $\langle \bot, \_ \rangle$;
7  for all $C \in S$ s.t. $rel(dst(C)) \in \mathcal{R}'$ and $rel(src(C)) \notin \mathcal{R}'$
    do
8      $Cand(rel(src(C))) \leftarrow$
         PROPAGATEBACKWARD($C$, $Cand$);
9      $\langle positive, Cand \rangle \leftarrow$
             EXTEND($S, \mathcal{R}' \cup \{rel(src(C))\}, Cand, Lemmas$);
10     if $\neg positive$ then return $\langle \bot, \_ \rangle$;
11 return $\langle \top, Cand \rangle$;
```

Both forward and backward propagation can be applied recursively for any set of candidates $Cand$ and a subset $\mathcal{R}' \subseteq \mathcal{R}$. This is shown formally in Alg. 2. After establishing the self-consistency of candidates (line 1), Alg. 2 extends $Cand$ by adding *inferred* candidates using forward propagation (line 4) for all CHCs C that have $rel(src(C)) \in \mathcal{R}'$ and $rel(dst(C)) \in \mathcal{R} \setminus \mathcal{R}'$, and *inferred* candidates using backward propagation (line 8) for all CHCs C that have $rel(dst(C)) \in \mathcal{R}'$ and $rel(src(C)) \in \mathcal{R} \setminus \mathcal{R}'$. Each round of propagation enlarges the set of symbols annotated by candidates $\mathcal{R}'$ as well as $Cand$, and Alg. 2 is called recursively (lines 5 and 9). If $\mathcal{R}' = \mathcal{R}$ then it is enough to check self-consistency of $Cand$ (and weaken it if needed) before returning $Cand$ as a set of lemmas.

Theorem 1. *Assuming termination of the quantifier elimination procedure and termination of each implication check, Alg. 2 always terminates.*

Algorithm 3: SOLVECHCS: overall algorithm.

```
Input: CHCs $S$ over $\mathcal{R}$
Output: $res \in \{\textsc{sat}, \textsc{unknown}\}$, $Lemmas : \mathcal{R} \to 2^{Expr}$
1  for all $\boldsymbol{inv} \in \mathcal{R}$ do
2      $Seeds \leftarrow SyntSeeds(\boldsymbol{inv}) \cup BehavSeeds(\boldsymbol{inv})$;
3      $G(\boldsymbol{inv}) \leftarrow \textsc{getGrammar}(Seeds)$;
4      $Lemmas(\boldsymbol{inv}) \leftarrow \varnothing$;
5  while $\forall C \in S \,.\, (dst(C) = \bot) \implies \Big( \bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \land body(C) \not\Longrightarrow \bot \Big)$
   do
6      if $\forall \boldsymbol{inv} \in \mathcal{R} \,.\, \textsc{allBlocked}(G(\boldsymbol{inv}))$ then
7          return $\langle \textsc{unknown}, \varnothing \rangle$;
8      $\boldsymbol{inv} \leftarrow \textsc{pickRelationalSymbol}(\mathcal{R})$;
9      $Cand(\boldsymbol{inv}) \leftarrow \textsc{sample}(G(\boldsymbol{inv}))$;
10     $\langle positive, Cand \rangle \leftarrow$
         $\textsc{extend}(S, \{\boldsymbol{inv}\}, Cand, Lemmas)$;
11     for all $\boldsymbol{inv} \in \mathcal{R}$ do
12         if $positive$ then
13             $Lemmas(\boldsymbol{inv}) \leftarrow$
                 $Lemmas(\boldsymbol{inv}) \cup \{Cand(\boldsymbol{inv})\}$;
14         $G(\boldsymbol{inv}) \leftarrow \textsc{block}(G(\boldsymbol{inv}), Cand(\boldsymbol{inv}), positive)$;
15 return $\langle \textsc{sat}, Lemmas \rangle$;
```

For theories which do not admit a terminating quantifier-elimination procedure, Alg. 2 can be safely modified by replacing the results of calling the propagation methods on lines 4 and 8 by constant $\top$.

E. Core algorithm

Our main contribution is an effective search strategy for a solution of a given system of CHCs S over a set of uninterpreted symbols $\mathcal{R}$. The search is over a set of candidate formulas for each $\boldsymbol{inv} \in \mathcal{R}$ which is described by a formal grammar $G(\boldsymbol{inv})$. In this section, we instantiate the setup outlined in Sect. III-A by the components that make the entire procedure practical. The pseudocode of the algorithm is shown in Alg. 3.

Alg. 3 starts by creating the sampling grammars $G(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$. Grammars are constructed automatically: first (line 2), by collecting *Seeds* as described in Sect. III-B and Sect. III-C; and then (line 3) by creating production rules that would be able to produce all *Seeds* recursively. We do not impose any restrictions on the implementation of this routine, and in practice, one could additionally add a normalization pass over all *Seeds* before processing them. Note that various unrollings, considered for constructing the behavior candidates, can be enhanced with the bodies of the query (and of other clauses if necessary) to be checked for the existence of counterexamples (recall Lemma 1). If no counterexamples are found, the algorithm starts *guessing and checking* candidate formulas $Cand(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$.

Simultaneous sampling from multiple grammars might lead to many iterations of Alg. 3. To be turned to a set of lemmas, each set of candidate formulas should be self-consistent. But if the candidates are sampled without taking into account any relationship among loops, the weakening by Alg. 1 might be too aggressive and might withdraw many good candidates. Instead, we propose to fix precisely one grammar (say, $G(\boldsymbol{inv})$ for some $\boldsymbol{inv} \in \mathcal{R}$) per iteration, to sample a candidate formula $Cand(\boldsymbol{inv})$ from $G(\boldsymbol{inv})$, and to propagate $Cand(\boldsymbol{inv})$ recursively to candidate formulas $Cand(\boldsymbol{inv}')$ for all $\boldsymbol{inv}' \in \mathcal{R}$ through all implications in S (lines 8-10).

In particular, at each iteration, Alg. 3 picks $\boldsymbol{inv} \in \mathcal{R}$ (in our implementation, we use Weak Topological Ordering [13], but any other heuristic can be used instead). Then the algorithm samples a formula $Cand(\boldsymbol{inv})$ – it could either be one of *Seeds* or a syntactically mutated formula. The goal now is to find candidate formulas for all other $\boldsymbol{inv}' \in \mathcal{R} \setminus \{\boldsymbol{inv}\}$ and to check all implications in CHCs. The algorithm performs inference of preconditions and postconditions using the routine described in Sect. III-D (Alg. 2).

Recall that Alg. 2 not only populates *Cand* with candidate formulas for some symbols but also drops some unsuccessful candidate formulas due to weakening. Note that Alg. 1 implements a simple strategy, in which a candidate formula $Cand(\boldsymbol{inv}_j)$ can only be dropped to $\top$ – this helps when $Cand(\boldsymbol{inv}_j)$ is conjunction-free. However, in case $Cand(\boldsymbol{inv}_j)$ is conjunctive (which could be due to quantifier elimination), a more careful weakening (e.g., [14], [15] or [16]) can be used. In the worst-case scenario, weakening ends up with an empty candidate, which means that nothing was learned at this iteration, and a new candidate formula should be sampled.

In the case when a sequence of weakening-propagation calls has converged, the entire *Cand* is learned as a lemma (line 13). The process is repeated until the conjunction of lemmas is strong enough to be a solution for the entire system (apply Lemma 2). Finally, for the progress of the algorithm, both failed and positive attempts are noted, and the algorithm ensures that the candidates are not sampled again in the future (line 14). If all candidates of all grammars are blocked, the algorithm terminates with an unknown result (line 6). The facts that each formal grammar admits only a finite number of candidates and that each candidate is considered only once enable us to prove the following theorem.

Theorem 2. *Alg. 3 always makes a finite number of iterations, and if it converges with* SAT*, the set of all learned lemmas constitutes a solution of the CHC system.*

Similarly to [8], the algorithm can be optimized by introducing *bootstrapping* and *sampling* stages, candidate batching and exploiting counterexamples-to-induction, and thus it can be effectively integrated with the elements of Generalized Property Directed Reachability (GPDR) [1], [4].

F. Extension to nonlinear CHCs

Definition 10. *A* nonlinear *CHC is a formula in first-order logic that has the form of one of three implications:*

$$\varphi(\vec{x_1}) \implies \boldsymbol{inv}_1(\vec{x_1})$$

$$\bigwedge_{0\leq i\leq n} \boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_0}, \ldots, \vec{x_{n+1}}) \implies \boldsymbol{inv}_{n+1}(\vec{x_{n+1}})$$

$$\bigwedge_{0\leq i\leq n} \boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_0}, \ldots, \vec{x_n}) \implies \bot$$

Our synthesis algorithm can be adapted to solve systems of nonlinear CHCs with limited backward propagation. The rest of the components operate in the same way: each $\boldsymbol{inv} \in \mathcal{R}$ gets its grammar, and candidates are iteratively sampled from them.

In the future, we would like to discover ways of effective backward propagation for nonlinear CHCs. In particular, a variant of (6) for nonlinear CHCs might be as follows:

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \boldsymbol{inv}_j(\vec{x_j}) \wedge \varphi(\vec{x_i}, \vec{x_j}, \vec{x_k}) \implies Cand(\boldsymbol{inv}_k)(\vec{x_k})$$

Applying quantifier elimination, we get candidates for conjunctions $Cand(\boldsymbol{inv}_i) \wedge Cand(\boldsymbol{inv}_j)$, but not necessarily for individual conjuncts $Cand(\boldsymbol{inv}_i)$ and $Cand(\boldsymbol{inv}_j)$.

IV. Implementation and Evaluation

We have implemented the algorithm from Sect. III-E on top of our previous implementation FreqHorn[3]. The tool takes a system of CHCs, automatically performs its unrolling, searches for counterexamples (if any), generates behavioral candidates, propagates and weakens candidates. To eliminate quantifiers, FreqHorn uses the technique based on Model-Based Projections [17]. For solving SMT queries, it uses Z3 [18]. For matrix operations, FreqHorn uses Armadillo [19], a C++ library for linear algebra.

We evaluated FreqHorn on **101** satisfiable CHC-systems[4] taken from the literature on program verification (e.g. [20]) and crafted by ourselves. There are 81 systems of CHCs over the theories of linear (LIA) and 20 over nonlinear integer arithmetic (NIA). All systems have two or more uninterpreted relation symbols. Because our quantifier-elimination engine has limited support for NIA, we disabled candidate propagation for the cases when the body of corresponding CHCs contains nonlinear arithmetic. In such cases, we assigned $\top$ to the propagated candidates and performed the self-consistency checks. Thus, disabling candidate propagation did not lead to incorrect results.

Among the 101 benchmarks, FreqHorn was able to solve **81** within a timeout of 5 minutes: 65 over LIA, and 16 over NIA. The remaining 20 benchmarks require disjunctive invariants which are difficult to find for FreqHorn. In order to evaluate the significance of candidate propagation, behavioral candidates, and candidates guessed from syntax, we performed controlled experiments with the corresponding features disabled. Fig. 2 gives the scatter plots that compare configurations on all benchmarks. Each point in a plot represents a pair of the runtime (sec) of the full configuration of FreqHorn (x-axis) and the runtime (sec) of the restricted configuration of FreqHorn (y-axis). In each plot, the color saturation roughly reflects the benefits of the full configuration, i.e., the delta between the runtimes.

The configuration of FreqHorn with candidate propagation disabled (thus, candidates for all unknowns had to be sampled independently) was able to solve 56 benchmarks, and it was on average three times slower than the full configuration. After disabling behavioral candidates (but with candidate propagation), FreqHorn was able to solve 60 benchmarks. Time-wise, this experiment gave less consistent results: for 15 benchmarks the restricted configuration outperformed the full one. Finally, after disabling syntactic candidates (but with candidate propagation and behavioral candidates), FreqHorn was able to solve only 37 benchmarks. The experiment confirmed that all features of our algorithm are essential for its efficacy, and it leaves room for devising heuristics to apply in specific contexts.

We also compared our tool to Spacer v.3 [4], μZ v.4.4.2 [1], and Eldarica v.1.3 [2] CHC solvers (shown in Fig. 3)[5][6]. Among the 101 benchmarks, Spacer was successful on 45, μZ on 42, and Eldarica on 71. FreqHorn solved 41 benchmarks on which Spacer diverged, 44 on which μZ diverged, 22 on which Eldarica diverged. In total, it solved **16** benchmarks on which all the competitors diverged, and 10 of them are over NIA.

In our benchmark selection, there are 8 tricky tasks which were solved by none of the tools. Investigating bottlenecks in solving them motivates our future work.

V. Related work

Conceptually, our algorithm for solving CHCs can be viewed as an extension of the syntax-guided invariant synthesizer [7] for transition systems (i.e., CHCs with one uninterpreted relation symbol). Thus, [7] is built around one sampling grammar, and does not require any candidate propagation. For arbitrary CHCs, as shown in our experiments, a naively extended approach of [7] does not scale well. Furthermore, in many cases, for convergence, it would require some symbolic constraints to be propagated across CHCs before the grammar is constructed (otherwise, the grammars might not be sufficient, and sometimes might be even empty). Our new solution is insensitive to these challenges.

Other instantiations of [7] include [8] and [9], but they still do not span beyond the transition systems. Our approach incorporates essential details of [8] and [9], namely enriching the grammars by externally created seeds. In particular, as in [9], we use polynomial equations as candidates for a

---

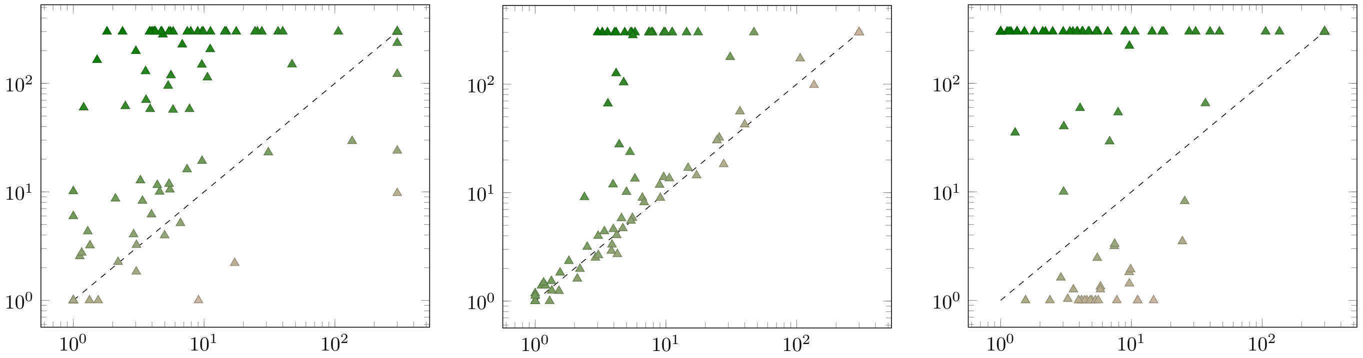
[3] The source code is available at https://github.com/grigoryfedyukovich/aeval/tree/rnd.

[4] Available at https://github.com/grigoryfedyukovich/aeval/tree/rnd/bench_horn_multiple, and also contributed to CHC-COMP: http://chc-comp.github.io/.

[5] We excluded the time needed to start Java Virtual Machine from the running time of Eldarica.

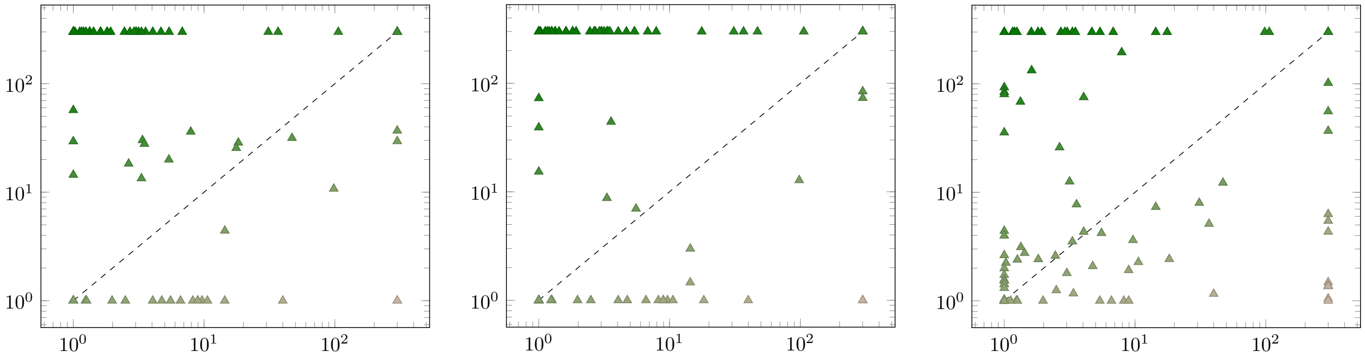
[6] Full statistics are available at https://goo.gl/ADZdez.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.



FreqHorn: **with** vs. **without** propagation. FreqHorn: **with** vs. **without** behavioral candidates. FreqHorn: **with** vs. **without** syntactic candidates.

Fig. 2: Internal statistics on FreqHorn (sec×sec): points above the diagonal represent runtimes for benchmarks on which full configuration outperformed the restricted configuration.



FreqHorn vs. Spacer. FreqHorn vs. μZ. FreqHorn vs. Eldarica.

Fig. 3: Comparison of FreqHorn (sec×sec) and external tools: points above the diagonal represent runtimes for benchmarks on which the best configuration of FreqHorn outperformed the competitor.

relation between variables, generated after analyzing models for unrollings of CHCs. But again, [9] does not deal with multiple uninterpreted relation symbols. Our approach required solutions to several new challenges. First, a satisfiable unrolling for every loop must be found to obtain behavioral data. Second, even if we get a good candidate for interpretation of one symbol, often a weakening or a strengthening of this candidate is needed to accommodate suitable candidates for other symbols. We have addressed these issues by introducing a concept of modular unrolling of a system of CHCs, and by considering the seeds obtained from data to bootstrap the grammar generation.

Apart from solving unrollings as in [9], there are prominently two ways to get behavioral data – from infeasible paths using interpolation [21], and from reachable states along feasible paths using test-based executions [22], [12], [23], [24]. These techniques are not only limited by the expressiveness of their grammar, which is fixed, they also take the naive approach to dealing with multiple loops, i.e., the candidates are learned independently for all loops. In contrast, we use behavioral seeds to bootstrap the grammar. Furthermore, we propagate candidates learned for one loop to obtain constraints on those for adjacent loops.

Propagation of candidates and search for inductive subsets is at the heart of the approaches based on Generalized Property Directed Reachability (GPDR) [1], [4]. In a nutshell, they are based on implicit unrollings of loops and a monotonic fixed-point computation, driven by spurious counterexamples. However, such methods often diverge due to failures to generalize an inductive invariant from counterexamples. In contrast, our approach does not perform a fixed-point computation, and propagates candidates only through a finite number of implications, specified directly in CHCs. Failures to propagate lead to withdrawing the candidate and generating a new guess from the grammar. In practice, this makes our solution effective on many benchmarks which are difficult for GPDR.

VI. CONCLUSIONS

We have presented an algorithm for solving systems of CHCs based on Syntax-Guided Synthesis. For each unknown predicate in CHCs, our algorithm generates a formal grammar from the syntax of the CHC system and models of various unrollings of the system. A solution for the system (i.e., an interpretation of each unknown predicate that makes all CHCs true) is then guessed from the corresponding grammars and checked by an SMT solver. It is crucial for the effectiveness of the approach to use modular unrollings of CHCs and to propagate candidates through all available implications in the CHC system. We have presented the evaluation of our prototype built on top of the FREQHORN tool and have confirmed that the algorithm is effective on a range of benchmarks originating from program verification tasks and competitive with state-of-the-art CHC solvers. As we go ahead, we plan to optimize the algorithm using heuristics, to develop effective strategies for backward candidate propagation in case of nonlinear CHCs, and to extend our tool with the support of CHCs over arrays, algebraic data types and bit-vectors.

Acknowledgements: This work was supported in part by NSF Grant 1525936. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF.

REFERENCES

[1] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, vol. 7317 of *LNCS*, pp. 157–171, Springer, 2012.

[2] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A verification toolkit for numerical transition systems - tool paper," in *FM*, vol. 7436 of *LNCS*, pp. 247–251, Springer, 2012.

[3] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, pp. 405–416, ACM, 2012.

[4] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV*, vol. 8559 of *LNCS*, pp. 17–34, 2014. https://bitbucket.org/spacer/code/branch/spacer3.

[5] H. Unno and T. Terauchi, "Inferring simple solutions to recursion-free Horn Clauses via sampling," in *TACAS*, vol. 9035 of *LNCS*, pp. 149–163, Springer, 2015.

[6] B. Kafle, J. P. Gallagher, and J. F. Morales, "Rahft: A tool for verifying Horn clauses using abstract interpretation and finite tree automata," in *CAV, Part I*, vol. 9779 of *LNCS*, pp. 261–268, Springer, 2016.

[7] G. Fedyukovich, S. Kaufman, and R. Bodík, "Sampling Invariants from Frequency Distributions," in *FMCAD*, pp. 100–107, IEEE, 2017.

[8] G. Fedyukovich and R. Bodík, "Accelerating Syntax-Guided Invariant Synthesis," in *TACAS, Part I*, vol. 10805 of *LNCS*, pp. 251–269, Springer, 2018.

[9] S. Prabhu, K. Madhukar, and R. Venkatesh, "Efficiently learning safety proofs from appearance as well as behaviours," in *SAS*, LNCS, Springer, 2018.

[10] 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*, pp. 1–17, IEEE, 2013.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, vol. 1579 of *LNCS*, pp. 193–207, Springer, 1999.

[12] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *ESOP*, vol. 7792 of *LNCS*, pp. 574–592, Springer, 2013.

[13] F. A. Bourdoncle, "Efficient Chaotic Iteration Strategies with Widenings," in *FMPA*, vol. 735 of *LNCS*, pp. 128–141, Springer, 1993.

[14] C. Flanagan and K. R. M. Leino, "Houdini: an Annotation Assistant for ESC/Java," in *FME*, vol. 2021 of *LNCS*, pp. 500–517, Springer, 2001.

[15] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Incremental verification of compiler optimizations," in *NFM*, vol. 8430 of *LNCS*, pp. 300–306, Springer, 2014.

[16] E. G. Karpenkov and D. Monniaux, "Formula slicing: Inductive invariants from preconditions," in *HVC*, vol. 10028 of *LNCS*, pp. 169–185, Springer, 2016.

[17] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Automated discovery of simulation between programs," in *LPAR*, vol. 9450 of *LNCS*, pp. 606–621, Springer, 2015.

[18] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, vol. 4963 of *LNCS*, pp. 337–340, Springer, 2008.

[19] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, 2016.

[20] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, pp. 443–456, ACM, 2013.

[21] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," in *J. of Symbolic Logic*, pp. 269–285, 1957.

[22] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *ICSE*, pp. 449–458, ACM, 2000.

[23] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *POPL*, pp. 499–512, ACM, 2016.

[24] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *CAV*, vol. 8559 of *LNCS*, pp. 88–105, Springer, 2014.

ISBN: 978-0-9835678-8-2.

Analysis of Relay Interlocking Systems via SMT-based Model Checking of Switched Multi-Domain Kirchhoff Networks

Roberto Cavada*, Alessandro Cimatti*, Sergio Mover‡, Mirko Sessa*†, Giuseppe Cadavero§, Giuseppe Scaglione§

*Fondazione Bruno Kessler
Trento, IT
{cavada, cimatti, sessa}@fbk.eu

‡University of Colorado Boulder
Boulder, USA
sergio.mover@colorado.edu

†University of Trento
Trento, IT

§Rete Ferroviaria Italiana
Rome, IT
{g.cadavero, g.scaglione}@rfi.it

Abstract—**Relay Interlocking Systems (RIS) are analog electromechanical networks traditionally applied in the safety-critical domain of railway signaling. RIS consist of networks of interconnected components such as power supplies, contacts, resistances, and electrically-controlled contacts (i.e. the relays). Due to cost and flexibility needs, RIS are progressively being replaced by equivalent computer-based systems. Unfortunately, RIS are often legacy systems, hard to understand at an abstract level, hence the valuable information they encoded in them is not available.**

In this paper, we propose a methodology and a tool chain to analyze and understand legacy RIS. A RIS is reduced to a Switched Multi-Domain Kirchhoff Network (SMDKN), which is in turn compiled into hybrid automata. SMT-based model checking supports various forms of formal analyses for SMDKN. The approach is based on the modeling of the RIS analog signals (i.e. currents and voltages) over continuous time, and their mapping in terms of railways control actions. Starting from the diagram representation, we overcome a key limitation of previous approaches based on purely Boolean models, i.e. the presence of spurious behaviors. The evaluation of the tool chain on a set of industrial-size railway RIS demonstrates practical scalability.

I. INTRODUCTION

Railway signaling systems guarantee the safe operation of train traffic. Trains run between points of the rail network, moving from section to section along exclusively allocated routes and crossing roads. Protection against catastrophic events, such as train-to-train and train-to-car collisions, is devoted to various devices such as semaphores, barriers at the level crossing, and train detection systems. These devices must be suitably controlled and coordinated by a logic that ensures the safety of operation even in case of multiple device faults.

Traditionally, the logic has been implemented by means of the Relay technology, in the form of networks of interconnected analog electro-mechanical components, such as power supplies, contacts, circuit breakers, and many forms of electrically-controlled contacts, also known as *relays*.

RIS are progressively being replaced by computer-based logics (CBL), that ensure greater flexibility and lower cost. The key question is how to ensure that the CBL is compliant with the (trusted) behavior of the relay-based interlocking being replaced. In some sense, the specification for the CBL is hidden in the relay circuit. Unfortunately, RIS are often old, legacy systems, hard to understand for software engineers at the level of abstraction required to specify the CBL. Thus, the valuable information they encode is not readily available.

Although relays may be thought of as Boolean components, that is just open or closed, this turns out to be a gross simplification. In order to operate (e.g. switching from open to closed), relays may require time, and go through transients required to fully excite the circuitry. Hence, a simple Boolean propagation is in fact a coarse abstraction of a sequence of intermediate states before stability. Furthermore, relays are subject to faults that may either delay or prevent the correct operation. Thus, relay networks are often designed in a redundant fashion in order to mitigate the effect of faults and to ensure safety (at the cost of liveness) in all conditions.

In this paper, we propose a methodology and a tool chain to analyze and understand legacy RIS, adopted in an ongoing research project of Rete Ferroviaria Italiana (RFI). At the surface, a graphical tool supports the component-based modeling of the RIS. The designer selects components from a palette of over 100 elements, and connects them according to the input description – typically, a printout of the electrical schematic. This step does not require any deep understanding of the nature of the circuit, and ensures that the semantic gap w.r.t. the legacy description is as limited as possible. The corresponding internal representation is reduced to a Switched Multi-Domain Kirchhoff Network (SMDKN), which has a semantic based on Differential Algebraic Equations (DAE). In turn, the SMDKN is compiled into a network of hybrid automata, based on the techniques proposed in [1]. Then, various forms of formal analysis are supported by means of SMT-based model checking. At its core, the approach is based on the modeling of the RIS analog signals (i.e. currents and voltages) over continuous time. The ability to analyze the circuit at the physical level supports a comprehensive understanding at the symbolic level in terms of railways control actions. This is done by defining suitable symbolic predicates in terms of the analog state: for example,

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

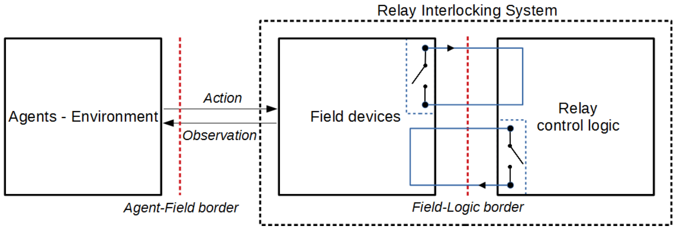


Fig. 1: Conceptual architecture of a RIS.

a green light to the train may correspond to a suitable current and voltage drop in the corresponding semaphore lamp.

The methodology is fully supported by an automated SMT-based verification tool chain. We evaluated the approach on a set of industrial-size railway RIS, with schematic having more than a thousand components and four-meter long plotter printouts. The results demonstrate practical scalability: we are able to prove (or disprove) conjectured properties, simulate scenarios, and construct fault-trees (FT) corresponding to undesirable events.

This approach was devised as a consequence of a previous unsatisfying modeling attempt we carried on basing our analysis on the traditional formal modeling at the Boolean level. Since relays are not instantaneous Boolean switches, substantial ingenuity from the modeler was required to bridge the gap with respect to the electrical semantics. This made the modeling task unmanageable in terms of conceptual hardness, and led to imprecise results (due to spurious behaviors) that we will report in the following sections. From a pragmatic perspective, the proposed approach provides invaluable support for the understanding of the legacy circuit (and ultimately the reverse-engineering of requirements for the CBL design).

The paper is structured as follows. In Section II we describe Relay Interlocking Systems. In Section III we overview SMDKN. In Section IV we describe the modeling approach. In Section V we present the analysis methods. In Sections VI and VII we present the tool chain and the experimental evaluation on a scalable industrial-size case study. In Sections VIII and IX we describe related work, draw some conclusions, and outline ongoing and future work.

II. RELAY INTERLOCKING SYSTEMS

A Relay Interlocking System (RIS) is an electromechanical system that conveys messages between the railway *agents* (e.g., trains, dispatchers, technicians). Fig. 1 shows the conceptual architecture of a RIS: the agents interacts with the *field devices* (e.g., semaphores, level crossing barriers, railroad switches) that are in turn controlled through the *relay control logic* (an interconnection of relays).

The agents interact with the field devices observing their state (e.g., if a semaphore light is on or off, the position of a barrier or of a railroad switch) and perform some actions (e.g., toggling an electrical contact, pushing a button) to change the current state of the RIS. The field devices are then connected to the relay control logic that reacts to the state change to implement the signaling system (e.g., lower the barrier of a level crossing when a train is approaching).

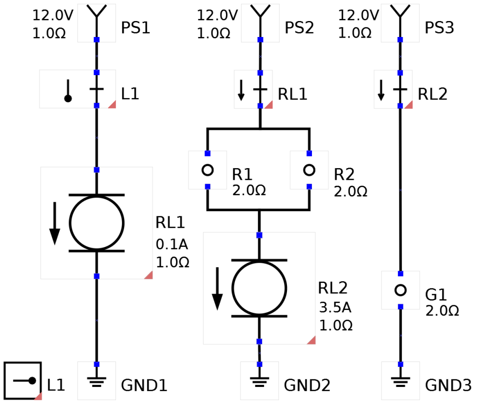


Fig. 2: Principle schemata of the RIS R2G1 that controls the semaphore lights for a RIS level crossing. The RIS is formed by 4 sub-circuits not connected electrically — from left to right: a lever handle, the lever sub-circuit, the sub-circuit that controls the red lights of the traffic semaphore, and a sub-circuit that controls the green light of the train semaphore.

The RIS is implemented as a network of switching electromechanical components where relays are the main switching components. Relays are electrically-controlled analog switches that implement the relay logic. A relay contains a mechanical contact that can open or close a contact (e.g., a relay can open or close the circuit of a semaphore light turning it on or off). A relay controls its contact with a coil that is physically disconnected from the contact itself. The relay switches the contact when the current that flows in the coil falls within or exceeds a *current threshold*. The relay is in the *dropped state* when the coil's current is below the threshold and it is in the *drawn state* otherwise. When a component in a RIS switches to a different state, for example when an agent pushes a button, it induces different circuit contacts and hence a different behavior of the currents and voltages in the RIS. The changes in the currents and voltages can in turn change the state of the relays in the circuit (e.g., the change of the current on the relay coil switches the state of the relay). Thus, a single state change in the RIS may generate a sequence of subsequent state changes.

A *principle schemata* is the standard graphical representation[1] of the design of a RIS. Fig. 2 shows the principle schemata for the RIS that controls the semaphore lights for a level crossing (we will refer to this example as R2G1). In the RIS a lever handle (the component named L_1 in the lower left part of the diagram) controls the semaphore for the level crossing (the red lights R_1 and R_2) and the semaphore for the train track (the green light G_1).

Each connected set of components in the RIS represents a sub-circuit (i.e., sub-circuits are not connected electrically to each other). In Fig. 2 there are 4 sub-circuits — from left to

---

[1] We use the graphical representation defined in the Italian railway regulation UNIFER-CEI S-461 [2].

right, the sub-circuits are the lever handle L_1 (note that the lever handle is by itself a sub-circuit), the sub-circuit that is controlled by L_1, the sub-circuit that controls the red lights, and the sub-circuit that controls the green light.

The sub-circuits are not connected electrically (i.e., with a wire), but are "connected" with some other means (e.g., mechanically, as for a lever, or magnetically, as for a relay coil). A component on one sub-circuit (e.g., a relay coil) opens or closes its contacts (e.g., the relay contacts) that are on other (electrically disconnected) sub-circuits. The principle schemata separates the representation of the components (e.g., a relay coil) and their contacts (e.g., the relay contact). In Fig. 3 we show the symbols for a relay coil and its contacts. In a schemata, the components and their contacts are identified by name: the contacts for a relay coil named RL_1 will be also named RL_1. In a well formed schemata the same component name is used only for a component and its contacts (e.g., two relay coils cannot have the same name) and a contact must have a correspondent component (e.g., if a schemata has a contact named RL_1, it must also have a relay coil named RL_1). We say that there is a *logical connection* between a component and its contacts. The contact symbols in the diagrams further define when the contact should be open or closed. The two left-most components in Fig. 3 are the relay coil RL_1 and an "open" contact RL_1 (in this case, the "open" qualifier identifies a contact that is open by default). The downward arrow shown on the left of the "open" relay contact specifies what will be the state of the contact (i.e. open or closed) depending on the state of its relay coil. In Fig. 3, the contact RL_1 is open when the relay coil RL_1 is dropped and closed otherwise. Note that for the "closed" contact RL_3 of Fig. 3 the downward arrow specifies that the contact is closed when the relay coil RL_3 is dropped, and open otherwise.

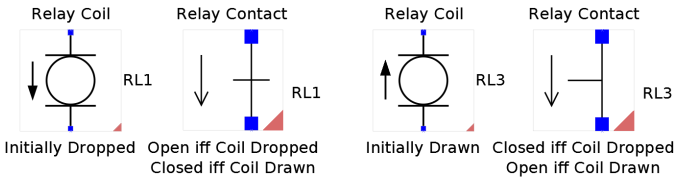


Fig. 3: Symbols of the relay coils and their contacts.

The graphical representation of the components further defines the electrical terminals of the components with blue square boxes and the electrical connections among terminals with black solid lines. The orientation of a component (important to determine the physical position, such as if a lever in the left, center, or right position), is uniquely represented with a red triangle in the bottom right corner of the component. The graphical representation describes also the initial state of switching components like lever handles and relay coils. For relay coils (see Fig. 3) the initial state is determined by the upward or downward arrow at the left of the component, while for lever handles the initial state is the position (left, center, right) of the lever handle (e.g., in the schemata of Fig. 2, the lever handle L_1 is initially in the left position).

In the RIS R2G1 we further have other electrical components like power generators (PS_1, PS_2, and PS_3) that generate a current on the sub-circuit and "ground components" (GND_1, GND_2, and GND_3) that determine the ground for each sub-circuit. The lever "open" contact L_1 in the RIS R2G1 is further closed only if the lever handle L_1 is in the center position (see the position of the lever on the left of the L_1 contact in Fig. 2).

The RIS R2G1 implements a control logic that ensure that every time the green light is on (i.e. the train can travel through the track section with the level crossing), the red lights are also on (i.e. the cars have to stop at the level crossing). In the initial configuration of the RIS R2G1 both the red lights and the green lights are off. This is because the lever handle L_1 is in the left position, thus the lever contact L_1 is open, and hence no current flows in the sub-circuit and the coil RL_1 is dropped. Since the coil RL_1 is dropped, the contact RL_1 is open and no current flows through the red lights and the relay coil RL_2, which are respectively off and dropped. The contact RL_2 is further open and the green light is off. When an operator moves the lever handle L_1 to the center position she starts a sequence of state changes in the RIS.

1) The operator moves the lever handle L_1 to the center position. This change instantaneously closes the lever contact L_1, and the current starts flowing on the coil RL_1.
2) After a small amount of time (the "transient" time of the relay), the relay coil RL_1 switches from the dropped to the drawn state, and the relay contact RL_1 closes. At this point, some current flows on the red lights and on the relay coil RL_2. The red lights turn on.
3) After a small amount of time, the relay coil RL_2 switches to the drawn state and the relay contact RL_2 closes, powering the green light that turns on.

III. SWITCHED MULTI-DOMAIN KIRCHHOFF NETWORKS

Switched Multi-Domain Kirchhoff Networks (SMDKN) are a formalism that models a network of components connected according to the Kirchhoff conservation laws. SMDKN models systems where the components are from different domains (e.g., electrical, hydraulic, mechanical).

The components of a SMDKN are hybrid systems that change a set of discrete modes instantaneously, with a discrete transition, and the value of the physical variables (e.g., the current on a branch) continuously as a function of time. For each possible combination of the discrete modes of the components the SMDKN has a different continuous behavior. Technically, for each configuration the continuous behavior of the SMDKN is defined with a Differential Algebraic Equation derived from the behavior of each single component of the network and the Kirchhoff conservation laws.

IV. MODELING APPROACH

A. *Choosing the modeling abstraction level for relays*

The physical behavior of a RIS is determined by the complex electromechanical phenomena of the relays. The "stationary" relay's states are the drawn and dropped states. However, the real behavior of a relay is more complex due to

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

inertial electromechanical phenomena: the transition between two stationary states is not instantaneous when the current on the relay's coil exceeds (or falls below) the threshold. Thus, we face the problem of modeling the relay's "transient states".

On the one hand a precise modeling of the "transient states" of the relays is challenging. First, such modeling requires complex differential equation; second, a RIS designer cannot reason precisely about the dynamic of the relay in the transient states. On the other hand, a purely "Boolean abstraction" approach that abstracts the physical quantities of the relay (e.g., the current on the coil) is also not adequate. Such abstraction does not permit reasoning about the physical quantities and the relative time between events.

We adopt an intermediate approach where we model the physical quantities of the system but we abstract the "transient state" of the relays. We model that after the relay's current crosses the threshold the change of state of the relay happens in a non-deterministic (but bounded) time interval. This time interval is a known design parameter of a relay. Our approach preserves the actual stationary physics of the system and enables automatic reasoning on the relative time distance between events, that are two key aspects for the designer. In our ongoing project we identified this abstraction level as the suitable trade-off between the designer's needs and the availability of precise and efficient model checking algorithms.

B. Modeling RIS *with* SMDKN

RIS are networks of components electrically connected by means of the Kirchhoff conservation laws. For this reason, we model RIS with SMDKN. The main advantages of the SMDKN modeling are: (i) **Preserve the RIS structure.** We model the RIS network as a SMDKN that has the same network structure (i.e. electrical connections on the components' terminals). Thus, RIS designer can easily model the RIS principle schemata as a SMDKN. (ii) **Compositional modeling.** SMDKN allow us to define the component behaviors independently. Our modeling effort is thus limited to create a library of components for the RIS domain. (iii) **SMDKN are an expressive and flexible modeling language**. SMDKN allow us to model the behavior of switching components as hybrid automata. With hybrid automata we can easily model the "abstraction level" described above. (iv) **Availability of formal analysis techniques**. There already exist efficient formal verification techniques SMDKN [3], [1] that we can apply off-the-shelf.

In the following, we describe in depth our modeling of the principle schemata as SMDKN, focusing on the components, their electrical connections, and the logical connections.

Components: we model a component in the RIS domain as a component in the SMDKN with a hybrid automaton. The hybrid automaton is standard [4]: it defines a finite set of discrete modes and continuous variables. In each discrete mode the automaton defines with a differential equation how the contiguous variables change in function of time, and with a conjunction of Boolean inequalities the invariant conditions. Transitions between discrete modes models the instantaneous state changes. Both RIS and SMDKN components have electrical terminals. We follow the standard approach in *acausal modeling* [5] to encode terminals with two variables, the flow and effort variables. In the electrical domain, the flow variable represents the current on the terminal, while the effort variable represent the potential on the terminal. Flow and effort variables will then be used to model the Kirchhoff conservation laws. The terminal implicitly has two continuous variables to represent flow and effort. Note that a component only exposes the effort and flow variables to the other components.

We describe in depth the modeling of a relay coil and of a faulty lamp. Both components are representative of the RIS library we developed that contains more than 100 components.

The model of the *delayed relay coil* shown in Fig. 4 follows the abstraction level described above where the transient states of the relay coil are modeled non-deterministically. The two modes *Dropped* and *Drawn* of the automaton represent two stable states where the coil has completely actuated its contacts. The two modes *Drawing* and *Dropping* encodes the transient states of the coil. The automaton uses a clock variable *clock* to encode the bounded and non-deterministic transition delays between the stable modes. In particular, the automaton transition from the *Dropped* to the *Drawn* mode only fires when the electrical current I through the coil continuously exceeds the current threshold I_{th} for a non-deterministic time within the specified time interval $[\Delta T_{-}, \Delta T_{+}]$. The same happens for the transition from the *Drawn* to the *Dropped* mode.

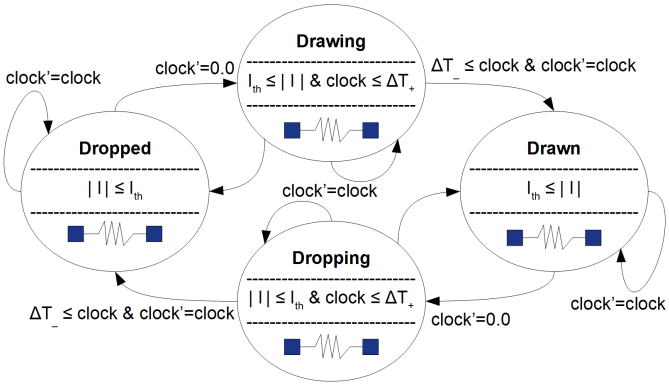


Fig. 4: Hybrid automaton of the delayed relay coil.

Fig. 5 shows the model of a *faulty lamp*, a lamp that can fail either creating a short-circuit or opening the circuit. The *Nominal* mode encodes the correct behavior of the lamp, which behaves as an ohmic load resistor. The automaton encodes the two fault conditions in the *FaultShort* and *FaultBlown* modes, where the lamp behaves respectively as a short-circuit and as an open circuit. The automaton can non-deterministically transition from the nominal mode to the two faulty modes. Since the lamp does not exhibit commutation delays, the hybrid automaton does not have continuous variables.

Physical connections: the semantics of the terminal connections follows the Kirchhoff's conservation laws. Given a set

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

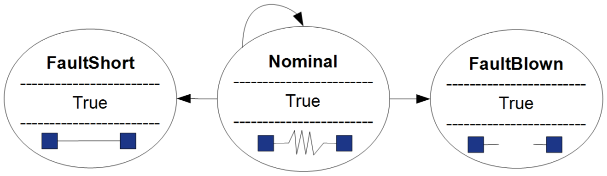


Fig. 5: Hybrid automaton of the faulty-lamp.

of connected terminals, all the effort variables of the terminals take the same value, and the sum of all the flow variables of the terminals equals zero. The SMDKN semantics already considers the Kirchhoff's law.

Logical connections: We model the logical connection among two components (e.g., a relay coil and one of its contacts) with additional synchronization constraints among the discrete modes of the hybrid automata of two components. For instance, for the relay coil RL_1 and the relay contact RL_1 of Fig. 3 the constraint encodes that the coil is in the *Dropped* mode if and only if the contact is in the *Open* mode, and in the *Closed* mode otherwise. Similarly, for the lever handle L_1 and lever contact L_1 of Fig. 2, we say that the handle is in the *Center* mode if and only if the contact is *Closed* mode.

Physical behavior of the running example: we present the relevant electrical behavior of the R2G1 system when lamps can fail either blown or short-circuited. The relay coil RL_2 of Fig. 2 senses the electrical current I_{PS_2} flowing through the parallel connection of the red lamps R_1 and R_2 in order to monitor their status. The current threshold of the coil RL_2 should be properly set to prevent inadvertent activation of the green lamp G_1 when the red lamps are either off or faulty. Tab. I shows the value of the current I_{PS_2} as a function of the 9 possible system modes resulting from the cross product of the 3 modes of the red lamps (see Fig. 5).

System mode	**Current I_{PS_2}**
Both red lamps failed blown	0.0 Ampere
One red lamp failed blown, one red lamp nominal	3.0 Ampere
Both lamps nominal	4.0 Ampere
At least one red lamp failed short-circuited	6.0 Ampere

TABLE I: Values of the electrical current I_{PS_2} sensed by the relay coil RL_2 when the red lamps are power supplied by the closed relay contact RL_1.

To detect the simultaneous activation of the red lamps, the current threshold of the relay RL_2 must be set in the interval $]3.0A, 4.0A[$, for instance to $3.5A$. Notice that, in the system design of Fig. 2, the configurations "both lamps nominal" and "at least one red lamp failed short-circuited" are indistinguishable to the coil RL_2 because in both cases the current I_{PS_2} exceeds the coil threshold of $3.5A$. In the following section, we discuss the implication of this consideration on the overall system safety and we show how the proposed methodology supports the designer on this kind of quantitative reasoning.

V. FORMAL ANALYSIS

In a RIS, the agents determine their next action observing the state of the field devices. Thus, the agents observe a partial-state of the system because the internal state of the control logic is hidden from their point of view. Nevertheless, the correctness of the signaling protocol is implicitly dependent from the implementation of the relay logic.

In our methodology, we propose to analyze the system at two levels of detail: at the higher *railway level* we consider only high-level properties over the field devices (e.g., the lamp emits light, the barrier is closed), despite the technological details of the control logic; at the lower *physical level* we consider properties that investigate the internal technological aspects of the control logic and of its physics (e.g., two terminals must be short-circuited when a relay is in a specific mode). This layered approach reduces the total effort to specify properties: the properties at the railway level are independent from the implementation of the control logic and can be reused for multiple control logic implementations.

Properties specification: a property at the physical level predicates on low level aspects of the system such as physical quantities and operating modes of the components. Focusing on the electrical domain, we can predicate either on the voltage drop ΔV across a pair of terminals, or on the current I that flows through a terminal. A similar approach holds in the mechanical domain replacing *current* and *voltage* with *torque* and *angular velocity*. A property can further predicate on the operational modes of the components.

A railway property is automatically mapped onto a combination of physical properties, hiding its implementation details. For instance, consider the sentence *"the lamp* G_1 *emits light"*. Since a lamp is electrically equivalent to an ohmic load resistor, the property is equivalent to *"the lamp* G_1 *consumes electrical power"* that in turns is equivalent to the first-order logical formula $I_{\mathrm{G}_1} \neq 0.0 \wedge \Delta V_{\mathrm{G}_1} \neq 0.0$. Notice that in the context of physical reasoning it is necessary to predicate on *both* currents and voltage drops in order to distinguish the nominal behavior of the lamp from the faulty ones (i.e. those in which the lamp is power supplied, but does not emit light). In fact, a short-circuited lamp is traversed by a non-null current ($I_{\mathrm{G}_1} \neq 0.0$), but its voltage drop is zero ($\Delta V_{\mathrm{G}_1} = 0.0$); similarly, a blown lamp is traversed by a null current ($I_{\mathrm{G}_1} = 0.0$) even if its voltage drop is different from zero ($\Delta V_{\mathrm{G}_1} \neq 0.0$). In our specification settings, we could also refine the property exploiting detailed information available to the designer. Assuming to know the range of nominal currents absorbed by the lamp (e.g., from its data sheet), we could rewrite the predicate $I_{\mathrm{G}_1} \neq 0.0$ into a more precise one such as $1.5 \leq |I_{\mathrm{G}_1}| \leq 2.3$.

Analysis of the running example: in the following we demonstrate the need of the quantitative reasoning, which is enabled by our modeling approach, using the RIS R2G1 of Fig. 2. We further consider variants of the R2G1 model changing the fault model for the red lamps and the current threshold of the relay coil RL_2. The red lamps may either not fail, or the red lamps may blown (see the *FaultBlown* state in Fig. 5), or the red lamps can introduce a short circuit (see the *FaultShort* state in Fig. 5). The current threshold on the relay coil RL_2 may be either 2.5A, or 3.5A, or 4.5A. We

ISBN: 978-0-9835678-8-2.

Nr.	R2G1 Variants		Verification results	
	Faults	RL_2 thresh.	RP	SP
1	None	2.5A	Hold	Hold
2	None	3.5A	Hold	Hold
3	None	4.5A	Doesn't hold	Hold
4	Blown	2.5A	Hold	Doesn't hold
5	Blown	3.5A	Hold	Hold
6	Blown	4.5A	Doesn't Hold	Hold
7	Short	2.5A	Hold	Doesn't hold
8	Short	3.5A	Hold	Doesn't hold
9	Short	4.5A	Hold	Doesn't hold

TABLE II: Verification results (property holds or does not hold) on variants of R2G1 introducing faults on the red lamps and changing the current threshold on the relay coil RL_2.

consider the reachability property $RP :=$ *"the green lamp* G_1 *can emit light"*, and the safety property $SP :=$ *"if the green lamp* G_1 *emits light, then both red lamps* R_1 *and* R_2 *emit light"*. We expect RP to hold for R2G1, witnessing an execution scenario where green lamp is on, and SP to hold to ensure the safety of the R2G1 system. The verification results are available in Tab. II.

When the current threshold of the relay coil RL_2 is over-dimensioned to 4.5A, the unexpected verification of the property RP proves that the green lamp cannot emit light because the relay contact RL_2 will never supply power to the lamp (rows 3, 6). Decreasing the threshold, RP always holds and this fact guarantees that the green lamp can turn on.

When the current threshold is under-dimensioned to 2.5A, the safety property SP is violated in the system variant with blown lamps (row 4). The counterexamples returned by the model checker provide execution scenarios able to reach the violation, but do not represent an exhaustive analysis. To determine all the minimal configurations of faults that lead to the violation, we perform formal safety assessment to compute fault-trees. For the system variant of row 4, the fault-tree of the safety property SP shows two possible fault configurations: when one red lamp fails blown, the other red lamp can still emit light absorbing 3.0A (see Tab. I) from the power supply PS_2. The 3.0A current exceeds the under-dimensioned threshold of 2.5A, thus the relay RL_2 inadvertently supplies power to the green lamp, violating the safety property. We fix this design flaw setting the coil threshold to 3.5A (row 5).

Unfortunately, the safety violation still occurs when the lamps fail short-circuited (row 8). The safety assessment process reveals that if any red lamp fails short-circuited, a current of 6.0A is drawn from PS_2 (see Tab. I), and the relay coil RL_2 is again deceived. This design flaw cannot be fixed by simply adjusting the electrical parameters of the system, but requires the upgrade of the entire design as shown in Fig. 6. In the system upgrade, the additional relay coil RL_3 is *Drawn* when the current I_{PS_2} exceeds the threshold of 4.5A, that makes its contact RL_3 open, thus preventing the green lamp from turning on if a red lamp is short-circuited.

Need of quantitative modeling for verification: we make a small digression to report the main limitations we encountered while applying the traditional Boolean modeling approach (i.e. the one based on the concept of conductive paths) that led us to this work. Referring to the upgraded R2G1 design of Fig. 6, Fig. 7 shows the value of the current I_{G_1} flowing through the green lamp G_1 as a function of the current I_{PS_2} sensed by the relay coils RL_2 and RL_3. Our physical modeling approach (Fig. 7-(2)) is able to properly discriminate the faulty scenarios (i.e. $I_{\mathrm{PS}_2} < 3.5A$ and $I_{\mathrm{PS}_2} > 4.5A$, where $3.5A$ is the RL_2 threshold and $4.5A$ is the RL_3 threshold), keeping the green lamp properly turned-off (i.e. $I_{\mathrm{G}_1} = 0.0A$). Differently, the expressiveness of the Boolean approach ((Fig. 7-(1))) cannot discern between different values that are greater than zero. This means that, for every current $I_{\mathrm{PS}_2} > 0.0A$, the relay coils RL_2 and RL_3 would be considered always Drawn, resulting in a spurious behavior with the green lamp always turned-off.

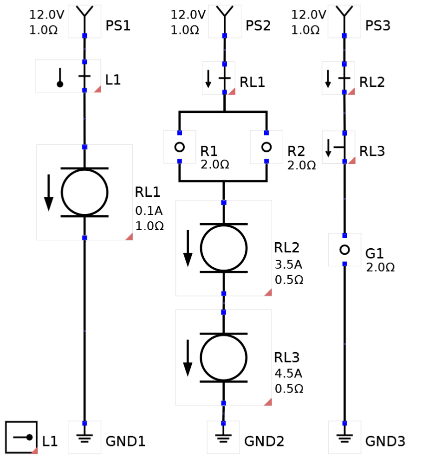


Fig. 6: Upgraded design of the RIS R2G1 from Fig. 2

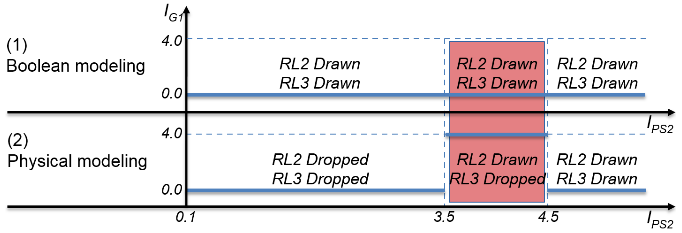
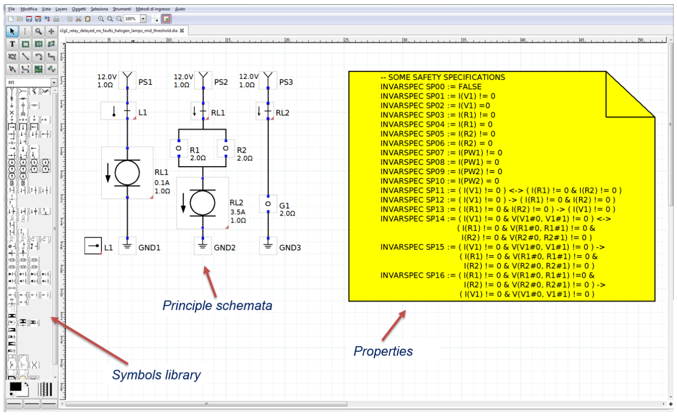


Fig. 7: Spurious behavior on the green lamp G_1 introduced by the Boolean modeling. The relay coils RL_2 and RL_3 are permanently Drawn, and keep G_1 always turned off.

VI. TOOL CHAIN

The proposed methodology was implemented in a tool chain composed of various blocks. The first block is a graphical front end (Fig. 8) based on a customization of the DIA [6] modeling environment. The palette of the front end supports over 100 distinct graphical symbols, corresponding to a subset of the components that can be found in RIS according to the Italian regulation. Each symbol is associated to an internal data structure, where parameters of various kinds are associated (e.g. delay in response time, resistance, and angular velocity).

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.



