fmcad.17

Proceedings of the 17th Conference on **Formal Methods in Computer-Aided Design (FMCAD 2017)**

TU Wien, Vienna, Austria, October 2-6, 2017

Edited by
Daryl Stewart and Georg Weissenbacher

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

acm In-Cooperation
SIGPLAN SIGSOFT

Technical co-sponsorship of IEEE

IEEE

Proceedings of the 17th Conference on

Formal Methods in Computer-Aided Design

FMCAD 2017

October 2-6, 2017

TU Wien, Vienna, Austria

Edited by Daryl Stewart and Georg Weissenbacher

ISBN: 978-0-9835678-7-5
Copyright owned jointly by the authors and FMCAD Inc.
Title page photo: Copyright TU Wien

Preface

The International Conference on Formal Methods in Computer Aided Design (FMCAD), held at TU Wien in Vienna, Austria, from October 2-6 in 2017, is the seventeenth 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 2017 comprises a tutorial day with three tutorials on security and concurrency (joint with the collocated MEMOCODE conference), two keynotes on the use of formal methods in industry, a forum for doctoral students, the Hardware Model Checking Competition 2017, the main program consisting of presentations of the accepted papers, and a Symposium in Memoriam of Helmut Veith.

The tutorial day features three presentations covering security and weak-memory concurrency (listed in the order of appearance in the program):

- "How Formal Methods and Analysis Helps Security of Entire Blockchain-based Systems", by Shin'ichiro Matsuo (MIT Media Lab, CELLOS Consortium, and BSafe.network)
- "Symbolic Security Analysis using the Tamarin Prover", by Cas Cremers (Oxford University)
- "Coalition, intrigue, ambush, destruction and pride: herding cats can be challenging", by Jade Alglave (University College London and Microsoft Research)

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

- "Automated Formal Reasoning About AWS Systems" by Byron Cook (Amazon Web Services and University College London)
- "Formal Methods in Industrial Dependable Systems Design - The TTTech Example" by Wilfried Steiner (TTTech Computertechnik AG)

FMCAD also hosts the fifth 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 2017 was organized by Keijo Heljanko and features posters and short presentations of thirteen accepted contributions. A detailed description of the Student Forum, listing all accepted contributions, is provided in the conference proceedings.

The Hardware Model Checking Competition 2017, affiliated with FMCAD 2017 and organized by Armin Biere, Tom van Dijk, and Keijo Heljanko, is a competitive event for hardware model checking tools from academia and industry. A description of this year's competition is provided in the proceedings.

The Symposium in Memoriam Helmut Veith, held on the last day of FMCAD 2017, is dedicated to the memory of Helmut Veith, who tragically passed away in March 2016. Helmut was one of the organizers of FMCAD 2016 and an active and much liked member of the FMCAD community. The Symposium honors Helmut and his contributions to the area of formal methods, which remain highly influential, with talks on model checking, synthesis, distributed algorithms, and security, given by collaborators, colleagues, and friends of Helmut and based on articles published in a Special Edition of the Journal on Formal Methods in System Design in Memoriam Helmut Veith. As part of the Symposium, a LogicLounge on Teaching Logic in Computer Science remembers Helmut's dedication to mentoring and his achievements in creating and shaping doctoral and master's programs on Logic and Computation at TU Wien. The LogicLounge is a series of discussions on computer science topics targeting a general audience, which was initiated by Helmut Veith at the Vienna Summer of Logic in 2014.

FMCAD 2017 received 87 abstracts, resulting in 67 submissions, of which 25 full papers and 4 short papers were accepted for publication in the conference proceedings. Each paper received at least four reviews, and the authors were given the opportunity to address the reviewers' concerns in a rebuttal phase. The topics of the accepted papers include solvers and decision procedures, verification of concurrent and distributed systems, analysis of hybrid and probabilistic systems, synthesis, run-time verification, a number of papers on the IC3 model checking paradigm, and applications of formal methods.

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 the Publication Chair Mitra Tabaei Befrouei (TU Wien) for her effort in preparing and assembling the conference proceedings, and Keijo Heljanko for organizing this year's FMCAD Student Forum. Our webmaster, Jens Katelaan, has our gratitude for maintaining and regularly updating the FMCAD website (which now features the new and sleek FMCAD logo designed by Anna Oberauer). 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) 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 (in alphabetical order) to our sponsors Amazon, ARM Ltd., Centaur Technology Inc., DiffBlue Ltd., Galois Inc., IBM, Microsoft Research, Oski Technology, Real Intent Inc., Synopsis, TTTech Computertechnik AG, and the Vienna Science and Technology Fund for their financial support of the conference, and the Austrian Ministry for Transport, Innovation and Technology and the National Science Foundation for their support of the Student Forum.

FMCAD 2017 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 2017 (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.

Daryl Stewart and Georg Weissenbacher
FMCAD 2017 Program Chairs
Vienna, Austria, September 2017

FMCAD 2017 is overshadowed by the death of Professor Michael J. C. Gordon, who recently (22 August 2017) passed away at the age of 69 after a brief illness. Professor Gordon was a leader in the use of mechanized formal methods to analyze hardware and software, and he was the original developer of the HOL theorem-proving system. Gordon was an expert in program semantics, and he was elected a Fellow of the Royal Society in 1994. The FMCAD community has benefited tremendously from Gordon's many contributions, and no doubt Gordon's efforts will continue to influence our community for many years to come. He will be sorely missed, not only for his wisdom and expertise but also for his distinctively generous and friendly spirit.

Warren A. Hunt Jr.
Chairman, FMCAD Steering Committee

Organization Committee

Program Co-Chairs

Daryl Stewart	ARM, UK
Georg Weissenbacher	TU Wien

Webmaster

Jens Katelaan	TU Wien

Publication Chair

Mitra Tabaei Befrouei	TU Wien

Student Forum Chair

Keijo Heljanko	Aalto University

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

Program Committee

Jade Alglave	University College London
Christel Baier	Technical University of Dresden
Roderick Bloem	TU Graz
Hana Chockler	King's College London
Andreas Griesmayer	ARM
Arie Gurfinkel	University of Waterloo
Fei He	School of Software, Tsinghua University
Keijo Heljanko	Aalto University
Alan J. Hu	University of British Columbia
Warren Hunt	University of Texas
Alexander Ivrii	IBM
Barbara Jobstmann	EPFL and Cadence Design Systems
Dejan Jovanović	SRI International
Gerwin Klein	Data61 and UNSW
Igor Konnov	TU Wien
Rebekah Leslie-Hurd	Intel
Ines Lynce	INESC-ID/IST, Universidade de Lisboa
Ken McMillan	Microsoft Research
Charles Morisset	Newcastle University
Lee Pike	Galois, Inc.
Mitra Purandare	IBM Research Laboratory Zurich
Ajitha Rajan	University of Edinburgh
Ahmed Rezine	Linköping University
Sean Safarpour	Synopsys
Roopsha Samanta	Purdue University
Martina Seidl	Johannes Kepler University Linz
Natasha Sharygina	Università della Svizzera italiana (USI Lugano, Switzerland)
Anna Slobodova	Centaur Technology
Ana Sokolova	University of Salzburg
Daryl Stewart	ARM
Murali Talupur	FormalSim
Michael Tautschnig	Queen Mary University of London
Thomas Wahl	Northeastern University
Chao Wang	University of Southern California
Georg Weissenbacher	TU Wien
Florian Zuleger	TU Wien

Additional Reviewers

Aleksandrowicz, Gadi
Arbel, Eli
Asadi, Sepideh
Ashar, Pranav
Athanasiou, Konstantinos
Auerbach, Gadiel

Bartocci, Ezio
Beringer, Lennart
Berryhill, Ryan
Bjesse, Per
Brain, Martin

Čadek, Pavel
Chapman, Erin
Chau, Cuong

Das, Shidhartha
de Paula, Flavio M.
Dubuisson, Thomas

Ebrahimi, Masoud
Elliott, Trevor
Even Mendoza, Karine

Fazekas, Katalin
Fedyukovich, Grigory
Fernandez, Matthew
Függer, Matthias

Ganai, Malay
Giefers, Heiner
Gu, Yijia
Guo, Shengjian
Günther, Henning

Hamza, Jad
Heule, Marijn
Hyvärinen, Antti

Ignatiev, Alexey
Iusupov, Rinat

Jacobs, Swen
Jain, Himanshu
Jaloyan, Georges-Axel
Janota, Mikolas

Karl, Anja
Katelaan, Jens
Khalimov, Ayrat
Kirsch, Christoph
Klüppelholz, Sascha
Koelbl, Alfred
Koyfman, Shlomit
Kragl, Bernhard
Krakovsky, Roi
Kukovec, Jure
Kusano, Markus

Laarman, Alfons
Lammich, Peter
Lazic, Marijana
Leslie-Hurd, Joe
Liu, Peizun

Malik, Sharad
Marescotti, Matteo
Mencía, Carlos
Morgado, Antonio
Märcker, Steffen

Nalla, Pradeep Kumar
Nevo, Ziv

Orni, Avigail

Pani, Thomas
Previti, Alessandro

Rabe, Markus N.
Radicek, Ivan
Ramachandran, Jaideep
Ravitch, Tristan
Rebola Pardo, Adrian
Ritirc, Daniela
Roveri, Marco

Saracino, Andrea
Schlaipfer, Matthias
Sethi, Divjyot
Sewell, Thomas
Singleton, Iain
Sison, Robert
Stoilkovska, Ilina
Sumners, Rob
Sung, Chungha
Swords, Sol

Tanaka, Miki
Tomb, Aaron
Tran, Thanh Hai

Wendler, Philipp
Winwood, Simon
Wu, Meng

Yin, Liangze

Table of Contents

How formal analysis and verification add security to blockchain-based systems

Shin'ichiro Matsuo
Keio University and BSafe.network

Extended Abstract of Tutorial Talk

***Abstract*—Blockchain is an integrated technology to ensure keeping record and process transactions with decentralized manner. It is thought as the foundation of future decentralized ecosystem, and collects much attention. However, the maturity of this technology including security of the fundamental protocol and its applications is not enough, thus we need more research on the security evaluation and verification of Blockchain technology This tutorial explains the current status of the security of this technology, its security layers and possibility of application of formal analysis and verification.**

***Index Terms*—Blockchain, Security Evaluation, Formal Method, Formal Verification, Domain Specific Language**

I. Introduction

A. Background

There are proposed many applications which aim to use blockchain technology as a fundamental distributed ledger. We expect considerable commercial interest in many new and novel applications using a blockchain. In spite of this burgeoning interest, academic research on the security model of blockchain technology and its application are at an early stage. Due to the Ethereum DAO debacle, the importance of analysis of the security of blockchain-based systems is rapidly increasing. Current research issues are to find a good framework to analyze the security of blockchain technology including defining the security requirements and the way to evaluate their security. Several existing researchers deal with how to figure out the security of blockchain by using formal analysis. To facilitate this direction of research, we need a more well-organized framework.

B. Structure of the tutorial

In this tutorial, we firstly figure out the security requirements needed for blockchain based systems and smart contracts. Then we propose technology layers for such systems and application and security considerations for each layer. Next we explore the applicability of formal analysis for each layer and pick three layers which are good targets of evaluation by formal analysis. Then, we propose the framework of applying formal analysis to help secure blockchain-based systems. An explanation of the limitations of formal *verification* follows. At the end of this tutorial, we conclude the direction to the framework to design application code and system which facilitate formal analysis and formal verification.

II. Security requirements for system and smart contract

The security definition of blockchain backbone protocol was proposed in [1], [2]. This security definition focuses on the difficulty of forgery of the block by introducing CommonPrefix property and Chain Quality property. By using these properties, we can estimate the probability which the adversary succeeds to manipulate the blockchain. This is the requirement only for protocol specification of the backbone protocol. From the system and application viewpoints, we should care about more aspects of security. Even on the protocol security, there are many assumptions in achieving its security goals. Cryptographic protocol assumes that the private cryptographic keys are kept secret at all nodes. We should analyze if the assumption surely holds.

For the application logic, there is possibility that some critical bugs remain in the program code. An adversary takes advantage of this bug to attack the application based on blockchain. The Ethereum DAO case gives us an important study that such attack may cause a rollback and a hard fork.

From above, we should cover not only the security requirements for backbone blockchain protocol, but also all mechanisms to ensure the assumptions and scripting language and codes to realize blockchain-based applications.

III. Security layers

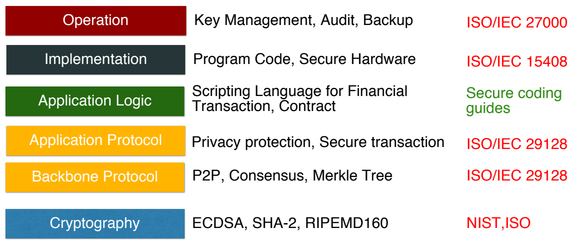
A. Technology layers and security consideration

In [3], Croman et al, proposed the technology layer of blockchain technology. This layers consist of network plane, consensus plane, storage plane, view plane and side plane. This structure is made to rethink the technology to provide more scalability.

From system and application security viewpoints, we set the technology layers by the target of evaluation. They consist of cryptography layer, backbone protocol, application protocol, application logic, implementation and operation (see Fig.1).

As this figure shows, each layer has international standards to analyze the security of the security mechanisms, except application logic. Cryptography layer is covered by standardization process of ISO, NIST and many effort by the cryptographic academic community. Security of backbone protocol is analyzed by using formal analysis and UC (Universal Composability) framework and ISO/IEC 29128 [8]. The security of implementation is certified by Common Criteria

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.



Operation	Key Management, Audit, Backup	ISO/IEC 27000
Implementation	Program Code, Secure Hardware	ISO/IEC 15408
Application Logic	Scripting Language for Financial Transaction, Contract	Secure coding guides
Application Protocol	Privacy protection, Secure transaction	ISO/IEC 29128
Backbone Protocol	P2P, Consensus, Merkle Tree	ISO/IEC 29128
Cryptography	ECDSA, SHA-2, RIPEMD160	NIST,ISO

Fig. 1. Technology layers and security consideration

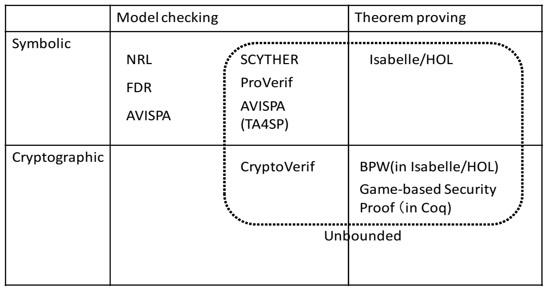


Fig. 2. Categorization of Formal Analysis for cryptographic protocol

(ISO/IEC 15408) [7] and operation of the system is defined and audited using ISMS and the framework of ISO/IEC 27000 series. Unfortunately, the application logic layer, which contains a scripting language for financial transaction and contract, does not yet have good standard to provide security analysis. Further research here is clearly required.

IV. Applying formal analysis

A. Abstract of formal analysis and formal verification

Here, we revisit the basis for the formal analysis and formal verification. Note that we distinguish between these two words. Formal analysis means evaluating the possibility of attack on the specification of the protocol, products or system by conducting some mathematical formalization of the security requirements, specifications and operational environment (an adversarial model). Is the description of the state spaces, axioms and changes both necessary and complete? Formal verification means to verify the correctness of the specification of the protocol, products or system formal methods such as automated axiomatic theorem proving or model checking. Formal analysis means a manner to use a mathematical formalization to evaluate the security and formal verification means checking if the specific protocol, product or system is qualified against the formal specification.