```
-- SOME SAFETY SPECIFICATIONS
INVARSPEC SP00 := FALSE
INVARSPEC SP01 := I(V1) != 0
INVARSPEC SP02 := I(V1) =0
INVARSPEC SP03 := I(R1) != 0
INVARSPEC SP04 := I(R1) = 0
INVARSPEC SP05 := I(R2) != 0
INVARSPEC SP06 := I(R2) = 0
INVARSPEC SP07 := I(PW1) != 0
INVARSPEC SP08 := I(PW1) = 0
INVARSPEC SP09 := I(PW2) != 0
INVARSPEC SP10 := I(PW2) = 0
INVARSPEC SP11 := ( I(V1) != 0 ) <-> ( I(R1) != 0 & I(R2) != 0 )
INVARSPEC SP12 := ( I(V1) != 0 ) -> ( I(R1) != 0 & I(R2) != 0 )
INVARSPEC SP13 := ( I(R1) != 0 & I(R2) != 0 ) -> ( I(V1) != 0 )
INVARSPEC SP14 := ( I(V1) != 0 & V(V1#0, V1#1) != 0 ) <->
                  ( I(R1) != 0 & V(R1#0, R1#1) != 0 &
                   I(R2) != 0 & V(R2#0, R2#1) != 0 )
INVARSPEC SP15 := ( I(V1) != 0 & V(V1#0, V1#1) != 0 ) ->
                   ( I(R1) != 0 & V(R1#0, R1#1) != 0 &
                    I(R2) != 0 & V(R2#0, R2#1) != 0 )
INVARSPEC SP16 := ( I(R1) != 0 & V(R1#0, R1#1) !=0 &
                    I(R2) != 0 & V(R2#0, R2#1) != 0 ) ->
                   ( I(V1) != 0 & V(V1#0, V1#1) != 0 )
```

Fig. 8: Front end of our design tool.

The front end supports the connection between components, and carries out a number of sanity checks to pinpoint errors such as dangling terminals, missed components in logical connections, and conflicting logical connections between incompatible symbols. The front end also supports the definition of railway predicates representing some relevant physical conditions. Properties are expressed in form of linear temporal logic over both railway and physical predicates.

The second block is a compiler from SMDKN to hybrid automata network, symbolically expressed in the HYDI language [7]. The compiler is written in Python, and implements the conversion traversing the network based on an extensible library of behavioral component descriptions.

The third block is the HYCOMP model checker [8], that processes the resulting HYDI network and carries out the required analyses, leveraging various SMT-based engines for model checking [9], together with XSAP [10] for safety analysis and fault-trees production.

VII. EXPERIMENTAL EVALUATION

Benchmarks: we evaluated the proposed methodology analyzing a scalable, industrial-size RIS referred to as RISCS. Fig.9 shows a simplified layout of the RISCS, omitting both the electrical connections among devices and other confidential details of the relay logic. The RISCS$_{[i]}$ system represents a railway section along a bidirectional train line containing a sequence of i level crossings, with $1 \leq i \leq 10$. The section is protected on each track side by a warning and a protection semaphore. The warning/protection semaphores have three yellow/red lamps (WYL/PRL) and two green/green lamps (WGL/PGL). The lamps of the same color are electrically connected in parallel to improve the redundancy of each semaphore. Every level crossing is protected on each street side by a barrier (LCB) and by a vehicular semaphore consisting of one red lamp (LCL). The presence of the train along the line is detected by means of the train approaching pedals (TAP) and of the train detection pedals (TDP). The maintainers can completely/partially disable the section acting on several maintenance levers (GML, TAML, LCML) at the maintenance place. The train dispatcher can activate the section acting on the section enabling lever (SEL) at the train station. The relay logic is electrically connected to all the devices shown in Fig.9. The relays sense the electrical currents flowing through every connected device and actuate a specific control sequence, transferring energy between the devices. For instance, when the train pushes the left train approaching pedal (left TAP), closing its sub-circuit, the logic checks the magnitude of the current flowing through the level crossing lamps (up/down LCL) of the vehicular semaphores, and, if all the lamps work properly, the logic powers on the engines of the barriers (LCB) to start the lowering sequence.

We modeled the RISCS case studies with our tool, selecting and modeling the components and their parameters, their interconnections, and verifying properties of interest. The overall modeling task lasted for about 3 weeks, including the creation of a reusable behavioral component library.

The largest system RISCS$_{[10]}$ contains 141 power supplies, 22 resistors, 113 relays, 15 levers, 12 pedals, 678 contacts, 40 lamps, 23 maintenance lights, and 54 circuit breakers (printed on twenty A4-sheets of paper). These components are distributed over 125 sub-circuits. The conversion of the corresponding SMDKN into hybrid automaton returns an SMT encoding that uses 437 Boolean variables to encode the discrete part, and 6281 real-valued variables to encodes the physical part. Clearly, the size of the state-space makes traditional manual inspection extremely time-consuming, expensive, and unfeasible in practice.

We presents the results of the analysis on the nominal and faulty variants of the RISCS system, where up to 80 electrical faults (i.e. blown or short-circuited lamp) are injected on the 40 semaphore lamps in the case of the RISCS$_{[10]}$ benchmark.

Verification: we model checked the RISCS system against 190 invariant properties, running the two verification algorithms IC3 [11] and BMC [12] that represent complementary techniques to either verify or falsify properties. We run the experiments on a 3.5 GHz cpu with 16GB RAM, with time out (TO) set to 3600 seconds. About half of the properties represent scenarios that are supposedly feasible, and are used to validate the system design. The first validation round reported that some scenarios were found to be (unexpectedly) unfeasible. Upon fixing some buggy components in the behavior library, all the scenarios were proved to be feasible, within the timeout of 3600s, in both the nominal and faulty case. The resulting execution traces were analyzed and validated by the domain experts. Examples of scenario include that every lamp of every semaphore can be turned on and then off, or that every barrier can be completely lowered and then raised.

The remaining properties express the absence of safety violations. Most of them are verified in the nominal case within the timeout, except for three properties on the synchronization among the warning and protection semaphores.

Some relevant properties expressing the proper synchronization between the semaphore lights and the barriers positions hold also under the non-nominal case (i.e. when components are subject to faults). For instance, the model guarantees that the green lamps of the protection semaphores are off when the

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

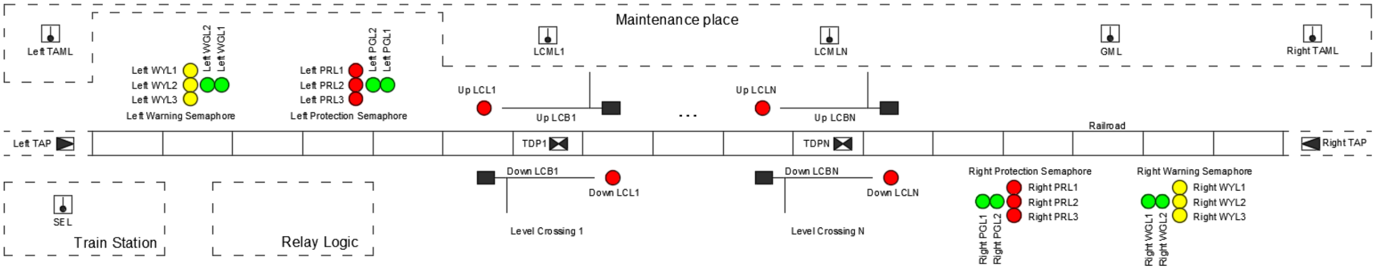


Fig. 9: Physical layout of the RISCS$_{[i]}$ case study. **Legend:** Warning Yellow Lamp (WYL), Warning Green Lamp (WGL), Protection Red Lamp (PRL), Protection Green Lamp (PGL), Level Crossing Lamp (LCL), Level Crossing Barrier (LCB), Train Approaching Pedal (TAP), Train Detection Pedal (TDP), Level Crossing Maintenance Lever (LCML), Train Approaching Maintenance Lever (TAML), General Maintenance Lever (GML), Section Enabling Lever (SEL).

level crossing barriers are not completely closed. Moreover, we are guaranteed that the colors of every semaphore are turned on in a mutually-exclusive way. Noteworthy, we successfully verified an electrical safety requirement (a low-level electrical property) prescribed by the national regulation: the level crossing lamps are short-circuited when the barriers are open and resting to prevent inadvertent activation.

59 safety properties were violated in the faulty case. Some of them check for each semaphore if there is always at least one lamp turned on. Of course, in case of multiple lamp faults, this condition cannot be avoided because all the lamp might fail. With safety analysis, we compute the fault tree responsible for the violations. For a warning semaphore, the fault tree shows that the violation might be reached in 7 distinct circumstances: either all yellow lamps are blown, or all green lamps are blown, or at least one yellow lamp is short-circuited, or at least one green lamp is short-circuited. The first two circumstances represent fault configurations of size 3 and 2, respectively the number of yellow and green lamps, that would be hard to spot by manual inspection.

VIII. RELATED WORK

Formal methods have been heavily applied in the railway domain. Important works on the verification of interlocking systems include (but are not limited to) [13], [14], [15], [16], [17], [18]. These works are not related, since they do not consider the specific case of relay circuits.

To the best of our knowledge, no works address the verification problem of a RIS based on its hybrid physical behavior. Closely related works are [19], [20], [21], [22]. While we model the evolution of continuous signals over time, the above works model Boolean signals evolving over discrete time. Furthermore, these works assume that the interaction with the environment is limited to one input per cycle to ensure that the internal micro-sequence of relay commutations started from an input command is fully extinguished (run to completion) before the arrival of the next input. In [22], two interesting observations are made. First, the discrete model of time does not support reasoning about relative time distances (e.g., between events, and on parasitic delays); second, the restriction on the number of inputs per execution cycle only works under the assumption that the control logic reacts “quickly enough” to every change in its environment. Our approach overcomes both limitations adopting a continuous model of time and not imposing restrictions on the environment. Thus, we deal with an arbitrary number of concurrent inputs and analyze the effect of inputs received in the middle of an internal micro-sequence.

We now analyze these works in more detail. The works [19], [20] present a practical approach to the RIS safety certification. A *Boolean* model is extracted from the RIS and analyzed via SAT-based abstraction-refinement. Our SMT-based approach enables more fine grained analyses, modeling the precise physics of the system and preventing spurious behaviors introduced by the Boolean abstraction. The work [21] builds a Boolean model based on the abstraction concept of *conductive path*: a relay coil is drawn iff all the conduction conditions along a conductive path from a power supply to the coil are satisfied. This approach is subject to several limitations: it is only valid under some assumptions on the system physics (e.g., all the power supplies are always up and running); it requires the enumeration of a potentially exponential number of conductive paths; it does not permit a quantitative reasoning (e.g., how much current flows through a conductive path). There is only one work [22] that considers risk analysis and the effects of single-mode faults on the system safety. These faults are Boolean and limited to the discrete state of relays (e.g., stuck at dropped/drawn). In our work we allow the designer to specify a larger class of faults, both on the discrete and physical state of components, with no limitation on the contemporaneity of fault occurrences.

IX. CONCLUSION

In this paper we proposed an approach to understand legacy relay circuits in the railway domain. We rely on an accurate representation at the physical level in form of Switched Kirchhoff Networks, that is then reduced to a symbolically represented network of hybrid automata, and then analyzed by means of SMT-based model checking. The experimental evaluation demonstrates the precision and scalability of the analyses. The proposed methodology is at the core of an ongoing research project aiming at the in-the-large analysis of legacy railway interlocking and the open specification of computer-based solutions. Directions for future research include the definition of a library of property patterns, the definition of specific verification engines, and the integrated animation of counterexamples.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

REFERENCES

[1] A. Cimatti, S. Mover, and M. Sessa, "SMT-based analysis of switching multi-domain linear Kirchhoff networks," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017*, 2017, pp. 188–195.

[2] Comitato Elettrico Italiano, "UNIFER-CEI S 461: sigle e segni grafici per gli schemi dei circuiti elettrici degli impianti di segnalamento ferroviario," Comitato Elettrico Italiano, Standard, 1976.

[3] A. Cimatti, S. Mover, and M. Sessa, "From Electrical Switched Networks to Hybrid Automata," in *FM 2016: Formal Methods, Proceedings*, 2016, pp. 164–181.

[4] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, 1996*, 1996.

[5] P. Fritzson, *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach.* John Wiley & Sons, 2014.

[6] GNOME, "Dia," https://gitlab.gnome.org/GNOME/dia, 2017.

[7] A. Cimatti, S. Mover, and S. Tonetta, "Hydi: A language for symbolic hybrid systems with discrete interaction," in *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011.* IEEE Computer Society, 2011, pp. 275–278.

[8] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "HyCOMP: An SMT-based model checker for hybrid systems," in *TACAS*, 2015.

[9] 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. Lecture Notes in Computer Science, vol. 8559, 2014, pp. 334–342.

[10] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri, "The xSAP safety analysis platform," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2016, Proceedings*, 2016, pp. 533–539.

[11] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Infinite-state invariant checking with IC3 and predicate abstraction," *Formal Methods in System Design*, vol. 49, no. 3, pp. 190–218, 2016.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[13] A. E. Haxthausen and P. H. Østergaard, "On the use of static checking in the verification of interlocking systems," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Proceedings, Part II*, 2016, pp. 266–278.

[14] L. V. Hong, A. E. Haxthausen, and J. Peleska, "Formal modelling and verification of interlocking systems featuring sequential release," *Sci. Comput. Program.*, vol. 133, pp. 91–115, 2017.

[15] A. Fantechi, A. E. Haxthausen, and H. D. Macedo, "Compositional verification of interlocking systems for large stations," in *Software Engineering and Formal Methods, SEFM 2017, Proceedings*, 2017, pp. 236–252.

[16] V. Hartonas-Garmhausen, S. V. A. Campos, A. Cimatti, E. M. Clarke, and F. Giunchiglia, "Verification of a safety-critical railway interlocking system with real-time constraints," *Sci. Comput. Program.*, vol. 36, no. 1, pp. 53–64, 2000.

[17] A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltsev, "Formal verification and validation of ERTMS industrial railway train spacing system," in *Computer Aided Verification, CAV 2012, Proceedings*, 2012, pp. 378–393.

[18] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso, "Formal verification of a railway interlocking system using model checking," *Formal Asp. Comput.*, vol. 10, no. 4, pp. 361–380, 1998.

[19] A. Bonacchi, A. Fantechi, S. Bacherini, and M. Tempestini, "Validation process for railway interlocking systems," *Sci. Comput. Program.*, vol. 128, pp. 2–21, 2016.

[20] A. Bonacchi, A. Fantechi, S. Bacherini, M. Tempestini, and L. Cipriani, "Validation of railway interlocking systems by formal verification, A case study," in *Software Engineering and Formal Methods - SEFM 2013*, 2013, pp. 237–252.

[21] A. E. Haxthausen, A. A. Kjær, and M. L. Bliguet, "Formal development of a tool for automated modelling and verification of relay interlocking systems," in *FM 2011: Formal Methods, Proceedings*, 2011, pp. 118–132.

[22] L. Eriksson, "Using formal methods in a retrospective safety case," in *Computer Safety, Reliability, and Security, SAFECOMP 2004, Proceedings*, 2004, pp. 31–44.

ISBN: 978-0-9835678-8-2.

Design-Time Railway Capacity Verification using SAT modulo Discrete Event Simulation

Bjørnar Luteberget
Railcomplete AS
Sandvika, Norway
Email: bjornar.luteberget@railcomplete.no

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden
Email: koen@chalmers.se

Christian Johansen
University of Oslo
Oslo, Norway
Email: cristi@ifi.uio.no

Abstract—**Railway capacity is complex to define and analyze, and existing tools and methods used in practice require comprehensive models of the railway network and its timetables. Design engineers working within the limited scope of construction projects report that only ad-hoc, experience-based methods of capacity analysis are available to them. Designs have subtle capacity pitfalls which are discovered too late, only when network-wide timetables are made – there is a mismatch between the scope of construction projects and the scope of capacity analysis, as currently practiced.**

We suggest a language for capacity specifications suited for construction projects, expressing properties such as running time, train frequency, overtaking and crossing. Verifying these properties amounts to solving a planning problem constrained by discrete control system logic, network topology, laws of motion, and sparse communication. To describe train dynamics one uses second-order linear differential equations which when solved analytically give rise to non-linear equations over real variables.

We argue that reasoning over the whole discrete/continuous solution space is not efficient with current state-of-the-art solvers. Instead, we have solved the problem by building a special-purpose solver which splits the problem into two: an abstracted SAT-based dispatch planning, and continuous-domain dynamics and timing constraints evaluated using discrete event simulation. The two components communicate in a CEGAR-loop (counterexample-guided abstraction refinement). We show that our method is fast enough at relevant scales to provide agile verification in a design setting, and we present case studies based on data from existing infrastructure and ongoing construction projects.

I. INTRODUCTION

This paper addresses a central problem that occurs when designing the layout and control systems for railway stations: Does the station infrastructure have the *capacity* to handle the amount of trains and the desired traveling times to provide adequate service in transportation of goods and passengers?

As an example, consider the question of crossing trains on a railway station. Fig. 1 shows two sequences of movements which result in such a crossing. There are a number of details of the railway design which can cause this scenario to become infeasible (or take an unacceptably long time), such as signal placement, detector placement, correct allocation and freeing of resources, track lengths, train lengths, etc.

Systematic capacity analysis for railways is typically performed on the scale of national railway networks, using comprehensive input on infrastructure and timetables, and only after the complete design is finished. Moreover, the widely used methods and tools for capacity analysis are heavy-duty methods, consisting of complicated simulations, and require specialized knowledge, thus not being suitable for agile design-time verification of railway stations. As a consequence, railway construction projects usually rely on informal, vague, or even non-existent capacity specifications, and engineers need to make ad-hoc/manual analyses of how the control system can provide this capacity.

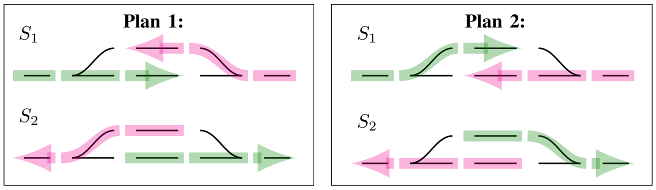


Fig. 1: Two alternative plans for achieving a crossing of two trains on a two-track station. The green areas show track segments which are currently occupied by a train going from left to right, while the pink areas show track segments which are currently occupied by a train going from right to left.

Our goal is to develop a verification technique and tool to help engineers specify capacity properties *at design time* and to check these automatically. To be agile, the tool needs to (1) have reasonable running times so that the verification can be run on the fly as the design is being updated by an engineer working in a drafting CAD application, and (2) keep the required input to the minimum of information needed to verify relevant properties. This style of verification gives engineers immediate feedback on their design decisions while requiring small amounts of specification and verification work.

The problem: We consider **the low-level railway infrastructure capacity verification problem**, which we define as follows:

> Given a railway station track plan including signaling components, rolling stock dynamic characteristics, and a performance/capacity specification, verify whether the specification can be satisfied and find a dispatch plan as a witness to prove it.

Solving this problem subsumes the following railway infrastructure design activities:

- Low-level **running time** analysis – verify the time required for getting from point A to point B.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

- Low-level **schedulability** analysis – verify frequency of trains arriving at a station, and simultaneous opportunities for crossing, parking, loading, etc.
- **Combinations** – verify running time requirements on schedulable operations.

Our approach: In this paper we suggest a formalization of capacity requirements as a set of operational scenarios involving a set of trains, a set of locations to visit, and a set of timing constraints.

Verification in this domain can in principle be encoded into the SMT [1], [2], [3] or PDDL+ [4] languages, essentially resulting in a SAT modulo non-linear real arithmetic problem [5], [6]. Many solvers can handle such problems [7], [8], [9], but we found that the problem size of our test cases, in terms of the number of planned actions and in terms of number of interacting Boolean and non-linear real logic terms, were out of reach for agile verification. Also, train dynamics using only constant acceleration $x'' = c$ is in some cases too simplistic for engineering. We would like to be able to extend the dynamics equations using e.g. polynomials of higher order or even numerical integration.

Therefore, we have developed a verification tool chain that uses a simple CEGAR-loop between a SAT-based planning tool that works on a discrete abstraction of control system commands, and a discrete event simulation engine (DES) [10] that calculates detailed continuous results for a specific plan, taking the physics of moving trains into account.

The SAT-based planner uses bounded model checking (BMC) [11] where time is reduced to a series of partially ordered actions with unknown durations, and the choice of actions are the available commands in the control system. The DES component verifies the continuous time/space results given the Boolean decisions of control system commands, and adds new SAT constraints excluding unsatisfactory solutions.

The separation of discrete and continuous domains also has the advantage that the simulation component can be extended to handle more complex models, such as engine power curves, tunnel air resistance, curve rolling resistance, train weight distribution, etc., without affecting the planning logic or its computational complexity.

We have tested our method and tool on practical examples from existing infrastructure and ongoing construction projects in collaboration with railway engineers in Railcomplete AS.

The rest of the paper is organized as follows: Sec. II contains an overview of the railway design process and the principles for analysis of these designs. We present a structure for capacity specifications, together with examples of how they can be used in construction projects. Sec. III describes the tool chain and the solver architecture that we propose to verify performance properties and integrate agile verification in the construction project workflow, and how each of the components of our solver are implemented. Sec. IV contains performance evaluations in a set of relevant case studies. Sec. V gives pointers to related work, and Sec. VI presents our conclusions.

II. DOMAIN BACKGROUND AND PROBLEM DESCRIPTION

Railway capacity is hard to define precisely (see [12], [13] for a discussion). Any capacity measure will necessarily make assumptions about the operation of the railway. One can say that the railway infrastructure does not have an inherent capacity, only capacity for specific use cases. As such, a fully accurate assessment of capacity can only be made under a fully specified timetable, meaning that every train's arrival and departure times at all stations in the network must be known. This makes for a highly coupled analysis, as constructing an actual timetable requires bringing together details about infrastructure, rolling stock, transportation demands, and crew schedules. Such work can be done using commercial tools like RailSys [14], OpenTrack [15], or LUKS [16]. Good overviews of methods are presented in [17] and [18].

The so-called analytical approaches to capacity analysis using networked queuing theory [19], maximum flow (originally posed as a railway capacity problem [20]), or max-plus algebra [21], can give preliminary or low-precision network-wide results, but fail to account for the critical low-level factors which are relevant for verification in construction projects, specifically discrete control system logic, communication, and train acceleration and braking dynamics.

Because the verification feedback loop between design and capacity analysis is either very time-consuming or too coarse-grained, railway engineers end up re-using proven design concepts or allowing sizable margins, e.g., in track lengths.

However, modern construction practice expects and demands optimization. When space requirements, performance requirements and costs are squeezed to the limit, the tradition-based railway engineering approach lacks the methods to accurately reason about the expectations of the finished system from partially finished design plans.

Using *agile verification* of high-level properties from the beginning of a design project, and in every step of the process, allows engineers to better see the consequence of each decision, and immediately uncover errors and shortcomings that would otherwise be discovered only months or years later.

Railway design

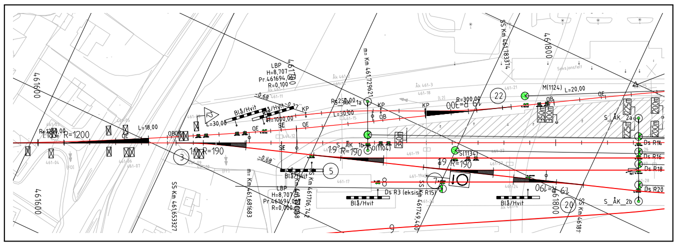
The railway design activity produces the following artifacts:

- Track and trackside component layout, describing the locations of tracks, switches, signals and detectors (see Fig. 2a).
- Interlocking specifications, describing the requirements for the logic of the control system (see Fig. 2b).

These design artifacts are the subject of verification, i.e. the *model*. Ensuring performance in the context of a construction project consists of verifying *properties* describing a set of trains moving on the tracks and the goals which need to be accomplished by these movements.

To verify performance properties, we need to find a sequence of trains and elementary routes for the train dispatcher, i.e., a *dispatch plan*, which when executed under safety and

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.



(a)

Elementary route	Start signal	End signal	Switch position	Track segments	Conflicts
AC	A	C	X right	1, 2, 4	AE, BF
AE	A	E	X left	1, 2, 3	AC, BD
BF	B	F	Y left	4, 5, 6	AC, BD
BD	B	D	Y right	3, 5, 6	AE, BF

(b)

Fig. 2: Railway design artifacts: (a) Cut-out from 2D geographical CAD model (construction drawing) of preliminary design of the Arna station signalling. (b) Simplified example of tabular interlocking (control system) specifications.

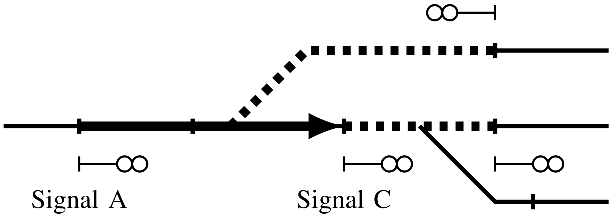


Fig. 3: Elementary route AC from signal A to the adjacent signal C. The thick line indicates track segments on the train's path which are reserved for this movement, and the dashed lines indicate reserved track segments outside the path.

correctness constraints (described in Sec. II-A below), demonstrate the properties described in the performance requirements (detailed in Sec. II-B below).

A. Safety and correctness of train movements

Low-level analysis of train movements covers a wide range of constraints given by the track layout, the control system, and operational procedures, to be certain that the analysis produces detailed, realistic results. The following subsections give an overview of these constraints, divided into four classes.

1) Physical infrastructure: Trains travel on a network of railway tracks which have physical properties such as length, gradient, curvature, etc. Tracks branch off using *switches*, whose *setting* determines where the train goes. Detectors on the track are used by the control system to determine whether track segments are occupied. The physical infrastructure also determines the *sight areas*: the set of locations where a train receives information from a given signal.

2) Allocation of resources: Avoiding collisions by exclusive use of resources is the responsibility of the interlocking, which takes requests from the dispatcher for activating **elementary routes**. An elementary route is the smallest unit of resources that can be allocated to a train, see Fig. 3. Route activation is a process which proceeds as follows:

1) Wait for all **required resources**, such as track segments and switches, to be free. Resources required by a route are typically any resource in the train path (or sometimes outside of it), which ensure that all movements are performed at a safe distance from each other.
2) **Movable elements** (e.g. switches) must be set to correct positions. If they are not, start a sub-process which moves the element into place, and wait for this process to finish before proceeding.
3) **Signals** are then set to show the 'proceed' aspect to the train when the above steps are finished. When the front of the train has passed the signal, it is immediately reset to show the 'stop' aspect.
4) A **release** process is started, which waits for the train to finish using the allocated resources (i.e. to travel over them) and frees them when this has happened.

3) Communication constraints: After movement has been allowed by the control system, the driver must be informed of this fact. When a route is activated, a train inside the sight area of the route's entry signal reads the signal's message that movement authority is given. The train driver may then drive the train forward until the next signal. The following types of signalling systems are common in railways:

- Traditional signaling with trackside lamps. Communication is limited by how many different aspects the lamps can show. To avoid high-speed trains slowing down at every signal, several consecutive elementary routes can be signaled in advance using so-called distant signals.
- Automatic train protection systems (ATP) work similarly to signals, but may give more information. Many ATP systems communicate information through magnets or short-range radio at specific locations on the track, corresponding to a signal sight area of zero length.
- The European Rail Traffic Management System (ERTMS) currently being implemented in many European countries replaces lamp signals with trackside marker boards, and uses long-range radio for communication. This effectively removes the communication constraint, as the radio can be used to update any train's movement authority at any time.

4) Laws of motion: Trains move within the limits of given maximum acceleration and braking power. Train drivers need to plan ahead for braking so that the train respects its given movement authority and speed restrictions at all times.

The speed increase from v_0 to v over a time interval Δt is limited by the train's maximum acceleration a:

$$v - v_0 \leq a\Delta t.$$

However, when there is a more restrictive speed restriction ahead, the driver must start braking in time to meet the restriction. A signal showing the 'stop' aspect can be treated as a speed restriction of zero. Since speed restrictions change

ISBN: 978-0-9835678-8-2.

with time, the driver must re-evaluate their actions whenever new information is received.

A train has the following constraint on its velocity v for each restriction,

$$v^2 - v_i^2 \leq 2bs_i,$$

where v_i is the maximum allowed speed, s_i is the distance to the location where the restriction starts, and b is the maximum retardation achieved by braking.

See [22] for a more in-depth description of railway operation principles.

B. Station performance requirements

To capture typical performance and capacity requirements in construction projects, we define an **operational scenario** $S = (V, M, C)$ as follows:

1) A set of **vehicle types** V, each defined by a length l, a maximum velocity $v_{\max}$, a maximum acceleration a, and a maximum braking retardation b.
2) A set of **movements** M, each defined by a vehicle type and an ordered sequence of visits. Each visit q is a set of alternative locations $\{l_i\}$ and an optional minimum dwelling time t_d.
3) A set of **timing constraints** C, which are two visits q_a, q_b, and an optional numerical constraint t_c on the minimum time between visit q_a and q_b. The two visits can come from different movements. If the time constraint t_c is omitted, the visits are only required to be ordered, so that $t_{q_a} < t_{q_b}$.

To demonstrate how this structure captures requirements of railway construction projects, we give some examples using the syntax of the file format used in our tool[1]. First, we define the following vehicle types:

```
vehicle passengertrain length 220.0
  accel 1.0 brake 0.9 maxspeed 55.0
vehicle goodstrain length 850.0
  accel 0.5 brake 0.5 maxspeed 20.0
```

The following set of **performance specifications** are selected prototypical versions of specifications that railway engineers have suggested as useful for automated verification:

- **Running time**: expresses an expectation of how long it should take for a train to travel between two locations. To specify this, we simply require that a train visits some location `b1` and later visits some other location `b2`. A timing constraint of 90.0s between these visits sets the running time requirement.