Formal analysis was originally used for check the existence of a bug in the circuit. Then it is applied to check the existence of bug in software code, design of the software and information system and security of cryptographic protocols.

B. State of formal analysis and checking Tools

The term formal methods refers to the use of methods for the mathematical modeling, calculation, and predication in the specification, design, analysis, construction, and assurance of hardware and software systems. These methods are distinguished as having a well-defined syntax, a semantics, and often a deductive system (or other machinery) for making semantically-sound statements about systems specified in the language of the formal method. Over the last two decades, the security community has made substantial advances in developing automated formal methods for analyzing cryptographic protocols and thereby preventing the kinds of attacks mentioned above. These methods and tools could be categorized by several points of view. Here we categorize them by “Symbolic versus Cryptographic”, “Bounded versus Unbounded”, and “Model checking versus Theorem proving” as Fig. 2.

C. Which security layer can formal method be applied?

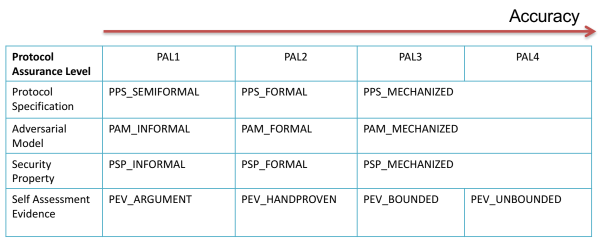
According to the past results and history of formal analysis, the following three layers are main targets of evaluation for formal analysis.

1) Implementation: This layer contains both software and hardware implementation of security mechanisms including cryptographic algorithm, protocols and key management mechanisms. Especially, crypto-token wallet programs used in general user device may become the weakest link and should be carefully implemented. In ISO/IEC 15408, there are seven EALs(evaluation assurance levels), and EAL 6 requires semiformal analysis on the design and implementation, and EAL 7 requires fully formal analysis on the design and implementation. There are many past examples and result of formal analysis in this layer.

2) Backbone protocol and application protocol: Formal analysis on the protocol specification has a long history and it gives many results to enhance the security of cryptographic protocols. ISO/IEC 9798 and 11770 are revised from results of formal analysis [4], [5]. Recently, formal analysis on TLS1.3, the latest version of TLS protocol, helps its sound development and the result is used in the IETF standardization process [6]. Recently, combination of mathematically rigorous proof (UC Framework and game-based proof) and formal analysis are used to apply formal analysis to a wider and complicated set of protocols.

3) Language for Smart Contracts: Checking the program code is the well-known application of formal analysis and we have extensive research in this area. It is not easy to check the highly complicated program by using formal analysis, there are many existing research to realize security assured language specification. For smart contracts, we will have good application by specifically defining new languages that are designed to lend themselves to formal analysis and verification.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.



Protocol Assurance Level	PAL1	PAL2	PAL3	PAL4
Protocol Specification	PPS_SEMIFORMAL	PPS_FORMAL	PPS_MECHANIZED	
Adversarial Model	PAM_INFORMAL	PAM_FORMAL	PAM_MECHANIZED	
Security Property	PSP_INFORMAL	PSP_FORMAL	PSP_MECHANIZED	
Self Assessment Evidence	PEV_ARGUMENT	PEV_HANDPROVEN	PEV_BOUNDED	PEV_UNBOUNDED

Fig. 3. PALs in ISO/IEC 29128

V. Proposal of the framework

A. Implementation

We can apply the same framework and methodology as Common Criteria (ISO/IEC 15408). Especially, wallet software or hardware should be secure against known attacking methodology like gray-box attack (side-channel attack) and white-box attack for software-only implementation. FIPS140-2 is also useful to make the framework for analyzing implementation. In this tutrial, we will provide past examples of formal analysis on the cryptographic implementation and how we can apply it to blockchain-based systems and devices.

B. Protocol

We can apply the same framework and methodology as ISO/IEC 29128 (verification of cryptographic protocols). It defines four PALs(Protocol Assurance Levels) according to the level of formalization for protocol specification, security requirements and operational environment. This framework covers combination of mathematical rigorous proof and formal analysis. In this tutrial, we will provide past examples of formal analysis on protocol specifications, how we write the report to align to this standard, and how we can apply it for analysis on backbone protocol and application protocol.

C. Language for smart contract

Analyzing the existence of bug in the program code is still fundamental research topic in computer science. We still do not have perfect results for general purpose language. The main problem is the openness of general purpose of programming language. As for the smart contract, Bhargavan et al. proposed a framework to analyze and verify both the runtime safety and the functional correctness of a Solidity contract by introducing an intermediate functional programming language suitable for verification [9]. Although the paper does not cover all EVM functionality at the time of writing this tutrial abstract, it seems a good approach to add limitation to operational environment to facilitate formal analysis.

In this tutrial, we additionally propose another approach to define a domain specific language for certain application domain, which has enough capability to write business logic and also suitable for formal verification. Then, we will present an example of the domain specific language for trade finance and trade facilitation.

VI. Limitation of formal verification and how we facilitate the use of it

In this part of the tutrial, we discuss about the limitation of the formal verification. Automated and tool-aided formal verification is strong approach to check the correctness of specification and code. However, there are two major issues when we use such automated tool. The first is on the limitation of the time and memory of the computer which executes the verification. In many formal methods, the tool finds the possibility of bug and security problems by exploring as many execution states as possible. In this case, the upper bound of runtime memory of the computer and execution time become the essential limitation for complicated programs and protocols. While there are many techniques to reduce the number of states to be explored, they are not generally sufficient for complicated software implemented in a general programming language.

The second issue is the correctness of the formalization. When we use the formal verification tool, we formalize the specification (code), security goals and operational environment. The result of execution of the tool depends on the accuracy of the formalization. However, we do not have a good tool check the accuracy. For arbitrary formalized systems, we need to check the correctness by reviewing the formalized code by humans. This limits the applicability of formal verification in general. Here, we need some kind of templates and code patterns in formalization.

From above perspective, limiting the number of states by tightly defining the language and preparing code patterns or templates are good direction to facilitate the use of formal analysis and formal verification. As for the implementation, the protection profile is the actual template for formalization. In the verification of cryptographic protools, there already exists evaluation reports which aligns to ISO/IEC 29128 and they can be used as templates. As for the language for smart contract, defining a domain specific language helps to reduce the number of states to be explored and creates a template of formalization.

VII. Conclusion

In this tutrial, we proposed the way to facilitate the application of formal analysis and formal verification by considering technology layers and their security concerns. We picked three layers, implementation, protocol and language, as targets of applications of formal analysis. Then, we propose a framework to apply formal analysis to each layer by using existing standards and results. We can use the same framework as

ISO/IEC15408 for implementation and ISO/IEC 29128 for protocol analysis. For the language, which was essential problem with the Ethereum DAO issue, defining a domain specific language is the new and effective solution and we showed an example for trade finance and trade facilitation. The domain specific language should have a design framework which facilitates formal analysis and, if possible, formal verification.

From the above, formal analysis research and technology development can deliver immediate value to the investments in blockchain technology with mutual benefits to all involved.

REFERENCES

[1] J. Garay, A. Kiayias and N. Leonardos, “The Bitcoin Backbone Protocol: Analysis and Applications,” Proceedings of Eurocrypt 2015.

[2] R. Pass, L. Seeman and A. Shelat, “Analysis of the Blockchain Protocol in Asynchronous Networks,” IACR ePrint Archive, https://eprint.iacr.org/2016/454.pdf

[3] K. Croman, C. Decker, I. Eyal, A. Efe Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, “On Scaling Decentralized Blockchains,” Proceedings of Bitcoin Workshop 2016.