```
movement passengertrain {
  visit #a [b1]; visit #b [b2] }
timing a <90.0 b
```

- **Train frequency**: a train station processes a set of trains arriving and departing with a fixed frequency. On a two-track station, we exemplify a sequence of four trains and their relative departure times.

```
movement passengertrain {
  visit [b1]
  visit [platform1,platform2] wait 60.0
  visit #e1 [b2] }
// ...3 more trains with visits e2, e3, e4.
timing e1 <90.0 e2
timing e2 <90.0 e3
timing e3 <90.0 e4
```

- **Overtaking**: trains traveling in the same direction can be reordered. For example, we specify a passenger train traveling from `b1` to `b2`, and a goods train with the same visits. Timing constraints ensure that the passenger train enters first while the goods train exits first.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement goodstrain {
  visit #g_in [b1]; visit #g_out [b2] }
timing p_in < g_in
timing g_out < p_out
```

- **Crossing**: trains traveling in *opposite directions* can visit this station simultaneously. This example is similar to the previous one, but the goods train now travels in the opposite direction, and the timing constraints require that the trains are inside the model simultaneously.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement goodstrain {
  visit #g_in [b2]; visit #g_out [b1] }
timing p_in < g_out
timing g_in < p_out
```

Similar specifications, and combinations of such specifications, are relevant in most railway construction projects. Since we typically only need to refer to locations such as model boundaries and loading/unloading locations, these specifications are not tied to a specific design, and can often be re-used even when the design of the station changes drastically.

III. TOOL CHAIN AND SOLVER ARCHITECTURE

We have investigated several logic-based approaches for the domain and problem described above. The PDDL+ language has been designed to express planning problems in mixed discrete/continuous domains. As each discrete change is represented by a planning step, our test case problem instances would need at least 50-100 steps to be solvable. We were only able to solve the most trivial test cases in less than one second using the SMTPlan+ solver.

Encoding into SMT can be done by expressing planning as BMC. This approach suffers from the same problem of having a high number of planning steps (some improvements can be made, s.a. making train driver choices implicit in constraints on the relation between velocity, distance and time).

In response to all these, we developed a CEGAR-style tool which exploits the limited number of control system commands to make an abstraction of the planning problem, see Fig. 4.

A verification tool chain which solves the low-level railway infrastructure capacity verification problem and supports agile

---

[1] For details of the input file formats, see https://luteberget.github.io/rollingdocs/usage.html

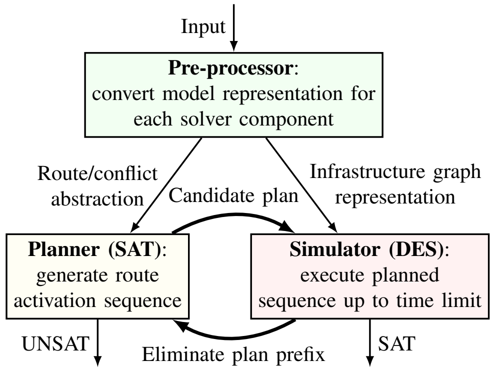


Fig. 4: Conceptual diagram of CEGAR architecture. Infrastructure, routes, train types, and movement specifications are transformed into (1) the planner's abstract representation, containing only elementary routes and train lengths, and (2) the detailed graph representation used in the simulator component.

verification in railway construction projects is outlined in Fig. 5. The manual, source code and test cases are available online[2]. The tool uses the MiniSAT v2.2.0 solver.

The tool is complementary to other verification techniques in railway design, such as static layout verification [23], [24], [25], static interlocking verification [26], [24], interlocking program verification [27], and timetable analysis [17].

The following input documents are used:

- **Operational scenarios** defining the performance properties to verify. Examples are given in Sec. II-B.
- **Infrastructure** given in the railML format [28], [29]. In our case studies we used the RailCOMPLETE software, a plugin for the widely used AutoCAD drafting software. Using a model taken directly from the drafting program means that no additional model preparation is needed.
- **Elementary routes** (*optional*), given in a custom format which is compatible with the upcoming railML interlocking format. Although subject to design, a decent guess of the content can be straight-forwardly derived from the infrastructure by listing resources in paths between adjacent signals, so this input is optional.
- **Dispatch plans** (*optional*) corresponding to each operational scenario. The verification tool can produce dispatch plans fulfilling the performance specification, so this input is optional.

An advantage of the separation of planner and simulator is that each component can be used separately. *The planner alone* may be used to enumerate different possibilities for train movements, which might be used in an operational testing situation. *The simulator alone* may be used to debug the execution of a specific dispatch plan to examine performance deficiencies, and educationally for demonstrating the workings of the railway system. Put together, the components provide automated verification, which is the main goal of our efforts.

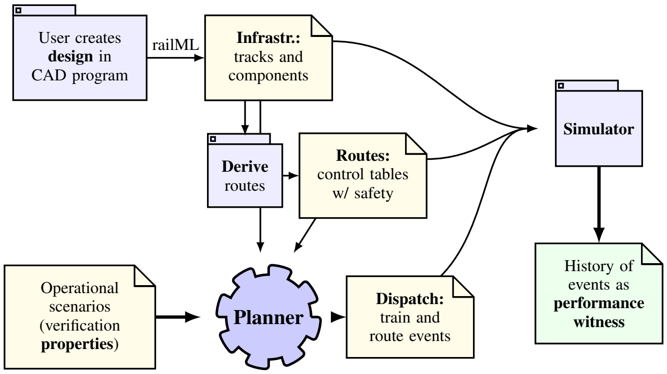


Fig. 5: Verification tool chain overview. Yellow boxes represent input documents. Note that only infrastructure and operational scenarios are strictly required – interlocking tables can be derived, and dispatch plans can be synthesized. Blue boxes represent programs. The green box represents the output document from the simulator, which is a history of events which is the witness that proves the performance requirement.

It would also, in principle, be possible to use one of the commercial simulation packages, such as OpenTrack or RailSys, provided that all input and simulation control can be given though a programmable interface (API).

A. Timing Evaluation using Simulation

Given a specific dispatch plan, we evaluate the time needed for executing it using discrete event simulation (DES), where a set of concurrent processes operate on a shared system state. Processes execute by reading or writing to the shared state, firing events, and then going to sleep until a specific event fires or a given amount of time has passed. When all processes are sleeping, the simulation timer is advanced to the earliest time when a process is scheduled to wake up.

Our DES for railway simulation has the following processes:

1) Elementary route activation (corr. Sec. II-A2): waits for resources, allocates them, sets switches to given positions and starts the following sub-processes:

- **Release trigger**: listens to a *trigger* detection section which is designated as the release trigger for a partial route. After the detection section has first been occupied, and later freed, resources are released for use in other elementary routes.
- **Signal catcher**: sets the route entry signal to the 'proceed' aspect, then waits for a given trigger section to become occupied before setting the signal to back 'stop'.

2) Train (corr. Sec. II-A3 and Sec. II-A4): evaluates movement authority using information from signals currently in sight, and takes one of the following actions: accelerate, brake, or coast/wait. Braking curves from velocity limitations are calculated, representing the train driver's plan for when to start braking. A guaranteed minimum time until further action is required from the driver is calculated by taking the minimum time until one of the following happen (see also Fig. 6):

---

[2] https://luteberget.github.io/rollingdocs and https://github.com/koengit/trainspotting

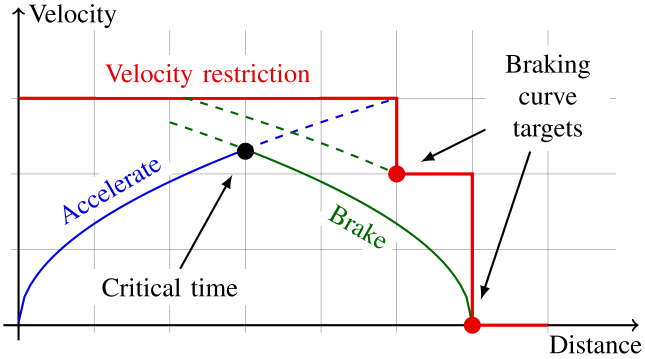


Fig. 6: The train driver's decision about when to accelerate/brake/coast happens at intersections between acceleration curves, braking curves and velocity restriction curves. In this example, the train can accelerate until the critical time where the acceleration intersects with the braking curve towards the second velocity restriction ahead (the first one is not critical).

- train arrives at a new node
- train reaches maximum velocity
- train enters the area of a new velocity restriction
- acceleration/coasting curve intersects braking curve

After this minimum time has passed, or any signals currently in sight have changed state, the train updates its position and velocity according to the chosen driver action and the laws of motion. Note that since we assume a constant maximum acceleration and braking, the equations of motion can be solved analytically, and there is no need for discretizing the time or space domains, except for the re-evaluation of the equations of motion at discrete events. This ensures that the train starts braking in time using only the information available to the driver at any given time.

B. Dispatch Planning using SAT

The planner solves the abstracted discrete planning problem of finding a dispatch plan, i.e. determining a sequence of trains and elementary routes which make the trains end up visiting locations according to the movements specification.

We encode an instance of the abstracted planning problem into an instance of the Boolean satisfiability problem (SAT). We consider the problem a model checking problem, and use the technique of bounded model checking (BMC) to unroll the transition relation of the system for a number of steps k, expressing state and transitions using propositional logic.

Using BMC for planning works by asserting the existence of a plan, so that when the corresponding SAT instance is satisfiable, it proves the fulfillment of the performance requirements and gives an example plan for it. When unsatisfiable, we are ensured that there is no plan within the number of steps k. In practice plans with higher number of steps are not of interest; i.e., the bound k is chosen based on practical considerations (twice the number of trains was sufficient in our case study). The SAT instance is built incrementally by solving with $k-1$ steps and then adding the k^{th} step if necessary.

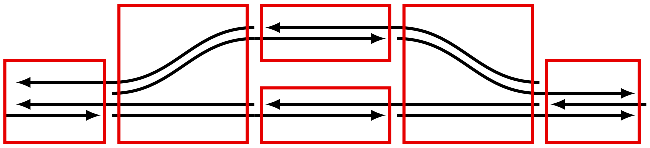


Fig. 7: The planner component takes an abstracted view of the railway infrastructure. Lines represent elementary routes with traveling direction given by the arrows. Boxes indicate routes in conflict, i.e. only one of them can be in use at a time.

The abstracted planning problem is encoded as a SAT instance by representing states, constraints on each state, and constraints on consecutive states. *State* i of the system in the planner component is represented as:

- Each route r_j has an **occupancy status** $o^i_{r_j}$: it can be free ($o^i_{r_j} = \text{Free}$) or it can be occupied by a specific train t_k ($o^j_{r_i} = t_k$). Each combination of route and train is represented by a Boolean variable, but we will write constraints with $o^i_{r_j}$ as a variable from the set of trains.
- Each train t_k has a Boolean representing **appearance status** b^i_k, used to propagate to future states that a train has started (used in constraint C2).
- Each visit l has a Boolean representing **required visits** v^i_l, which is used to propagate to future states that a visitation requirement has been fulfilled (used in constraint C5).
- Each combination of route r_j and train t_k has a Boolean representing **deferred progress** $p^i_{j,k}$, used to propagate to future states that a train is not progressing, and must resolve the conflict in the future (used in constraint C8).

A dispatch plan is produced directly from the occupancy status $o^i_{r_j}$ of states by taking the difference between consecutive states and then dispatching any trains and routes which become active from one state to the next.

Constraints on states ensure the following:

- The plan is viable for execution (i.e., correctness):
  (C1) Conflicting routes are not activated simultaneously.
  (C2) Each train can only take one continuous path.
  (C3) An elementary route must be allocated as a unit, but its parts may be deallocated separately.
  (C4) (Partial) routes are deallocated only after a train has fully passed over them.
- The plan fulfills performance specifications:
  (C5) Trains perform their specified visits.
  (C6) Visits happen in specified order.
- Equivalent solutions are eliminated (for performance):
  (C7) Routes are deallocated immediately after the train has fully passed over them.
  (C8) A train's path is extended as far as possible in the current time step, unless hindered by a conflicting train.

Equivalent plans, which result in the same trains traversing the same paths and conflicting in the same locations, should have the same representation so that enumeration of different plans produces meaningful alternatives. This equivalence is

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

demonstrated in the crossing example in Fig. 1, where the two plans shown are the *only* alternatives given by the planner.

The simulator component, which evaluates the time consumption of plans, reports which parts of the plan fail the timing constraints, and the negation of this partial plan is added to the SAT instance. Since the timing calculations are path dependent, we use the part of the plan starting from the beginning and going up to the step where the timing specification violation occurs. This way of refining the abstraction can cause performance problems when many different choices are possible early in the plan, and the timing violation can only be found near the end of the plan, as demonstrated in Sec. IV. Finding a way to make more precise refinements could be necessary for larger problem instances.

The implementation of each of these constraints as propositional logic statements is described below. Constraints apply separately to all states i unless noted otherwise.

1) Resource conflicts (C1): Any two routes which require the same resources cannot both be allocated in the same state.

$$\forall r_a \in \text{Routes} : \forall r_b \in \text{conflict}(r_a) : o^i_{r_a} = \text{Free} \lor o^i_{r_b} = \text{Free}.$$

2) Train path (C2): At most one alternative route is taken by a train in a single state. First, ensure that only one route from a given start signal may be taken at any time.

$$\begin{aligned}&\forall t \in \text{Trains} : \forall s \in \text{Signal} :\\ &\text{atMostOne}(\{o^i_r = t \mid \text{entry}(r) = s\})\end{aligned}$$

We use a standard sequential encoding to encode atMostOne and other similar constraints, as explained in e.g. [30]. Note that entry signals for all routes entering from a model boundary share the same null value, so that this constraint also excludes plans where a single train appears in several positions at once. Each train should only enter the plan once, thus the appearance Boolean changes to true in exactly one transition.

$$\forall t \in \text{Trains} : b^i_t \Rightarrow b^{i+1}_t.$$

$$\forall t \in \text{Trains} : \text{exactlyOne}\left(\left\{\neg b^j_t \land b^{j+1}_t \mid j \in \text{States}\right\}\right),$$

A train appears when an entry boundary route is allocated:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \{r \in \text{Routes} \mid \text{entry}(r) = \text{null}\} :\\ &\qquad\qquad\left(o^i_r \neq t \land o^{i+1}_r = t\right) \Rightarrow b^{i+1}_t.\end{aligned}$$

Routes which are not entry routes can only be allocated to a train when they extend some other route which was already allocated to the same train, i.e. consecutive routes must match so that the exit signal of one is the entry signal of the next:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \{r \in \text{Routes} \mid \text{entry}(r) \neq \text{null}\} :\\ &\left(o^i_r \neq t \land o^{i+1}_r = t\right) \Rightarrow\\ &\bigvee\left\{o^{i+1}_{r_x} = t \mid r_x \in \text{Routes}, \text{entry}(r) = \text{exit}(r_x)\right\}\end{aligned}$$

3) Partial release (C3): Partial release is represented by splitting each elementary route into separate routes for each component which is released separately. The set *Partial* contains such sets of routes. Partial routes are allocated together:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall q \in \text{Partial} :\\ &\text{allEqual}(\{o^i_r \neq t \land o^{i+1}_r = t \mid r \in q\})\end{aligned}$$

4) Deallocation (C4, C7): Routes are freed when sufficient length has been allocated ahead to fully contain the train.

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \text{Routes} :\\ &o^i_r = t \Rightarrow (o^{i+1}_r = t) = \text{freeable}_{r,t}(\{o^i\}),\end{aligned}$$

Note that the equality sign on the right hand side implies that deallocation is both allowed (C4), and required (C7). The freeable predicate is a disjunction of paths (conjunction of routes) ahead which are long enough to contain the train.

5) Visits (C5, C6): Visits and their order are given by the set VisitOrder, which contains pairs of (t, v), where t is a train and v is a set of alternative routes. Visits must happen using any of the alternative routes, and must be in an order such that the visit (t_1, v_1) comes before (t_2, v_2):

$$\begin{aligned}&\forall((t_1, v_1), (t_2, v_2)) \in \text{VisitOrder} :\\ &\bigvee\{o^i_{r_a} = t_1 \land o^j_{r_b} = t_2 \land i \leq j\\ &\qquad \mid r_a \in (v_1), r_b \in (v_2), i, j \in \text{States}\}\end{aligned}$$

6) Forced progress (C8): In addition to the constraints on allocation and freeing that are required to produce a valid plan, we also add constraints which force each train to get allocated routes further along a path forward unless there is a conflict. Routes ahead are either allocated, or the train is deferred p:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \text{Routes} :\\ &o^i_r \Rightarrow p^i_{t,r} \lor \bigvee\left\{o^i_{r_x} \mid r_x \in \text{Routes}, \text{entry}(r_x) = \text{exit}(r)\right\}\end{aligned}$$

Deferred progress must be resolved by freeing a conflicting route, and then allocating it to the train in the following step:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \text{Routes} :\\ &p^i_{t,r} \Rightarrow p^{i+1}_{t,r} \lor \bigvee\left\{o^i_{r_c} \neq \text{Free} \land o^i_{r_x} \neq t \land o^{i+1}_{r_x} = t\right.\\ &\left.\mid r_c, r_x \in \text{Routes}, \text{exit}(r) = \text{entry}(r_x), r_c \in \text{conflict}(r)\right\}\end{aligned}$$

When i is the last state, $p^{i+1}_{t,r}$ is considered to be false, which forces the deferred progress to be resolved eventually. Note that it is not required that the conflicting trains are distinct.

IV. CASE STUDIES AND PERFORMANCE

This section presents running times for different typical performance specifications on different types of railway infrastructure where the size and complexity of the model is typical for the scope of railway construction projects. Verification performance on various test examples as well as real stations is presented in Table I. The table shows the time spent in each solver component, and also shows the number of invocations n_{DES} of the simulator, which is very low in most of the practical cases. This supports our hypothesis that the chosen abstraction and CEGAR loop is efficient. The two-track station used in Fig. 1 is not too complex, having only 6 elementary routes. Even so, this scale is still interesting for verification in practice, since there are many possible mistakes to uncover.

The Norwegian railway infrastructure manager Bane NOR has supplied a railML infrastructure model of the whole national railway network [31] from which we have extracted some more complex examples. Fig. 8 shows cut-outs from the

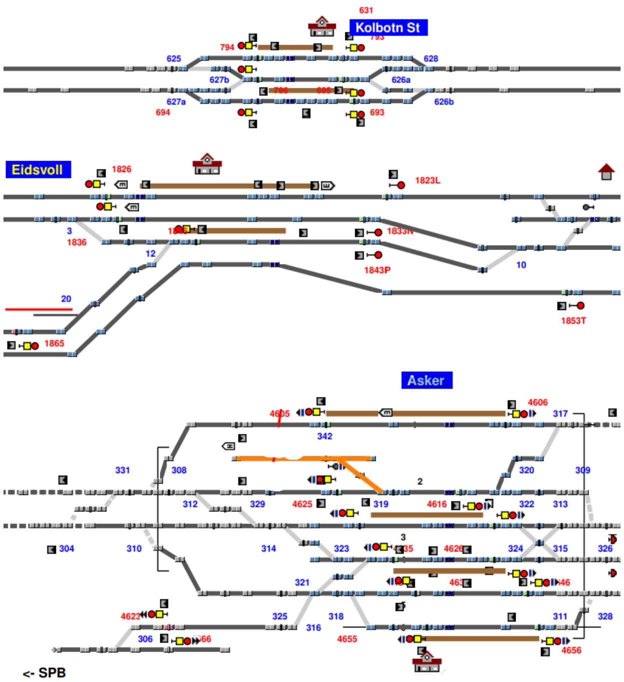


Fig. 8: Stations Kolbotn, Eidsvoll, and Asker from Bane NOR's model of the Norwegian national network [31].

Infrastructure	**Property**	**Result**	n_{DES}	t_{SAT}	t_{DES}	t_{total}
Simple (3 elem.)	Run.time	Sat.	1	0.00	0.00	0.00
	Crossing	Unsat.	0	0.00	0.00	0.00
Two track (14 elem.)	Run.time	Sat.	1	0.01	0.00	0.01
	Frequency	Sat.	1	0.01	0.00	0.01
	Overtaking 2	Sat.	1	0.00	0.00	0.01
	Overtaking 3	Unsat.	0	0.01	0.00	0.01
	Crossing 3	Unsat.	0	0.01	0.00	0.01
Kolbotn (BN) (56 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 4	Sat.	1	0.05	0.00	0.06
	Overtake 3	Unsat.	0	0.05	0.00	0.06
Eidsvoll (BN) (64 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 2	Sat.	1	0.08	0.00	0.08
	Crossing 3	Sat.	1	0.04	0.00	0.04
	Crossing 4	Unsat.	0	0.21	0.00	0.21
Asker (BN) (170 elem.)	Overtaking 2	Sat.	1	0.20	0.00	0.21
	Overtaking 3	Unsat.	1	0.73	0.00	0.74
	Crossing 4	Sat.	0	0.75	0.00	0.77
Arna (CAD) (258 elem.)	Run. time	Sat.	1	0.02	0.00	0.04
	Overtaking 2	Sat.	1	0.50	0.00	0.51
	Overtaking 3	Sat.	1	1.43	0.00	1.45
	Crossing 4	Sat.	1	1.73	0.00	1.74
Gen. 3x3 (74 elem.)	High time	Sat.	1	0.01	0.00	0.01
	Low time	Unsat.	27	0.18	0.01	0.19
Gen. 4x4 (196 elem.)	High time	Sat.	1	0.01	0.00	0.03
	Low time	Unsat.	256	2.08	0.26	2.34
Gen. 5x5 (437 elem.)	High time	Sat.	1	0.06	0.00	0.09
	Low time	Unsat.	3125	38.89	4.35	43.24

TABLE I: Verification performance on test cases, including Bane NOR (BN) and RailCOMPLETE (CAD) infrastructure models. The number of elementary routes (*elem.*) is shown for each infrastructure to indicate the model's size. n_{DES} is the number simulator runs, t_{SAT} the time in seconds spent in SAT solver, t_{DES} the time in seconds spent in DES, and t_{total} the total calculation time in seconds.

visual representation of these models, i.e., the stations Kolbotn, Eidsvoll, and Asker were converted from the railML models.

We have also tested against an infrastructure model from the Arna construction project that uses the RailCOMPLETE CAD design software, a realistic use case for agile verification.

Finally, to test the limitations of scalability in our method, we construct a set of examples where m stations each with n parallel tracks each are serially connected by a single track. In this case, when a timing bound is slightly too small to be satisfiable, the planner will have to come up with n^m plans for timing evaluation. This scenario is outside the intended use case for our method: path selection can on this scale instead be based on static speed profiles. Capacity over many stations is better suited for the established timetabling tooling.

We attempted an alternative implementation using the PDDL+ solver SMTPlan+, but found that even for greatly simplified models, the required number of steps and numerical constraints put all our case studies out of reach for sub-second verification times.

V. RELATED WORK

Railway timetabling and capacity analysis has often been posed as a planning problem and solved using mixed integer programming and similar approaches. Zwaneveld et al. [32] use integer programming on a problem closely related to our low-level railway infrastructure capacity verification problem. Isobe et al. [33] formulate a similar model in timed CSP, representing train locations, velocities, and control logic. Our definition of the problem in this paper includes non-linear constraints on train dynamics (acceleration/braking power) and communication constraints (trains must slow down if they have not been informed of movement authority), which are relevant in construction projects but less relevant in timetabling.

Many variations on discrete event simulation are used in railway dynamic analysis, see e.g. [34], [35], [36].

In the planning literature, the PDDL+ language [4] has been introduced to capture mixed discrete/continuous planning problems such as the one studied in this paper. General-purpose solvers have recently been developed, using time domain discretization (DiNo [37]) or the SMT theory of non-linear real arithmetic (SMTPlan+ [38]).

VI. CONCLUSIONS AND FURTHER WORK

The goal of our suggested tool chain for railway engineering is (1) to allow fully automated performance verification and (2) use minimal input documentation for the verification. Both of these aspects encourage bringing in performance verification into frequently changing early-stage design projects, avoiding the costly and time-consuming backtracking required when later-stage analysis reveals unacceptable performance.

As future work we plan to integrate the current prototype in the RailCOMPLETE tool and test the usability with the engineers using this tool in their design work.

Acknowledgments: We thank the engineers at Railcomplete AS, especially senior engineer Claus Feyling, for guidance on railway operations and design methodology.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

REFERENCES

[1] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_11

[2] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[3] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006. [Online]. Available: http://doi.acm.org/10.1145/1217856.1217859

[4] M. Fox and D. Long, "Modelling mixed discrete-continuous domains for planning," *J. Artif. Intell. Res.*, vol. 27, pp. 235–297, 2006. [Online]. Available: https://doi.org/10.1613/jair.2044

[5] M. Franzle, 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. [Online]. Available: https://satassociation.org/jsat/index.php/jsat/article/view/16

[6] D. Jovanovic and L. de Moura, "Solving non-linear arithmetic," *ACM Comm. Computer Algebra*, vol. 46, no. 3/4, pp. 104–105, 2012. [Online]. Available: http://doi.acm.org/10.1145/2429135.2429155

[7] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[8] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction*, ser. Lecture Notes in Computer Science, M. P. Bonacina, Ed., vol. 7898. Springer, 2013, pp. 208–214. [Online]. Available: https://doi.org/10.1007/978-3-642-38574-2_14

[9] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th Conf., CAV 2014*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[10] S. Robinson, *Simulation: The Practice of Model Development and Use*. USA: John Wiley & Sons, Inc., 2004.

[11] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu *et al.*, "Bounded model checking." *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003. [Online]. Available: https://doi.org/10.1016/S0065-2458(03)58003-2

[12] M. Abril, F. Barber, L. Ingolotti, M. Salido, P. Tormos, and A. Lova, "An assessment of railway capacity," *Transportation Research Part E: Logistics and Transportation Review*, vol. 44, no. 5, pp. 774 – 806, 2008. [Online]. Available: https://doi.org/10.1016/j.tre.2007.04.001

[13] A. Landex, "Methods to estimate railway capacity and passenger delays," Ph.D. dissertation, 2008. [Online]. Available: http://orbit.dtu.dk/en/publications/id(f5578206-74c3-4c94-ba0d-43f7da82bf95).html

[14] "RMCon RailSys," 2018. [Online]. Available: http://www.rmcon.de/railsys-en/

[15] "OpenTrack: Simulation of railway networks," 2018. [Online]. Available: http://www.opentrack.ch/

[16] "LUKS: Analysis of lines and junctions," 2018. [Online]. Available: http://www.via-con.de/development/luks

[17] S. S. Harrod, "A tutorial on fundamental model structures for railway timetable optimization," *Surveys in Operations Research and Management Science*, vol. 17, no. 2, pp. 85 – 96, 2012. [Online]. Available: https://doi.org/10.1016/j.sorms.2012.08.002

[18] A. A. Assad, "Models for rail transportation," *Transportation Research Part A: General*, vol. 14, no. 3, pp. 205 – 220, 1980. [Online]. Available: https://doi.org/10.1016/0191-2607(80)90017-5

[19] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. New York, NY, USA: Wiley-Interscience, 1975.

[20] T. Harris and F. Ross, "Fundamentals of a method for evaluating rail net capacities," Rand Corp., Tech. Rep., 1955.

[21] R. M. Goverde, "Railway timetable stability analysis using max-plus system theory," *Transportation Research Part B: Methodological*, vol. 41, no. 2, pp. 179 – 201, 2007, advanced Modelling of Train Operations in Stations and Networks. [Online]. Available: https://doi.org/10.1016/j.trb.2006.02.003

[22] J. Pachl, *Railway Operation and Control*. VTD Rail Publishing, 2015.

[23] B. Luteberget, C. Johansen, and M. Steffen, "Rule-based consistency checking of railway infrastructure designs," in *Integrated Formal Methods 2016*, ser. Lecture Notes in Computer Science, E. Ábrahám and M. Huisman, Eds., vol. 9681. Springer, 2016, pp. 491–507. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-33693-0_31

[24] B. Luteberget and C. Johansen, "Efficient verification of railway infrastructure designs against standard regulations," *Formal Methods in System Design*, vol. 52, no. 1, pp. 1–32, Feb 2018. [Online]. Available: https://doi.org/10.1007/s10703-017-0281-z

[25] B. Luteberget, J. J. Camilleri, C. Johansen, and G. Schneider, "Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL," in *International Conference on Software Engineering and Formal Methods (SEFM)*, ser. Lecture Notes in Computer Science, A. Cimatti and M. Sirjani, Eds., vol. 10469. Springer International Publishing, 2017, pp. 87–103. [Online]. Available: https://doi.org/10.1007/978-3-319-66197-1_6

[26] A. E. Haxthausen and P. H. Østergaard, "On the Use of Static Checking in the Verification of Interlocking Systems," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, T. Margaria and B. Steffen, Eds. Springer, 2016, pp. 266–278. [Online]. Available: https://doi.org/10.1007/978-3-319-47169-3_19

[27] A. Borälv and G. Stålmarck, "Formal verification in railways," in *Industrial-Strength Formal Methods in Practice*, M. G. Hinchey and J. P. Bowen, Eds. Springer, 1999, pp. 329–350. [Online]. Available: https://doi.org/10.1007/978-1-4471-0523-7_15

[28] A. Nash, D. Huerlimann, J. Schütte, and V. P. Krauss, "RailML — a standard data interface for railroad applications," in *Computers in Railways IX*. WIT Press, 2004, pp. 233–240.

[29] "railML. The XML interface for railway applications," 2018. [Online]. Available: http://www.railml.org

[30] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," in *Principles and Practice of Constraint Programming - CP 2005*, ser. Lecture Notes in Computer Science, P. van Beek, Ed., vol. 3709. Springer, 2005, pp. 827–831. [Online]. Available: https://doi.org/10.1007/11564751_73

[31] "Bane NOR: Model of the Norwegian rail network," 2016. [Online]. Available: http://www.banenor.no/en/startpage1/Market1/Model-of-the-national-rail-network/

[32] P. J. Zwaneveld, L. G. Kroon, and S. P. van Hoesel, "Routing trains through a railway station based on a node packing model," *European Journal of Operational Research*, vol. 128, no. 1, pp. 14 – 33, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221700000874

[33] Y. Isobe, F. Moller, H. N. Nguyen, and M. Roggenbach, "Safety and line capacity in railways – an approach in timed csp," in *Integrated Formal Methods*, J. Derrick, S. Gnesi, D. Latella, and H. Treharne, Eds. Springer, 2012, pp. 54–68. [Online]. Available: https://doi.org/10.1007/978-3-642-30729-4_5

[34] M. Montigel, "Modellierung und gewährleistung von abhängigkeiten in eisenbahnsicherungsanlagen," Ph.D. dissertation, ETH Zurich, 1994. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/47983

[35] D. Hürlimann, "Objektorientierte modellierung von infrastrukturelementen und betriebsvorgängen im eisenbahnwesen," Ph.D. dissertation, ETH Zurich, 2002. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/47957

[36] E. Kamburjan and R. Hähnle, "Uniform modeling of railway operations," in *Formal Techniques for Safety-Critical Systems FTSCS 2016*, ser. Communications in Computer and Information Science, vol. 694. Springer, 2016, pp. 55–71. [Online]. Available: https://doi.org/10.1007/978-3-319-53946-1_4

[37] W. M. Piotrowski, M. Fox, D. Long, D. Magazzeni, and F. Mercorio, "Heuristic planning for PDDL+ domains," in *International Joint Conference on Artificial Intelligence, IJCAI 2016*, S. Kambhampati, Ed. IJCAI/AAAI Press, 2016, pp. 3213–3219. [Online]. Available: http://www.ijcai.org/Abstract/16/455

[38] M. Cashmore, M. Fox, D. Long, and D. Magazzeni, "A compilation of the full PDDL+ language into SMT," in *International Conference on Automated Planning and Scheduling, ICAPS 2016*, A. J. Coles, A. Coles, S. Edelkamp, D. Magazzeni, and S. Sanner, Eds. AAAI Press, 2016, pp. 79–87. [Online]. Available: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13101

ISBN: 978-0-9835678-8-2.

Complete and Efficient DRAT Proof Checking

Adrián Rebola-Pardo
TU Wien
`arebolap@forsyte.com`

Luís Cruz-Filipe
University of Southern Denmark
`lcf@imada.sdu.dk`

***Abstract*—DRAT proofs have become the standard for verifying unsatisfiability proofs emitted by modern SAT solvers. However, recent work showed that the specification of the format differs from its implementation in existing tools due to optimizations necessary for efficiency. Although such differences do not compromise soundness of DRAT checkers, the sets of correct proofs according to the specification and to the implementation are incomparable. We discuss how it is possible to design DRAT checkers faithful to the specification by carefully modifying the standard optimization techniques. We implemented such modifications in a configurable DRAT checker. Our experimental results show negligible overhead due to these modifications, suggesting that efficient verification of the DRAT specification is possible. Furthermore, we show that the differences between specification and implementation of DRAT often arise in practice.**

I. INTRODUCTION

Recent years have seen SAT solvers become increasingly popular, with many sucess stories in their application to several open problems, e.g. the recent computation of the Schur number five [11]. Popularity has also brought about the question of reliability: how much can we trust an answer provided by a SAT solver? A satisfiability result can be easily checked, since SAT solvers output a satisfying assignment. In the case of unsatisfiability results, several formats have been developed aimed at representing proofs of unsatisfiability in a way that is both compact and efficient to check. In this paper we focus on the DRAT format [10], [16], which has been widely adopted in SAT competitions and can represent most inferences done by SAT solvers. DRAT proofs can be checked both by efficient, untrusted programs such as `DRAT-trim`, and by certified, slower programs that work on extended formats such as LRAT [2] and GRAT [12].

A mismatch between the definition of DRAT proofs and the results of state-of-the-art proof checkers has been recently exposed [15]. The class of correct DRAT proofs and that of proofs accepted by modern checkers are incomparable: simple proofs which are correct but rejected, or incorrect but accepted, exist. This is not as catastrophic as it may sound, since it can be shown that whenever checkers accept a DRAT refutation of a formula, the latter is indeed unsatisfiable. Hence, one may consider state-of-the-art checkers as *implicitly* defining a proof system of their own. These two notions of correct DRAT refutations have been refered to as *flavors*: the original definition of a DRAT proof corresponds to the *specified* flavor, whereas the one defined by the results of DRAT checkers is the *operational* flavor. The fundamental difference between them is that in the operational flavor specific clause deletion instructions, called *unit deletions*, are ignored.

While this issue attracted some interest within the SAT solving community, a discussion on the convenience of either flavor is hindered by the absence of specified-DRAT checkers. The reason for this unavailability lies deep down at the heart of how DRAT checkers work. Deleting unit clauses breaks invariants required by some lazy data structures for unit propagation, which are necessary for the huge efficiency of checkers. Without specified-DRAT checkers, it is virtually impossible to assess how often discrepancies between the two flavors occur in proofs produced by SAT solvers in practice.

In this paper, we explain how an efficient specified-DRAT checker can be implemented. By carefully repairing the involved data structures, the invariants necessary for effective unit propagation can be restored. Extensively applying these repairs would be extremely expensive; we identify restrictions that greatly curb the induced overhead. To measure the reparation overhead in specified-DRAT checking, we implemented our method in a configurable checker, which can be run to check proofs on either flavor. To the best of our knowledge, this is the first specified-DRAT checker available. Experimental data suggests that the overhead of checking specified-DRAT proofs over checking operational-DRAT proofs is negligible. Furthermore, we find that discrepancies between both flavors occur relatively often in practice, and are not just an artifact of carefully handcrafted proofs.

Related work: There is extensive literature on clausal proof generation and checking for SAT solvers [5], [6], [8], [10], [16]. Several methods to validate correctness results of DRAT checkers through certified means have been proposed [2], [7], [12], although none of them covers incorrectness results. The incompleteness of state-of-the-art DRAT checkers and its relation with unit clause deletion has been observed and acknowledged [4], [10], [15].

II. PRELIMINARIES

Given a variable x, we denote its *complement* by $\overline{x}$. A *literal* is a variable or its complement. A *clause* is a disjunction of literals; we denote clauses by juxtaposition, i.e. $x \vee y \vee \overline{z}$ is denoted by $xy\overline{z}$. We assume that clauses do not contain complementary literals. The *unsatisfiable* or *empty* clause is denoted by $\square$. A *CNF formula* is a conjunction of clauses. We follow the usual definitions of *satisfiability* and *entailment*. We construe CNF formulas as clause sets and clauses as literal sets. For a clause C, we denote by $\overline{C}$ the set of clauses

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