[4] C. Cremers and M. Hovert, “Improving the ISO/IEC 11770 standard for key management techniques,” International Journal of Information Security, November 2016, Volume 15, Issue 6, pp 659?673.

[5] D. Basin, C. Cremers, S, Meier, “Provably repairing the ISO/IEC 9798 standard for entity authentication,” Journal of Computer Security - Security and Trust Principles, Volume 21 Issue 6, November 2013, Pages 817-846.

[6] K. Paterson and T. van der Merwe, “Reactive and Proactive Standardisation of TLS,” In Proc. of SSR 2016.

[7] ”Information technology – Security techniques – Evaluation criteria for IT security – Part 1: Introduction and general model,” ISO/IEC 15408-1:2009

[8] ”Information technology – Security techniques – Verification of cryptographic protocols,” ISO/IEC 29128:2011

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy and S. Zanella-Béguelin “Short Paper: Formal Verification of Smart Contracts,” http://www.cs.umd.edu/ aseem/solidetherplas.pdf

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Symbolic Security Analysis using the Tamarin Prover

Cas Cremers
Oxford University

Abstract of Tutorial Talk

In this talk I will present the Tamarin Prover, an analysis tool for symbolic security analysis of systems. A prime example of systems that fall within its scope are security protocols that are executed in the presence of an active attacker. Tamarins state-of-the-art analysis of such systems requires dealing with unbounded replication of processes, loops, the prolific behaviour of the attacker, and equational theories to model cryptographic operations as accurately as possible within the symbolic model.

This tutorial covers Tamarins system specification, execution model, and property specification language. I will demonstrate how Tamarin can automatically analyse systems, and how its extensive interactive mode aids in the analysis of more complex systems. Finally, I will touch upon Tamarins more advanced features and larger succesful case studies, such as the upcoming TLS 1.3 internet standard.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Coalition, intrigue, ambush, destruction and pride: herding cats can be challenging.

Jade Alglave
University College London and Microsoft Research Cambridge

Abstract of Tutorial Talk

***Abstract*—Herding cats can lead to coalition (of cheetahs), intrigue (of kittens), ambush (of tigers), destruction (of wild cats) or pride (of lions). In this tutorial, I will present the cat language to write consistency models as a set of constraints on the executions of concurrent programs. A cat model can be executed within the herd tool [3], which I will use during the tutorial.**

Concurrent programming can be difficult: how are concurrent programs supposed to behave? Do they behave correctly on exotic hardware? Formal *consistency models* can help answer these questions. Unfortunately, very often, the consistency models of the machines or operating systems we run our software on are not precisely defined. Our software itself may be written in languages whose concurrency semantics is a work in progress. To try to remedy this, the past decade has been quite rich in works aiming at describing the consistency models of hardware [20], [5], [19], [18], [6], [7], [12], [13], programming languages [10], [9], [8], [17] and more [11].

Most of these models belong to one of two formal styles: they are *operational* or *axiomatic* models. Operational models describe the executions of a concurrent program as sequences of steps: for example, reading from memory or writing to a store buffer. Axiomatic models describe executions as relations over events which represent the semantics of instructions: relations represent for example the order in which instructions are executed, or who reads from where. Both styles have advantages: operational models can be quite close to hardware designs, thus becoming a good device to communicate with hardware folks. Axiomatic models can be quite abstract, which leads to concise models and efficient verification [2].

The cat language [7] is a domain-specific language which allows the user to describe axiomatic consistency models as a set of constraints on executions. It has been used to describe hardware models such as ARMv7 and IBM Power [7], Nvidia GPUs [1], HSA GPUs [4], C++ and OpenCL [8]. More recently, ARM has released an official cat file as part of their formalisation of their ARMv8 consistency model [15]. A cat model can be executed by the herd tool [3], to answer questions about the semantics of concurrent code.

In this tutorial, I will present the cat language and the herd tool. By the end of this tutorial, you should have the skills required to build several models amongst the following: Sequential Consistency [14], Total Store Order [20], IBM Power [7], ARM [12], Nvidia GPUs [1], C++ [8] and Linux [16]. I hope to make this tutorial interactive, using the herd tool. For this to go smoothly, I would suggest downloading and installing the herd tool from http://diy.inria.fr.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

References

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *ASPLOS 2015*.

[2] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient Bounded Model Checking of concurrent software. In *CAV 2013*, pages 141–157.

[3] Jade Alglave and Luc Maranget. The diy7 tool suite. http://diy.inria.fr/, 2011–2017.

[4] Jade Alglave and Luc Maranget. Towards a formalisation of the HSA memory model in the cat language. http://www.hsafoundation.com/?ddownload=5381, 2015.

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV 2010*, pages 258–272.

[6] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *FMSD 2012*, 40(2):170–205.

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

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

[9] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From c++11 to POWER. In *POPL 2012*, pages 509–520.

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

[11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR 2015*, pages 58–71.

[12] 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 2016*, pages 608–621.

[13] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL 2017*, pages 429–442.

[14] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[15] ARM Ltd., editor. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Limited, 2017.

[16] Paul E. McKenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern. Linux-kernel memory ordering: Help arrives at last! In *LinuxCon Europe*, 2016. http://www.rdrop.com/users/paulmck/scalability/paper/LinuxMM.2016.10.04c.LCE.pdf.

[17] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633.

[18] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *PLDI 2012*, pages 311–322.

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

[20] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *CACM 2010*, 53(7):89–97.

Automated Formal Reasoning About AWS Systems

Byron Cook
Senior Principal Engineer, Amazon Web Services
Professor, University College London

Abstract of Invited Talk

Automatic and semiautomatic formal verification tools are now being developed and used within Amazon Web Services (AWS) to find proofs that prove or disprove desired properties of key AWS components. In this session, we outline these efforts and discuss how tools are used to play and then replay found proofs of desired properties when software artifacts or networks are modified, thus helping provide security throughout the lifetime of the AWS system.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Formal Methods in Industrial Dependable Systems Design - The TTTech Example

Wilfried Steiner
TTTech

Abstract of Invited Talk

Over the last decades the field of dependable computer systems has gained tremendous significance in our modern society. We rely on the dependability of automobiles, railways, airplanes, medical devices, critical infrastructures, like the electrical grid or industrial production facilities, and many more. These dependable systems frequently implement non-trivial mechanisms, for example, to coordinate between redundant components, and a guarantee of correctness of these mechanisms is therefore crucial to avoid catastrophic incidents. Consequently, formal methods are frequently used in industrial dependable system design and in this talk I will discuss the various aspects in which formal methods are and have been deployed for specification, verification, and configuration at TTTech for critical networking products.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Hardware Model Checking Competition 2017

Armin Biere armin.biere@jku.at
Tom van Dijk tom.vandijk@jku.at
Johannes Kepler University Linz, Austria
Keijo Heljanko keijo.heljanko@aalto.fi
Aalto University, Finland

The Hardware Model Checking Competition (HWMCC) 2017 affiliated to the International Conference on Formal Methods in Computer Aided Design (FMCAD) in 2017 in Vienna was the 9th competitive event for hardware model checkers we organized. After HWMCC'15 affiliated with FMCAD'15 in Austin, the competition took a break in 2016.

The competition has its roots in the model checking community with focus on hardware verification, a former central theme in International Conference on Computer-Aided Verification (CAV) and the first three incarnations of the competition in 2007, 2008 and 2010 were affiliated with CAV. This topic is now more at home at FMCAD, the primary place for research in formal methods for hardware. Accordingly the hardware model checking competition stays with FMCAD (2011,2012,2013,2015,2017) except when CAV is part of the Federated Logic Conference (FLoC) as in 2014 [4].

The goal in organizing this competition is to keep up the driving force in improving hardware model checkers. We also want to motivate implementors to present their work to a broader audience. Another important objective is to collect realistic benchmarks and to make them available to the research community. Both academia and industry is invited to submit solvers and benchmarks. Competiting model checkers have to solve benchmarks in the AIGER format [2], [3].

The competition in 2017 had multiple tracks. The most important track was the single safety property track (SINGLE). As in previous years we also had a (single) liveness property track (LIVE), and a deep bound track (DEEP), but no multiple property track. The winner of the deep bound track received an award of $500 sponsored by Oski Technology.

The tracks were run in the same way as in the previous four incarnations of the competition, except that we were using our new cluster running Ubuntu 16.04.2 64 bit. Each cluster node had two Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz CPUs and 128 GB of main memory.

Each solver had full access to both processors on one node, thus combined 16 cores (32 virtual cores) and 128 GB of main memory. Accordingly a memory limit of 120GB was enforced. As in the last competition in 2015 affiliated to FMCAD'15 we were further using a time limit of 1 hour of wall clock-time.

Also as before the number of submissions was restricted to at most two model checkers per submitter and model checkers were required to produce witnesses in the SINGLE track. These witnesses were checked by the AIGSIM tool, which is part of the AIGER tools [1].

Supported by Austrian Science Fund (FWF) NFN S11408-N23 (RiSE)

Except for the new hardware, competition rules, as well as input and output formats [2] did not change compared to previous competitions. As starting with HWMCC'12 model checkers competing in the DEEP bound track were requested to print the bounds reached during running in the SINGLE track. In the SINGLE track model checkers were required to print witnesses traces if a bad state was claimed to be reachable. These witnesses serve as certificates for satisfiable bad state properties and were checked for correctness.

Again as in HWMCC'14 and HWMCC'15, in order to avoid glitches in interpreting the format, the SINGLE track only used AIGER pre 1.9 single property benchmarks [2], with the single bad state property encoded as an output (MILOA header with O = 1). All latches were assumed to be initialized implicitly to zero as it is the default in the pre 1.9 AIGER format [2].

There was no change in the LIVE track which of course used the AIGER 1.9 format [3] nor in the DEEP track. Solvers intended to participate in the DEEP track were run in the SINGLE track and were expected to print reached bounds as in previous years (see for instance HWMCC'12).

In the previous competition HWMCC'15 we were proposing to completely switch to the AIGER 1.9 format [3] (also in the SINGLE track), add back the multiple property track, provide support for fuzzing and delta-debugging, and last but not least to establish a word-level track. However, due to lack of resources, we had to postpone these changes again.

References

[1] AIGER library and tools. http://fmv.jku.at/aiger.

[2] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

[3] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[4] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko. Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:135–172, 2014 (published 2016).

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

The FMCAD 2017 Graduate Student Forum

Keijo Heljanko
Aalto University, Finland

***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 2017, the event took place in Vienna, Austria, as integral part of the FMCAD conference. Thirteen 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 automated reasoning, model checking of hardware, software, as well as parameterized systems, verification of concurrent programs, and checking of floating point properties.**

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 2017 Graduate Student Forum follows the tradition of its predecessors, which took place in Mountain View, CA, USA in 2016 [1], Austin, Texas, USA in 2015 [2], Lausanne, Switzerland in 2014 [3], and in Portland, Oregon, USA in 2013 [4].

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, 13 high quality submissions were accepted. 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 range of topics in the field of verification, such as automated reasoning, model checking of hardware, software, as well as parameterized systems, verification of concurrent programs, and checking of floating point properties.

The following contributions have been accepted:

- Yulia Demyanova, Thomas Pani, Helmut Veith and Florian Zuleger: *Empirical Software Metrics for Benchmarking of Verification Tools*
- Sepideh Asadi, Karine Even-Mendoza, Grigory Fedyukovich, Antti Hyvärinen, Hana Chockler and Natasha Sharygina: *HiFrog: Interpolation-based Software Verification using Theory Refinement*
- Thanh Hai Tran and Jure Kukovec: *Pattern-based abstractions for parameterized model checking of distributed algorithms*
- David Declerck, Sylvain Conchon and Fatiha Zaidi: *A Backward Reachability Algorithm for Parameterized Systems on Weak Memory*
- William Hallahan, Ruzica Piskac and Anton Xue: *Building a Symbolic Execution Engine for Haskell*
- Samuel Pastva: *Discrete Bifurcation Analysis of Reactive Systems*
- Rohit Dureja and Kristin Yvonne Rozier: *From One To Many: Checking A Set Of Models*
- Adrian Rebola Pardo: *Satisfiability-preserving Reasoning in Software Verification*
- Ákos Hajdu and Zoltan Micskei: *Towards Using Multiple Counterexamples for Abstraction Refinement*
- Yiji Zhang, Lenore Zuck and Kedar Namjoshi: *An LLVM Refinement Checker and its Applications*
- Andreas Fellner: *Model-based, mutation-driven test case generation via heuristic-guided branching search*
- Lucas Martinelli Tabajara: *Synthesis via CNF Decomposition*
- Jaideep Ramachandran: *Unified Solver Strategy for Floating-Point*

The 2017 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 generous support of the National Science Foundation and FMCAD's sponsors enabled us to subsidize the travel cost of the participating students. The help and advice of Georg Weissenbacher, FMCAD 2017 General and PC chair who organized the earlier FMCAD 2015 student forum was invaluable. Also help from Warren Hunt and Lindy Aleshire was instrumental in providing and administering the NSF grant for subsidizing travel costs.

References

[1] H. Hojjat, "The FMCAD 2016 graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD).* FMCAD Inc, 2016, p. 8.

[2] G. Weissenbacher, "The FMCAD 2015 graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD).* FMCAD Inc, 2015, p. 8.

[3] R. Piskac, "The FMCAD 2014 graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD).* IEEE, 2014, p. 13.

[4] T. Wahl, "The FMCAD graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD).* IEEE, 2013, pp. 16–17.

[1]http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD17/

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

goSAT: Floating-point Satisfiability as Global Optimization

M. Ammar Ben Khadra, Dominik Stoffel, Wolfgang Kunz
Department of Electrical and Computer Engineering
University of Kaiserslautern, Germany
{khadra,stoffel,kunz}@eit.uni-kl.de

***Abstract*—We introduce goSAT, a fast and publicly available SMT solver for the theory of floating-point arithmetic. We build on the recently proposed XSat solver [1] which casts the satisfiability problem to a corresponding global optimization problem. Compared to XSat, goSAT is an integrated tool combining JIT compilation of SMT formulas and NLopt, a feature-rich mathematical optimization backend. We evaluate our tool using several optimization algorithms and compare it to XSat, Z3, and MathSat. Our evaluation demonstrates promising results.**

***Index Terms*—satisfiability modulo theories, decision procedure, floating-point, global optimization**

I. Introduction

Automated bit-precise reasoning over floating-point arithmetic (FPA) is essential for a wide range of applications. For instance, test generation and program synthesis. SMT solvers are often used as a backend to implement such reasoning. Improving the support for FPA theory has been tackled in several recent works [2]–[5]. Despite these advances, the performance of SMT solvers regarding FPA theory still suffers from relatively poor scalability. Moreover, clauses involving common non-linear functions, e.g., trigonometric, remain particularly difficult. In fact, modern SMT solvers are based on DPLL(T) as their central framework. Therefore, their core SAT engines can be ineffective in deducing facts that might otherwise be "obvious" at the theory level [3]. In the following, we elaborate on two key challenges raised by FPA theory.

Standard complexity. The IEEE 754-2008 standard defines seven core operations that need to be correctly rounded, namely, {+,−,⋆,/,rem,sqrt,fma}. The result of a core operation is affected by the rounding mode, five defined modes, and whether it involves a special number {NaN, $\pm\infty$}. Also, rules for type conversion and exception handling, e.g., overflow, need to be considered.