containing the size-one clause $\bar{l}$ for each literal $l \in C$. A *partial assignment* is a finite, complement-free set of literals I. For any literal l, we define $I(l)$ as follows: $I(l) = 1$ if $l \in I$; $I(l) = 0$ if $\bar{l} \in I$; and $I(l) = ?$ otherwise.

A clause C is called *unit* w.r.t. a partial assignment I whenever there is a literal $l \in C$ with $I(l) = 1$, and for any other literal $k \in C \setminus \{l\}$ we have $I(k) = 0$. We say that a CNF formula F *implies* a literal l *by unit propagation* whenever there is a finite sequence $l_1, \dots, l_n$ of literals such that $l_n = l$, and we can find a clause $C_i \in F$ with $l_i \in C_i$ and $C_i \setminus \{l_i\} \subseteq \{\overline{l_1}, \dots, \overline{l_{i-1}}\}$ for $1 \leq i \leq n$. Furthermore, we say that F *implies a conflict by unit propagation* whenever there are two complementary literals l and $\bar{l}$ implied by unit propagation over F. A clause C is a *reverse unit propagation* (RUP) clause in F whenever $F \cup \overline{C}$ implies a conflict by unit propagation. Moreover, C is called a *resolution asymmetric tautology* (RAT) in F upon a literal $l \in C$ whenever the clause $C \vee (D \setminus \{\bar{l}\})$ is a RUP in F, for all clauses $D \in F$ with $\bar{l} \in D$. We assume that clauses contain at least two literals. In practice, the empty clause is never introduced in the data structures, but size-one clauses are. For simplicity, we assume that a new literal $\top$ is made true by all partial assignments. Then, we replace size-one clauses l by the size-two clause $l\,\overline{\top}$.

Modern SAT solvers are able to generate unsatisfiability certificates called *DRAT proofs*. A DRAT proof is a string of instructions $i_1, \dots, i_n$; every instruction is either a *clause introduction* $\mathbf{i}\colon C$ or a *clause deletion* $\mathbf{d}\colon C$, for a clause C. Given a DRAT proof π and a CNF formula F, the *accumulated formula* $F[\pi]$ by F through π is recursively defined as follows:

$$\begin{aligned} F[\epsilon] &= F \\ F[\mathbf{i}\colon C,\ \pi] &= (F \cup \{C\})[\pi] \\ F[\mathbf{d}\colon C,\ \pi] &= (F \setminus \{C\})[\pi] \end{aligned}$$

The set of literals implied by unit propagation from the formula accumulated by F through π is called the *accumulated partial assignment*. In [15], the accumulated partial assignment was characterized as the minimal UP-model of $F[\pi]$.

Given a CNF formula F, a DRAT proof $i_1, \dots, i_n$ is called a *correct* DRAT proof of F if $\square = i_m$ for some $1 \leq m \leq n$, and for every $1 \leq j \leq n$ either of the following holds:

- i_j is a deletion instruction $\mathbf{d}\colon C$.
- i_j is an introduction instruction $\mathbf{i}\colon C$, and C is a either a RUP or a RAT in $F[i_1, \dots, i_{j-1}]$.

Example 1. Throughout this paper we use the following running example. We consider a CNF formula F containing the following clauses:

$$\begin{array}{cccc} x_1 & x_5x_6 & \overline{x_3}\,\overline{x_6}x_8 & \overline{x_4}\,\overline{x_9}x_{10} \\ \overline{x_1}x_2 & \overline{x_2}\,\overline{x_5}x_7 & \overline{x_6}\,\overline{x_4}x_3 & \overline{x_{10}}x_9 \\ \overline{x_1}\,\overline{x_2}x_3 & \overline{x_1}\,\overline{x_5}x_6 & \overline{x_8}x_5 & \overline{x_9}x_7 \\ \overline{x_1}\,\overline{x_3}x_4 & \overline{x_5}\,\overline{x_6}x_4 & \overline{x_3}x_9x_{10} & \overline{x_7}\,\overline{x_8}\,\overline{x_9}\,\overline{x_{10}} \end{array}$$

Furthermore, we consider the following two DRAT proofs:

$$\pi = \mathbf{i}\colon x_5,\ \mathbf{d}\colon \overline{x_1}x_2,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square \qquad \pi' = \mathbf{i}\colon x_5,\ \mathbf{i}\colon x_9,\ \mathbf{i}\colon \square$$

Both π and π' are correct DRAT proofs. Let us check that the instruction $\mathbf{i}\colon x_9$ in π is correct. The accumulated formula at that point is $F' = (F \setminus \{\overline{x_1}x_2\}) \cup \{x_5\}$. $F' \cup \{\overline{x_9}\}$ implies both x_9 and $\overline{x_9}$ by unit propagation, so x_9 is a RUP in F'.

The proofs π and π' do not contain any RAT introduction instruction. As an example, clause $\overline{x_5}$ is not a RUP in F, but it is a RAT in F. The formula $F \cup \{x_5\}$ implies by unit propagation exactly the literals $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$, so $\overline{x_5}$ is not a RUP in F. To show that it is a RAT in F upon $\overline{x_5}$, we check that $\overline{x_5}x_6 = \overline{x_5} \vee (x_5x_6 \setminus \{x_5\})$ and $\overline{x_5}x_8 = \overline{x_5} \vee (x_5x_8 \setminus \{x_5\})$ are RUPs in F. This holds, for $F \cup \{x_5\overline{x_8}\}$ (resp. $F \cup \{x_5\overline{x_6}\}$) implies by unit propagation x_8 and $\overline{x_8}$ (resp. x_6 and $\overline{x_6}$). ■

Our definition of a DRAT proof, reflecting the original from [9], [10], is central to this paper. *DRAT checkers* are programs that determine whether a DRAT proof is correct or not. DRAT checking is computationally challenging, due to the sheer size of proofs and the need for unit propagation to check introduction instructions. Several DRAT checkers are available. `DRAT-trim`[1] is the *de facto* standard checker, and is used in SAT Competitions to certify unsatisfiability results [1], [10]. Some data structure improvements have been shown to induce notable improvements over `DRAT-trim` [12].

However, recent work exposed critical differences between the way DRAT proofs are defined and the way DRAT proofs are checked [15]. DRAT checkers ignore deletion instructions removing clauses that are unit w.r.t. the accumulated assignment. Hence, whereas the notion of correctness stays the same, DRAT checkers compute the accumulated formula differently: $F[\mathbf{d}\colon C,\ \pi]$ is defined as $F[\pi]$ if C is a unit clause w.r.t. the acumulated assignment for F; and $(F \setminus \{C\})[\pi]$ as usual otherwise. Proofs that are correct but rejected by DRAT checkers exist, and vice versa. We refer to the original definition as the *specified* flavor of DRAT, whereas the *operational* flavor uses the modified definition for accumulated formula.

A. Data structures for DRAT checking

Modern DRAT checkers are relatively complex programs. Efficient unit propagation is required to check the correctness of RUP and RAT introductions. This is achieved through the same *two-watched literal* schema CDCL SAT solvers are based upon, where each clause is *watched* on two distinct literals, and the clauses watched on literal l are stored in the *watchlist* for l [13]. Also as in SAT solvers, a *trace* of the assigned literals is kept as a stack. The trace stores the accumulated assignment (i.e. the literals implied by unit propagation by the accumulated formula), together with information about the order on which they were assigned and the reason clause that triggered that propagation. Moreover, watchlists keep track of clauses that are candidate to trigger future unit propagations. Both data structures maintain invariants throughout the execution of the DRAT checker, which are required so that all available unit propagations are appropriately detected.

---

[1] https://github.com/marijnheule/drat-trim

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

At a given stage during checking, the j-th instruction is considered. The trace then contains the accumulated assignment I_j for the accumulated formula F_j. Remarkably, literals in the trace occur in the same order as they were assigned. In fact, they are *staged*: the trace behaves like a stack that grows monotonically throughout the proof, so it can be divided in sections such that the first j' sections correspond to the accumulated assignment $I_{j'}$. Furthermore, every clause is watched in such a way that that the following invariant holds:

Invariant 1. *If a clause is watched on literals l and k, and the current trace I_j falsifies l, then I_j satisfies k.*

A DRAT checker can decide whether a CNF formula together with some *assumed literals* implies a conflict by unit propagation using a well-known procedure [13]. After assigning each assumed literal l, the watchlist for $\bar{l}$ is traversed. By Invariant 1, clauses that trigger new propagations must be watched on $\bar{l}$, so they are all eventually encountered. The checker tries to relocate the watches in each clause so that Invariant 1 is satisfied. Two conditions may prevent this. In one case, the trace falsifies all literals, hence a conflict is reported. In the other case, all literals are falsified but for one unassigned literal k. In this case, k is implied by unit propagation, so it can be assigned to true. In turn, this triggers new propagations, which are detected when the watchlist for $\bar{k}$ is traversed. If no further watchlists for previously assigned literals remain to be processed, and a conflict has not been reached, the checker can conclude there is no conflict by unit propagation. Preparing the data structures to check if a new set of assumed literals implies a conflict by unit propagation only requires to unassign the literals in the trace: any watch choice satisfies Invariant 1 correct afterwards.

B. Double-sweep DRAT checking

The described procedure can already check DRAT proofs: to check if C is a RUP in F, it suffices to assume $\bar{C}$ and perform unit propagation, and RAT checking can be done via several RUP checks. There is however much room for improvement. DRAT checkers implement a number of techniques to speed checking up, e.g. resolution candidate caching [12] and core-first propagation [8]. Two techniques are especially relevant to our work: an undocumented technique we call *incremental prepropagation*, and *backwards checking* [8]. DRAT checkers perform two sweeps through the proof. In the first sweep, incremental prepropagation traverses the proof forwards, caching propagation information that will be used in the second sweep. Incremental prepropagation performs no proper checking. Instead, the second sweep called backwards checking performs RUP or RAT checks for introduction instructions, traversing the proof backwards. Backwards checking allows to skip irrelevant parts of the proof by performing conflict analysis.

Incremental prepropagation: The description of the unit propagation algorithm above implicitly assumes that the trace starts empty. This is unnecessary: as long as the watches satisfy Invariant 1, the initial trace may contain literals. Invariant 1 also implies that the trace contains all literals implied by unit propagation. DRAT checkers exploit this by preserving the anterior part of the trace stack between instructions during the first sweep, in such a way that the trace grows monotonically.

Incremental prepropagation traverses the CNF instance and the DRAT proof forwards. Every premise or introduction instruction adds a clause C to the clause database; deletion instructions are discussed later in this section. After a clause is introduced, the trace and watchlists are updated. New literals implied by unit propagation are *incrementally* added to the trace stack. Hence, the trace has the form $I_0 I_1 \ldots I_m$, and the substack $I_0 \ldots I_j$ is the accumulated assignment after the j-th instruction. The data structures can be updated in three ways:

- If watches for C respecting Invariant 1 exist, no further literals are propagated. C is added to the relevant watchlists, and the checker moves on to the next instruction.
- If C is falsified by the trace, then C is a RUP in F, and moreover $\Box$ is a RUP in $F \cup \{C\}$. This can be treated as the end of the proof, and backwards checking starts.
- Otherwise, C only contains falsified literals except for one unassigned literal l. In this case, C is watched in l and in some other literal, and l follows by unit propagation. Hence, l is pushed into the trace stack, and the propagation procedure is called to derive new literals.

As observed above, the stack structure of the trace is monotonic with respect to the proof: to recover the trace computed before introducing C, if C was the reason to propagate l, it suffices to drop the latter part of the stack starting with l. When doing so, watches need not be modified, although this is not so obvious; again, we defer this discussion to Section III-C, when we will have the tools to explain the reason for this.

Example 2. Let us reconsider the proofs from Example 1:

$$\pi = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{d}\colon \overline{x_1} x_2,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \Box \qquad \pi' = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \Box$$

where we have introduced the literal $\overline{\top}$ to prevent size-one clauses. Figure 1 shows the evolution of the trace throughout incremental prepropagation. Observe that the trace evolution for π is non-monotonic, since some literals are removed from the trace, whereas the one for π' is monotonic. The reason for this difference is the deletion of reason clause $\overline{x_1} x_2$ in π. State-of-the-art checkers would ignore this deletion instruction in π because $\overline{x_1} x_2$ is a unit clause w.r.t. the trace before the deletion, thus *implicitly* checking proof π'. Therefore, checking π and π' is equivalent under the operational flavor. Observe that the procedure described above to restore previous traces works well in all cases except for recovering the trace "after $\mathbf{i}\colon \overline{\top} x_5$" from "$\mathbf{d}\colon \overline{x_1} x_2$" in π. As we will see later, this is the reason why unit clause deletions are ignored. ■

Backwards checking: Once a conflict in the accumulated assignment is reached, the second sweep starts. Backwards checking traverses the proof from the conflict point towards the beginning of the proof. Introduction instructions are checked for RUP or RAT by restoring the trace to its state before that instruction during incremental inprocessing. RUP checks for a clause C are performed by assumming $\overline{C}$ and propagating; RAT checks can be reduced to a number of RUP checks.

trace preprocessing for $\pi = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{d}\colon \overline{x_1} x_2,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \square$							
start		after $\mathbf{i}\colon \overline{\top} x_5$		after $\mathbf{d}\colon \overline{x_1} x_2$		after $\mathbf{i}\colon \overline{\top} x_9$	
x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$
x_2:	$\overline{x_1} x_2$	x_2:	$\overline{x_1} x_2$	x_5:	$\overline{\top} x_5$	x_5:	$\overline{\top} x_5$
x_3:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$	x_6:	$\overline{x_1 x_5} x_6$	x_6:	$\overline{x_1 x_5} x_6$
x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_5 x_6} x_4$	x_4:	$\overline{x_5 x_6} x_4$
		x_5:	$\overline{\top} x_5$	x_3:	$\overline{x_6 x_4} x_3$	x_3:	$\overline{x_6 x_4} x_3$
		x_6:	$\overline{x_1 x_5} x_6$	x_8:	$\overline{x_3 x_6} x_8$	x_8:	$\overline{x_3 x_6} x_8$
		x_7:	$\overline{x_2 x_5} x_7$			x_9:	$\overline{\top} \overline{x_9}$
		x_8:	$\overline{x_3 x_6} x_8$			x_7:	$\overline{x_9} x_7$
						x_{10}:	$\overline{x_4 x_9} x_{10}$
						$\overline{x_{10}}$:	$\overline{x_7 x_8 x_9 x_{10}}$

trace preprocessing for $\pi' = \mathbf{i}\colon \overline{\top} x_5,\ \mathbf{i}\colon \overline{\top} x_9,\ \mathbf{i}\colon \square$					
start		after $\mathbf{i}\colon \overline{\top} x_5$		after $\mathbf{i}\colon \overline{\top} x_9$	
x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$	x_1:	$\overline{\top} x_1$
x_2:	$\overline{x_1} x_2$	x_2:	$\overline{x_1} x_2$	x_2:	$\overline{x_1} x_2$
x_3:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$	x_3:	$\overline{x_1 x_2} x_3$
x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$	x_4:	$\overline{x_1 x_3} x_4$
		x_5:	$\overline{\top} x_5$	x_5:	$\overline{\top} x_5$
		x_6:	$\overline{x_1 x_5} x_6$	x_6:	$\overline{x_1 x_5} x_6$
		x_7:	$\overline{x_2 x_5} x_7$	x_7:	$\overline{x_2 x_5} x_7$
		x_8:	$\overline{x_3 x_6} x_8$	x_8:	$\overline{x_3 x_6} x_8$
				x_9:	$\overline{\top} x_9$
				x_{10}:	$\overline{x_4 x_9} x_{10}$
				$x_{\overline{10}}$:	$\overline{x_7 x_8 x_9 x_{10}}$

Fig. 1. Trace evolution throughout incremental prepropagation for proofs π and π' from Example 1. Reason clauses for each propagated literal are indicated.

Done naïvely, restoring the trace would mean storing the trace for each instruction in the proof, and then retrieving the appropriate trace for every instruction. Watches would then need to be relocated too, incurring in large costs. Fortunately, as explained above, the checker can restore a previous trace can be recovered by simply removing the latter part of the trace stack. Also, this makes watch relocation unnecessary.

This does not justify checking the proof backwards: the same effect can be obtained by checking introductions during the first sweep. However, by performing conflict analysis on each conflict similarly to CDCL [13], the checker can determine which clauses were involved in the conflict. These clauses get *marked*; unmarked clauses are skipped during backwards checking, since they are unnecessary to derive $\square$.

Ignoring unit clause deletions: We had let aside the issue of deletion instructions in incremental prepropagation. Clauses that were not involved in trace propagation can be safely removed from the clause database and watchlists. Otherwise, C triggered the propagation of a literal l in the trace; we refer to C as a *reason clause* for l. Removing a reason clauses is cumbersome. For one, the propagated literal l may be used to propagate later literals in the trace. For another, l (or any of the subsequently propagated literals) may *still* be implied by unit propagation, just through a different propagation sequence.

The solution adopted by state-of-the-art checkers is rather pragmatic: ignore such deletions. If the checker only ignored reason clauses, the results would be unpredictable, for reason clauses depend on arbitrarities like the order of clauses in the formula or the order of literals within clauses. Instead, a more semantic criterion is used: a deletion instruction for C is ignored whenever C is a unit w.r.t. the accumulated assignment, which is stored in the trace. This is a necessary condition for being a reason clause, albeit not a sufficient one.

Example 3. Consider the instruction $\mathbf{d}\colon \overline{x_1} x_2$ in proof π in our running example. At this point, the trace is storing the accumulated assignment $\{x_1, x_2, x_3, x_4, x_5, x_7, x_6, x_8\}$, and the clause $\overline{x_1} x_2$ is a unit w.r.t. this assignment. Therefore this deletion instruction is simply ignored by DRAT checkers. ■

This criterion makes the results of DRAT checkers stable, i.e. equivalent representations of proofs yield the same correctness result. However, ignoring unit clause deletions changes the class of accepted proofs: DRAT checkers are checking *something else* instead. The implicitly defined proof system is sound, i.e. it can only prove unsatisfiable formulas. However, its class of correct proofs is incomparable to that of correct DRAT proofs. The implicit proof system has been formalized and named *operational-DRAT*, in contrast to the originally defined *specified-DRAT* proof system. A comparison between the two flavors and a discussion on the need for specified-DRAT checkers can be found in [15].

III. (NAÏVELY) CHECKING SPECIFIED-DRAT PROOFS

Due to the problems discussed in Section II-B, no DRAT checkers for the specified flavor are available: the invariants broken by unit clause deletion are precisely those that make DRAT checking efficient. In this section, we describe how to restore broken invariants after unit clause deletion. The operations described in this section are expensive, but the optimizations in Section IV vastly curb this overhead.

Our first goal is to construct the trace after a reason clause deletion during incremental propagation, such as the trace "after $\mathbf{d}\colon \overline{x_1} x_2$" in Example 2. A very inefficient way to do that would be simply to discard the trace and the watches and reconstruct them from scratch. We aim to improve over this by reusing the trace before the deletion as much as possible.

We construct the trace after deleting the reason clause C for literal l in two stages. First, we identify which literals in the trace used l to be derived by unit propagation; we call these literals the *propagation cone* of l. After removing the propagation cone from the trace, the second stage restores into the trace the removed literals that are still implied by unit propagation. These two stages are illustrated in Example 4.

A. Computing the propagation cone

Intuitively, the propagation cone $P(l)$ for literal l with respect to a trace is determined inductively by two rules:

- The literal l is in the propagation cone.
- A literal k from the trace with reason clause D is in the propagation cone if D contains a (necessarily falsified) literal $m \neq k$ where $\overline{m}$ is in the propagation cone.

after **i**: $\overline{\top}x_5$	after cone removal	after reinsertion	after propagation
x_1: $\overline{\top}x_1$	x_1: $\overline{\top}x_1$	x_1: $\overline{\top}x_1$	x_1: $\overline{\top}x_1$
x_2: $\overline{x_1}x_2$	x_5: $\overline{\top}x_5$	x_5: $\overline{\top}x_5$	x_5: $\overline{\top}x_5$
x_3: $\overline{x_1x_2}x_3$	x_6: $\overline{x_1x_5}x_6$	x_6: $\overline{x_1x_5}x_6$	x_6: $\overline{x_1x_5}x_6$
x_4: $\overline{x_1x_3}x_4$		x_4: $\overline{x_5x_6}x_4$	x_4: $\overline{x_5x_6}x_4$
x_5: $\overline{\top}x_5$			x_3: $\overline{x_6x_4}x_3$
x_6: $\overline{x_1x_5}x_6$			x_8: $\overline{x_3x_6}x_8$
x_7: $\overline{x_2x_5}x_7$			
x_8: $\overline{x_3x_6}x_8$			

Fig. 2. Constructing the trace "after **d**: $\overline{x_1}x_2$" from π in Example 2.

To compute the propagation cone $P(l)$ w.r.t. a trace inducing the partial interpretation I, let $P_0(l) = \{l\}$, and

$$P_{n+1}(l) = P_n(l) \cup \{k \in I \mid \exists m \in R_k \setminus \{k\},\ \overline{m} \in P_n(l)\}$$

for each $n \geq 0$, where we denote by R_k the reason clause for literal k in the trace. The propagation cone is then the fixpoint $P(l) = \bigcup_{n \geq 0} P_n(l)$, which exists and is reachable because the sequence $(P_n(l))_{n \in \mathbb{N}}$ is increasing and $P(l)$ is finite. Because the reason clauses for trace literals are stored for conflict analysis purposes, all information needed for computing the propagation cone is available. The cone $P(l)$ is then removed from the trace, keeping the order of remaining literals.

B. Reintroducing literals implied by unit propagation

The fact that a literal k is in the propagation cone of l only means that l was used to derive k by unit propagation in the original trace; but k might still be implied through a different propagation sequence. Such literals must be restored into the trace; to find them, we exploit that unit propagation only requires Invariant 1 to discover all propagations. To satisfy it, we can relocate the watches; calling unit propagation would then do the heavy work. Again, the simple way is to relocate watches for each clause; again, we can outperform this.

Let I and J be the partial assignments defined by the traces before and after the removal of the propagation cone. Invariant 1 is satisfied by I, but possibly violated by J. This only happens for clauses D with watched literals k and m such that $J(k) = 0$ and $J(m) \neq 1$. Removing literals from I can only unassign literals; in particular, we infer that $I(k) = 0$. By Invariant 1 we conclude that $I(m) = 1$, and so m got unassigned by the removal of the propagation cone. Hence, m was in the propagation cone.

This means that the only clauses whose watches may need to be relocated are watched in a literal from the propagation cone. In order to enforce Invariant 1, one can traverse the watchlist for every literal m in the propagation cone $P(l)$ and relocate watches. When this cannot be done, then Invariant 1 is enforced by assigning literal m back into the trace. Furthermore, in the latter case, all subsequent clauses watched in m have correct watches, so we can move on to the next propagation cone literal. This procedure may reassign some literals, which may in turn lead to new propagations. Since Invariant 1 is satisfied afterwards, we can simply perform unit propagation to find them out. Our procedure always reintroduces these literals in the latter part of the stack; this will become very relevant in Section III-C. An overview of the procedure is depicted in Figure 3.

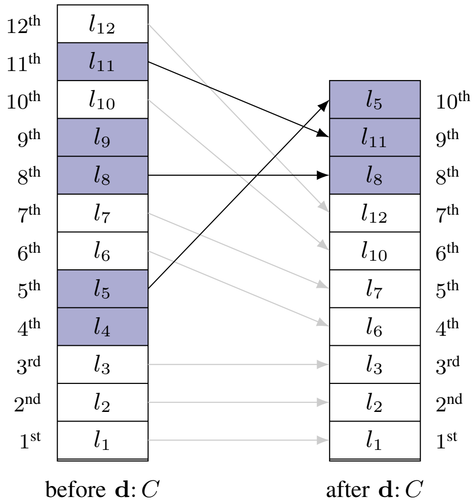


Fig. 3. Stack structure of trace reconstruction after a unit deletion. In this case, the deleted clause C is the reason clause for l_4. The propagation cone $P(l_4)$ is shaded on the left; as explained in Section III-A these literals are removed, and the unshaded part in the stack on the right is obtained. Some literals from $P(l_4)$ may be reinserted as presented in Section III-B; these are the shaded literals on the right, which need not preserve the order on the left.

Example 4. Let us consider the traces for π from Example 2. Starting from the trace "after **i**: $\overline{\top}x_5$", we construct the trace "after **d**: $\overline{x_1}x_2$". Let us assume the following watch choices (shown as dots and only for clauses of size larger than 2):

$$\overline{x_1\dot{x}_2}\dot{x}_3 \qquad \overline{x_2\dot{x}_5}\dot{x}_7 \qquad \overline{x_5\dot{x}_6}\dot{x}_4 \qquad \overline{\dot{x}_6x_4}\dot{x}_3 \qquad \overline{x_4\dot{x}_9}\dot{x}_{10}$$

$$\overline{x_1\dot{x}_3}\dot{x}_4 \qquad \overline{x_1\dot{x}_5}\dot{x}_6 \qquad \overline{x_3\dot{x}_6}\dot{x}_8 \qquad \overline{x_3}\dot{x}_9\dot{x}_{10} \qquad \overline{x_7x_8\dot{x}_9\dot{x}_{10}}$$

Clause $\overline{x_1}x_2$ is the reason for literal x_2 in the trace "after **i**: $\overline{\top}x_5$". The propagation cone $P(x_2)$ contains the literals x_2, x_3, x_4, x_7, x_8. By removing those literals from the trace, we obtain the trace "after cone removal" in Figure 2. The procedure above can be applied to the watchlists for literals in $P(x_2)$. We perform the following changes:

- Watchlist for literal x_3: clause $\overline{\dot{x}_6x_4}\dot{x}_3$ becomes $\overline{x_6\dot{x}_4}\dot{x}_3$.
- Watchlist for literal x_4: clause $\overline{x_5\dot{x}_6}\dot{x}_4$ causes literal x_4 to be reinserted in the trace.
- Watchlist for literal x_7: clause $\overline{x_2\dot{x}_5}\dot{x}_7$ becomes $\overline{\dot{x}_2x_5}\dot{x}_7$.
- Watchlist for literal x_8: clause $\overline{x_3\dot{x}_6}\dot{x}_8$ becomes $\overline{\dot{x}_3x_6}\dot{x}_8$.

This yields the trace "after reinsertion". Unit propagation then finds clause $\overline{x_6\dot{x}_4}\dot{x}_3$ in the watchlist for $\overline{x_4}$, propagating x_3, and clause $\overline{\dot{x}_3x_6}\dot{x}_8$ in the watchlist of $\overline{x_3}$, propagating x_8. We obtain the trace "after propagation", which corresponds to the trace "after **d**: $\overline{x_1}x_2$". ■

C. Restoring trace and watches in backwards checking

The methods explained above apply to the incremental prepropagation sweep. It nevertheless remains unclear how would this work during backwards checking. One problem is recovering the trace before the deletion: removing the latter part of the trace as in Section II-B does not work anymore: fter reverting a clause deletion, some unassigned literals may become assigned. In terms of Example 2, what we need to do is to recover the trace "after $\mathbf{i}\colon \overline{\top} x_5$" from "$\mathbf{d}\colon \overline{x_1} x_2$" for π.

For the time being, our solution is simple: store the trace every time a unit clause deletion is processed during incremental propagation, and then restore it back when the deletion is reverted during backwards checking. This does not solve all the problems, though. In Section II-B, the trace is restored by removing its latter part. As we mentioned there, Invariant 1 is satisfied after doing so; let us inspect the reasons for this.

Removing *arbitrary* literals from the trace can violate Invariant 1, which is required for exhaustive unit propagation. For example, a clause $x_1 x_2$ satisfies the Invariant 1 for a trace containing x_1 and $\overline{x_2}$, but violates it after x_1 is dropped from the trace. Operational-DRAT checkers must be somehow preventing this situation. It is apparent from Invariant 1 and from the monotonic growth of the trace stack in operational-DRAT checking that, once a watched literal is satisfied by the trace during stack prepropagation, further watch relocation is unnecessary. This is not a only an efficiency hack, but also needed to maintain Invariant 1 during backwards checking too: this ensures that, in the conditions above, if x_1 (resp. $\overline{x_2}$) was added to the trace in the j_1-th (resp. j_2-th) instruction during trace preprocessing, then $j_2 \geq j_1$. Hence, during backwards checking, $\overline{x_2}$ is dropped from the trace before or at the same time as x_1, and so the problematic situation above never arises.

Invariant 2. *Consider a clause F in the current accumulated formula for the c-th instruction F_c that is watched on a literal l satisfied by the current trace I_c. Let $p < c$ the largest index such that I_p does not satisfy l, and k be the other watched literal in D. Then either of the following holds:*

a) *$D \notin F_r$ for some index $p \leq r < c$*
b) *$I_r(k) \neq 0$ for some index $p \leq r \leq c$*

This invariant is preserved by operational-DRAT checkers, and forces Invariant 1 to hold after the removal of the latter part of the trace stack when reverting a clause introduction during backwards checking. Unfortunately, reverting a unit clause deletion by restoring the stored trace violates Invariant 2, and this eventually causes Invariant 1 to be violated.

Example 5. Consider now the clause $\overline{x_2}\,\dot{\overline{x_5}}\,\dot{x}_7$ during backwards checking in proof π from Example 1. After instruction $\mathbf{d}\colon \overline{x_1} x_2$, literals $\overline{x_2}$ and x_7 are unassigned, so Invariant 1 holds. However, Invariant 2 is violated with this watch choice: the literal x_7 is last not satisfied in the "start" trace, but this trace falsifies $\overline{x_2}$. Invariant 1 is eventually violated too. In "after $\mathbf{i}\colon \overline{\top} x_5$", literal $\overline{x_2}$ becomes falsified and x_7 becomes satisfied, and so Invariant 1 is still satisfied. Once backwards checking moves on to "start", x_7 is unassigned while $\overline{x_2}$ is still falsified, and this violates Invariant 1. RUP checks may then report false negatives: if literal x_5 is added to the trace, then literal x_7 must be propagated, but since the clause is not watched on literal $\overline{x_5}$ the checker will not inspect this clause. ■

The reason why Invariant 2 is broken in Example 5 lies on the non-monotonic changes that reverting the reason clause deletion $\mathbf{d}\colon \overline{x_1} x_2$ causes in the trace. Restoring Invariant 2 is difficult, since this requires storing the traces after instructions. Instead, we establish an invariant that is strong enough to force Invariant 1 and weak enough to be simple to maintain.

Invariant 3. *Consider a clause D in the current accumulated formula F_c for the c-th instruction that is watched on a literal l satisfied by the current trace I_c. Let $p < c$ the largest index such that I_p does not satisfy l, and k be the other watched literal in D. Then either of the following holds:*

a) *$D \notin F_r$ for some index $p \leq r < c$*
b) *$I_r(k) \neq 0$ for some index $p \leq r \leq c$*
c) *$\overline{k}$ is in the propagation cone from Section III-A at a deletion in some index $p < r \leq c$.*

Together, Invariants 1 and 3 are preserved when reverting an introduction instruction $\mathbf{i}\colon C$ during backwards checking at index c. Assume that they both hold at the c-th instruction. If Invariant 1 was violated at index $c-1$ by some clause $D \in F_{c-1}$, then the value of p would necessarily be $c-1$, and $I_c(k) = I_{c-1}(k) = 0$. Since $\mathbf{i}\colon C$ is an introduction instruction, Invariant 3 would be violated at index $c-1$, which is a contradiction. On the other hand, if Invariant 3 was violated at index $c-1$, then we have $I_{c-1}(l) = I_c(l) = 1$, and furthermore $D \in F_r$ for all $p \leq r < c-1$; $I_r(k) = 0$ for all $p \leq r \leq c$; and $\overline{k}$ is never removed as a part of a propagation cone at an index $p < r \leq c-1$. Because $\mathbf{i}\colon C$ is an introduction instruction $I_c(k) = I_{c-1}(k) = 0$ holds, and $\overline{k}$ is also not removed as a part of a propagation cone at index c. But then Invariant 3 would be violated at index c, which is again a contradiction.

The previous paragraph shows that Invariant 3 is strong enough to guarantee the same good behavior as Invariant 2. However, in the specified-DRAT case we also need to consider reverting deletion instructions $\mathbf{d}\colon C$ during backwards checking at index c, and in general Invariant 3 is not preserved by this operation (although it *almost* is, as we will see in Section IV-C). Instead, we explicitly reestablish the invariant by relocating the watches in every clause D in the accumulated formula F_{c-1} before the deletion. If D is not a unit clause w.r.t. I_{c-1}, we choose as watches any two non-falsified literals. Otherwise, it contains one satisfied literal l, which is chosen as one of the watches. All other literals $k \in D \setminus \{l\}$ are falsified by I_{c-1}. We choose as the second watch the k such that $\overline{k}$ occurs the latest in the trace stack I_{c-1}. Finding k is computationally simple, since the trace is stored as an array in memory, and so it boils down to pointer comparison.

This watch choice trivially satisfies Invariant 1; we show that Invariant 3 is attained too. The former case is straightforward; we explain the case when D is a unit w.r.t. I_{c-1}. Assume D violates Invariant 3. Then we have $I_{c-1}(l) = 1$, and furthermore $D \in F_r$ for all $p \leq r < c-1$; $I_r(k) = 0$ for all $p \leq r \leq c$; and $\overline{k}$ is never removed as a part of a propagation cone at an index $p < r \leq c-1$; where p is defined as in Invariant 3. The trace I_p at the p-th instruction is saturated under unit propagation, so $I_p(l) \neq 1$ implies that there is some $m \in D \setminus \{l\}$ such that $I_p(m) \neq 0$. Our choice of watch k implies that $\overline{m}$ occurs strictly earlier in I_{c-1} than $\overline{k}$. Now consider the instruction at the $(c-1)$-th index.

- If it is an introduction, then I_{c-1} is obtained from I_{c-2} by appending literals in the later part of the stack. Because $I_{c-2}(k) = 0$, $\overline{k}$ is not one of the appended literals; and $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{c-1}, so neither is $\overline{m}$. We conclude that $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{c-2}.
- If it is a deletion, I_{c-1} is obtained from I_{c-2} by removing a propagation cone P, and reinserting some literals from P into the result. We know that $\overline{k} \notin P$; in particular $\overline{k}$ is not reintroduced. As observed at the end of Section III-B, literals are reintroduced at the later part of the stack; so if $\overline{m} \in P$ held true, $\overline{m}$ would occur later than $\overline{k}$ in I_{c-2}, but we have the opposite case. Thus, $\overline{m} \notin P$, and so $\overline{m}$ occurs strictly earlier than $\overline{k}$ also in I_{c-2}.

Iterating this argument shows that $\overline{m}$ occurs strictly earlier than $\overline{k}$ in I_{p+1}. Now, $I_p(l) \neq 1 = I_{p+1}(l)$, so the instruction at index p must be an introduction. Then, I_p is obtained from I_{p+1} by removing literals in the later part of the stack. Now, $I_p(k) = I_{p+1}(k) = 0$, so $\overline{k}$ is not removed; and $\overline{m}$ occurs earlier than $\overline{k}$, so neither is $\overline{m}$. But then $I_p(m) = I_{p+1}(m) = 0$ contradicts our choice of m. Therefore, Invariant 3 is fulfilled.

This completes our method for checking specified-DRAT proofs with incremental preprocessing and backwards checking. To summarize, we give a method that behaves essentially like operational-DRAT checkers, the only difference being the treatment of unit clause deletion instructions. During incremental preprocessing, our method is able to construct a trace reflecting the accumulated assignment after the deletion, and relocate watches in a suitable way. By storing this assignment to memory, we are able to restore it when the same unit clause deletion is encountered during backwards checking; at that point, watches for all clauses must be relocated.

IV. OPTIMIZING UNIT CLAUSE DELETION

The methods from Section III are computationally expensive, and in practice they make specified-DRAT checking much less efficient than operational-DRAT checking. This overhead is mainly due to three causes. First, the fixpoint computation for the propagation cone involves traversing the trace quadratically many times. Second, storing each trace before a deletion instruction may have a notable impact in memory even if the changes in the trace are minimal. Last, the watch relocation method in Section III-C involves relocating the watches for every clause in the formula. We now explain optimizations that greatly reduce the clause deletion-induced overhead in specified-DRAT checking.

A. Linearly computing propagation cones

In order to efficiently compute propagation cones, yet another invariant maintained by traces can be exploited:

Invariant 4. *Let l be a literal in the trace with reason clause R_l. Then, every literal $k \in R_l \setminus \{l\}$ is falsified by the trace, and $\overline{k}$ either is $\top$, or occurs earlier than l in the trace stack.*