Tunable approximation. FPA is an approximation of reals by definition. In practice, FPA implementations are *tunable* depending on the required performance and precision. For example, the flag `-ffast-math` instructs GCC to enable FP optimizations that are less precise. Moreover, a function like `sin` might be evaluated using a software library or a single hardware instruction with potentially different results [6]. Further, function `sin` might even be evaluated at compile time with correct rounding[1]. Therefore, sound reasoning about FPA should take into account the semantics of various approximate implementations of a single function. This can overwhelm SMT solvers particularly in the case of non-linear functions.

To address this, Fu et al. recently proposed XSat [1], an SMT solver for FPA based on mathematical optimization. XSat works by transforming a quantifier-free SMT instance $\mathcal{F}(\vec{x})$, where $\vec{x} \in \mathrm{FP}^n$, to a corresponding objective function $\mathcal{G}(\vec{x})$. The latter represents a distance value that needs to be minimized by Global Optimization (GO) techniques [7]. The goal is to find an assignment α satisfying $\mathcal{G}(\alpha) = 0$. The key advantage of XSat is that it doesn't need to explicitly encode FPA semantics. Rather, it can guide its reasoning purely by observing the outputs of $\mathcal{G}(\vec{x})$. Consequently, it can generally reason about any *executable* function. The original implementation of XSat consists of (1) a code generator that generates $\mathcal{G}(\vec{x})$ in C language, and (2) a Python tool that invokes Basin Hopping (BH) [8], a GO algorithm built in Scipy[2], to find a satisfying α. Note that the C code of $\mathcal{G}(\vec{x})$ needs to be compiled as a C extension to Python in a separate step which makes XSat difficult to use.

In this work, we build on the ideas proposed in XSat. We make a number of contributions. First, goSAT is an integrated tool that generates the objective function $\mathcal{G}(\vec{x})$ using Just-in-Time (JIT) compilation and directly attempts to solve it on-the-fly. Second, our backend is based on the feature-rich non-linear optimization library NLopt [9]. In contrast, XSat is restricted to the BH algorithm. Third, in addition to its native solving mode, goSAT has a code generation mode similar to XSat. This enables experimenting with various optimization libraries that are not yet natively supported by goSAT. Fourth, we evaluate our tool on the same benchmarks used in XSat. We employ various GO algorithms available in NLopt and compare them with the BH algorithm. Finally, we make our tool publicly available at (https://github.com/abenkhadra/gosat).

II. Background

We discuss here the theoretical basis of goSAT. Given an SMT formula $\mathcal{F}(\vec{x})$, where $\vec{x} \in \mathrm{FP}^n$, we need to systematically derive a corresponding objective function $\mathcal{G}(\vec{x})$. Evaluating $\mathcal{G}(\vec{x})$ for a particular assignment α returns a distance value that becomes smaller as we get closer to the global minimum at zero. In order to establish the equivalence between

[1]GCC supports compile-time evaluation of built-in functions that have constant arguments since v4.3: https://gcc.gnu.org/gcc-4.3/changes.html

[2]Popular Python library for scientific computing: https://www.scipy.org/

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

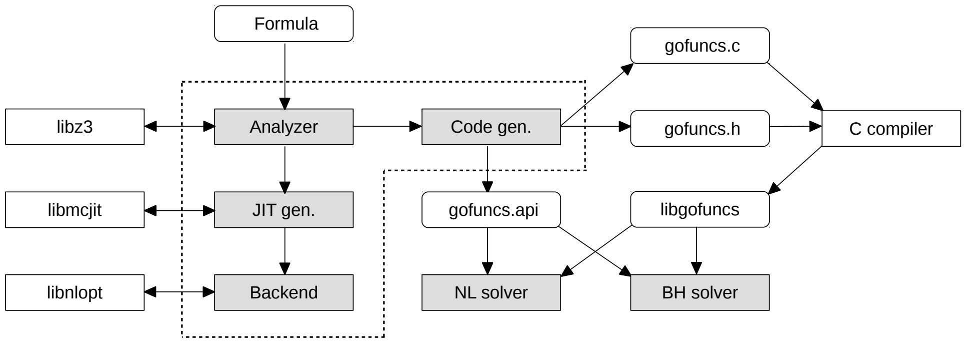


Fig. 1. goSAT architecture

satisfiability of $\mathcal{F}(\vec{x})$ and global optimization of $\mathcal{G}(\vec{x})$, the function $\mathcal{G}(\vec{x})$ must satisfy (R1) $\forall \vec{x} \in \mathrm{FP}^n, \mathcal{G}(\vec{x}) \geq 0$ and (R2) $\mathcal{G}(\alpha) = 0 \Leftrightarrow \alpha \models \mathcal{F}(\vec{x})$.

Consider $\mathcal{F}(\vec{x})$ to be in the language $\mathcal{L}_{fp}$ defined over quantifier-free FPA. Our $\mathcal{L}_{fp}$ is slightly modified to that found in XSat, namely,

Boolean constraints	$\pi \ := \ \neg\pi' \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid e_1 \bowtie e_2$
Arithmetic expressions	$e \ := \ c \mid x \mid e1 \otimes e2 \mid H(e_1, ...e_n)$

where $\bowtie \in \{<, \leq, >, \geq, ==, \neq\}$, $\otimes \in \{+, -, *, /\}$, c is a floating-point constant, x is a variable, and H can be any user-defined function, e.g., logarithm.

Let $\mathcal{F}_c(\vec{x})$ be $\mathcal{F}(\vec{x})$ after eliminating $\neg$ using De-Morgan's law and transforming it to CNF,

$$\mathcal{F}_c(\vec{x}) \overset{\text{def}}{=} \bigwedge_{i \in I} \bigvee_{j \in J} e_{i,j} \bowtie_{i,j} e'_{i,j} \tag{1}$$

we derive $\mathcal{G}(\vec{x})$ from $\mathcal{F}_c(\vec{x})$ as follows:

$$\mathcal{G}(\vec{x}) \overset{\text{def}}{=} \sum_{i \in I} \prod_{j \in J} d(\bowtie_{i,j}, e_{i,j}, e'_{i,j}) \tag{2}$$

where,

$$d(\leq, e_1, e_2) \ \overset{\text{def}}{=} \ e_1 \leq e_2 \ ? \ 0 \ : \ \theta(e_1, e_2) \tag{3}$$
$$d(<, e_1, e_2) \ \overset{\text{def}}{=} \ e_1 < e_2 \ ? \ 0 \ : \ \theta(e_1, e_2) + 1 \tag{4}$$
$$d(\geq, e_1, e_2) \ \overset{\text{def}}{=} \ e_1 \geq e_2 \ ? \ 0 \ : \ \theta(e_1, e_2) \tag{5}$$
$$d(>, e_1, e_2) \ \overset{\text{def}}{=} \ e_1 > e_2 \ ? \ 0 \ : \ \theta(e_1, e_2) + 1 \tag{6}$$
$$d(==, e_1, e_2) \overset{\text{def}}{=} \ \theta(e_1, e_2) \tag{7}$$
$$d(\neq, e_1, e_2) \ \overset{\text{def}}{=} \ e_1 \neq e_2 \ ? \ 0 \ : \ 1 \tag{8}$$

Function $\theta(x_1, x_2)$ represents the distance between bit representations of x_1 and x_2. It has the following key properties:

$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) \geq 0 \tag{9}$$
$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) = 0 \quad \Rightarrow x_1 = x_2 \tag{10}$$
$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) = \theta(x_2, x_1) \tag{11}$$

From equations (2) to (11), it can be shown that $\mathcal{G}(\vec{x})$ satisfies requirements R1 and R2. Consequently, goSAT provides a sound method for proving FPA satisfiability. However, completeness of goSAT depends on the applied GO algorithm.