```
$P(l) := \{l\}$
for $k$, trace literal after $l$ do
    if there is a literal $m \in R_k$ with $\overline{m} \in P(l)$ then
        $P(l) := P(l) \cup \{k\}$
    end if
end for
```

Fig. 4. Algorithm to linearly compute the implication cone

literal	position index	reason
x_2	3rd	$\overline{x_1}x_2$
x_3	4th	$\overline{x_1 x_2}x_3$
x_4	5th	$\overline{x_1 x_3}x_4$
x_7	8th	$\overline{x_2 x_5}x_7$
x_8	9th	$\overline{x_3 x_6}x_8$

Fig. 5. Information stored to reconstruct trace "after **i**: $\overline{\top}x_5$" from trace "after **d**: $\overline{x_1}x_2$" in Example 1.

The algorithm in Figure 4 exploits Invariant 4 to compute the implication cone in a single pass through the trace[2].

B. Storing deleted traces as permutations

Rather than storing each trace before a reason clause deletion during incremental prepropagation and restoring it during backwards checking, we can store the permutation that the trace undergoes. By deleting a clause, no literal is derived: some literals are removed from the trace, and some others are moved to the latter part of the trace stack. From Figure 3 it is apparent that storing the original reasons and positions within the trace for propagation cone literals is enough to restore the trace before deletion from the trace after deletion. Following Example 1, we store the information in Figure 5 to reconstruct the trace "after **i**: $\overline{\top}x_5$" from the trace "after **d**: $\overline{x_1}x_2$".

C. On-demand watch relocation

Our previous analysis required the relocation of watches during backwards checking for all clauses in the accumulated formula. This is immensely wasteful: our preliminary experiments showed that doing so takes up to 85% of the checking runtime. This can however be vastly improved, reducing the runtime share spent on this sort of watch relocation negligible.

Consider a clause deletion $\mathbf{d}: C$ at the c-th index, which removed the propagation cone P from the trace I_{c-1}, reintroducing afterwards a set $R \subseteq P$ of literals to obtain I_c. Let D be a clause in F_c watched on l and k, and assume it satisfies Invariants 1 and 3 at the c-th instruction. If $\overline{k}$ and $\overline{l}$ do not occur in P, it is easy to check that both invariants also hold at the $(c-1)$-th instruction. In other words: the watch relocation explained in Section III-C is only needed for clauses in the watchlist of $\overline{l}$ for every literal l in the propagation cone.

---

[2] An anonymous reviewer pointed out that `MiniSAT` contains a similar algorithm in its `analyzeFinal` function [3].

V. EXPERIMENTAL EVALUATION

The ideas described in this paper were implemented in a proof-of-concept DRAT checker `rupee`. Our DRAT checker can be run in operational or specified modes; the operational mode is designed to be as close as possible to a standard DRAT checker, whereas the specified mode includes the unit deletion processing methods described in this paper. Being a proof-of-concept implementation, this checker lacks of many optimizations, including efficient proof parsing, exploitation of CPU cache, core-first propagation, and resolution candidate caching. We thus expect worse performance than state-of-the-art checkers. However, our goal is to measure the overhead induced by specified-DRAT checking compared to operational-DRAT checking, and for this we needed a system that we completely understood to minimally change the behavior between the two modes. To the best of our knowledge, there is no reason to think that the aforementioned optimizations are incompatible with our methods for specified-DRAT checking.

An LRAT certificate [2] can be generated for instances that `rupee` reports as correct. For instances reported as incorrect, `rupee` reports information on the state of the trace at the end of RUP and RAT checks on failing instructions. To the best of our knowledge, `rupee` reports the right result in both modes.

We selected 11 benchmarks which were solved fast by solvers in the SAT Competition 2017. DRAT proofs for these benchmarks were generated by 4 participant solvers: `COMiniSatPS_Pulsar_drup`, `glucose-4.1`, `Maple_LCM_Dist`, and `cadical-sc17-proof`. The 44 resulting proofs were checked with `rupee` in both modes, as well as with the state-of-the-art `DRAT-trim` as a baseline[3].

`DRAT-trim` and `rupee` in operational mode agree on all instances, as expected; `rupee` in specified mode only agrees on 18 instances, rejecting all remaining instances. Hence, discrepancies between specified-DRAT and operational-DRAT occur rather frequently. Despite the semantic complexity of the interaction between RAT introduction and clause deletion [14], [15], this is not the cause of discrepancies: none of the discrepant proofs contains RAT clauses. The distribution of the discrepancies gives some insight in this regard: `cadical-sc17-proof` produced no discrepancies; for the other three solvers 8 out of 11 proofs were discrepant. We conjecture that the cause of discrepancies may be in the `MiniSAT` patch which most checkers use for proof generation in the CDCL loop, since `cadical-sc17-proof` implements its own method.

Figure 6 shows runtime results. We only compared results on instances where all three checkers accepted the proof; comparing discrepant instances would be meaningless, since execution stops as soon as an instruction is declared incorrect. `DRAT-trim` performs about one order of magnitude better than `rupee`; this is expectable due to the lack of optimizations in our tool. However, the runtimes of `rupee` in both its modes are comparable, with the specified mode outperforming the operational mode in hard instances. We conclude that the overhead of checking specified-DRAT proofs as compared to operational-DRAT proofs can be made negligible. Further research is required to verify the observed speed-up; one possible explanation would be that, by deleting more clauses in the specified mode, less resolution candidates are available for RAT checks, and so less RUP check calls need to be made.

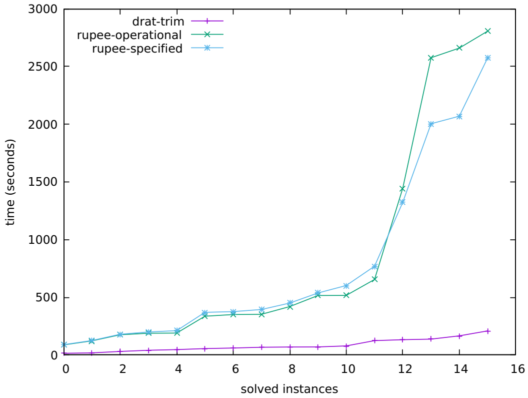


Fig. 6. Performance of `DRAT-trim` compared to both versions of `rupee`.

VI. CONCLUSION

The notion of a correct DRAT proof in the specification differs from the used in the implementation of DRAT checkers. We discussed the practical reasons for this, which lie on data structure invariants that are broken if the original definition of DRAT were to be respected. We proposed several changes in DRAT checkers' data structures and algorithms to check DRAT proofs according to the specification in an efficient way. In particular, we explained how to maintain slightly more intricate invariants so that unit clause deletions can be applied, and explored ways to vastly reduce the induced overhead.

We implemented these enhanced algorithms in a tool `rupee`, and used it to verify DRAT proofs produced by modern SAT solvers. Our results show that the discrepancy between the DRAT definition and the operational notion of correctness arises relatively often in practice. Our tool has a negligible overhead over checking with respect to the operational semantics, although further efforts in optimization must be done in order to attain similar performance to state-of-the-art DRAT checkers. Our data also suggests that discrepancies might have their root cause in an anomalous behavior of the CDCL proof logging method underlying many solvers. This suggests that future work should be directed towards efficient, specification-complying proof generation.

Acknowledgments: We would like to acknowledge anonymous reviewers who pointed out several relevant details. This work was supported by the Austrian National Research Network S11403-N23 (RiSE), the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF), the Vienna Science and Technology Fund (WWTF) through grant VRG11-005 and Microsoft Research through its PhD Scholarship Programme.

---

[3] https://github.com/arpj-rebola/fmcad2018

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

REFERENCES

[1] Tomas Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In *AAAI Conference on Artificial Intelligence*, pages 5061–5063, 2017.

[2] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *CADE*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919, pages 502–518. Springer, 2004.

[4] Walter Forkel, Tobias Philipp, Adrian Rebola-Pardo, and Elias Werner. Fuzzing and verifying RAT refutations with deletion information. In *Florida Artificial Intelligence Research Society Conference*, pages 190–193. AAAI Press, 2017.

[5] Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[6] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 886–891. IEEE, 2003.

[7] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving*, volume 10499 of *LNCS*, pages 269–284. Springer, 2017.

[8] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[9] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *CADE*, volume 7898 of *LNCS*, pages 345–359. Springer, 2013.

[10] Marijn J. H. Heule. The DRAT format and drat-trim checker. *CoRR*, abs/1610.06229, 2016.

[11] Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*. AAAI Press, 2018.

[12] Peter Lammich. Efficient verified (UN)SAT certificate checking. In *CADE*, volume 10395 of *LNCS*, pages 237–254. Springer, 2017.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[14] Tobias Philipp and Adrián Rebola-Pardo. Towards a semantics of unsatisfiability proofs with inprocessing. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 65–84. EasyChair, 2017.

[15] Adrián Rebola-Pardo and Armin Biere. Two flavors of DRAT. EasyChair Preprint no. 457, EasyChair, 2018.

[16] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

ISBN: 978-0-9835678-8-2.

Semantic-based Automated Reasoning for AWS Access Policies using SMT

John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek,
Kasper Luckow, Neha Rungta, Oksana Tkachuk, Carsten Varming
Amazon Web Services

***Abstract*—Cloud computing provides on-demand access to IT resources via the Internet. Permissions for these resources are defined by expressive access control policies. This paper presents a formalization of the Amazon Web Services (AWS) policy language and a corresponding analysis tool, called ZELKOVA, for verifying policy properties. ZELKOVA encodes the semantics of policies into SMT, compares behaviors, and verifies properties. It provides users a sound mechanism to detect misconfigurations of their policies. ZELKOVA solves a PSPACE-complete problem and is invoked many millions of times daily.**

I. INTRODUCTION

Cloud computing provides on-demand access to IT resources via the Internet. The convenience of accessing resources in the cloud is made secure by user-specified *access control policies*. An access control policy is an expressive specification of what resources can be accessed, by whom, and under what conditions. Properly configured policies are an important part of an organization's security posture. The scale and diversity of cloud-based services is constantly growing (*e.g.*, serverless computing, streaming analytics, edge-computing devices), and each new offering used by an organization may require a different access policy configuration. Moreover, customers are combining these services, which means that the complexity is increasingly moving into policies. Thus the security challenge for many customers is becoming one of reasoning about static policies for their dynamic systems. Cloud customers want a tool that allows them to check policy configurations based on their security requirements.

Amazon Web Services (AWS) defines a policy language that lets users govern access to AWS resources. The permissions granted by a policy rely on the interactions of different statements and conditions. The policy language supports the interplay of statements that either grant access (*allow* statements) or revoke access (*deny* statements). Furthermore, conditions within statements can be based on access details such as the source address, encryption, and other configuration options.

Users want assurances that their policies grant the right permissions. To validate that policies express what is intended, some AWS users have implemented heuristic-based syntactic checks that detect certain patterns in policies, *e.g.*, the use of a wildcard that makes resources publicly accessible. Although helpful, heuristic-based syntactic checks are unsound, since they do not fully take into account the semantics of the policy language. Others attempt to explicitly enumerate all possible requests to a policy but quickly find this intractable.

In this paper, we present the development and application of ZELKOVA, a policy analysis tool designed to reason about the semantics of AWS access control policies. ZELKOVA translates policies and properties into Satisfiability Modulo Theories (SMT) formulas and uses SMT solvers to check the validity of the properties. We use off-the-shelf solvers and an in-house extension of Z3 called Z3AUTOMATA.

ZELKOVA reasons about all possible permissions allowed by a policy in order to verify properties. For example, ZELKOVA can answer the questions "Is this resource accessible by a particular user?" and "Can an arbitrary user write to this resource?". The property to be verified is specified in the policy language itself, eliminating the need for a different specification or formalism for properties. In addition, ZELKOVA provides many built-in checks for common properties.

The SMT encoding uses the theory of strings, regular expressions, bit vectors, and integer comparisons. The use of the wildcards $*$ (any number of characters) and ? (exactly one character) in the string constraints makes the decision problem PSPACE-complete. However, our experience with real-world policies is that 99% of policy questions can be answered in less than 160 milliseconds.

ZELKOVA is the underlying policy analysis engine for a growing number of AWS services. Used many millions of times every day, ZELKOVA analyzes policies attached to resources with compute, storage, messaging, search, analytics, and other capabilities. A sample of AWS services that integrate ZELKOVA includes Amazon S3 (object storage), AWS Config (change-based resource auditor), Amazon Macie (security service), AWS Trusted Advisor (compliance to AWS best practices), and Amazon GuardDuty (intelligent threat detection). Also, ZELKOVA is used by internal AWS Security auditing tools to enforce security best-practices for policy configurations, *e.g.*, public access to the resources is prohibited.

A. Related work

Policy languages have been used in a variety of domains, *e.g.*, trust management, distributed authorization, role-based access, access control of resources [1]–[6]. Several policy languages are defined as Datalog programs since it enables efficient verification of properties [2], [6]–[10]. The AWS policy language is defined with respect to a JSON serialization, and is designed to be used across various cloud services and scenarios of access control. ZELKOVA combines all the components of the policy language in a single analysis tool.

ISBN: 978-0-9835678-8-2.

Fisler *et al.* define a policy formalism that consists of transitions between different states of the environment that determine access control in policies [2]. The access control model in AWS also uses a policy and a dynamic environment request context to determine permissions, but the environment does not evolve during a single access request. Other policy frameworks, *e.g.*, XACML, allow policies across different applications to be combined [11], [12]. In a closely related work, Hughes and Bultan transform XACML policies into Boolean satisfiability problems and use a SAT solver to check partial orders between policies using a bounded analysis. Bounding the analyses, however, makes it unsound. In contrast, the encoding to SMT in ZELKOVA is sound. The TRBAC policy model uses concrete units of time to grant or revoke access [13]. This is accomplished in the AWS policy language with conditions on date and time. Finally, the SecGuru tool [14] compares network connectivity policies using the SMT theory of bit vectors.

Our present work stands out most along three dimensions. First, we use an existing industrial policy language, which has evolved to suit the needs of millions users and use cases. The language is robust and flexible, with features that have arisen from practical needs. Second, we work closely with service teams to integrate our tool and to develop custom pre-built properties that are relevant to each service's users. Finally, we have reached an audience of many millions with our tool.

II. APPROACH

When an access request is made to an AWS service, a *request context* is generated which includes the *principal* making the request, the *resource* being requested, and the specific *action* being requested. A policy evaluation engine compares this request context against the policies for the user and the resource to determine if access is granted or denied.

ZELKOVA verifies AWS policies by reasoning over all possible request contexts. The fundamental mechanism of ZELKOVA is the ability to say if one policy is less-or-equally-permissive than another. Properties can be specified as *boundary policies* that represent either upper or lower bounds on desired behavior. ZELKOVA's less-or-equally-permissive check then establishes the correctness of these bounds or finds a counterexample.

A. Policy language overview

The AWS policy language is defined as serialized JSON[1], however, in this paper we describe the core constructs of the policy language in a simplified abstract syntax. The examples in this paper are also presented using this abstract syntax.

Policy → *Statement**
Statement → (*Effect*, *Principal*, *Action*, *Resource*, *Condition*?)
Effect → `allow` | `deny`
Principal → `principal`: *string**
Action → `action`: *string**
Resource → `resource`: *string**
Condition → `condition`: *Operator**
Operator → (*OpName*, *KeyName*, *Value**)
OpName → `StringEquals` | `StringEqualsIfExists` | `StringLike` |
`StringNotEquals` | `IpAddress` | ...
KeyName → `aws:sourceVpc` | `aws:sourceIp` | `s3:prefix` | ...
Value → *string* | *num* | *bool*

Fig. 1. Simplified abstract syntax for the AWS policy language

Fig. 1 shows the abstract syntax for the policy language. In this syntax, ? denotes optional elements and * denotes list valued elements. A policy is a list of statements. Each *Statement* consists of a tuple (*Principal*, *Effect*, *Action*, *Resource*, *Condition*?). The *Condition* is an optional element in the policy while the others are required. The *Effect* construct states whether the statement allows or denies access. By default, access to a resource is denied. Allow statements override the default permissions, and deny statements override the permissions granted by allow statements. In other words, to get access to a resource, there must be some allow statement that grants access and no deny statement that revokes that access. There is no ordering constraints on statements in a policy.

The *Principal* construct is used in policies to specify which users, accounts, services, or entities are granted or denied access to resources. The principals are identified uniquely by string values. The *Action* construct specifies the list of actions that are either allowed or denied on the corresponding resource. Various AWS services publish the set of actions that can be performed by the user for the resources specific to those services. The *Resource* construct specifies the list of service specific resources to which access is either granted or denied. Every AWS service has its own set of resources and each AWS resource is uniquely identified by a string value. String values for *Action* and *Resource* can contain the wildcard $*$ which matches any number of characters and the wildcard ? which matches exactly one character.

The *Condition* construct specifies conditions under which access is granted or denied. In the *Condition* construct expressions are constructed using *Operators* on condition key value pairs. The condition operators are grouped by their types: String, Numeric, Date and Time, Boolean, Binary, IP address, and others. The operator name (*OpName*) indicates the type and the comparator. String condition operators provide comparison on string conditions, *e.g.*, `StringEquals` checks string equality, `StringLike` checks a string against a pattern. The complete list of operators is defined in the IAM documentation[2] and is supported in our implementation. The operators are applied to condition keys (*ConditionKey*). Each condition key is mapped to a corresponding value. Certain condition keys are defined globally across all services, *e.g.*, `aws:sourceIp`, while other condition keys are service specific, `s3:prefix`.

---

[1] https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html

[2] https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition_operators.html

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

```
(( allow,
   principal  :  students,
   action     :  getObject,
   resource   :  cs240/Exam.pdf),
 ( allow,
   principal  :  tas,
   action     :  getObject,
   resource   :  (cs240/Exam.pdf,
                  cs240/Answer.pdf)))
```

(a) Policy X

```
(( allow,
   principal  :  *,
   action     :  getObject,
   resource   :  cs240/*),
 ( deny,
   principal  :  students,
   action     :  getObject,
   resource   :  cs240/Answer.pdf))
```

(b) Policy Y

Fig. 2. Example policies for students and TAs access to exams and answers.

$$\begin{aligned}
X_0 &: a = \text{“}\mathit{getObject}\text{”} \wedge p = \text{“}\mathit{students}\text{”} \wedge r = \text{“}\mathit{cs240/Exam.pdf}\text{”}\\
X_1 &: a = \text{“}\mathit{getObject}\text{”} \wedge p = \text{“}\mathit{tas}\text{”} \wedge\\
&\qquad (r = \text{“}\mathit{cs240/Exam.pdf}\text{”} \vee r = \text{“}\mathit{cs240/Answer.pdf}\text{”})\\
X &: X_0 \vee X_1\\
Y_0 &: a = \text{“}\mathit{getObject}\text{”} \wedge r = \text{“}\mathit{cs240/*}\text{”}\\
Y_1 &: a = \text{“}\mathit{getObject}\text{”} \wedge p = \text{“}\mathit{students}\text{”} \wedge r = \text{“}\mathit{cs240/Answer.pdf}\text{”}\\
Y &: Y_0 \wedge \neg Y_1
\end{aligned}$$

Fig. 3. SMT encoding of policies X and Y from Fig. 2.

B. Example

A *policy* in the simplified abstract syntax for the Amazon Simple Storage Service (S3) is shown in Fig. 2. Amazon S3 is an object store where a logical unit of storage is called a *bucket*. S3 stores data as objects in these buckets. Each resource, *e.g.*, the bucket and the objects in the bucket, is uniquely identified through an Amazon Resource Name (ARN). The policy attached to the bucket controls access to the bucket and the objects in the bucket. The policy in Fig. 2(a) states that students can read the exam and teaching assistants can read both the exam and its answers. The other policy, shown in Fig. 2(b), says that everybody can access all the contents of *cs240/* except that students cannot access the answers.

Fig. 3 shows the encoding of the policies from Fig. 2. The encoding for each policy is a formula over three variables p, a, and r that correspond to the principal, action, and resource in the resource request context. The formula evaluates to true whenever the policy grants access. Since policy X has two allow statements that can grant access, it is represented by their disjunction. On the other hand, policy Y has one allow statement Y_0 and one deny statement Y_1. Thus policy Y only grants access if Y_0 allows access and Y_1 does not deny it: $Y_0 \wedge \neg Y_1$. Note that we are abusing notation in Y_0 to say $r = \text{“}\mathit{cs240/*}\text{”}$ since this, in fact, will correspond to a form of string matching rather than equality. We discuss the details of string matching in Section III-A.

To determine if policy X is less-or-equally-permissive than policy Y, ZELKOVA uses SMT solvers to check if

$$(X_0 \vee X_1) \implies (Y_0 \wedge \neg Y_1)$$

is valid, which is true. The result of this check states that all requests allowed by policy X are allowed by policy Y.

However, policy Y allows additional permissions. The resource *“cs240/*”* in the allow statement in policy Y allows the *“students”* and *“tas”* principals access to objects (files) *other* than *“Exam.pdf”* and *“Answer.pdf”*, such as *“ClassRoster.pdf”*. Policy Y additionally grants principals other than *“students”* and *“tas”* access to the resources in the bucket *“cs240”*, since the deny statement only denies *“students”* access to the *“Answer.pdf”*. This leads to a publicly readable bucket since any other principal can perform the *getObject* action on the contents of the bucket. Thus this policy does not represent the user's intentions, and it violates security best practices. This shows the need for sound analysis of policies. ZELKOVA provides this by reducing policies to mathematical formulas and verifying their properties using SMT solvers.

III. SMT ENCODING

In this section, we describe ZELKOVA's SMT encoding. The encoding uses the theory of strings, regular expressions, bit vectors, and integer comparisons. The policy language is declarative, with no programming constructs such as loops or dynamically allocated arrays. The semantics of the policy language are encoded as an SMT formula. The permissions granted by the policy are encoded as all the permissions granted by allow statements and not revoked by deny statements:

$$\left(\bigvee_{S \in \mathit{Allow}} [\![S]\!]\right) \wedge \neg \left(\bigvee_{S \in \mathit{Deny}} [\![S]\!]\right) \tag{1}$$

Here *Allow* and *Deny* are the set of allow and deny statements in a policy. The semantic meaning of each statement, $[\![S]\!]$, is the set of permissions granted by an allow statement or the set of permissions revoked by a deny statement.

Each statement in a policy encodes the constraints over the principal, action, resource, and conditions:

$$\begin{aligned}
[\![S]\!] := \left(\bigvee_{v \in P(S)} p = v\right) \wedge \left(\bigvee_{v \in A(S)} a = v\right) \wedge\\
\left(\bigvee_{v \in R(S)} r = v\right) \wedge \left(\bigwedge_{O \in C(S)} [\![O]\!]\right)
\end{aligned} \tag{2}$$

The function $P(S)$ returns all the string values specified for a principal. Similarly, $A(S)$ and $R(S)$ return the string values for the actions and resources in the statement. The function $C(S)$ returns the set of condition operators for a given statement. The variables p, a, and r map respectively to the principal, action, and resource values. The permissions in a statement are granted as a disjunction over string values of the principal, action, and resource values as well as a conjunction over the conditions as shown in Eq. (2).

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

```
( allow,
  principal   :   *,
  action      :   getObject,
  resource    :   cs240,
  condition   :   (StringEquals, aws:sourceVpc, vpc-111bbb222),
                  (StringLike, s3:prefix, cs240/Exam*))
```

Fig. 4. Example policy with two conditions.

Each condition in a policy encodes a constraint over the corresponding condition key:

$$[\![O]\!] := \bigwedge_{\langle op,k,V\rangle \in CO(O)} \left(\textit{condExists}_k \wedge \left(\bigvee_{v \in V} op(k,v) \right) \right) \tag{3}$$

Each condition maps to an operator name, a key name, and a list of values via the function $CO(O)$. The meaning of a condition is encoded by a disjunction over all the listed values. The Boolean variable $\textit{condExists}_k$ states that condition key, k, must exist in the request context. The variable k represents the value of the condition key when it exists. The operator (op) defines the operations on the key and value pair (k, v), *e.g.*, equality or inequality.

Next, we present the encoding of a few important classes of condition operators.

A. String constraints

The encoding of policies in Zelkova is largely through the use of string constraints. This includes both string equality and inequality constraints, as well as pattern matching against regular expressions. The principal, action, and resources constructs in the policy are encoded as string constraints. String operators and their corresponding condition keys are also encoded as string constraints. An example policy with conditions is shown in Fig. 4. The operator `StringEquals` is applied to the condition key `aws:sourceVpc` with a value of *“vpc-111bbb222”*, which restricts access to a specific virtual private network (VPC) in the AWS cloud[3]. The string operator `StringLike` is applied to the condition key `s3:prefix` with a value of *“grades/∗”*, which limits access so that only objects under the *“grades/”* directory may be listed.

Fig. 5 shows the SMT encoding for this example. The Boolean variables *vpcExists* and *s3PrefixExists* encode whether the conditions `aws:sourceVpc` and `s3:prefix` are present in the request context. The constraint *“grades/” prefixOf s3Prefix* encodes that *“grades/”* is a prefix of the variable *s3Prefix*. The following request context corresponds to a satisfying assignment to the set of constraints in Fig. 5:

```
{principal: bob,
 action : listBucket,
 resource : cs240,
 condition: {aws:sourceVpc: vpc-111bbb222,
             s3:prefix: grades/2018/final/}}
```

$$\begin{aligned} & a = \texttt{"listBucket"} \wedge r = \textit{“cs240”} \wedge \\ & \textit{vpcExists} \wedge \textit{vpc} = \textit{“vpc-111bbb222”} \wedge \\ & \textit{s3PrefixExists} \wedge \textit{“grades/” prefixOf s3Prefix} \end{aligned}$$

Fig. 5. SMT encoding of policy in Fig. 4

```
( allow,
  principal   :   *,
  action      :   listBucket,
  resource    :   *,
  condition   :   (StringEquals, s3:prefix, UpLoads),
                  (StringEqualsIgnoreCase, s3:prefix, Uploads))
```

Fig. 6. Example policy with mixed case conditions.

In order to encode $*$ wildcards in strings we use the *prefixOf*, *suffixOf*, and *contains* string operators. With this encoding we can support up to two $*$ wildcards. Later we will see a different encoding for additional wildcards. Examples of the current encoding are given in (4).

$$\begin{aligned} &\textit{“cs2*/Exam*”} \mapsto \textit{“cs2”}\ \texttt{prefixOf}\ \textit{Var} \wedge \textit{Var}\ \texttt{contains}\ \textit{“/Exam”} \\ &\textit{“cs2*/*Exam”} \mapsto \\ &\quad \textit{“cs2”}\ \texttt{prefixOf}\ \textit{Var} \wedge \textit{Var}\ \texttt{contains}\ \textit{“/”} \wedge \textit{“Exam”}\ \texttt{suffixOf}\ \textit{Var} \\ &\textit{“*240/*Exam”} \mapsto \textit{Var}\ \texttt{contains}\ \textit{“240/”} \wedge \textit{“Exam”}\ \texttt{suffixOf}\ \textit{Var} \end{aligned} \tag{4}$$

When different parts of a pattern can overlap, we disallow the possible overlaps. For example, *“ab∗bc”* translates to $\textit{“ab”}\ \texttt{prefixOf}\ \textit{Var} \wedge \textit{“bc”}\ \texttt{suffixOf}\ \textit{Var} \wedge \textit{Var} \neq \textit{“abc”}$. Note that *“abc”* would otherwise satisfy the prefix and suffix constraints, yet it does not match the pattern *“ab∗bc”*.

B. Regular expression constraints

More complicated string constraints require a more powerful encoding. In particular, the encoding described above is unable to represent constraints with the ? wildcard or more than two $*$ wildcards. For example, the following encoding fails because it does not restrict *“b”* to appear before *“c”*.

$$\begin{aligned} \textit{“a*b*c*d”} \mapsto\ & \textit{“a”}\ \texttt{prefixOf}\ \textit{Var} \wedge \textit{Var}\ \texttt{contains}\ \textit{“b”} \wedge \\ & \textit{Var}\ \texttt{contains}\ \textit{“c”} \wedge \textit{“d”}\ \texttt{suffixOf}\ \textit{Var} \end{aligned} \tag{5}$$

In such cases, we use regular expressions to encode these constraints. For example, (6) shows two encodings based on the traditional regular expression pattern format where “.” stands for any single character and “*” is the Kleene star operator representing zero or more occurrences of the previous character.

$$\begin{aligned} &\textit{“cs???/Exam*”} \mapsto \textit{Var}\ \texttt{matches}\ \textit{“cs.../Exam.*”} \\ &\textit{“cs2*/Exam/*/Results/*”} \mapsto \\ &\qquad\qquad \textit{Var}\ \texttt{matches}\ \textit{“cs2.*/Exam/.*/Results/.*”} \end{aligned} \tag{6}$$

Some condition operators are case sensitive (*StringEquals*, *StringLike*) while others are case insensitive (*StringEqualsIgnoreCase*, *Bool*). Which type of operators are used on the same condition key determines the exact encoding for case sensitivity. When a condition key is constrained with only case sensitive operators, nothing special needs to be done. When a condition key is constrained with only case insensitive operators, the targets of all those comparisons are converted to lowercase which solves the problem. The difficult case is

---

[3] https://aws.amazon.com/vpc/

$$
\begin{aligned}
&a = \text{“}\texttt{listBucket}\text{”} \land \mathit{s3PrefixExists} \land \\
&\mathit{s3Prefix}\ \texttt{matches}\ \text{“}\mathit{UpLoads}\text{”} \land \\
&\mathit{s3Prefix}\ \texttt{matches}\ \text{“}\mathit{[uU][pP][lL][oO][aA][dD][sS]}\text{”}
\end{aligned}
$$

Fig. 7. SMT encoding of policy in Fig. 6

when a condition key is constrained with both case sensitive and case insensitive operators. The previous method of converting to lowercase all targets of case insensitive operators would interfere with the case sensitive operators. Instead, case sensitive comparisons are treated normally while the targets of case insensitive comparisons are encoded into a regular expression that represents all possible case combinations. For example, consider the contrived combinations of conditions in shown in Fig. 6. Here there is both a case sensitive and a case insensitive constraint on the `s3:prefix` condition key. The ZELKOVA encoding of these constraints is shown in Fig. 7 where we use character classes of the form *[xX]* to represent a regular expression which matches a single character, either *“x”* or *“X”*.

C. Bit vector constraints

The `IpAddress` condition operator allows users to restrict access based on IP addresses. The `IpAddress` operator is used in combination with the `aws:SourceIp` condition. The values of `aws:SourceIp` have to be in the Classless Inter-Domain Routing (CIDR) format. The CIDR format associates net masks as part of the IP address specification. For example, the IPv4 in CIDR notation 11.22.33.0/24 means that the first 24 bits of the IP address are considered significant. Consider the translation of two conditions, one where `aws:SourceIp` is set to 11.22.33.0/24 and the other set to 11.22.0.0/16:

$$
\begin{aligned}
&C_0 : (\texttt{IpAddress}, \texttt{aws:SourceIp}, 11.22.33.0/24) \mapsto \\
&\qquad \mathit{ipV4Exists} \land (\mathit{0x0B162100} = (\mathit{ipV4}\ \&\ \mathit{0xFFFFFF00})) \\
&C_1 : (\texttt{IpAddress}, \texttt{aws:SourceIp}, 11.22.0.0/16) \mapsto \\
&\qquad \mathit{ipV4Exists} \land (\mathit{0x0B160000} = (\mathit{ipV4}\ \&\ \mathit{0xFFFF0000}))
\end{aligned} \tag{7}
$$

The Boolean variable *ipV4Exists* encodes the existence of condition `aws:SourceIp`, and the bit vector variable *ipV4* encodes the actual value. A bitwise AND operation is used to mask the insignificant bits of the IP address in the constraint.

With this encoding we have $[\![C_0]\!] \implies [\![C_1]\!]$ is valid. There are 24 significant bits in the IP address in constraint C_0 and only 16 significant bits in the IP address in the constraint C_1. The common routing prefix is the same. Thus, request contexts that are allowed by C_0 are also allowed by C_1.

D. Other operators

The conditions on numeric operators only perform integer comparisons. There are no arithmetic operations in the policy language and no interactions between numeric values and string values, *e.g.*, you cannot take the length of a string. The conditions applicable to the Boolean operators are simply encoded as Boolean constraints. Conditions with the `IfExists` suffix check existence of the condition key in the request context. This suffix can be added to other condition operators such as `StringEquals` which results in a new operator `StringEqualsIfExists`. The resulting operator can be applied to the `aws:sourceVpc` condition key. For example:

$$
\begin{aligned}
&(\texttt{StringEqualsIfExists}, \texttt{aws:sourceVpc}, \text{“}\mathit{vpc\text{-}111bbb222}\text{”}) \mapsto \\
&\quad \mathit{awsSourceVpcExists} \implies \mathit{awsSourceVpc} = \text{“}\mathit{vpc\text{-}111bbb222}\text{”}
\end{aligned} \tag{8}
$$

	Z3	CVC4	Z3AUTOMATA
UNSAT	965,092	34,908	0
SAT	959,543	39,932	525

Fig. 8. Number of times each solver was the fastest for one million UNSAT and one million SAT property checks.

IV. Z3AUTOMATA

Z3AUTOMATA is an in-house extension of Z3 designed to provide a complete decision procedure for the theory of regular expressions. As described in Section III, ZELKOVA uses the regular expressions for problems that involve more than two $*$ wildcards, any ? wildcards, or tricky combinations such as mixing case-sensitive and case-insensitive string comparisons. Such cases are rare in general, but common at our scale where we receive many millions of queries every day.

Z3 and CVC4 aim to efficiently solve problems over word equations, a strictly more general problem than regular expression matching. This sometimes results in degraded performance for pure regular expression problems. For example, both fail to answer the query “Does there exist a string that matches *‘ab∗b∗b∗b’* but not *‘a∗b∗b∗b’*?”. More generally, both solvers seem very sensitive to small changes in the input encoding, where a quickly solved problem in our domain becomes non-terminating. Yet, the theory of regular expressions is decidable, and our problems stay within that theory. Thus Z3AUTOMATA fills an important niche for our domain.

Fig. 8 shows which solver was the fastest for one million UNSAT and one million SAT Zelkova property checks, both randomly selected. Note that for UNSAT problems, Z3AUTOMATA is never the fastest solver. The SMT problems that ZELKOVA generates contain a mix of both simple and complex string constraints. For the properties that ZELKOVA checks, an UNSAT result is, in our experience, always due to the simple string constraints being unsatisfiable. Z3 and CVC4 can easily and efficiently handle that case, thus Z3AUTOMATA never wins. In the case where the constraints are satisfiable, all the constraints must be considered including the complex ones. Here, Z3AUTOMATA is able to win, often in cases where Z3 and CVC4 are non-terminating.

Z3AUTOMATA solves regular expression problems using the standard translation to deterministic finite automata (DFAs) via non-deterministic finite automata (NFAs). It uses Hopcroft’s algorithm for DFA minimization [15]. Z3AUTOMATA is parametric with respect to the character set and strives to produce strings using only the printable subsets of a character set. The

```
( allow,
  principal  :  *,
  action     :  getObject,
  resource   :  *,
  condition  :  (StringEquals, aws:sourceVpc, vpc-111bbb222)))
( allow,
  principal  :  *,
  action     :  putObject, listBucket, ...,
  resource   :  *)
```

Fig. 9. A policy check that allows *getObject* requests only from *vpc-111bbb222*.

full range of regular expression (and automata) features are supported including intersection, union, and complement.

Z3AUTOMATA currently integrates with Z3 only on the SAT level and treats each regular expression match as an atom. A good future challenge for the SMT community to solve is how to integrate this into the traditional Nelson-Oppen framework.

V. ZELKOVA PROPERTIES

Organizations using cloud services want assurances that policies being authored or modified by users do not violate general security best-practices, adhere to the security guidelines defined by the organization, and do not deny access to the intended users. Examples of these properties are as follows: *"Ensure that unrestricted public write is not allowed to a particular resource."* (security best-practice), *"Ensure access to a resource is only allowed from a certain range of IP addresses."* (organizational security check), and *"Ensure a particular user is allowed to perform a specific action on a resource"* (availability property). These properties can be specified in the policy language and checked by ZELKOVA. Verification of properties by ZELKOVA provides assurance that there are no inappropriately configured resources within an organization.

A. Organizational security checks

We use the example in Section II to describe how an organization can specify a property in the policy language such that it can be checked by ZELKOVA. The example in Fig. 2(b) allows principal "$*$" access to the *cs240* resource and denies *students* access to *Answer.pdf*. The principal being set to a wildcard can lead to unauthorized access of objects by users who are not members of the University as described in Section II. As a safeguard measure, suppose, the University administrator wants to ensure that there is no unauthorized access to data in the buckets. The administrator and the security lead of the University decide that an appropriate property to check would be "*the getObject action on the CS department S3 buckets is only allowed on requests from vpc-111bbb222.*" The VPC is owned by the University, and so access requests from within the VPC are trusted.

A policy that specifies the property, "*getObject actions are only allowed from vpc-111bbb222*" is shown in Fig. 9. The first allow statement in Fig. 9 permits *getObject* only when the request comes from *vpc-111bbb222*. The second allow statement permits all other unrelated actions that are not relevant to the comparison. The policy in Fig. 9 represents a desired upper bound on the set of request contexts that should be allowed. This bound will only be violated if the input policy allows a request which Fig. 9 does not allow. In such a case, the request must be a *getObject* request (since all other requests are allowed by the second allow statement in Fig. 9) and it must come from outside of *vpc-111bbb222* (since all *putObject* requests inside the VPC are allowed by the first allow statement). Such a request would indeed violate the proposed property. On the other hand, if ZELKOVA shows that the input policy implies the policy in Fig. 9 then the upper bound is establish and the proposed property holds true.

```
(( allow,
   principal  :  *,
   action     :  sendMessage,
   resource   :  *,
   condition  :  (ArnEquals, aws:sourceArn, mytopic)))
```

(a)

```
(( allow,
   principal  :  *,
   action     :  sendMessage,
   resource   :  *,
   condition  :  (ForAllValues:ArnEquals, aws:sourceArn, mytopic)))
```

(b)

Fig. 10. Policies constrained by `aws:sourceArn`. (a) Policy does not allow world writability. (b) `ForAllValues` semantics allow world writability.

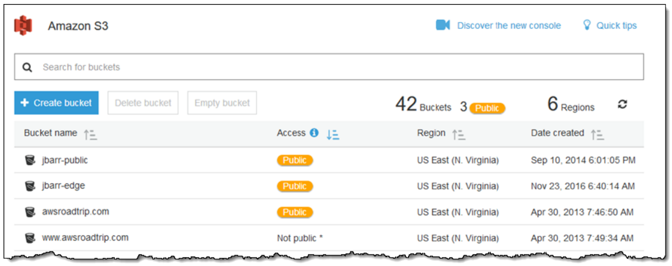
B. Security best-practices

ZELKOVA supports several built-in checks that can be leveraged to check a variety of security best-practices. Examples of these include checking whether a policy allows world accessibility for services such as Amazon S3, Amazon SQS, Amazon SNS, Amazon Glacier, Amazon Elasticsearch, and AWS Lambda. These AWS services provide compute, storage, messaging, and search capabilities. These checks are used internally by AWS to check adherence to security best practices and also available to external customers through services such as Amazon Macie, AWS Config, AWS Trusted Advisor, and the Amazon S3 console. The built-in checks provide greater security assurances without requiring the users to define the properties.

Consider the case of Amazon SQS, a fully managed message queueing service. ZELKOVA provides a built-in check for whether an Amazon SQS policy is world accessible. Fig. 10(a) shows an example SQS policy which which allows *sendMessage* to any resource by any principal, predicated on a condition. The condition restricts the source (`aws:sourceArn`) of the message to be a specific source (*mytopic*). A similar policy is shown in Fig. 10(b). Here, the operator `ForAllValues:ArnEquals` is applied to the condition `aws:sourceArn` whose value is restricted to *mytopic*. The semantics for the operator prefix `ForAllValues` states that if the condition *aws:sourceArn* exists, then its value is *mytopic*. The SMT formula for that is as follows:

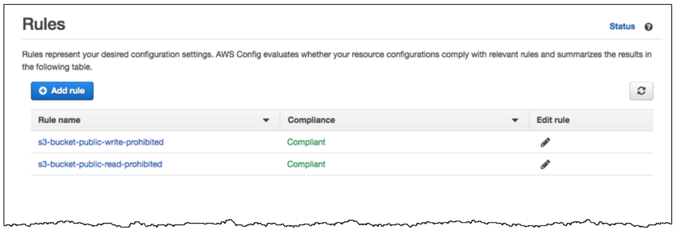
$$\mathtt{awsSourceArnExists} \implies (\mathtt{awsSourceArn} = \mathit{mytopic})$$

When a request context does not have the condition key `aws:sourceArn` set, the above formula is true. Thus any



Bucket name	Access	Region	Date created
jbarr-public	Public	US East (N. Virginia)	Sep 10, 2014 6:01:05 PM
jbarr-edge	Public	US East (N. Virginia)	Nov 23, 2016 6:40:14 AM
awsroadtrip.com	Public	US East (N. Virginia)	Apr 30, 2013 7:46:50 AM
www.awsroadtrip.com	Not public *	US East (N. Virginia)	Apr 30, 2013 7:49:34 AM

Fig. 11. S3 Console: Buckets marked Public or Not Public using ZELKOVA checks.



Rules represent your desired configuration settings. AWS Config evaluates whether your resource configurations comply with relevant rules and summarizes the results in the following table.

Rule name	Compliance	Edit rule
s3-bucket-public-write-prohibited	Compliant	
s3-bucket-public-read-prohibited	Compliant	

Fig. 12. Rules in AWS Config that check public read is prohibited (s3-bucket-public-read-prohibited) and public write is prohibited (s3-bucket-public-write-prohibited) for an S3 bucket using ZELKOVA.

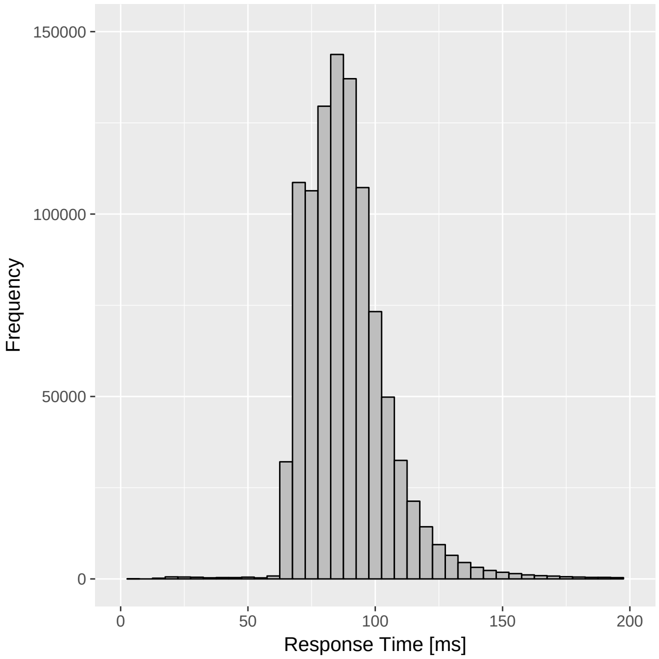


Fig. 13. Performance of ZELKOVA on one million random policy questions

principal can send a message to the SQS queue. The ZELKOVA built-in check for SQS world accessibility correctly marks Fig. 10(a) as not world accessible and Fig. 10(b) as world accessible.

VI. INDUSTRIAL EXPERIENCE

ZELKOVA is integrated in many AWS services including Amazon S3, AWS Config, Amazon Macie, AWS Trusted Advisor, and Amazon GuardDuty. In addition, ZELKOVA is used by an internal security auditor by the AWS Security team.

The Amazon S3 Console is a web-based interface where users can provision buckets; manage buckets, objects, and folders; and set permissions to buckets and objects. A recent release of the console added a view showing whether a bucket is publicly accessible (`Public`) or not (`Not Public`). The underlying check is performed by ZELKOVA. Fig. 11 shows an example of this view.

AWS Config currently supports several managed rules based on ZELKOVA[4], such as a check for AWS Lambda Functions granting unrestricted access, a check for S3 buckets granting unrestricted read access, a check for S3 buckets granting unrestricted write access, deny *putObject* requests that do not have server side encryption, and deny actions that do not allow https traffic. Config will trigger a new ZELKOVA-based check whenever a new resource is created or the policy attached to it is changed. Using the Config console, customers can determine compliance of their S3 buckets against these rules, as shown in Fig. 12, and receive notifications when permissions change or view the permissions history in the console. The checks available in the Amazon Macie and AWS Trusted Advisor services are similar to those in AWS Config.

ZELKOVA is used by internal security auditing tools, owned by the AWS Security team, that scan all *internal* AWS accounts to check for unintended configurations of resources. Internal accounts are all AWS accounts owned by the AWS development teams and personnel. These include policies attached to various resources such as S3 buckets, SQS queues, SNS topics, Glacier Vaults, KMS Keys, ElasticSearch Domains, and AWS Lambda Functions. The security auditing tools periodically scan all the resources and check compliance of the resources policies according to the security best practices. Violations of checks are automatically ticketed as discovered, assigned to the owners, and automatically resolved when policies are fixed. The auditing tools require no manual intervention by the security engineering team.

While the checks available in Amazon Macie, AWS Config, Amazon S3 Console, and AWS Trusted Advisor check safety properties, the ZELKOVA integration in Amazon GuardDuty checks for an availability property. ZELKOVA ensures that the requisite permissions are enabled in a user's policy when they are on-boarding onto the service.

A. Implementation

ZELKOVA runs on AWS Lambda, a serverless computing platform that runs applications without users needing to provision or manage servers. The input to ZELKOVA is a JSON structure that consists of the policies that are being compared, or a policy and the name of a built-in ZELKOVA check. The response from ZELKOVA is also a JSON structure with the answer to the query. For a comparison of policies, it returns

---

[4] https://docs.aws.amazon.com/config/latest/developerguide/managed-rules-by-aws-config.html

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

whether the first policy in the payload is *less permissive*, *more permissive*, *equivalent*, or *incomparable* with respect to the second policy in the payload. For each of the built-in checks, ZELKOVA takes a policy and returns *true* or *false* based on whether the check is satisfied. If ZELKOVA is unable to handle any construct in the policy or the solver times out, it returns *unknown*.

ZELKOVA uses the solvers Z3, Z3AUTOMATA, and CVC4 in the backend to solve queries [16], [17]. The solvers provide a combination of string, regular expression, bit vector, and integer comparison theories. ZELKOVA invokes the solvers in parallel and returns the results as soon as one of the solvers provides the answer. We use the Z3 solver with its traditional sequence string solver. Experiments with other solvers such as Z3Str3 [18] and other automata-based solvers [19] is part of our future work.

B. Usage statistics

The total number of invocations of ZELKOVA ranges from a few million to tens of millions in a single day. The number of invocations varies based on the services invoking ZELKOVA. Certain services invoke ZELKOVA at some regular cadence, *e.g.*, the internal security auditing tools, while other services, *e.g.*, AWS Config, invokes ZELKOVA when a change is detected in the policies.

Fig. 13 shows the performance of ZELKOVA on one million randomly selected policy questions. These contain both policy comparisons and built-in checks. The total time includes time to parse the input JSON, encode the policies into SMT, perform the check, and construct the resulting JSON that is returned. The y-axis represent the count, i.e., number of policies solved within the time. The graph shows that 99% of policies are solved within 160 milliseconds.

VII. CONCLUSION

In this paper, we have presented a formalization of the AWS policy language that controls access to resources. This is the first instance of formalizing the AWS policy language as SMT formulas. The advantage of this approach is that it allows us to use off-the-shelf SMT solvers to verify safety and availability properties. Given the distributed nature of the policy language where different services establish their own list of condition keys, this work provides a single consolidated service to reason about the semantics of policies applicable across different services in AWS. The previous state of the art in policy checks for AWS services used syntactic checks for policies. Alternatively, given a concrete request context, the policy evaluation engine allows users to test access control. In contrast, our formalization into SMT provides the ability to soundly reason about properties of a policy for all valid request contexts.

For customers of AWS services, ZELKOVA provides deeper insights into the policy language, its semantics, and its implications. The tool enables customers to automatically maintain their security posture. For people in the SMT and verification community, this work shows how SMT can verify properties of a complex industrial policy language that is used by millions on a daily basis. Moreover, this work is one of the largest and most widespread uses of formal methods in industry.

There are two avenues of future work. One avenue is to improve the existing functionality provided in ZELKOVA. This includes further work on Z3AUTOMATA to make it more competitive. The second avenue is to enhance the functionality of the ZELKOVA engine itself. For example, we want to add support in ZELKOVA to return to the user a concrete request context using the model generated by the SMT solver when performing the check. The concrete request context will provide information to the user on why a certain check passed or failed. We also want to add support for recommending policy repairs in cases when the policy fails a certain check.

REFERENCES

[1] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *IJCAR*, vol. 4130. Springer, 2006, pp. 632–646.

[2] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 196–205.

[3] D. P. Guelev, M. Ryan, and P.-Y. Schobbens, "Model-checking access control policies," in *ISC*, vol. 3225. Springer, 2004, pp. 219–230.

[4] G. Hughes and T. Bultan, "Automated verification of access control policies using a SAT solver," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 6, pp. 503–520, 2008.

[5] G. Kolaczek, "Specification and verification of constraints in role based access control," in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003x2. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*. IEEE, 2003, pp. 190–195.

[6] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: A logic-based approach to distributed authorization," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, no. 1, pp. 128–171, 2003.

[7] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management, applied to electronic health records," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 2004, pp. 139–154.

[8] J. DeTreville, "Binder, a logic-based security language," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 2002, pp. 105–113.

[9] T. Jim, "SD3: A trust management system with certified evaluation," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 106–115.

[10] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *Padl*, vol. 3. Springer, 2003, pp. 58–73.

[11] A. Anderson, A. Nadalin, B. Parducci, D. Engovatov, H. Lockhart, M. Kudo, P. Humenn, S. Godik, S. Anderson, S. Crocker *et al.*, "Extensible access control markup language (xacml) version 1.0," *OASIS*, 2003.

[12] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati, "An algebra for composing access control policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, no. 1, pp. 1–35, 2002.

[13] E. Bertino, P. A. Bonatti, and E. Ferrari, "TRBAC: A temporal role-based access control model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 191–233, 2001.

[14] K. Jayaraman, N. Bjorner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft Research, Tech. Rep. MSR-TR-2014-102, 2014.

[15] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, New York, 1971, pp. 189–196.

[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

ISBN: 978-0-9835678-8-2.

[17] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[18] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," *Formal Methods in Computer-Aided Design FMCAD 2017*, vol. 10, no. 14, p. 55, 2017.

[19] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 255–272.

ISBN: 978-0-9835678-8-2.

A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4

Heiko Becker*, Nikita Zyuzin*, Raphaël Monat†, Eva Darulova*, Magnus O. Myreen‡ and Anthony Fox§
*MPI-SWS, †ENS Lyon, ‡Chalmers University of Technology, §University of Cambridge
*{hbecker,zyuzin,eva}@mpi-sws.org, †raphael.monat@ens-lyon.org, ‡myreen@chalmers.se, §anthony.fox@arm.com

Abstract—**Being able to soundly estimate roundoff errors of finite-precision computations is important for many applications in embedded systems and scientific computing. Due to the discrepancy between continuous reals and discrete finite-precision values, automated static analysis tools are highly valuable to estimate roundoff errors. The results, however, are only as correct as the implementations of the static analysis tools. This paper presents a formally verified and modular tool which fully automatically checks the correctness of finite-precision roundoff error bounds encoded in a certificate. We present implementations of certificate generation and checking for both Coq and HOL4 and evaluate it on a number of examples from the literature. The experiments use both in-logic evaluation of Coq and HOL4, and execution of extracted code outside of the logics: we benchmark Coq extracted unverified OCaml code and a CakeML-generated verified binary.**

I. INTRODUCTION

Numerical programs, common in scientific computing or embedded systems, are often implemented in finite-precision arithmetic. This approximation of real numbers inevitably introduces roundoff errors, potentially making the computed results unacceptably inaccurate. The discrepancy between discrete finite-precision arithmetic and continuous real arithmetic make accurate and sound error estimation challenging. Automated tool support is thus highly valuable.

This fact was already recognized previously and resulted in a number of static analysis techniques and tools [18, 38, 11, 14] for computing sound worst-case absolute error bounds on numerical errors. The results of such static analysis tools are, however, only as correct as the tools' implementation.

Some of these tools provide independently checkable formal proofs, however we found that none of the current certificate producing tools, FPTaylor [38], PRECiSa [32] and Gappa [14] go far enough. FPTaylor produces a proof certificate in HOL-Light, relying on an in-logic decision procedure [37]. Its analysis is specific to floating-point arithmetic and does not support other finite precisions. PRECiSa and Gappa generate a proof certificate by instantiating library theorems, explicitly encoding verification steps. Any tool that explicitly encodes verification steps, or is to be used interactively [15, 35] requires expert knowledge in IEEE754 floating-point semantics [21] or formal verification; in contrast our goal is to make our tool usable by non-experts. Finally, in-logic verification of certificates can often become unreasonably slow.

This paper describes a new fully automated tool, called *FloVer*, which checks proof certificates of finite-precision roundoff error bounds generated by static analysis tools. Certificates checked by FloVer encode only the minimal static analysis result, and thus using FloVer does not require formal verification expertise. Separately from FloVer, we implement fully automated certificate generation in the static analysis tool Daisy [13], demonstrating our envisioned tool-chain.

FloVer supports straight-line arithmetic kernels, floating-point as well as fixed-point arithmetic, mixed-precision evaluation (including floating-point type inference), and local variable declarations. For floating-point expressions, FloVer proves correctness of each analyzed expression with respect to the concrete bit-level IEEE754 floating-point semantics [21]. Our tool is formally verified in both Coq and HOL4. A succesful run of FloVer shows that the encoded roundoff error is a valid upper bound and that the analyzed function can be run without any errors (e.g. division-by-zero).

In order to handle both floating-point and fixed-point arithmetic, FloVer supports a forward dataflow static analysis. FloVer is furthermore built modularly to allow reusability and easy extensions, and supports dataflow analysis with both interval and affine arithmetic abstract domains.

We have implemented and verified FloVer in two theorem provers to be able to connect to projects in both provers and thereby make FloVer widely applicable. In Coq, we hope to link to the CompCert compiler [27] and CertiCoq [2]; and in HOL4 we already link to CakeML [39].

The connection to CakeML allows us to provide efficient certificate checking: using the CakeML toolchain [39, 34] we produce a verified binary of our certificate checker. At the time of writing, CertiCoq was not capable of extracting our checker functions, thus we extract an unverified binary from Coq and compare its perfomance with the verified CakeML binary.

Our evaluation on standard benchmarks from embedded systems and scientific computing shows that roundoff errors verified by FloVer are competitive with the state of the art, and extracted certificate checking times are significantly faster than in-logic verification.

Contributions

- We explain our modular, fully automated and self-contained approach to certification of absolute finite-precision roundoff error bounds (Section IV and V).
- We implement and prove FloVer correct in both Coq and HOL4. The sources are available at https://gitlab.mpi-sws.org/AVA/FloVer.

ISBN: 978-0-9835678-8-2.

- We are the first to provide an efficient and verified way of checking finite-precision error certificates by extracting a verified binary version of FloVer from HOL4 (Section VI).
- We experimentally evaluate (in Section VII) implementations of FloVer on examples from the literature. The results are competitive and show that our approach to certificate checking is feasible. During our experiments, we found a subtle bug in the Daisy static analyzer.

II. OVERVIEW

In this section, we give a high-level overview of our certificate generation and checking approach. The next section provides the necessary background on finite precision arithmetic and static dataflow analysis for roundoff errors. Section IV describes the technical details of FloVer.

A certificate (in Coq or HOL4) checked by FloVer encodes the *result* of a forward dataflow static analysis of roundoff errors, but not the analysis or correctness proofs themselves. For each analyzed arithmetic expression (consisting of $+, -, *, /$, `FMA`, and local variables), the certificate contains:

- the expression f, as an abstract syntax tree (AST)
- a precondition P, specifying the domain (interval) of all input variables
- a (possibly mixed-precision) type assignment Γ for all input variables and optionally let-bound variables,
- the analysis result which consists of a range $\Phi_{\mathcal{R}}$ and an error bound $\Phi_{\mathcal{E}}$ for each intermediate subexpression

FloVer then checks the analysis result recursively, by verifying for each AST node that the error bound is a sound upper bound on the worst-case absolute roundoff error:

$$\max_{x \in [a,b]} \; |f(x) - \tilde{f}(\tilde{x})| \tag{1}$$

where f and x are the real-valued expression and variable, respectively, and $\tilde{f}$ and $\tilde{x}$ their finite-precision counterparts. The interval $[a, b]$ is the domain of x given by precondition P. Ranges for input variables as well as the analysis result are necessary as (absolute) finite-precision roundoff errors depend on the magnitude of the computed values. In the absence of input ranges, roundoff errors are unbounded in general.

FloVer splits the certification into several subtasks and runs separate validator functions (see also Figure 1):

- `validRealRange` validates the range result $\Phi_{\mathcal{R}}$,
- `validTypes` infers and checks types (given in Γ) of all subexpressions
- `validErrors` validates the error results $\Phi_{\mathcal{E}}$,
- `validMachineRanges` validates that no overflow and NaN's (not-a-number special values) occur.

We have implemented the validators in both Coq and HOL4 and proven an overall soundness theorem: when all validators return successfully, then the computed error bounds (for each subexpression) are soundly overapproximating the finite-precision roundoff errors.

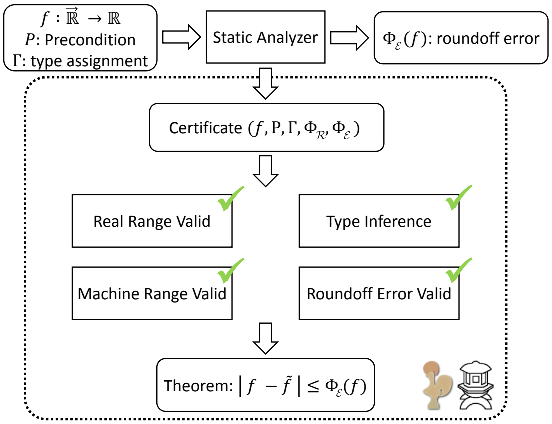


Fig. 1. Overview of the FloVer framework

To verify a certificate, one can run the validator functions in Coq or HOL4 directly. However, while both provers natively support evaluation of functions, this is not particularly efficient. To speed up the certificate checkers, we have used the CakeML in-logic compilation toolchain [34], to extract a *verified* binary from our HOL4 checker definitions. Since the CakeML compiler is fully verified, the binary enjoys the very same correctness guarantees as the certificate checkers implemented in HOL4. Similarly, we have used the extraction mechanism [28] in Coq to extract an, albeit unverified, binary. The binary implementations of the checkers run natively and are thus significantly more efficient, as our experiments in Section VII demonstrate.

III. BACKGROUND

A. Finite-Precision Arithmetic

FloVer uses a general abstraction for finite-precision arithmetic relating it to operations on real numbers:

$$x \circ_{fp} y = (x \circ y) + \mathrm{error}(x \circ y, fp) \tag{2}$$

where $\circ \in \{+, -, *, /\}$ and $\circ_{fp}$ denotes the corresponding finite-precision operation at type fp. Function $\mathrm{error}(e, fp)$ computes the error from representing the real value e in the finite-precision type fp. An input x may not be representable in finite-precision arithmetic, and thus FloVer considers an initial error on the input: $|x - \tilde{x}| \leq \mathrm{error}(x, fp)$.

For floating-point arithmetic, we assume IEEE754 [21] semantics with rounding-to-nearest rounding mode and the standard abstraction of arithmetic operations:

$$\mathrm{error}(e, fp) = e * \delta \quad |\delta| \leq \varepsilon_{fp} \tag{3}$$

Constant ε_{fp} is the so-called machine epsilon for precision fp ($fp = 16$, 32 or 64 bits) and represents the maximum relative error for a single arithmetic operation. In addition to binary operations, FloVer also supports unary negation, which does not incur a roundoff error, and fused-multiply-add instructions, where $\mathrm{FMA}(x, y, z)_{fp} = (x * y + z) + \mathrm{error}(x * y + z, fp)$.

Equation 3 holds under IEEE754 floating-point semantics only for normal floating-point values, and thus FloVer reports ranges containing only subnormals, infinity or not a number (NaN) special values as errors. We discuss the proof of correctness wrt. to IEEE754 semantics in Section V-A.

Fixed-point arithmetic is an alternative to floating-points which does not require dedicated hardware and is thus a common choice in embedded systems. No standard exists, but we follow the common representation [3] of fixed-point values as bit vectors with an integer and a fractional part, separated by an implicit radix point, which has to be precomputed at compile-time. We assume truncation as the rounding mode for arithmetic operations. The absolute roundoff error at each operation is determined by the fixed-point format, i.e. the (implicit) number of fractional bits available, which in turn can be computed from the range of possible values at that operation. Since this information must be computed by any static analysis on fixed-point programs, we encode fractional bits as part of our fixed-point type and rely on the certificate containing a full (unverified) map Γ from expressions to types for fixed-point kernels.

B. Static Dataflow Roundoff Error Analysis

FloVer's range and error validators perform dataflow roundoff error analysis and for this follow the same approach for computing absolute error bounds as Rosa [11], Fluctuat [18], Gappa [14] and Daisy [13].

The magnitude of absolute finite-precision roundoff errors depends on the magnitude of values of all intermediate subexpressions (this can be seen e.g. from Equation 3). Thus, in order to accurately bound roundoff errors, the analysis first needs to be able to bound the ranges of all (intermediate) expressions.

At a conceptual level, dataflow analysis computes roundoff error bounds in two steps:

range analysis computes sound range bounds for all intermediate expressions,

error analysis propagates errors from subexpressions and computes the new worst-case roundoffs using the previously computed ranges.

Both steps are performed recursively on the AST of the arithmetic expressions. A side effect of this separation is that it provides us with a modular approach: we can choose different range arithmetics with different accuracy-efficiency tradeoffs for ranges and errors. Common choices for range arithmetics are interval arithmetic (IA) [31] and affine arithmetic(AA) [16].

IV. CERTIFICATION OF ERROR ANALYSIS RESULTS

Next, we focus on the technical details of our certificate checking. The certificates in Coq, HOL4 and for the extracted binaries are structurally the same and only differ in syntax. Figure 2 shows a sample structure of a certificate in Coq and HOL4, including the types of encoded results. Γ represents a type assignment to all free variables in the analyzed function. Expressions (of type `expr`) are parametric in the type of constants. $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ map each AST node of the analyzed function to an interval and a positive (absolute) error bound represented by a single fraction, respectively. We discuss the differing types of $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ in Section V-C.

The validator functions, which check the certificate, also have the same structure in both Coq and HOL4 and we describe them here independently of the particular prover.

A. Checking Range Analysis Results

The range validator is implemented in the function `validRealRange`$(e, P, \Phi_{\mathcal{R}})$ which takes as input an expression e, the precondition P, which captures the constraints on the input variables, and the real-valued ranges which are to be checked in $\Phi_{\mathcal{R}}$. `validRealRange` verifies by structural recursion on the AST that for each subexpression e' of e, $\Phi_{\mathcal{R}}(e')$ returns a sound enclosure of the true range, which is computed inside the theorem prover with interval or affine arithmetic. That is, we check the ranges in $\Phi_{\mathcal{R}}$ by effectively recomputing them inside the prover.

Since FloVer supports let-bindings in the input program to reuse evaluation results, both at runtime as well as in the certificate validator, we extend `validRealRange` to handle let-bound variables without recomputing results.

B. Mixed-precision Support

Mixed-precision evaluation allows different arithmetic operations to be executed in different precisions. This often allows to speed up computations as evaluation in lower precisions is usually faster. Instead of requiring e.g. uniform 64-bit precision, each subexpression in FloVer can be evaluated in 16, 32 or 64 bit floating-point precisions (each with the corresponding machine epsilon ε_p). FloVer supports the same semantics as C and Scala: for two operands with different precisions, the lower one is implicitly cast to the higher precision, but an explicit cast is required when decreasing precision (e.g. when assigning a 64 bit value to a 32 bit variable).

The typing environment Γ assigns a machine precision to every free variable of the analyzed expression. We further require any constant in the AST, as well as casts to be annotated with its (resulting) precision.

For floating-point precisions, FloVer infers the remaining types automatically, i.e. the user only has to provide this necessary minimal information, and in particular does not need to annotate all intermediate operations.

We can reuse the existing infrastructure to support fixed-point arithmetic. A fixed-point type in FloVer is then represented as a pair of word length w and number of fractional bits f. For fixed-point precisions, FloVer avoids recomputing the fractional bits and thus relies on the information being encoded in Γ.

When checking a certificate, FloVer computes a full type-map $\Phi_{\mathcal{T}}$ from the (partial) type map Γ to avoid recomputing results. To this end we implement the function `validTypes`(Γ, e). The function returns $\Phi_{\mathcal{T}}$ if and only if all types encoded in Γ are valid types for their respective subexpressions. We reuse $\Phi_{\mathcal{T}}$ in both the error validator and the machine range validator.

ISBN: 978-0-9835678-8-2.

```
Definition f:cmd Q := <AST f>.
(* Type assignment for free variables *)
Definition Gamma: expr Q → option mType := <Γ>.
(* range constraints on free variables of f *)
Definition Precondition: nat → (Q ∗ Q) := <P>.
(* map from sub-expressions to ranges and errors *)
Definition AbsEnv: expr Q → option ((Q ∗ Q) ∗ Q) :=
  <Φℛ,Φℰ>.

Theorem CertificateCheckingSuccedes =
 CertificateChecker f Gamma Precond AbsEnv = true.
Proof.
  vm_compute; auto.
Qed.
```