Generally, GO algorithms can be classified into deterministic [10] and stochastic [11]. The former are complete by providing a guarantee of finding a global minimum within a finite time. However, their applicability usually depends on the type of considered function, e.g., convex functions. Also, they often require the user to provide first and/or second derivatives (gradient and Hessian, respectively). In comparison, stochastic methods are more flexible by being applicable to functions as black box. This comes at the expense of not guaranteeing convergence to global minimum.

III. Implementation details

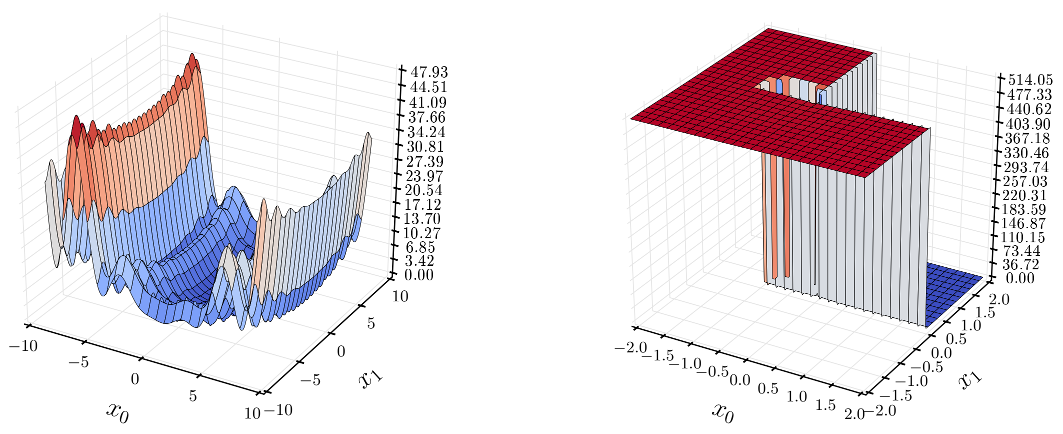
Now we discuss the implementation of goSAT. We begin with its native solving mode. Then, we move to discuss its code generation mode and helper utilities, namely, `NL solver` and `BH solver`. Finally, we discuss our choice of optimization algorithms and their parameter tuning. Our discussion will be based on Fig. 1. Highlighted components are part of our contribution. Our implementation language is C++ except for the `BH solver` which is written in Python.

A. Native solving mode

This is the default mode of goSAT where it accepts an SMT file as input. The `Analyzer` parses the input file using the facilities of `libz3` to get an expression (`expr`) representing the formula. Then, the `Analyzer` constructs an LLVM module that contains the objective function $\mathcal{G}(\vec{x})$. The latter is passed to a `JIT generator` that traverses `expr` in a post-order manner in order to generate the corresponding LLVM IR. The translation process is syntax-directed resembling equations (2) to (11) discussed previously. Next, function $\mathcal{G}(\vec{x})$ is just-in-time compiled (jitted) and optimized using `libmcjit` from the LLVM framework. A pointer to the jitted $\mathcal{G}(\vec{x})$ is provided to our `Backend` alongside other required data structures. Finally, the `Backend` configures and invokes `libnlopt` on function $\mathcal{G}(\vec{x})$ in order to find a satisfying model.

B. Code generation mode

This tool mode is similar to what is implemented in XSat. We developed it in order to facilitate experimentation with GO algorithms that we still do not natively support in goSAT.



(a) (b)

Fig. 2. Topologies of (a) `levy` function compared to (b) `f23` function generated by goSAT. Functions generated by goSAT are non-smooth, however, they exhibit more regularity which is a key property for goSAT to work in practice.

Additionally, we provide two utilities, `NL solver` and `BH solver`, to demonstrate its use. The former depends on NLopt as its backend while the latter uses Scipy as its backend. Note that Scipy currently supports only one GO algorithm, namely, basin hopping. We were able of reproducing (most) results published in XSat using our `BH Solver`.

This goSAT mode is mainly implemented in the code generation component, refer to Fig. 1, which receives an `expr` after parsing the input formula by `Analyzer`. Code generation is realized using syntax-directed translation similar to the native solving mode. The output of this mode are C code and header files. These need to be compiled to obtain a shared library `libgofuncs`. Additionally, goSAT generates an `api` text file which is required to properly call the functions in `libgofuncs`. The `api` file, in its simplest forms, lists the name and dimension (variable count) of each $\mathcal{G}(\vec{x})$.

C. *Optimization algorithms*

We decided to use NLopt as our backend since it is publicly available and supports several derivative-free non-linear GO algorithms. There are, however, other open source packages for large-scale non-linear optimization, e.g., IpOpt [12]. Unfortunately, they generally have restrictions regarding the types of supported functions and the availability of derivatives. Note that open-source derivative-free GO algorithms still lack in performance compared to commercial implementations [13].

Our next step was to profile various GO algorithms implemented in NLopt to experiment with their efficiency and reliability. To this end, we chose several standard functions that have multiple local minima, e.g., `levy`, `griewank`, and `rastrigin`. These functions are commonly used for benchmarking GO algorithms [14]. We ended up choosing four promising derivative-free algorithms, namely, the deterministic algorithm `DIRECT` and the stochastic algorithms `CRS2`, `ISRES`, and `MLSL`[3]. Note that algorithm parameters play a crucial rule in convergence to global minima. For example, consider the `levy` function depicted in Fig. 2a which has a global minimum $\mathcal{G}(\vec{x}) = 0$ for $\vec{x} = (1, 1)$. Basin Hopping (BH) with default parameters and an initial guess $x = (-8.2, 1)$ was unable of "hopping" over the barrier and was trapped at a local minimum 6.056. Convergence to the global minimum required increasing the Monte-Carlo step size to 2.0. Fortunately, the transformation implemented in goSAT produces functions with more regularity. For example, consider formula `f23` depicted in Fig. 2b which is taken from the Griggio benchmarks [15]. BH quickly converged to the satisfiable area using default parameters despite setting an initial guess that is far away at $\vec{x} = (-10^9, -10^9)$. Actually, it is easy see, from equations (3)-(8), that $\mathcal{G}(\vec{x})$ generated by goSAT are non-smooth due to the use of conditional statements. However, they exhibit some regular structure that makes them easier to solve compared to standard GO benchmarking functions.

[3]Please refer to NLopt algorithm documentation for further details.

IV. Evaluation

We evaluated goSAT on the entire Griggio benchmark set (214 instances). The GO algorithms used in the evaluation are `DIRECT`, `CRS2`, `ISRES`, and `MLSL`. In order to draw a comparison with XSat (`BH` algorithm), we used goSAT to generate a `libgofuncs` library representing the same benchmark instances. Then, we provided `libgofuncs` as input to our `BH solver`.

We "reasonably" tuned algorithm parameters in order to provide a fair comparison. The initial guess for all algorithms was set to zeros, step size to 0.5, and timeout to 600s. Each algorithm was executed once per instance. This makes `BH solver` achieve slightly different results to those reported in XSat. The latter uses a restart strategy using multiple initial guesses. Note that native goSAT has a small extra overhead compared to `NL solver` since it needs first to parse and JIT the input formula. We draw a comparison with Z3 v4.5 and MathSat v5.3.14. Both solvers were used with their default parameters. Experiments were conducted on a Linux machine with 8 GB RAM and Intel® Core i7 processors.

TABLE I
EVALUATION RESULTS

	sat	unsat	timeout	errors	avg. time
CRS2	91	123	0	0	2.60
ISRES	88	126	0	0	2.89
BH	89	113	0	12	4.43
MLSL	56	116	0	42	5.30
DIRECT	45	169	0	0	13.60
MathSat	100	68	46	0	55.54
Z3	85	60	65	4	71.39

Results are summarized in Tab. I. We provide the number of `sat`, `unsat`, `timeout`, and `error` instances together with the average query time in seconds (excluding timeout and error instances). Some GO algorithms faced numerical errors, e.g., round-off. Z3 encountered 4 out-of-memory exceptions. In the case of goSAT, error instances can be considered `unsat` since GO algorithms are generally incomplete. We used Z3 to externally validate all `sat` models returned by goSAT.