```
val f_def = Define ‘f: real cmd = <AST f>‘;
(* Type assignment for free variables *)
val Gamma_def = Define
  ‘Gamma: real expr → mType option = <Γ>‘;
(* range constraints on free variables of f *)
val Precondition_def = Define ‘
  P: num → (real ∗ real) = <P>‘;
(* map from sub-expressions to ranges and errors *)
val AbsEnv_def = Define ‘
  AbsEnv: real expr → ((real ∗ real) ∗ real) option =
    <Φℛ,Φℰ>‘;
val CertificateCheckingSucceeds = prove (
 ‘‘CertificateChecker f Gamma Precond AbsEnv‘‘,
 daisy_eval_tac);
```

Fig. 2. Certificate structure with corresponding types in Coq (left) and HOL4 (right)

C. *Checking Error Analysis Results*

The error validator `validErrors`$(e, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}})$ takes as input the expression e, a type assignment to subexpressions $\Phi_{\mathcal{T}}$, the range analysis result $\Phi_{\mathcal{R}}$ and the error analysis result $\Phi_{\mathcal{E}}$, which is to be checked.That is, `validErrors` assumes that the ranges and types have been verified independently. As for the range validator, we extend `validErrors` to reuse results of let-bound variables. The validator function checks by structural recursion on the AST of e that for each subexpression e' of e, $\Phi_{\mathcal{E}}(e')$ is a sound upper bound on the absolute roundoff error.

For constants and variables, the error bounds are straightforwardly derived using Equation 2 and the range analysis result. For arithmetic operations, the error check is more involved. Using Equation 2, Equation 1 and the triangle inequality, we obtain for an addition:

$$\begin{aligned} |(e_1 + e_2) - (\tilde{e}_1 +_{fp} \tilde{e}_2)| \leq& \\ & |e_1 - \tilde{e}_1| + |e_2 - \tilde{e}_2| + \mathrm{error}((\tilde{e}_1 + \tilde{e}_2), fp) \end{aligned} \tag{4}$$

$|e_1 - \tilde{e}_1|$ and $|e_2 - \tilde{e}_2|$ are the roundoff errors of the operands, which are propagated simply by addition. $\mathrm{error}((\tilde{e}_1 + \tilde{e}_2), fp)$ is the new roundoff error commited by the addition at precision fp. The new roundoff error depends on the magnitude of the operands and thus on the ranges of $\tilde{e}_1$ and $\tilde{e}_2$.

The computation of an upper bound to Equation 4 then uses the range analysis result from $\Phi_{\mathcal{R}}$, the already verified error bounds on the subexpressions e_1 and e_2 in $\Phi_{\mathcal{E}}$, and basic properties of range arithmetic.

Similar bounds can be derived for the other arithmetic operations. However, for multiplication and division, the propagation of errors is more involved. For $e_1 * e_2$ we obtain $|(e_1 * e_2) - (\tilde{e_1} *_{fp} \tilde{e_2})| \leq |e_1 * e_2 - \tilde{e_1} * \tilde{e_2}| + \mathrm{error}(\tilde{e_1} * \tilde{e_2}, fp)$ and similarly for division:

$$\begin{aligned} |(e_1/e_2) - (\tilde{e_1}/_{fp}\tilde{e_2})| \leq& \\ & |e_1 * (1/e_2) - \tilde{e_1} * (1/\tilde{e_2})| + \mathrm{error}(\tilde{e_1} * 1/\tilde{e_2}, fp) \end{aligned}$$

FloVer checks whether a division by zero may occur during the execution of the analyzed function under the real-valued as well as the finite-precision semantics. If it detects that a division by zero can occur in any of the executions, certificate checking fails.

D. *Supported Range Arithmetics*

FloVer currently supports interval arithmetic (IA) [31] in both provers and affine arithmetic (AA) [16] in Coq to check real-valued ranges. The support for AA in the error validator in Coq as well as the HOL4 development in general is currently work in progress. Arithmetic operations in IA are efficiently computed as: $x \circ^{\#} y = [\min(x \circ y), \max(x \circ y)], \quad \circ \in \{+, -, *, /\}$. IA cannot track correlations between variables (e.g. it cannot show that $e_1 - e_1 \in [0, 0]$). Affine arithmetic is a simple relational analysis which tracks linear correlations and thus computes ranges for linear operations exactly (like the $e_1 - e_1$); for nonlinear operations it nonetheless has to compute an over-approximation.

V. SOUNDNESS

We have proven in both Coq and HOL4 that it suffices to run the validator functions on a certificate to show a) that the static analysis result is correct, and b) that the analyzed function will always evaluate to a finite value. The overall soundness proof relates a succeeding run of the validators `validTypes`, `validRealRange`, `validMachineRanges` and `validErrors` to the semantics of the analyzed function.

We have formalized the semantics of functions according to Equation 2. The rule for binary addition, for instance, is

$$\frac{\begin{array}{c} m_+ = m_1 \sqcup m_2 \\ \Phi_{\mathcal{T}}(e_1) = m_1 \quad \Phi_{\mathcal{T}}(e_2) = m_2 \quad \Phi_{\mathcal{T}}(e_1 + e_2) = m_+ \\ (e_1, E, \Phi_{\mathcal{T}}) \Downarrow (v_1, m_1) \qquad (e_2, E, \Phi_{\mathcal{T}}) \Downarrow (v_2, m_2) \end{array}}{(e_1 + e_2, E, \Phi_{\mathcal{T}}) \Downarrow ((v_1 + v_2) + \mathrm{error}(v_1 + v_2,\ m_+))}$$

E is the environment tracking values of bound variables, and Γ tracks precisions of variables. $(e_1, E, \Phi_{\mathcal{T}}) \Downarrow (v_1, m_1)$ means that expression e_1 big-step evaluates for the variable environment E and the type assignment $\Phi_{\mathcal{T}}$ to value v_1 in precision m_1. $m_1 \sqcup m_2$ is an upper bound operator on precisions, returning the most precise of m_1 and m_2.

Real-valued executions map every variable, constant and cast operation to infinite (real-valued) precision, which we denote by $m = \infty$. The rules for subtraction, multiplication, division, casts, and FMA's are defined analogously. Unary negation does not introduce a new roundoff error and keeps the precision of the operand.

Analogously to expressions, we will use E to refer to the idealized real-valued environment and $\tilde{E}$ for the finite-precision environment. The overall soundness theorem is then

Theorem 1. *Let f be a real-valued function, E a real-valued environment, $\tilde{E}$ its finite-precision counterpart, P a precondition constraining the free variables of f, Γ a map from all free variables of f to a precision, $\Phi_{\mathcal{R}}$ a range analysis result, $\Phi_{\mathcal{T}}$ a type-map and $\Phi_{\mathcal{E}}$ an error analysis result. Then*

$$\begin{gathered} E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E} \wedge \\ \mathit{validTypes}(\Gamma, f) = \Phi_{\mathcal{T}} \ \wedge \mathit{validRealRange}(f, P, \Phi_{\mathcal{R}}) \ \wedge \\ \mathit{validMachineRanges}(f, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}}) \ \wedge \\ \mathit{validErrors}(f, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}}) \implies \\ \exists\, v\, \tilde{v}_1\, m_1.\ (f, E, \Phi_{\mathcal{T}}) \Downarrow (v, \infty) \ \wedge (\tilde{f}, \tilde{E}, \Phi_{\mathcal{T}}) \Downarrow (\tilde{v}_1, m_1) \ \wedge \\ (\forall \tilde{v}_2\, m_2.\ (\tilde{f}, \tilde{E}, \Phi_{\mathcal{T}}) \Downarrow (v_2, m_2) \Rightarrow |v - \tilde{v}_2| \leq \Phi_{\mathcal{E}}(f)) \end{gathered}$$

The assumption $E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E}$ states that the real-valued environment E and the finite-precision environment $\tilde{E}$ agree up to a fixed δ on the values of the variables in the sets $\mathcal{V}$ and $\mathcal{D}$. We give the full explanation when explaining soundness of the error validator. To prove the theorem, we have split the proof into separate soundness proofs for each validator function. Each theorem is shown by structural induction on e.

a) Type Validator: Giving the full type map $\Phi_{\mathcal{T}}$ is tedious to do for a user. FloVer thus requires only annotations for casts, constants and (let-bound) variables, and infers the remaining types ($\Phi_{\mathcal{T}}$) fully automatically for floating-point expressions. For fixed-point types only, we require Γ to be a complete map since we rely on the fractional bits to be inferred externally.

Soundness of the type inference `validTypes` means that when $\Phi_{\mathcal{T}}(e) = m_t$ and evaluation of e gives value v and precision m_v, then $m_t = m_v$. Thus, we need not recompute type information once the type map has been computed and reuse it in the other validators.

b) Real Range Validator: For `validRealRange`, the soundness theorem proves that if E binds variables in e to values that are within the range given by the precondition P, then e evaluates for environment E to v under a real-valued semantics and v is contained in $\Phi_{\mathcal{R}}(e)$.

c) Machine Range Validator: We prove that whenever `validMachineRanges` succeeds on expression e, valid type-map $\Phi_{\mathcal{T}}$ and valid error map $\Phi_{\mathcal{E}}$, then any evaluation of e results in a finite, representable value for the type of e in $\Phi_{\mathcal{T}}$.

For floating-point precisions this means that v is a finite value according to IEEE754 (i.e. either 0, subnormal or normal). For fixed-point precisions with word size w and f fractional bits, this means that v is within the range of representable values ($|v| \leq \frac{2^{w-1}-1}{2^f}$) and no overflow occurs (i.e. the fractional bits were correctly inferred).

FloVer uses Equation 3 to compute an error for floating-point precisions which is only valid in the presence of IEEE754 *normal* numbers or 0. We note that the roundoff error of the *biggest representable subnormal number* is smaller than the roundoff error of *normal numbers* in general. We add this condition as a check to function `validMachineRanges` by checking that the floating-point range contains at least one normal number.

d) Error Validator: If `validErrors`$(e, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}})$ succeeds, and e evaluates to v, then we want to show that $\tilde{e}$ evaluates to $\tilde{v}$, and that $|v - \tilde{v}| \leq \Phi_{\mathcal{E}}(e)$. The challenge in this proof lies in the fact that we reason about two different executions of similar expressions, e and $\tilde{e}$.

Given a free variable x in the analyzed expression e, the value $E(x)$ may not be representable as a finite-precision value. Thus the values for the related variables x and $\tilde{x}$ will not in general agree. This is the case for every free variable occurring in e. Additionally, the roundoff error of any variable depends on its precision. As a consequence we introduce an inductive approximation relation $\sim_{(\mathcal{V},\Phi_{\mathcal{T}})}$ between values provided by E and $\tilde{E}$ for variables in $\mathcal{V}$ so that we can prove the error bound. Given $E \sim_{(\mathcal{V},\Phi_{\mathcal{T}})} \tilde{E}$, both environments are defined for every variable $v \in \mathcal{V}$. In addition, the difference between $E(v)$ and $\tilde{E}(\tilde{v})$ at precision p is upper bounded by $\text{error}(v, p)$, where p is $\Phi_{\mathcal{T}}(v)$. In the proofs we instantiate $\mathcal{V}$ by the free variables of the analyzed expression. Two empty environments are trivially related under the empty set $((\_ \mapsto \bot) \sim_{(\emptyset,\, \Phi_{\mathcal{T}})} (\_ \mapsto \bot))$ and for free variables we have:

$$\text{FreeVar}\ \dfrac{\begin{array}{c} E \sim_{(\mathcal{V},\Phi_{\mathcal{T}})} \tilde{E} \qquad x \notin \mathcal{V} \\ \Phi_{\mathcal{T}}(x) = m \qquad |\, v - \tilde{v}\, | \ \leq \ \text{error}(v, m) \end{array}}{(E\, [x \mapsto v]) \sim_{(\{x\} \cup \mathcal{V}, \Phi_{\mathcal{T}})} (\tilde{E}\, [\tilde{x} \mapsto \tilde{v}])}$$

To prove soundness for let-bindings, we will extend the relation with a rule for defined variables later.

$\Phi_{\mathcal{E}}$ maps expressions to rationals, representing absolute error bounds. FloVer computes error bounds from intervals from $\Phi_{\mathcal{R}}$ and the error bounds on subterms. The propagation errors for multiplication and division depend on both the real-valued and the float-valued ranges. Therefore the soundness proof requires solving 16 and 32 sub-cases for multiplication and division, respectively.

e) Let-Bindings: To extend the soundness proofs to let-bindings, we have to check that the analyzed function f is in SSA form (since $\Phi_{\mathcal{T}}$, $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ are maps, variables cannot be redefined). For this we use the formalization of SSA defined in the LVC framework [36]. Furthermore, we adapt the approximation relation $\sim$ to include let-bound variables:

$$\text{DefinedVar}\ \dfrac{\begin{array}{c} E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E} \qquad x \notin \mathcal{V} \cup \mathcal{D} \\ \Phi_{\mathcal{T}}(x) = m \qquad |\, v - \tilde{v}\, | \ \leq \ \Phi_{\mathcal{E}}\,(x) \end{array}}{(E\, [x \mapsto v]) \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\{x\} \cup \mathcal{D}, \Phi_{\mathcal{T}})} (\tilde{E}\, [\tilde{x} \mapsto \tilde{v}])}$$

Set $\mathcal{D}$, tracks variables added to both environments using let-bindings and $\Phi_{\mathcal{E}}$ is the error analysis result. The sets $\mathcal{D}$ and $\mathcal{V}$ are used to distinguish whether a variable x is free or let-bound.

f) Using Flover: We obtain the overall soundness of FloVer (Theorem 1) as the conjunction of the results of the functions `validTypes`, `validRealRange`, `validMachineRanges` and `validErrors`. Theorem 1 holds only if checking of the certificate succeeds. If the static analysis result in a certificate is incorrect, e.g. a computed range or roundoff error is incorrect, FloVer fails checking the certificate. Our tool can be used by any other roundoff error analysis tool that computes real-valued ranges, roundoff error bounds and knows about variable types. Using FloVer is then as easy as implementing a pretty-printer for this information.

FloVer performs sound dataflow analysis, which necessarily computes an overapproximation of the true roundoff errors. It is thus possible that FloVer cannot verify a certificate even though the error bounds are indeed correct. Different range arithmetics, which influence the accuracy of FloVer's analysis, commit different overapproximations. Thus we use our implementations of IA and AA in Coq in a portfolio approach and run both when checking range analysis results.

A. Connecting FloVer to IEEE754

We connect our formalization to formalizations of IEEE754 floating-point arithmetic in HOL4 [17] and the Flocq library in Coq [6] by proving that if checking the certificate succeeds, we can evaluate the analyzed function using IEEE754 semantics and the roundoff error bound is valid for this execution.

Theorem 2. *Let* $\tilde{f}$ *be a function on 64-bit floating-points and* f *its real-valued counterpart,* E *a real-valued environment,* $\tilde{E}$ *its 64-bit floating-point counterpart,* P *a precondition constraining the free variables of* $\tilde{f}$*,* Γ *a map from all free variables of* $\tilde{f}$ *to 64-bit precision,* $\Phi_{\mathcal{R}}$ *a range analysis result, and* $\Phi_{\mathcal{E}}$ *an error analysis result, Then*

$$\begin{gathered} E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Gamma)} \tilde{E} \wedge \\ \texttt{CertificateChecker}\,(f,P,\Gamma,\Phi_{\mathcal{R}},\Phi_{\mathcal{E}}) \wedge \\ \texttt{IEEEevalAvoidsSubnormals}\,\tilde{f} \implies \\ \exists\, v\, \tilde{v}.\ (f,E) \Downarrow v \ \wedge\ (\tilde{f},\tilde{E}) \Downarrow_{\mathit{IEEE}} \tilde{v} \ \wedge\ |v-\tilde{v}| \le \Phi_{\mathcal{E}}(f) \end{gathered}$$

The proof of Theorem 2 is an extension of FloVer's soundness theorem (Theorem 1). To show that the roundoff error bounds are valid for the IEEE754 operations, we use the soundness theorem of `validMachineRanges` to establish that all values obtained form an evaluation are finite.

The formalization in HOL4 (currently) does support neither cast operations nor reasoning about roundoff errors for subnormal values. Until these are supported, we assume Γ to map every variable to 64-bit double precision and disallow subnormal values to occur during evaluation. To this end, we define the function `IEEEevalAvoidsSubnormals`(e, E), as a temporary workaround. The function returns true only if every subexpression of e evaluates to a normal value or 0.

B. Division Bug Found

We use Daisy [13] to generate certificates for our evaluation. During this, we found a subtle bug in the tool's static analysis of the division operator. The error bounds are only sound in the absence of division-by-zero errors, but only the real-valued range of the denominator was checked for whether it contains zero. It is possible, however, that the real-valued range does *not* contain zero, while the corresponding floating-point range does, essentially due to large enough roundoff errors.

C. Formalization Details

Executions inside FloVer are represented in both Coq and HOL4 as big-step relations using Equation 2. These formalizations do not depend on external libraries. Only the connection to IEEE754 semantics uses external libraries.

Roundoff errors and our theorems relate real-valued executions to finite-precision ones and we thus need a way to represent the numbers and also compute on them. However, the latter is problematic for infinite-precision reals. We use rationals to represent the values in the certificates. To relate these values to the real-valued ($\mathbb{R}$) executions in the theorem statement, we use the fact that rationals are a subset of the `real` type in HOL4, and in Coq we use the translation `Q2R`:$\mathbb{Q} \to \mathbb{R}$ and exploit that our AST is parametric in the constant type by instantiating it with $\mathbb{Q}$ for computations and $\mathbb{R}$ for theorems.

VI. Extracting a Verified Binary with CakeML

Running the range and error checker functions in Coq and HOL4 directly is quite inefficient (see our experiments in Section VII). We have thus extracted a verified binary from our HOL4 checker function definitions, and an unverified binary for Coq. We are aware of the work on certified extraction from Coq in the CertiCoq [2] project, but at the time of writing, the tool could not handle our checker definitions.

We have implemented in HOL4 and Coq an unverified lexer and parser for the encoding of the certificates, which are included in the extracted binaries in both Coq and HOL4.

a) Extracting from HOL4: For extracting a binary from HOL4, we use the CakeML proof-producing synthesis tool [34] which translates ML-like HOL4 functions into deeply embedded CakeML programs that exhibit the same behaviour. In HOL4 we use the `real` type to store the rational bounds in $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$. For each of the arithmetic operations over the `real` type that we used in the HOL4 development, we define a translation into a representation of the arbitrary-precision rationals in CakeML.

CakeML and HOL4 have different notions of equality. Since we perform equality tests in the certificate checkers, we had to prove that our newly defined representation of real numbers respects CakeML's semantics for structural equality. For this purpose, we had to require and prove that our representation of rationals maintains a gcd of one between nominator and denominator.

When translating a HOL4 function into CakeML code, the CakeML toolchain generates preconditions that exclude runtime exceptions, e.g. divisions by zero. We have shown that all generated preconditions are always satisfied, hence the specification theorem for the generated ML code does not have any unproved preconditions left.

Having compiled the CakeML libraries beforehand, we can compile the checking functions into a verified binary in around 90 minutes on the same machine as we used for the experiments in Section VII. Checking the certificate with the

Benchmark	FloVer		FPTaylor
	interval	affine	
ballbeam	2.141e-12	2.141e-12	1.746e-12
bspline1	1.517e-15	1.601e-15	5.149e-16
bspline2	1.406e-15	1.448e-15	5.431e-16
bspline3	1.295e-16	1.295e-16	8.327e-17
doppler (m)	9.766e-05	7.445e-04	3.111e-05
floudas1	1.052e-12	1.074e-12	5.755e-13
floudas26	7.292e-13	7.292e-13	7.740e-13
floudas33	3.109e-15	3.109e-15	6.199e-13
himmilbeau (m)	4.876e-04	4.876e-04	3.641e-04
invertedPendulum	5.369e-14	5.369e-14	3.843e-14
kepler0 (m)	2.948e-05	2.948e-05	1.758e-05
kepler1 (m)	9.948e-05	9.948e-05	5.902e-05
kepler2 (m)	3.732e-04	3.732e-04	1.433e-04
rigidBody1 (m)	4.023e-05	4.023e-05	2.146e-05
rigidBody2 (m)	6.438e-03	6.438e-03	9.871e-03
traincar1-out1	5.406e-12	5.406e-12	4.601e-12
traincar1-state1	5.421e-15	5.421e-15	4.753e-15
traincar1-state2	8.862e-15	8.862e-15	8.099e-15
traincar1-state3	7.784e-15	7.784e-15	7.013e-15
turbine1 (m)	1.356e-05	1.356e-05	3.192e-06
turbine2 (m)	2.034e-05	2.034e-05	4.970e-06
turbine3 (m)	9.038e-06	9.038e-06	1.671e-06

TABLE I
ROUNDOFF ERRORS VERIFIED BY FLOVER AND FPTAYLOR.

Benchmark	# ops	Daisy	Coq		HOL4	CakeML	OCaml
			Interval	Affine			
ballBeam	7	4.62	3.50	3.26	89.04	<0.01	0.02
invertedPendulum	7	3.62	3.59	3.27	112.61	0.01	0.02
bicycle	13	4.31	4.01	4.08	156.76	0.01	0.04
doppler (m)	17	4.86	5.28	12.21	610.67	0.05	0.02
dcMotor	26	5.19	4.97	4.50	316.75	0.02	0.08
himmilbeau (m)	26	3.52	4.11	4.40	65.48	0.02	0.03
bspline	28	4.21	4.61	4.07	298.44	0.03	0.08
rigidbody (m)	33	5.04	7.14	4.52	88.92	0.03	0.06
science	35	5.64	11.69	567.36	1471.96	0.07	0.07
traincar1	36	4.85	10.87	9.84	932.93	0.07	0.11
batchProcessor	56	6.46	8.49	7.43	997.77	0.06	0.16
batchReactor	58	6.84	11.45	9.53	1117.48	0.07	0.17
turbine (m)	82	5.99	18.69	24.90	4095.56	0.25	0.11
traincar2	89	7.90	29.79	28.58	3967.88	0.23	0.27
floudas	99	7.76	13.99	12.76	565.68	0.14	0.27
kepler (m)	158	4.89	21.56	22.70	3848.75	0.21	0.21
traincar3	168	9.14	68.53	68.14	9594.07	0.58	0.49
traincar4	269	10.6	116.94	115.38	17429.3	1.10	0.77

TABLE II
RUNNING TIMES OF DAISY AND FLOVER IN SECONDS.

binary is then extremely fast, since no theorem prover logic is loaded.

b) Extracting from Coq: Coq natively supports unverified extraction into OCaml code [28]. We used the existing libraries for translating Coq numbers into OCaml's `Big_int` type from the base library. The extracted code is compiled using the OCaml native-code compiler ("ocamlopt") in our experiments.

VII. EVALUATION

To evaluate the performance of FloVer, we have extended the static analyzer Daisy to generate certificates of its analysis. As Daisy already computes all the information that needs to be encoded in a certificate, implementing the certificate generation was similar to implementing a pretty-printer for analysis results (we have switched off a few optimizations, which however do not affect the error bounds significantly). Using the certificate generation, we have evaluated Daisy and FloVer on examples taken from the Rosa [10] and real2float [30] projects. Each benchmark consists of one or more separate functions. Daisy analyzes all functions of one benchmark together and produces one certificate containing a call to the certificate checker for each separate function.

We compare error bounds verified by FloVer with those verified by FPTaylor, as FPTaylor generally computes the most accurate bounds [38, 12]. Furthermore, Rosa [10], Fluctuat [19] and Gappa [14] use the same technique to compute roundoff errors as Daisy and FloVer. We also compare FloVer's certificate checking times with FPTaylor's, as the tool also provides a proof certificate.We note that FPTaylor can compute less precise error bounds with shorter running times, here we opt for the off-the-shelve solution without additional parameters.

a) Accuracy: Table I gives a subset of the roundoff errors certified by FloVer as well as roundoff errors computed by FPTaylor for comparison (we give the full table in our technical report [4]). PRECiSa and Gappa compute similar results; we provide them here for two benchmarks for reference. For the ballBeam benchmark, Precisa and Gappa show an error of 1.085e-07 and 1.240e-12 resp., and for the invertedPendulum benchmark, the errors are 3.531e-12 and 3.217e-14. The focus of FloVer is not to compute the most precise bounds possible, but rather to develop the necessary infrastructure for future extensions. Nevertheless, the roundoff errors verified by it are usually close to those proven by FPTaylor. Benchmarks marked with '(m)' are in mixed-precision, otherwise the roundoff errors are evaluated under uniform double (64 bit) floating-point precision (FPTaylor does not support fixed-point precision).

b) Efficiency: In Table II, we compare running times of in-logic evaluation of FloVer in Coq and HOL4, the *verified* binary extracted with the CakeML toolchain and the *unverified* binary extracted from Coq. For our experiments we used a machine with a four core Intel i3 processor with 3.3GHz, 8 GB of RAM, running Debian 9. For the in-logic evaluation in Coq we show range analysis in interval and affine arithmetic, for all other runs we use interval arithmetic. As for the accuracy evaluation, benchmarks marked with '(m)' are in mixed-precision, double precision otherwise.

In Table II, 'OCaml' refers to the Coq binary compiled with the OCaml native compiler. The '# ops' column gives the number of arithmetic operations in the whole benchmark (summed for all functions) and gives an intuition about the complexity of the benchmark. For all columns, the running

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

times are the end-to-end times measured by the UNIX *time* command in seconds. This time includes parsing and generating the certificate for Daisy, checking the proof that FloVer succeeds for Coq and HOL4 in-logic, and running FloVer in the binaries. The running times for Daisy, Coq and HOL4 are the average running times for a single run over three runs. For the binaries we report the average running time of a single run from 300 executions (due to the small runtime).

We give the running times for FPTaylor's certificate checking in our technical report [4] and note that they are larger, but of the same order of magnitude as our Coq in-logic evaluation. Note that FPTaylor's checker requires either a two hour starting time or external checkpointing. FloVer's certificate checking time for fixed-point arithmetic is similar to floating-point checking; we give the detailed running times in our technical report [4].

The evaluation of FloVer's Coq checker is faster than the evaluation of the HOL4 checker. This is probably because we benefit from Coq's `vm_compute` tactic in the Coq evaluation. The tactic translates terms to OCaml and evaluates them using a virtual machine. A Coq term is reconstructed from the result. HOL4's `EVAL_TAC` instead uses a simple call-by-value evaluation strategy. We further observe that the evaluation using affine arithmetic sometimes is as fast as the one using intervals. We suspect that the reason for this is that the affine arithmetic checker must memorize polynomials for sub-expressions and thus does not recompute them. The interval validator, however, currently does not memorize sub-expressions, but only let-bound variables.

VIII. RELATED WORK

a) Sound Accuracy Analysis: The tools FPTaylor [38], Gappa [14], PRECiSa [32], real2float [30] and VCFloat [35] are most closely related to our work as they formally verify floating-point roundoff errors. Each tool handles mixed-precision floating-point arithmetic, but other features differ slightly between tools. FloVer is the only tool with the combination of support for both Coq and HOL4, floating-point as well as fixed-point arithmetic and two abstract domains, interval and affine arithmetic. FloVer is fully automated and FloVer and FPTaylor are the only tools that generate certificates using in-logic decision procedures. While FPTaylor and PRECiSa handle transcendental functions (which FloVer does not), both tools do not handle fixed-point arithmetic. Gappa has some support for fixed-points, but FloVer is the only tool with formalized affine arithmetic. Finally, FloVer is the first tool to provide *efficient* certificate checking with a verified binary. Fluctuat [18], Gappa++ [29] and Rosa [12] statically bound finite-precision roundoff errors using affine arithmetic [16], but do not provide formal guarantees.

FloVer currently does not handle conditionals and loops. These are—to some extent—supported by Fluctuat [19] and Rosa [12], however not formally verified. PRECiSa [32] provides an initial formalization of these approaches, but scalability is unclear [10, 12]. FloVer furthermore focuses, like most tools, on certifying absolute error bounds. Bounding relative errors is challenging due to the increased complexity as well as due to the issue that often the error is not even well-defined due to an inherent division by zero [23]. Gappa does provide verified relative error support by optimizing a constraint based on Equation 3. This approach has been shown to not provide tight bounds once input ranges and expressions become larger [23]. Finally, note that input ranges are also necessary for computing *concrete* relative error bounds.

b) Sound Verification of Floating-point Computations: Absence of runtime errors in floating-point computations can be shown with abstract interpretation, where different abstract domains have been developed for this purpose [5, 9, 25], which are sound w.r.t. floating-point arithmetic. Jourdan et al. [26] have also formalized some of these abstract domains in Coq. Note, however, that these domains do not quantify the difference between a real-valued and the finite-precision semantics and can only show the absence of runtime errors.

Moscato et al. [33] have built a formalization and implementation of AA for computation of real-valued ranges in PVS. This development does not handle division, which we do. Immler [22] has formalized AA in Isabelle/HOL; our own formalization shares a similar structure.

Coq has also been used to prove entire programs correct w.r.t. numerical uncertainties such as roundoff errors [7]. However, in these efforts much of the work is still manual. Our current development can be seen as complementary as it could potentially provide automation for the verification of roundoff error bounds. The CompCert compiler also supports floating-point computations [8], but only shows semantics preservation and not roundoff error bounds. Harrison [20] has formally verified a floating point implementation of the exponential function inside HOL-Light. The analysis is detailed and specific to this particular function. In contrast, our work aims to provide a fully automated verified analysis for arbitrary real-valued expressions, but at a higher level of abstraction.

c) Real Arithmetic and Finite-precision Formalizations: Formalizations of floating-point arithmetic exist in HOL-Light [24], in Coq in the Flocq library [6] as well as in Isabelle [40] and HOL4 [17]. We found using these formalizations in Coq and HOL4 more complex than was necessary for reasoning inside FloVer, thus we use them only to show a connection to IEEE754. Fixed-point arithmetic has been formalized in HOL4 [1], focusing on its hardware implementation, whereas our focus is on relating their execution to real-valued semantics.

IX. CONCLUSION

We have presented our modular, reusable and easily extendable approach to certificate checking for error bound analysis in FloVer. Our checker is fully-automated and requires neither user interaction, nor expert knowledge. All of the theorems about FloVer have been proven in both Coq and HOL4. We are the first to extract a verified binary for checking finite-precision roundoff errors using the CakeML toolchain and have shown that we achieve significant performance improvements when using the binary.

REFERENCES

[1] B. Akbarpour, S. Tahar, and A. Dekdouk. Formalization of Fixed-Point Arithmetic in HOL. *Formal Methods in System Design*, 27(1-2):173–200, 2005.

[2] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A Verified Compiler for Coq. In *Coq for Programming Languages (CoqPL)*, 2017.

[3] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic Verification of Control System Implementations. In *International Conference on Embedded Software (EMSOFT)*, 2010.

[4] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. *arXiv preprint arXiv:1707.02115*, 2018.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation (PLDI)*, 2003.

[6] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *IEEE Symposium on Computer Arithmetic (ARITH)*, 2011.

[7] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[8] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[9] L. Chen, A. Miné, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[10] E. Darulova. Rosa - The real compiler. https://github.com/malyzajko/rosa, 2015.

[11] E. Darulova and V. Kuncak. Sound Compilation of Reals. In *Symposium on Principles of Programming Languages (POPL)*, 2014.

[12] E. Darulova and V. Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2), 2017.

[13] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. Daisy-Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 270–287. Springer, 2018.

[14] M. Daumas and G. Melquiond. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Transactions on Mathematical Software*, 37(1):2:1–2:20, 2010.

[15] F. De Dinechin, C. Q. Lauter, and G. Melquiond. Assisted Verification of Elementary Functions using Gappa. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[16] L. H. de Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 37 (1-4), 2004.

[17] A. Fox, J. Harrison, and B. Akbarpour. A Formal Model of IEEE Floating Point Arithmetic. *HOL4 Theorem Prover Library*, Apr. 2017. https://github.com/HOL-Theorem-Prover/HOL/tree/master/src/floating-point.

[18] E. Goubault and S. Putot. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

[19] E. Goubault and S. Putot. Robustness Analysis of Finite Precision Implementations. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.

[20] J. Harrison. Floating Point Verification in HOL Light: The Exponential Function. *Formal Methods in System Design*, 16(3), 2000.

[21] C. S. IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[22] F. Immler. A Verified Algorithm for Geometric Zonotope/Hyperplane Intersection. In *International Conference on Certified Programs and Proofs (CPP)*, pages 129–136. ACM, 2015.

[23] A. Izycheva and E. Darulova. On Sound Relative Error Bounds for Floating-Point Arithmetic. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[24] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan. A Parameterized Floating-Point Formalizaton in HOL Light. *Electronic Notes in Theoretical Computer Science*, 317: 101–107, 2015.

[25] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*, 2009.

[26] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *Symposium on Principles of Programming Languages (POPL)*, 2015.

[27] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), 2009.

[28] P. Letouzey. A New Extraction for Coq. In *International Workshop on Types for Proofs and Programs (TYPES)*, 2002.

[29] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards Program Optimization through Automated Analysis of Numerical Precision. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[30] V. Magron, G. A. Constantinides, and A. F. Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):34, 2017.

[31] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[32] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *International Con-*

ference on Computer Safety, Reliability, and Security (SAFECOMP), pages 213–229. Springer, 2017.

[33] M. M. Moscato, C. A. Muñoz, and A. P. Smith. Affine Arithmetic and Applications to Real-Number Proving. In *International Conference on Interactive Theorem Proving (ITP)*, 2015.

[34] M. O. Myreen and S. Owens. Proof-Producing Synthesis of ML from Higher-Order Logic. In *International Conference on Functional Programming (ICFP)*, 2012.

[35] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Certified Programs and Proofs (CPP)*, 2016.

[36] S. Schneider, G. Smolka, and S. Hack. A Linear First-Order Functional Intermediate Language for Verified Compilers. In *International Conference on Interactive Theorem Proving (ITP)*, 2015.

[37] A. Solovyev and T. C. Hales. Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations. In *NASA Formal Methods Symposium (NFM)*, 2013.

[38] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *International Symposium on Formal Methods (FM)*, 2015.

[39] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming (ICFP)*, 2016.

[40] L. Yu. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs*, July 2013. ISSN 2150-914x. http://isa-afp.org/entries/IEEE_Floating_Point.shtml.

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

Certifying Proofs for LTL Model Checking

Alberto Griggio, Marco Roveri and Stefano Tonetta
FBK-irst, Trento, Italy
Email: {griggio,roveri,tonettas}@fbk.eu

Abstract—**In the context of formal verification, certifying proofs are proofs of the correctness of a model in a deduction system produced automatically as outcome of the verification. They are quite appealing for high-assurance systems because they can be verified independently by proof checkers, which are usually simpler to certify than the proof-generating tools.**

Model checking is one of the most prominent approaches to formal verification of temporal properties and is based on an algorithmic search of the system state space. Although modern algorithms integrate deductive methods, the generation of proofs is typically restricted to invariant properties only.

In this paper, we solve this issue in the context of Linear-time Temporal Logic. By exploiting the k-liveness algorithm, we show how to extend proof generation capabilities for invariant checking to cover full LTL properties, in a simple and efficient manner, with essentially no overhead for the model checker. We implemented the technique on top of an IC3 engine, and show the feasibility of the approach on a variety of benchmarks.

I. INTRODUCTION

The application of formal methods in the certification of high-assurance systems demands the qualification of the verification tools to ensure a sufficient level of confidence in their results. However, verification tools such as model checkers can be quite complex with numerous heuristics and combinations of techniques. The idea of certifying model checking [1] to generate deductive proofs as byproduct of the verification is therefore quite appealing, because the proof can be verified by independent proof checkers, which are usually simpler to certify than the proof-generating tools.

Most modern model checking techniques integrate search-based and deductive methods such as induction. In particular, many current model checking algorithms are based on a sequence of SAT queries to find inductive invariants incrementally (e.g., IC3 [2]). Nevertheless, most works on certifying model checkers go back a decade, are mainly theoretical and based on μ-calculus, while practical SAT-based approaches are currently limited to invariant properties.

In this paper, we address the problem of generating a proof in the context of Linear Temporal Logic (LTL) [3] model checking. The main obstacle to proof generation is due to the various transformations applied to the problem: model checking is reduced by contradiction to finding a counterexample; LTL formulae are encoded into symbolically-represented automata; multiple fairness conditions resulting from such encoding are reduced to one; liveness is reduced typically to safety.

We propose a sound and complete approach, where the proof is generated from the inductive invariant obtained with the k-liveness algorithm [4] by combining standard resolution with inference rules specific for LTL, and reasoning by contradiction: by assuming that initially the negation of the property holds, we prove that a certain fairness condition can be visited at most k times, in contradiction with the validity of the fairness condition itself. The resulting approach is simple and efficient, and it can be implemented on top of any state-of-the-art LTL model checker based on the combination of k-liveness with an engine for invariant properties that is capable of producing inductive invariants (e.g., IC3 [2]). The proposed approach results in essentially no overhead for the model checker. We have implemented the technique on top of the IC3IA [5] engine leveraging on the MATHSAT [6] SMT solver as backend. Our experimental results show the feasibility of the approach on a variety of benchmarks taken from the literature, and confirm the small impact of the proof construction on the overall verification process. Finally, we also implemented a prototype proof-checker in Python to check the correctness of the generated proofs, and we executed it on each of the generated proofs. The results show that, for our prototype implementation, the cost of proof checking is comparable with the cost of verification.

This paper is structured as follows. In Sect. II we analyze the related works. In Sect. III we provide the needed background. In Sect. IV we discuss the proposed approach to compute proofs for LTL model checking, and in Sect. V we show the results of our experimental evaluation. Finally, in Sect. VI we draw conclusions and outline future work.

II. RELATED WORK AND CONTRIBUTIONS

Deductive systems for temporal logics have been widely studied [7]–[11]. The idea of certifying model checking is to generate deductive proofs automatically as byproduct of a model checking algorithm.

The closest work to ours is [1], which describes a deductive proof system for verifying properties expressed in the μ-calculus, and shows how to generate a proof in this system from a model checking run. The proposed approach is applicable both for explicit state and symbolic search. The proof system and the proof generation process draw on results which relate model checking for the μ-calculus to winning parity games [12]. The system was implemented (as a prototype) on top of a BDD-based engine (COSPAN [13]);

This work has received funding from the EU's H2020 research and innovation programme under the Grant Agreement No. 700665 (project CITADEL) and from the EU's ECSEL JU and Italy's MIUR under the Grant Agreement No. 692474 (project AMASS).

ISBN: 978-0-9835678-8-2.

it is however unclear how to adapt it to modern SAT-based engines. Our approach instead implements proof generation on top of SAT-based algorithms without any substantial overhead or modification of the model checking engine. Moreover, although in terms of expressiveness LTL is more restricted than μ-calculus, in [1], LTL is assumed to be encoded and it is not shown how to convert the proof for the resulting μ-calculus formula to a proof for LTL. Our work instead produces a proof using inference rules for LTL, automatically reverting the internal automata construction.

Other approaches targeting model checking of linear-time properties include [14], [15], [16], [17] and [18]. These works are however mostly theoretical, and to the best of our knowledge, with no implementation available.

Related, but slightly-different, problems are addressed in [19], [20], and [21]. The first work gives a technique to incrementally build a (partial) deductive proof from the search performed by a model checker for incomplete (partially specified) systems while proving a given LTL property holds; the second focuses on runtime monitoring, proposing a local proof system for LTL and showing how such a system can be used for the construction of online runtime monitors; the third work instead discusses a proof system to provide evidence why a trace violates an LTL specification, as opposed to certifying why the property holds on the system under verification.

The work in [22] presents an LTL model checker whose code has been completely verified using the Isabelle theorem prover. The proof consists of the formal verification of a few hundred lines of "formalized pseudocode", and a verified refinement step in which mathematical sets and other abstract structures are replaced by implementations of efficient structures. The resulting checker is slower than unverified checkers, but it can be used as a trusted reference implementation.

Finally, some theorem provers for LTL can produce proofs, such as TRP++ [23] and TeMP [24]. Both systems are based on the temporal resolution calculus and can produce fine-grained proofs, which can then be inspected and checked to certify the correctness. However, no automatic proof checkers are available.

Overall, to the best of our knowledge, no previous work provides the following contributions:

- a proof-generation technique extending a SAT-based LTL model checking algorithm;
- a proof-generation technique for LTL based on symbolic encoding, reverting the internal automata construction;
- a proof-generation technique for LTL validity based on model checking;
- an available effective implementation of proof generation from LTL model checking.

III. BACKGROUND

We work in the setting of Boolean (i.e. propositional) logic, with the standard notions of satisfiability, validity, interpretations and models. We denote propositional variables with v, x, y, and formulae with $\phi, \psi, f, \alpha, \beta, I, T$, possibly with subscripts or primes (e.g v_1, x'). If V, V' are (disjoint) sets of variables, we write $\phi(V, V')$ to stress that all the variables occurring in ϕ belong to $V \cup V'$. We use $ite(\phi_c, \phi_t, \phi_e)$ as a shorthand for $(\phi_c \rightarrow \phi_t) \wedge (\neg\phi_c \rightarrow \phi_e)$. Given a variable v, a formula ϕ and a formula ψ not containing v, we denote with $\phi[v := \psi]$ the result of substituting v with ψ everywhere in ϕ. We extend this to sets of variables in a pointwise manner. If V and V' are two disjoint sets of variables, we might write $\phi[V := V']$ as ϕ'. A counter is an integer-valued variable c that occurs in two kinds of predicates: comparisons with constants, such as $c = 0$ or $c \leq 10$; and conditional increments, such as $ite(f, c' = c + 1, c' = c)$. Abusing notation, and for the sake of readability, in the following we sometimes use counters to denote their equivalent propositional encoding.[1]

A. Transition Systems

A *transition system* M is a tuple $M = \langle V, I, T\rangle$ where V is a set of (propositional) state variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula representing the transitions.

A *state* of M is an assignment to the variables V. We denote with Σ_V the set of states. We say that a state $s \in \Sigma_V$ is a model for a formula $\phi(V)$ (denoted $s \models \phi(V)$) if substituting in ϕ the values of the variables in s, the formula ϕ evaluates to $\top$. A [finite] *path* of M is an infinite sequence $s_0, s_1, \ldots$ [resp., finite sequence $s_0, s_1, \ldots, s_k$] of states such that $s_0 \models I$ and, for all $i \geq 0$ [resp., $0 \leq i < k$], $s_i, s'_{i+1} \models T$. Given $\sigma := s_0, s_1, \ldots$, with $\sigma[j]$ we denote the state s_j, and with σ^j the path $s_j, s_{j+1}, \ldots$. Given two transitions systems $M_1 = \langle V_1, I_1, T_1\rangle$ and $M_2 = \langle V_2, I_2, T_2\rangle$, we denote with $M_1 \times M_2$ the synchronous product $\langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2\rangle$.

B. Invariant Properties

Given a Boolean combination ϕ of predicates, the invariant model checking problem, denoted with $M \models_{fin} \phi$, is the problem to check if, for all finite paths $s_0, s_1, \ldots, s_k$ of M, $s_k \models \phi$.

Most model checkers prove an invariant property by generating a stronger invariant formula ψ that is *inductive*, i.e. such that: (i) $I \rightarrow \psi$; (ii) $\psi \wedge T \rightarrow \psi'$; and (iii) $\psi \rightarrow \phi$.

C. LTL

Given a set of propositional variables V, LTL formulae are built using Boolean connectives and the temporal operators **X** ("next") and **U** ("until"). Formally,

- a variable $v \in V$ is an LTL formula,
- if ϕ_1 and ϕ_2 are LTL formulae, then $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\mathbf{X}\phi_1$ and $\phi_1 \mathbf{U} \phi_2$ are LTL formulae.

We use the standard abbreviations: $\top := p \vee \neg p$, $\bot := p \wedge \neg p$, $\mathbf{F}\phi := \top \mathbf{U} \phi$, and $\mathbf{G}\phi := \neg\mathbf{F}\neg\phi$.

Given an LTL formula ϕ, a sequence σ of assignments to V, and an index i, we define $\sigma, i \models \phi$, i.e., that σ satisfies the formula ϕ in i, as follows:

- $\sigma, i \models v$ iff $\sigma[i] \models v$

---

[1] This can be done in a standard way, using e.g. a unary or a binary encoding for integer numbers and the required comparison and increment operations; we refrain from giving the full details of this in order to save space.

ISBN: 978-0-9835678-8-2.

- $\sigma, i \models \phi \land \psi$ iff $\sigma, i \models \phi$ and $\sigma, i \models \psi$
- $\sigma, i \models \neg\phi$ iff $\sigma, i \not\models \phi$
- $\sigma, i \models \mathbf{X}\phi$ iff $\sigma, i+1 \models \phi$
- $\sigma, i \models \phi\mathbf{U}\psi$ iff for some $j \geq i$, $\sigma, j \models \psi$ and for all $i \leq k < j$, $\sigma, k \models \phi$.

Finally, $\sigma \models \phi$ iff $\sigma, 0 \models \phi$.

Given an LTL formula ϕ, the LTL model checking problem, denoted with $M \models \phi$, is the problem to check if, for all (infinite) paths σ of M, $\sigma \models \phi$.

Given an LTL formula ϕ, the LTL validity problem, denoted by $\models \phi$, is the problem of checking if $\sigma \models \phi$ for all (infinite) paths over Σ_V. The validity problem can be reduced to the model checking problem by considering the universal model $M_U = \langle V, \top, \top \rangle$. It is easy to prove that $\models \phi$ iff $M_U \models \phi$.

D. Symbolic LTL Model Checking

The automata-based approach [25] to LTL model checking is to build a transition system $M_{\neg\phi}$ with a set of fairness conditions $F_{\neg\phi}$ such that $M \models \phi$ iff $M \times M_{\neg\phi} \models \neg \bigwedge_{f \in F_{\neg\phi}} \mathbf{GF}f$. This reduces to finding a counterexample as a fair path, i.e., a path of the system that visits each fairness condition in $F_{\neg\phi}$ infinitely many times.

Following [26], the encoding of an LTL formula ϕ over variables V into a transition system $M_{\neg\phi} = \langle V_{\neg\phi}, I_{\neg\phi}, T_{\neg\phi} \rangle$ with fairness conditions $F_{\neg\phi}$ is defined as follows:

- $V_{\neg\phi} = V \cup \{v_{X\beta} \mid \mathbf{X}\beta \in Sub(\phi)\} \cup \{v_{\mathbf{X}(\beta_1\mathbf{U}\beta_2)} \mid \beta_1\mathbf{U}\beta_2 \in Sub(\phi)\}$
- $I_{\neg\phi} = Enc(\neg\phi)$
- $T_{\neg\phi} = \bigwedge_{v_{\mathbf{X}\beta} \in V_{\neg\phi}} \beta \leftrightarrow Enc(\beta)'$
- $F_{\neg\phi} = \{Enc(\beta_1\mathbf{U}\beta_2 \rightarrow \beta_2) \mid \beta_1\mathbf{U}\beta_2 \in Sub(\phi)\}$

where Sub is a function that maps a formula ϕ to the set of its subformulae, and Enc is defined recursively as:

- $Enc(v) = v$
- $Enc(\phi_1 \land \phi_2) = Enc(\phi_1) \land Enc(\phi_2)$
- $Enc(\neg\phi_1) = \neg Enc(\phi_1)$
- $Enc(\mathbf{X}\phi_1) = v_{\mathbf{X}\phi_1}$
- $Enc(\phi_1\mathbf{U}\phi_2) = Enc(\phi_2) \lor (Enc(\phi_1) \land v_{\mathbf{X}(\phi_1\mathbf{U}\phi_2)})$

E. Degeneralization

In explicit-state model checking, the standard way to encode a Generalized Büchi Automaton with n fairness conditions into an equivalent "degeneralized" one (i.e., with one fairness), is to fix an order on the fairness conditions, replicate the automaton n times, and move from the i-th copy to the next one as soon as the i-th fairness condition is visited. Symbolically, this can be achieved as follows.

Given a transition system $M = \langle V, I, T \rangle$ with fairness conditions $F = \{f_1, \ldots, f_n\}$, we build an equivalent system with a single fairness condition f by considering $M \times M_{deg}$, where $M_{deg} = \langle V_{deg}, I_{deg}, T_{deg} \rangle$ is defined as follows:

- $V_{deg} = V \cup \{s\}$
- $I_{deg} = s = 0$
- $T_{deg} = \bigwedge_{0 \leq i < n-1} (s = i \rightarrow ite(f_{i+1}, s' = s + 1, s' = s)) \land (s = n - 1 \rightarrow ite(f_n, s' = 0, s' = s))$

and $f = s = 0 \land f_1$.

Most standard symbolic model checkers use a different encoding, which does not fix an ordering on the fairness conditions: one propositional variable per fairness condition is set to true whenever the fairness condition is visited, and when all the variables are true they are reset to false. The proof generation described in the next section is based on the above encoding with fixed ordering (see Section IV-D for details on the reason). We analyze the impact of this choice experimentally in Section V.

F. K-Liveness and SAT-based Symbolic Model Checking

SAT-based algorithms take as input a propositional transition system and a property, and try to solve the verification problem with a series of satisfiability queries. IC3 [2] is a symbolic model checking algorithm for the verification of invariant properties. It builds an over-approximation of the reachable state space, using clauses obtained by generalization while disproving candidate counterexamples. In the case of finite-state systems, the algorithm is implemented on top of Boolean SAT solvers, fully leveraging their features. IC3 has demonstrated to be extremely effective, and it is a fundamental core in all the engines in hardware verification.

K-liveness [4] is an algorithm recently proposed to reduce liveness checking (and so also LTL verification) to a sequence of invariant checking problems. K-liveness uses the standard approach, outlined above, to reduce the LTL verification problem $M \models \varphi$ to $M \times M_{\neg\varphi} \times M_{deg} \models \neg\mathbf{GF}f$. Its key insight is that, for finite-state systems, this is equivalent to find a k such that f is visited at most k times, which in turn can be reduced to invariant checking.

In [4], it is proved that, for finite-state systems, $M \models \neg\mathbf{GF}f$ iff there exists k such that f can be visited at most k times along a path of M. The last check can be reduced to an invariant checking problem of the form $M \times M_c \models_{fin} (c \leq k)$, where $M_c := \langle V_c, I_c, T_c \rangle$ is defined as follows: $V_c := \{c\}$, $I_c := c = 0$, $T_c := ite(f, c' = c + 1, c' = c)$. K-liveness is therefore a simple loop that increases k at every iteration and calls a subroutine SAFE to check the invariant $(c \leq k)$ on $M \times M_c$. In particular, the implementation in [4] uses IC3 as SAFE and exploits the incrementality of IC3 to solve the sequence of invariant problems in an efficient way.

G. Deduction Systems

A deduction system consists of a set of axiom schemes and inference rules. We use natural deduction [27] notation to represent proofs. A proof is a tree of formulae where leaves are axioms or hypothesis, and any other formula is obtained by the application of an inference rule. Proofs for propositional formulae can be built using the following resolution rule and set of axioms (see e.g. [28]):

$$\frac{\alpha_1 \lor \psi \quad \neg\psi \lor \alpha_2}{\alpha_1 \lor \alpha_2}\ \text{RES}$$

$$\frac{}{\neg(a \lor b) \lor a \lor b}\ \text{OR-L} \qquad \frac{}{\neg a \lor (a \lor b)}\ \text{OR-R}$$

$$\frac{}{\neg(a \land b) \lor a}\ \text{AND-L} \qquad \frac{}{\neg a \lor \neg b \lor (a \land b)}\ \text{AND-R}$$

ISBN: 978-0-9835678-8-2. Copyright owned jointly by the authors and FMCAD, Inc.

In order to use resolution proofs inside other proofs, we use the reductio ad absurdum rule. If a proof of $\bot$ can be derived using $\neg\alpha$ as hypothesis, we can extend it to a proof of α, removing α from the hypothesis.

$$\begin{array}{c}[\neg\alpha]\\ \vdots\\ \dfrac{\bot}{\alpha}\ \text{RAA}\end{array}$$

As for temporal operators, we use the following generalization inference rule: if a proof of α can be derived without any hypothesis, then we can derive $\mathbf{G}\alpha$, and thus also $\mathbf{X}\alpha$:

$$\frac{\overline{\alpha}}{\mathbf{G}\alpha}\ \text{G} \qquad\qquad \frac{\overline{\alpha}}{\mathbf{X}\alpha}\ \text{X}$$

and we use the following axioms:

$$\frac{}{(a\mathbf{U}b) \leftrightarrow (b \vee (a \wedge \mathbf{X}(a\mathbf{U}b)))}\ \text{UNTIL}$$

$$\frac{}{(a\mathbf{U}(b_1 \vee b_2)) \leftrightarrow ((a\mathbf{U}b_1) \vee (a\mathbf{U}b_2))}\ \text{UNTIL-OR}$$

$$\frac{}{((b_1 \wedge b_2)\mathbf{U}a) \leftrightarrow ((b_1\mathbf{U}a) \wedge (b_2\mathbf{U}a))}\ \text{UNTIL-AND}$$

$$\frac{}{\mathbf{X}\neg a \leftrightarrow \neg\mathbf{X}a}\ \text{NEXT-NOT}$$

$$\frac{}{\mathbf{X}(a \vee b) \leftrightarrow (\mathbf{X}a \vee \mathbf{X}b)}\ \text{NEXT-OR}$$

$$\frac{}{\mathbf{X}(a \wedge b) \leftrightarrow (\mathbf{X}a \wedge \mathbf{X}b)}\ \text{NEXT-AND}$$

The following are abbreviations of multiple applications of the above rules:

- LTL expansion is obtained by multiple application of UNTIL, AND-L, AND-R, OR-L, OR-R:

  $$\frac{}{\alpha \leftrightarrow Exp(\alpha)}\ \text{EXP}$$

  where Exp is defined recursively as:
  - $Exp(v) = v$
  - $Exp(\phi_1 \wedge \phi_2) = Exp(\phi_1) \wedge Exp(\phi_2)$
  - $Exp(\neg\phi_1) = \neg Exp(\phi_1)$
  - $Exp(\mathbf{X}\phi_1) = \mathbf{X}\phi_1$
  - $Exp(\phi_1\mathbf{U}\phi_2) = Exp(\phi_2) \vee (Exp(\phi_1) \wedge \mathbf{X}(\phi_1\mathbf{U}\phi_2))$
- X distribution is obtained by multiple application of NEXT-NOT, NEXT-AND, NEXT-OR:

  $$\frac{\mathbf{X}\alpha}{Next(\alpha)}\ \text{XDIS}$$

  where $Next$ is defined recursively as:
  - $Next(v) = \mathbf{X}v$
  - $Next(\phi_1 \wedge \phi_2) = Next(\phi_1) \wedge Next(\phi_2)$
  - $Next(\neg\phi_1) = \neg Next(\phi_1)$
  - $Next(\mathbf{X}\phi_1) = \mathbf{XX}\phi_1$
  - $Next(\phi_1\mathbf{U}\phi_2) = Next(\phi_2) \vee (Next(\phi_1) \wedge \mathbf{XX}(\phi_1\mathbf{U}\phi_2))$
- $\wedge$ removal is obtained by combining RES with AND-L:

  $$\frac{\alpha_1 \wedge \alpha_2}{\alpha_1}\ \text{ANDL} \qquad\qquad \frac{\alpha_1 \wedge \alpha_2}{\alpha_2}\ \text{ANDR}$$
- $\mathbf{U}$ distribution is obtained by combining RES with UNTIL-AND:

  $$\frac{(\alpha_1 \wedge \alpha_2)\mathbf{U}\beta}{\alpha_1\mathbf{U}\beta}\ \text{UAL} \qquad\qquad \frac{(\alpha_1 \wedge \alpha_2)\mathbf{U}\beta}{\alpha_2\mathbf{U}\beta}\ \text{UAR}$$
- $\mathbf{G}$ distribution is obtained by combining RES with UNTIL-OR:

  $$\frac{\mathbf{G}(\alpha_1 \wedge \alpha_2)}{\mathbf{G}\alpha_1}\ \text{GAL} \qquad\qquad \frac{\mathbf{G}(\alpha_1 \wedge \alpha_2)}{\mathbf{G}\alpha_2}\ \text{GAR}$$
- $\mathbf{G}$ removal is obtained by combining UNTIL and OR-R:

  $$\frac{\mathbf{G}\alpha}{\alpha}\ \text{GN}$$

H. Resolution Proofs for Invariant Properties

In case of an invariant property ϕ, an inductive invariant ψ can be used to generate a proof of ϕ. In fact, since the formulae $I \rightarrow \psi$, $\psi \wedge T \rightarrow \psi'$, $\psi \rightarrow \phi$ are valid, we can obtain a resolution proof for each of them. Using an inductive inference rule, we can then deduce that ϕ holds in all reachable states.

IV. Certifying Proofs for LTL Model Checking

A. Overview of the Approach

As described in Section III, the standard symbolic LTL model checking approach proceeds through a sequence of transformations. Thus, from the original problem $M \models \phi$, we arrive at the problem $M \times M_{\neg\phi} \times M_{deg} \times M_c \models_{fin} c \leq k$ from which we can extract an inductive invariant ψ. In order to generate the proof for the original problem, we conceptually reverse this sequence showing how to generate a proof for each step. In Section IV-C, we show how to generate a proof from ψ of $M \times M_{\neg\phi} \times M_{deg} \models \neg\mathbf{GF}f$; in Section IV-D, we show how to generate a proof from ψ of $M \times M_{\neg\phi} \models \neg(\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n)$; finally, in Section IV-E, we show how to generate a proof from ψ of $M \models \phi$.

B. LTL Model Checking and LTL Validity

Consider the LTL model checking problem $M \models \phi$, where $M = \langle V, I, T\rangle$. With abuse of notation, we consider T also as an LTL formula, identifying v' with $\mathbf{X}v$ for every variable $v \in V$. In order to prove that $M \models \phi$, we provide a proof of $(I \wedge \mathbf{G}T) \rightarrow \phi$.

Note that, in case the original problem is the validity of an LTL formula ϕ, we reduce it to the model checking problem $M_U \models \phi$ (as explained in Section III-D) generating a proof of ϕ since the initial and transition conditions of M_U are true.

C. Certifying Proofs for K-Liveness

We consider first the special case of proving $M \models \neg\mathbf{GF}f$, where f is a propositional formula over V. In order to prove $(I \wedge \mathbf{G}T) \rightarrow \neg\mathbf{GF}f$, we use the following inference rule, denoted with KL (see Section IV-F for proof of correctness):

$$\frac{(Pi)\quad (Pn_0)\quad (Pp_0)\quad \ldots\quad (Pn_k)\quad (Pp_k)}{(\iota \wedge \mathbf{G}\tau) \rightarrow \neg\mathbf{GF}\rho}\ \text{KL}$$

where the premises of the rule KL are:

$$\iota \rightarrow \alpha_0 \qquad (Pi)$$
$$\mathbf{G}((\alpha_0 \wedge \tau \wedge \neg\rho) \rightarrow \mathbf{X}\alpha_0) \qquad (Pn_0)$$
$$\mathbf{G}((\alpha_0 \wedge \tau \wedge \rho) \rightarrow \mathbf{X}\alpha_1) \qquad (Pp_0)$$
$$\ldots$$
$$\mathbf{G}((\alpha_k \wedge \tau \wedge \neg\rho) \rightarrow \mathbf{X}\alpha_k) \qquad (Pn_k)$$
$$\mathbf{G}((\alpha_k \wedge \tau \wedge \rho) \rightarrow \bot) \qquad (Pp_k)$$

Intuitively, this means that if we have conditions $\alpha_0, \ldots, \alpha_{k+1}$ such that α_0 is implied by ι, α_i is inductive relative to $\neg\rho$ for $0 \leq i \leq k$, α_{i+1} is implied by $\alpha_i \wedge \rho$ after a transition for $0 \leq i \leq k$, and $\alpha_{k+1} = \bot$, then any path starting from ι can visit ρ finitely many times only.

When checking $M \models \neg\mathbf{GF}f$, we instantiate the rule using $\iota = I$, $\tau = T$, $\rho = f$, and α_i are obtained by the inductive invariant generated with k-liveness.

If c is the counter introduced by k-liveness to count the occurrences of f and ψ is the inductive invariant over $V \cup \{c\}$ obtained to prove that $c \leq k$, then we instantiate the rule KL using $\alpha_i = \psi[c := i]$.

Since ψ is the inductive invariant obtained with k-liveness we know that the following propositional formulae are valid:

$$I \wedge c = 0 \rightarrow \psi$$
$$\psi \wedge T \wedge ite(f, c' = c + 1, c' = c) \rightarrow \psi'$$
$$\psi \rightarrow c \leq k$$

Therefore also the following formulae are valid:

$$I \rightarrow \psi[c := 0] \qquad (p0)$$
$$(\psi[c := 0] \wedge T \wedge \neg f) \rightarrow \mathbf{X}\psi[c := 0] \qquad (p00)$$
$$(\psi[c := 0] \wedge T \wedge f) \rightarrow \mathbf{X}\psi[c := 1] \qquad (p01)$$
$$\ldots$$
$$(\psi[c := k] \wedge T \wedge \neg f) \rightarrow \mathbf{X}\psi[c := k] \qquad (pkk)$$
$$(\psi[c := k] \wedge T \wedge f) \rightarrow \bot \qquad (pk)$$

Note that, the formulae $\alpha_0, \ldots, \alpha_k$ above are not required to be in any specific form. In particular, when instantiating them with the inductive invariant ψ, we can apply standard equivalence-preserving simplifications (e.g. $\alpha \wedge \top \equiv \alpha$) after the substitution of counter values.

For each formula $p \in \{p0, p00, p01, ..., pk\}$, we can obtain a resolution proof:

$$\begin{array}{c} p \\ \vdots \\ \bot \end{array}$$

Using rules RAA and G, we obtain a proof for each premise of the rule KL, obtaining a proof of $(I \wedge \mathbf{G}T) \rightarrow \neg\mathbf{GF}f$ in the following form:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}[\neg p0]\\ \vdots \\ \bot\end{array}}{p0}\,\text{RAA}}{\mathbf{G}(p0)}\,\text{G} \quad \dfrac{\dfrac{\begin{array}{c}[\neg p00]\\ \vdots \\ \bot\end{array}}{p00}\,\text{RAA}}{\mathbf{G}(p00)}\,\text{G} \quad \dfrac{\dfrac{\begin{array}{c}[\neg p01]\\ \vdots \\ \bot\end{array}}{p01}\,\text{RAA}}{\mathbf{G}(p01)}\,\text{G} \quad \ldots \quad \dfrac{\dfrac{\begin{array}{c}[\neg pk]\\ \vdots \\ \bot\end{array}}{pk}\,\text{RAA}}{\mathbf{G}(pk)}\,\text{G}}{(I \wedge \mathbf{G}T) \rightarrow \neg\mathbf{GF}f}\,\text{KL}$$

D. Generalization to Multiple Fairness Conditions

We now consider the case $M \models \neg(\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n)$, where $f_1, \ldots, f_n$ are propositional formulae over V. In order to prove $(I \wedge \mathbf{G}T) \rightarrow \neg(\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n)$, we generalize the rule KL into rule GKL. Rule GKL derives $(\iota \wedge \mathbf{G}\tau) \rightarrow \neg(\mathbf{GF}\rho_1 \wedge \ldots \wedge \mathbf{GF}\rho_n)$ from the following premises:

$$\iota \rightarrow \alpha_{01} \qquad (P_{01})$$
for $0 \leq i \leq k, 1 \leq j \leq n$
$$\mathbf{G}((\alpha_{ij} \wedge \tau \wedge \neg\rho_j) \rightarrow \mathbf{X}\alpha_{ij}) \qquad (Pn_{ij})$$
for $0 \leq i \leq k, 1 \leq j < n$
$$\mathbf{G}((\alpha_{ij} \wedge \tau \wedge \rho_j) \rightarrow \mathbf{X}\alpha_{ij'}) \qquad (Pp_{ij})$$
for $0 \leq i < k$
$$\mathbf{G}((\alpha_{in} \wedge \tau \wedge \rho_n) \rightarrow \mathbf{X}\alpha_{i'1}) \qquad (P_{in})$$
$$\mathbf{G}((\alpha_{kn} \wedge \tau \wedge \rho_n) \rightarrow \bot) \qquad (P_{kn})$$

where $j' = j + 1$ and $i' = i + 1$.

Again, this rule can be instantiated from the inductive invariant generated by k-liveness when using the degeneralization described in Section III-E. More concretely, if c is the counter used to count the occurrences of the fairness conditions, s is the counter used to track if the i-th fairness has been visited, and ψ is the inductive invariant, we set $\alpha_{ij} = \psi[c := i, s := j - 1]$ and generate a resolution proof for the following valid formulae (as in the previous case, we can simplify the formulae after substituting counter values, before generating the proofs):

$$I \rightarrow \psi[c := 0, s := 0] \qquad (p_{01})$$
for $0 \leq i \leq k, 1 \leq j \leq n$
$$(\psi[c := i, s := j - 1] \wedge T \wedge \neg f_j) \rightarrow \mathbf{X}\psi[c := i, s := j - 1] \qquad (pn_{ij})$$
for $0 \leq i \leq k, 1 \leq j < n$
$$(\psi[c := i, s := j - 1] \wedge T \wedge f_j) \rightarrow \mathbf{X}\psi[c := i, s := j] \qquad (pp_{ij})$$
for $0 \leq i < k$
$$(\psi[c := i, s := n-1] \wedge T \wedge f_n) \rightarrow \mathbf{X}\psi[c := i+1, s := 0] \qquad (p_{in})$$
$$(\psi[c := k, s := n - 1] \wedge T \wedge f_n) \rightarrow \bot \qquad (p_{kn})$$

Similarly to the previous case, we can transform the resolution proofs for these lemmas in temporal proofs for the premises of the rule GKL.

E. Certifying Proofs for LTL

We consider here the general case of $M \models \phi$. The procedure described in Section III-F reduces the problem to $M \times M_{\neg\phi} \models \neg(\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n)$, where $M \times M_{\neg\phi} = \langle V \cup V_{\neg\phi}, I \wedge I_{\neg\phi}, T \wedge T_{\neg\phi} \rangle$. Applying the procedure described above, we obtain a temporal proof of $(I \wedge I_{\neg\phi} \wedge \mathbf{G}(T \wedge T_{\neg\phi})) \rightarrow \neg(\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n)$.

Every variable $v_{\mathbf{X}\beta} \in V_{\neg\phi}$ is associated with a temporal formula $\mathbf{X}\beta$. We denote by $Enc^{-1}(\alpha)$ the formula obtained from α by substituting every $v_{\mathbf{X}\beta}$ with $\mathbf{X}\beta$. By applying this substitution in the mentioned proof, we obtain a proof of $(I \wedge Enc^{-1}(I_{\neg\phi}) \wedge \mathbf{G}(T \wedge Enc^{-1}(T_{\neg\phi}))) \rightarrow \neg(\mathbf{GF}Enc^{-1}(f_1) \wedge \ldots \wedge \mathbf{GF}Enc^{-1}(f_n))$.

From this, as shown in Figure 1, we derive a proof of $(I \wedge \mathbf{G}T) \rightarrow \phi$ with three resolution steps, using $\mathbf{GF}Enc^{-1}(f_i)$, $\mathbf{G}Enc^{-1}(T_{\neg\phi})$, and $\neg Enc^{-1}(I_{\neg\phi}) \rightarrow \phi$ as lemmas.

Finally, we provide a proof for each lemma. Note that, given the specific construction of $M_{\neg\phi}$, $Enc^{-1}(T_{\neg\phi})$ and $\mathbf{GF}Enc^{-1}(f_i)$ are always valid formulae. Moreover, note that $Enc^{-1}(I_{\neg\phi}) = Exp(\neg\phi)$ and that $Enc^{-1}(T_{\neg\phi})$ is in the form $\bigwedge_\beta \mathbf{X}\beta \leftrightarrow Next(Exp(\beta))$.

The following are therefore proofs for the above lemmas:

$$\dfrac{\dfrac{}{\neg\phi \leftrightarrow Exp(\phi)}\;\text{EXP}}{\neg Enc^{-1}(I_{\neg\phi}) \rightarrow \phi}\;\text{ANDL} \qquad \dfrac{\dfrac{\dfrac{}{\beta \leftrightarrow Exp(\beta)}\;\text{EXP}}{\mathbf{X}(\beta \leftrightarrow Exp(\beta))}\;\text{X}}{\mathbf{X}\beta \leftrightarrow Next(Exp(\beta))}\;\text{XDIS}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\mathbf{G}(\beta_1\mathbf{U}\beta_2 \wedge \neg\beta_2)]}{\mathbf{G}\beta_1\mathbf{U}\beta_2}\;\text{GAL}}{\beta_1\mathbf{U}\beta_2}\;\text{GN}}{\mathbf{F}\beta_2}\;\text{UAR} \qquad \dfrac{[\mathbf{G}(\beta_1\mathbf{U}\beta_2 \wedge \neg\beta_2)]}{\mathbf{G}\neg\beta_2}\;\text{GAR}}{\bot}\;\text{RES}}{\mathbf{F}(\beta_1\mathbf{U}\beta_2 \rightarrow \beta_2)}\;\text{RAA}}{\mathbf{GF}(\beta_1\mathbf{U}\beta_2 \rightarrow \beta_2)}\;\text{G}$$

Example 1: We work out a full example showing the different steps from model checking to proof generation.

Le us consider the transition system $M = \langle V, I, T\rangle$ where:

$$V := \{x, y, z\} \qquad I := \top \qquad T := (x \rightarrow y') \wedge (y \rightarrow z')$$

and let us consider the property $\phi = \mathbf{G}(x \rightarrow \mathbf{F}z)$.

ϕ contains two $\mathbf{U}$-formulae: $\mathbf{F}(\neg(x \rightarrow \mathbf{F}z))$, which we abbreviate by $\mathbf{F}_1$, and $\mathbf{F}z$.

The transition system for the negation $\neg\phi$ is $M_{\neg\phi} = \langle V, I_{\neg\phi}, T_{\neg\phi}\rangle$ where:

- $V_{\neg\phi} = \{x, z, v_{\mathbf{XF}_1}, v_{\mathbf{XF}z}\}$
- $I_{\neg\phi} = Enc(\neg\phi) = (x \wedge \neg(z \vee v_{\mathbf{XF}z})) \vee v_{\mathbf{XF}_1}$
- $T_{\neg\phi} = (v_{\mathbf{XF}_1} \leftrightarrow ((x' \wedge \neg(z' \vee v'_{\mathbf{XF}z})) \vee v'_{\mathbf{XF}_1})) \wedge (v_{\mathbf{XF}z} \leftrightarrow (z' \vee v'_{\mathbf{XF}z}))$

with fairness conditions $Enc(f_1)$ and $Enc(f_2)$ where:

$$f_1 = \neg\mathbf{F}_1 \vee \neg(x \rightarrow \mathbf{F}z) \qquad f_2 = \neg\mathbf{F}z \vee z$$

M_{deg} and M_c are defined as in Sections III-E and III-F.

Let us suppose that k-liveness produces the following inductive invariant:

$$\begin{aligned}\psi =&(Enc(\neg\phi) \wedge (\neg x \vee z \vee v_{\mathbf{XF}z}) \wedge s = 0) \vee \\ &(y \wedge \neg z \wedge v_{\mathbf{XG}\neg z} \wedge s = 1)\end{aligned}$$

After substituting and simplifying, we obtain:
$\alpha_{01} = Enc(\neg\phi) \wedge (\neg x \vee z \vee v_{\mathbf{XF}z})$
$\alpha_{02} = y \wedge \neg(z \vee v_{\mathbf{XF}z})$
$\alpha_{11} = \alpha_{12} = \bot$

Let us consider only a non-trivial case and produce a proof for $TL := (Enc(\neg\phi) \wedge (\neg x \vee z \vee \mathbf{XF}z) \wedge T \wedge T_{\neg\phi} \wedge f_1) \rightarrow (\mathbf{X}y \wedge \mathbf{X}\neg z \wedge \neg\mathbf{XF}z)$.

From the SAT solver we can obtain the following resolution proof for $L = Enc(\neg\phi) \wedge (\neg x \vee Enc(\mathbf{F}z)) \wedge T \wedge T_{\neg\phi} \wedge Enc(f_1) \wedge (\neg y' \vee Enc(\mathbf{F}z)')$:

$$\dfrac{\dfrac{\dfrac{L}{x \rightarrow y'} \qquad \dfrac{\dfrac{\dfrac{L}{Enc(\neg\phi)} \quad \dfrac{L}{Enc(f_1)}}{x \wedge \neg Enc(\mathbf{F}z)}}{x}}{y'} \qquad \dfrac{\dfrac{\dfrac{L}{T_{\neg\phi}} \quad \dfrac{\dfrac{\dfrac{L}{Enc(\neg\phi)} \quad \dfrac{L}{Enc(f_1)}}{x \wedge \neg Enc(\mathbf{F}z)}}{\neg Enc(\mathbf{F}z)}}{\neg Enc(\mathbf{F}z)'} \qquad \dfrac{L}{\neg y' \vee Enc(\mathbf{F}z)'}}{\neg y'}}{\bot}$$

In order to obtain a proof of TL it is sufficient to substitute in the above proof the variables $v_{\mathbf{XF}_1}$ and $v_{\mathbf{XF}z}$ with respectively $\mathbf{XF}_1$ and $\mathbf{XF}z$.

Finally, to obtain a proof of $(I \wedge \mathbf{G}T) \rightarrow \phi$ we instantiate the lemmas to remove $I_{\neg\phi}$, $T_{\neg\phi}$, and the fairness conditions.

For example, the proof for the lemma $\mathbf{GF}f_1$ is obtained by substituting β_1 with $\mathbf{F}_1$ and β_2 with $\neg(x \rightarrow \mathbf{F}z)$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\mathbf{G}\neg f_1]}{\mathbf{GF}_1}\;\text{GAL}}{\mathbf{F}_1}\;\text{GN}}{\mathbf{F}(\neg(x \rightarrow \mathbf{F}z))}\;\text{UAR} \qquad \dfrac{[\mathbf{G}\neg f_1]}{\mathbf{G}\neg(\neg(x \rightarrow \mathbf{F}z))}\;\text{GAR}}{\bot}\;\text{RES}}{\mathbf{F}f_1}\;\text{RAA}}{\mathbf{GF}f_1}\;\text{G}$$

F. Correctness

In the above proofs, we only used the rules defined in Section III-G and the new rule GKL (rule KL is a special case of GKL where $n = 1$).

Let us denote deducibility with this set of rules by $\vdash_{\text{GKL}}$. In the following, we prove soundness and completeness of the proofs.

Theorem 1: If $\vdash_{\text{GKL}} \alpha$ then $\models \alpha$.

Proof. All rules and axioms described above are trivial apart from rule GKL. So, we prove that if σ satisfies (P_{01}), (Pn_{ij}) for $0 \leq i \leq k, 1 \leq j \leq n$, (Pp_{ij}) for $0 \leq i \leq k, 1 \leq j < n$, and (P_{in}) for $0 \leq i \leq k$, then $\sigma \models (\iota \wedge \mathbf{G}\tau) \rightarrow \neg(\mathbf{GF}\rho_1 \wedge \ldots \wedge \mathbf{GF}\rho_n)$ holds. By contradiction, suppose $\sigma \models (\iota \wedge \mathbf{G}\tau) \wedge (\mathbf{GF}\rho_1 \wedge \ldots \wedge \mathbf{GF}\rho_n)$, then σ satisfies each ρ_j infinitely many times. So, let us define $(k+1) \times n$ points t_{ij} such that $t_{00} = 0$ and for all i, j, $0 \leq i \leq k$, $1 \leq j \leq n$, $t_{i,j}$ is such that for all h, $t_{i,j-1} \leq h < t_{i,j}$ $\sigma, h \not\models \rho_j$ and $\sigma, t_{ij} \models \rho_j$ and, for all i, $0 \leq i < k$, $t_{i+1,0} = t_{i,n} + 1$. Due to (P_{01}), $\sigma, t_{00} \models \alpha_{01}$. Due to (Pn_{ij}), for all i, j, $0 \leq i \leq k$, $1 \leq j \leq n$, $\sigma, t_{ij} \models \alpha_{i,j}$. Due to (Pp_{ij}), for all i, j, $0 \leq i \leq k$, $1 \leq j < n$, $\sigma, t_{ij} + 1 \models \alpha_{ij'}$. Due to (P_{in}), for all i, $1 \leq i \leq k$, $\sigma, t_{i0} \models \alpha_{i1}$. Due to (P_{kn}), $\sigma, t_{k,n} \models \bot$, which is a contradiction. Therefore $\sigma \models (\iota \wedge \mathbf{G}\tau) \rightarrow \neg(\mathbf{GF}\rho_1 \wedge \ldots \wedge \mathbf{GF}\rho_n)$. ◇

Corollary 1: If $\vdash_{\text{GKL}} (I \wedge \mathbf{G}T) \rightarrow \phi$ then $M \models \phi$.

Theorem 2: If $M \models \phi$ then $\vdash_{\text{GKL}} (I \wedge \mathbf{G}T) \rightarrow \phi$.

Proof. Let $S := M \times M_{\neg\phi} \times M_{deg} \times M_c$, where $M_{\neg\phi}$ has fairness conditions $f_1, \ldots, f_n$ and accepts the language of $\neg\phi$, M_{deg} has a fairness condition f and accepts the language of $\mathbf{GF}f_1 \wedge \ldots \wedge \mathbf{GF}f_n$, and M_c has a counter that counts the occurrence of f. Since M has finitely many states, if $M \models \phi$, then there exists

$$\dfrac{\dfrac{\dfrac{(I \wedge I_{\neg\phi} \wedge \mathbf{G}(T \wedge T_{\neg\phi})) \rightarrow \neg(\bigwedge_{1 \le i \le n} \mathbf{GF} f_i) \quad \bigwedge_{1 \le i \le n} \mathbf{GF} f_i}{(I \wedge Enc^{-1}(I_{\neg\phi}) \wedge \mathbf{G}T \wedge \mathbf{G}Enc^{-1}(T_{\neg\phi})) \rightarrow \bot}\ \text{RES} \qquad \mathbf{G}Enc^{-1}(T_{\neg\phi})}{(I \wedge \mathbf{G}T) \rightarrow \neg Enc^{-1}(I_{\neg\phi})}\ \text{RES} \qquad \neg Enc^{-1}(I_{\neg\phi}) \rightarrow \phi}{(I \wedge \mathbf{G}T) \rightarrow \phi}\ \text{RES}$$

Fig. 1. Overall proof structure for $M \models \phi$

k such that $S \models_{fin} c \le k$, and thus there exists an inductive invariant ψ such that $I_S \rightarrow \psi$, $\psi \wedge T_S \rightarrow \psi'$, and $\psi \rightarrow c \le k$. Then, formulae (p_{01}), (pn_{ij}) for $0 \le i \le k, 1 \le j \le n$, (pp_{ij}) for $0 \le i \le k, 1 \le j < n$, and (p_{in}) for $0 \le i \le k$ are all valid. Following the construction shown in Sections IV-D and IV-E, we can generate a proof of $(I \wedge \mathbf{G}T) \rightarrow \phi$. ⋄

Corollary 2: If $\models \alpha$ then $\vdash_{\text{GKL}} \alpha$.

V. EXPERIMENTAL EVALUATION

We have implemented our proof generation procedure on top of IC3IA, a simple, open-source implementation of IC3 that uses the MATHSAT [6] SMT solver as backend. The tool supports LTL model checking of both finite and infinite-state systems (using a combination of implicit abstraction and well-founded relations, as described in [5]), but currently proof generation is only available for finite-state systems. Upon successful verification, IC3IA generates a proof certificate which can be checked by a simple companion proof checker, using purely-syntactic operations. The resolution proofs for the individual proof obligations, as described in the previous sections, are generated using the off-the-shelf proof-production capabilities provided by MATHSAT. The core of the (prototype) proof checker consists of about 500 lines of Python code. The source code of both IC3IA and the proof checker is available at http://es.fbk.eu/people/griggio/papers/fmcad2018-ltlproofs.tar.bz2, together with the benchmark instances used in our experimental evaluation, the log files of our results and the scripts to reproduce them.

For our evaluation, we have collected a total of 1150 instances from three different sources:

- the 63 safe LTL model checking problems from the 2015 hardware model checking competition (denoted HWMCC in the following); all the instances in this set are non-trivial for the model checker, with several that are very challenging also for state-of-the-art tools; all the properties in this family are of the form $\neg \bigwedge_i (\mathbf{GF} f_i)$;[2]
- 519 unsatisfiable LTL formulae from a benchmark set used in previous work on LTL satisfiability checking [29] (denoted Schuppan in the following); this set contains instances of varying difficulty, ranging from trivial to moderately-challenging; several instances are randomly-generated;
- 568 LTL model checking problems resulting from the verification of contracts of a component-based model of an aircraft wheel braking system [30] (denoted WBS in the following); the instances are typically easy, and many are in fact trivial. [3]

The main objective of our experimental analysis is to demonstrate the feasibility of proof generation in practice. For this, we performed three sets of experiments. In all cases, we used a timeout of 1200 seconds and a memory limit of 7Gb; all experiments were run on a cluster of Linux machines with 2.10GHz Intel Xeon E5-2620 CPUs and 128Gb of RAM.

A. Performance impact at model-checking time

In the first experiment, we evaluated the performance impact of the modified monitor for handling multiple fairness constraints with k-liveness, which is the only modification required at model checking time for being able to produce proofs. The results are shown in the scatter plot of Fig. 2, in which we compare the results of running IC3IA with the modified monitor that records the fairness conditions in a fixed order (x-axis) against the results when running using the standard monitor that doesn't impose any order for recording the fairness conditions (y-axis). The plot shows no clear trend for the vast majority of the instances, suggesting that the two encodings are essentially equivalent in terms of performance on average.

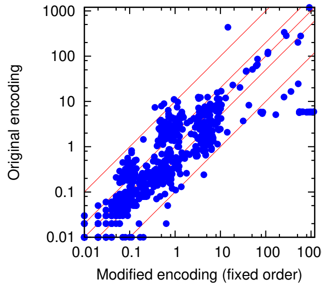


Fig. 2. Performance impact of the modified encoding for handling multiple fairness constraints.

A notable exception is the subset of problems in the `TRP/N12y` group of the Schuppan set: for these instances, the modified monitor results in a significant slowdown (up to two orders of magnitude on some instances), leading to 13 more timeouts. For these instances, it seems that the initial ordering of fairness conditions used by IC3IA, which is based on the unique internal IDs of expressions, is particularly problematic. Randomly shuffling the initial list of fairness conditions greatly mitigates the problem in this case. Although further more in-depth analyses of the correlation between the introduced overhead and the structure of the LTL properties under consideration are out of the scope of the present paper, and therefore left for future work, we can however observe that

---

[2] The benchmarks are in the Aiger format, which doesn't support arbitrary LTL properties, but only liveness properties of the above form. For most of the benchmarks, the input system therefore already corresponds to $M \times M_{\neg\phi}$ for some LTL property ϕ.

[3] This is the case e.g. for some proof obligations generated for components with a trivial assumption.

the choice of which encoding to use for handling multiple fairness conditions can have an impact on performance for two different, and at least partially conflicting, reasons. On one hand, forcing to record fairness conditions in a fixed order and one at a time has the effect of making model checker consider longer sequences of transitions before it can converge to an inductive invariant (e.g. for IC3 this causes the exploration of a longer sequence of relatively-inductive frames before reaching the fixpoint); on the other hand, however, using the modified monitor allows k-liveness to prove properties with smaller values of k, which in turn might allow the model checker to converge faster. We illustrate both situations with a simple example.

Example 2: Consider the following system $M := \langle V, I, T \rangle$:

$$V := \{c, f_1, \dots, f_{n+1}\} \qquad I := c = 0 \wedge \textstyle\bigwedge_{i=1}^{n+1} \neg f_i$$
$$T := ite(c < n, c' = c + 1, c' = c) \wedge \textstyle\bigwedge_{i=1}^{n+1} (f_i' \leftrightarrow (c < n))$$

and suppose that $n \geq 1$. M clearly satisfies the property $\varphi := \neg(\bigwedge_{i=1}^{n+1} \mathbf{GF} f_i)$, since all the $f_i's$ will stabilize to false after $n+1$ transition steps. When using the monitor that doesn't force an ordering for recording the fairness conditions, the k value needed for a k-liveness proof is n, since all fairness conditions are true for the first n steps. However, when using the modified monitor, $M \models \varphi$ can be proved with $k = 1$.

Consider instead the following variant of M, in which T is modified as follows:

$$T := ite(c < 1, c' = c + 1, c' = c) \wedge \textstyle\bigwedge_{i=1}^{n+1} (f_i' \leftrightarrow (c < 1)).$$

In this case, $k = 1$ is enough in both cases. However, the modified monitor will cause IC3 to explore a much deeper sequence of frames before finding an inductive invariant.

B. Overhead of proof generation

In our second experiment, we evaluated the impact of proof generation on the total execution time. Fig.3 shows a plot comparing the total time taken by IC3IA (x-axis) against the time required to model-check the instances, without generating a proof certificate (y-axis).

As we can see, the overhead of generating a proof gets progressively smaller as the instances become harder for the model checker. Overall, enabling proof generation results in only one lost instance compared to model checking only when using the same encoding for handling multiple fairness conditions; compared to the original encoding, 14 instances are lost.[4] A summary of the performance of the different configurations of IC3IA is reported in Table I, where the number of successfully solved instances for each benchmark family is shown.

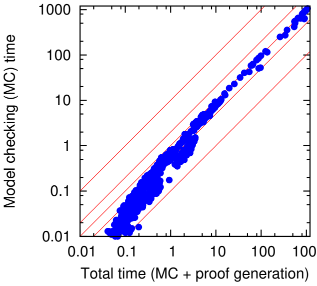


Fig. 3. Performance impact of proof generation.

---

[4] See the discussion above about this.

TABLE I
SUCCESSFULLY SOLVED INSTANCES BY BENCHMARK FAMILY.

	HWMCC	Schuppan	WBS	All
Model Checking only (original monitor for multi fairness)	31 / 63	495 / 519	568 / 568	1094 / 1150
Model Checking only (modified monitor for multi fairness)	31 / 63	482 / 519	568 / 568	1081 / 1150
MC + Proof Generation	31 / 63	481 / 519	568 / 568	1080 / 1150

TABLE II
STATISTICS ON THE SIZE OF GENERATED PROOFS.

	HWMCC	Schuppan	WBS	All
Proof size				
Median	31	151	11	31
9th percentile	64	363	31	307
Min	4	4	4	4
Max	78	723	51	723
Proof steps				
Median	125858	9601	1215	1590
9th percentile	524519	169854	17054	128901
Min	46	5	5	5
Max	6377311	1025799	1674373	6377311
Temporal steps				
Median	0	1031	9	17
9th percentile	0	15073	1	8705
Min	0	0	111	0
Max	0	128921	355	128921
Fairness conditions				
Median	4	37	2	5
9th percentile	8	90	7	76
Min	1	1	1	1
Max	10	180	12	180
Memory used (MB)				
Median	56.2	19.7	15.5	16.1
9th percentile	1163.8	31.7	29.3	31.3
Min	13.7	12.7	12.8	12.7
Max	1620.0	191.1	793.1	1620.0

Proof size: number of resolution proofs (generated by the SMT solver) for proving $I \wedge \mathbf{G}T \rightarrow \neg(\bigwedge_i \mathbf{GF} f_i)$.
Proof steps: total number of inference rules applied.
Temporal steps: total number of inference rules involving temporal axioms (from $M_{\neg\phi}$).

C. Cost of proof checking

We conclude the section presenting some data about the performance of the proof checker.

Fig. 4 shows a scatter plot comparing, for each instance, verification (x-axis) and proof checking (y-axis) times, whereas Table II presents some statistics about the size of the generated proofs. We remark though that while IC3IA is written in C++, the current implementation of the proof checker is a prototype written in Python. We expect that reimplementing the checker in C++ would lead to very significant performance improvements.

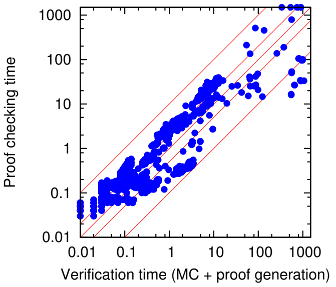


Fig. 4. Performance of proof checking (y-axis) vs verification time (x-axis).

VI. Conclusions and Future Work

In this paper, we have presented a sound and complete approach for generating proofs for LTL model checking problems using the k-liveness algorithm. The technique can be easily and efficiently implemented on top of modern SAT-based model checkers, as demonstrated by our experimental evaluation, and results in proofs that can be efficiently checked (by independent tools) using purely-syntactic rules.

We see several directions for future work. First, we would like to extend the technique to be applicable also to other SAT-based LTL model checking algorithms, such as the liveness-to-safety transformation of [31] and the FAIR algorithm of [32]. We would also like to investigate generalizations of the approach to infinite-state systems, using model checking algorithms that combine liveness-to-safety, k-liveness and ranking function synthesis [5]. Finally, from the practical perspective, we will enhance our implementation and extend it from the current prototype to a state-of-the art tool like nuXmv [33].

References

[1] K. S. Namjoshi, "Certifying model checkers," in *CAV*, ser. LNCS, vol. 2102. Springer, 2001.

[2] A. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, ser. LNCS, vol. 6538. Springer, 2011, pp. 70–87.

[3] A. Pnueli, "The Temporal Logic of Programs," in *FOCS*, 1977, pp. 46–57.

[4] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 52–59.

[5] J. Daniel, A. Cimatti, A. Griggio, S. Tonetta, and S. Mover, "Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations," in *CAV (1)*, ser. LNCS, vol. 9779. Springer, 2016.

[6] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013.

[7] M. Ben-Ari, *Mathematical logic for computer science*, ser. Prentice Hall International series in computer science. Prentice Hall, 1993.

[8] E. A. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 995–1072.

[9] Z. Manna and A. Pnueli, "Completing the Temporal Picture," *Theor. Comput. Sci.*, vol. 83, no. 1, pp. 91–130, 1991.

[10] ——, *Temporal verification of reactive systems - safety*. Springer, 1995.

[11] A. Pnueli and Y. Kesten, "A Deductive Proof System for CTL*," in *CONCUR*, 2002, pp. 24–40.

[12] E. A. Emerson, C. S. Jutla, and A. P. Sistla, "On model checking for the μ-calculus and its fragments," *Theor. Comput. Sci.*, vol. 258, no. 1-2, pp. 491–522, 2001.

[13] K. Fisler and R. P. Kurshan, "Verifying VHDL designs with COSPAN," in *FHV*, ser. LNCS, vol. 1287. Springer, 1997, pp. 206–247.

[14] D. A. Peled and L. D. Zuck, "From model checking to a temporal proof," in *SPIN*, ser. LNCS, vol. 2057. Springer, 2001.

[15] D. A. Peled, A. Pnueli, and L. D. Zuck, "From falsification to verification," in *FST TCS 2001*, ser. LNCS, R. Hariharan, M. Mukund, and V. Vinay, Eds., vol. 2245. Springer, December 2001, pp. 292–304.

[16] O. Kupferman and M. Y. Vardi, "From complementation to certification," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 83–100, 2005.

[17] C. Dax, M. Hofmann, and M. Lange, "A proof system for the linear time μ-calculus," in *FSTTCS*, ser. LNCS, vol. 4337. Springer, 2006, pp. 273–284.

[18] I. Kokkinis and T. Studer, "Cyclic proofs for linear temporal logic," *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, 171-192 2016.

[19] A. Bernasconi, C. Menghi, P. Spoletini, L. D. Zuck, and C. Ghezzi, "From model checking to a temporal proof for partial models," in *SEFM*, ser. LNCS, vol. 10469. Springer, 2017, pp. 54–69.

[20] C. Cini and A. Francalanza, "An LTL proof system for runtime verification," in *TACAS*, ser. LNCS, vol. 9035. Springer, 2015, pp. 581–595.

[21] D. Basin, B. N. Bhatt, and D. Traytel, "Optimal proofs for linear temporal logic on lasso words," 2018, available at: https://www21.in.tum.de/~traytel/papers/expl/expl.pdf.

[22] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus, "A fully verified executable LTL model checker," *Archive of Formal Proofs*, vol. 2014, 2014.

[23] U. Hustadt and B. Konev, "TRP++2.0: A temporal resolution prover," in *CADE-19*, ser. LNCS, vol. 2741. Springer, 2003.

[24] U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov, "Temp: A temporal monodic prover," in *IJCAR*, ser. LNCS, vol. 3097. Springer, 2004.

[25] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Banff Higher Order Workshop*, ser. LNCS, vol. 1043. Springer, 1995, pp. 238–266.

[26] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," *Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997.

[27] D. Prawitz, *Natural Deduction: A Proof-Theoretical Study*, ser. Dover Books on Mathematics. Dover Publications, 2006.

[28] L. M. de Moura and N. Bjørner, "Proofs and refutations, and Z3," in *LPAR Workshops*, ser. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org, 2008.

[29] V. Schuppan and L. Darmawan, "Evaluating LTL Satisfiability Solvers," in *ATVA*, ser. LNCS, vol. 6996. Springer, 2011.

[30] M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, "Formal design and safety analysis of AIR6110 wheel brake system," in *CAV (1)*, ser. LNCS, vol. 9206. Springer, 2015, pp. 518–535.

[31] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electr. Notes Theor. Comput. Sci.*, vol. 66, no. 2, pp. 160–177, 2002.

[32] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*. FMCAD Inc., 2011, pp. 144–153.

[33] 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, vol. 8559. Springer, 2014, pp. 334–342.

ISBN: 978-0-9835678-8-2.