Our results provide a rough comparison since algorithm parameters can be tuned further. For instance, using the same function evaluation limit of 5×10^5, the deterministic `DIRECT` algorithm needed more time and found fewer `sat` instances compared to the stochastic `CRS2`. Fig. 3 compares the solving time of `BH` algorithm to `CRS2` and `DIRECT` (fastest and slowest in goSAT respectively). Note that the performance of `DIRECT` varies relatively widely across the benchmarks. Also, `BH` needed a maximum of 488s for one instance while `CRS2` was able to respond in about 25% of that time at most.

Overall, GO algorithms can provide a viable alternative to conventional SMT solvers for FPA particularly in the case of formulas involving non-linear functions. Moreover, they can assist them in special applications, e.g., in Optimization-Modulo-Theory (OMT) [16], [17]. Note, however, that SMT solvers often need to reason about multiple theories which is still not possible in goSAT. The theory of quantifier-free bit-vectors (BV) can be particularly relevant in combination with FPA in the domains of software verification and synthesis. Recently, Fröhlich et al. [18] demonstrated promising results in applying stochastic search for solving BV satisfiability directly on the theory level. This provides potential ideas for combining BV and FPA to be solved using stochastic search.

V. Conclusion

We introduced goSAT, an SMT solver for the theory of FPA. In contrast to XSat, goSAT is capable of natively solving SMT formulas and is publicly available. Unlike conventional solvers, goSAT is based on mathematical optimization which enables it to reason, in principle, about any executable function. There are, however, several areas for future improvement. Most notably, we plan to exploit the particular structure of $\mathcal{G}(\vec{x})$ generated by goSAT in order to improve solving effectiveness. Also, our restriction to derivative-free GO algorithms might be too strict. Relaxing this restriction might be possible using automatic differentiation techniques.

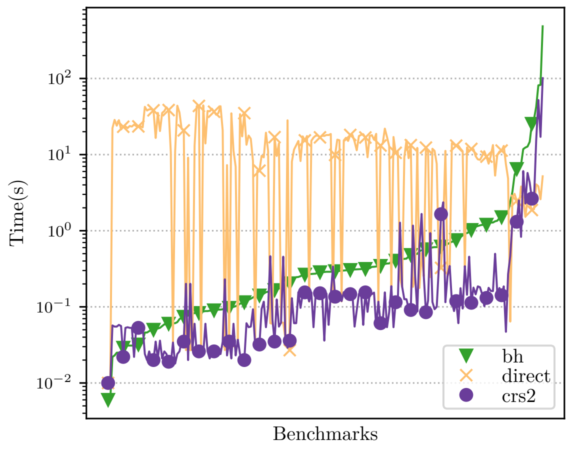


Fig. 3. Solving time of CRS2 and DIRECT compared to BH used in XSat.

References

[1] Z. Fu and Z. Su, "XSat: A Fast Floating-Point Satisfiability Solver," in *Computer Aided Verification (CAV'16)*. Springer, 2016, pp. 187–209.
[2] K. Scheibler, F. Neubauer, A. Mahdi, M. Franzle, T. Teige, T. Bienmuller, D. Fehrer, and B. Becker, "Accurate ICP-based floating-point reasoning," in *Formal Methods in Computer-Aided Design (FMCAD'16)*. IEEE, 2016, pp. 177–184.
[3] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *Proceeding of Formal Methods in Computer-Aided Design (FMCAD'12)*, 2012, pp. 131–140.
[4] A. Zeljić, C. M. Wintersteiger, and P. Rümmer, "Approximations for Model Construction," in *Proceedings of 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, 2014, pp. 344–359.
[5] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Deciding floating-point logic with abstract conflict driven clause learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.
[6] S. Duplichan, "Intel overstates FPU accuracy." [Online]. Available: http://notabs.org/fpuaccuracy/
[7] R. Horst, P. M. Pardalos, and N. V. Thoai, *Introduction to global optimization*, 2nd ed. Springer, 2000.
[8] D. J. Wales and J. P. K. Doye, "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms," *Journal of Physical Chemistry*, 1997.
[9] S. G. Johnson, "The NLopt nonlinear-optimization package." [Online]. Available: http://ab-initio.mit.edu/nlopt
[10] C. A. Floudas and C. E. Gounaris, "A review of recent advances in global optimization," *Journal of Global Optimization*, vol. 45, no. 1, pp. 3–38, 2009.
[11] J. C. Spall, *Introduction to Stochastic Search and Optimization*. Hoboken, NJ, USA: John Wiley & Sons, Inc., mar 2003.
[12] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.
[13] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.
[14] M. Jamil and X. S. Yang, "A literature survey of benchmark functions for global optimisation problems," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, p. 150, 2013.
[15] "Benchmarks of QF_FP track in SMT-COMP (2015)." [Online]. Available: http://www.cs.nyu.edu/~barrett/smtlib/QF_FP_Hierarchy.zip
[16] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with SMT solvers," in *Proceedings of 41st ACM Symposium on Principles of Programming Languages (POPL'14)*. ACM, 2014, pp. 607–618.
[17] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ - An Optimizing SMT Solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 194–199.
[18] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi, "Stochastic Local Search for Satisfiability Modulo Theories," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2015.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

On Sound Relative Error Bounds for Floating-Point Arithmetic

Anastasiia Izycheva and Eva Darulova
Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany
Email: izycheva@mpi-sws.org, eva@mpi-sws.org

***Abstract*—State-of-the-art static analysis tools for verifying finite-precision code compute worst-case absolute error bounds on numerical errors. These are, however, often not a good estimate of accuracy as they do not take into account the magnitude of the computed values. Relative errors, which compute errors relative to the value's magnitude, are thus preferable. While today's tools do report relative error bounds, these are merely computed via absolute errors and thus not necessarily tight or more informative. Furthermore, whenever the computed value is close to zero on part of the domain, the tools do not report any relative error estimate at all. Surprisingly, the quality of relative error bounds computed by today's tools has not been systematically studied or reported to date.**

In this paper, we investigate how state-of-the-art static techniques for computing sound absolute error bounds can be used, extended and combined for the computation of relative errors. Our experiments on a standard benchmark set show that computing relative errors *directly*, as opposed to via absolute errors, is often beneficial and can provide error estimates up to six orders of magnitude tighter, i.e. more accurate. We also show that interval subdivision, another commonly used technique to reduce over-approximations, has less benefit when computing relative errors directly, but it can help to alleviate the effects of the inherent issue of relative error estimates close to zero.

I. Introduction

Numerical software, common in embedded systems or scientific computing, is usually designed in real-valued arithmetic, but has to be implemented in finite precision on digital computers. Finite precision, however, introduces unavoidable roundoff errors which are individually usually small, but which can accumulate and affect the validity of computed results. It is thus important to compute *sound* worst-case roundoff error bounds to ensure that results are accurate enough - especially for safety-critical applications. Due to the unintuitive nature of finite-precision arithmetic and its discrepancy with continuous real arithmetic, *automated* tool support is essential.

One way to measure worst-case roundoff is *absolute error*:

$$err_{\text{abs}} = \max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right| \tag{1}$$

where f and x denote a possibly multivariate real-valued function and variable respectively, and $\hat{f}$ and $\hat{x}$ their finite-precision counter-parts. Note that absolute roundoff errors are only meaningful on a restricted domain, as for unrestricted x the error would be unbounded in general. In this paper, we consider interval constraints on input variables, that is for each input variable $x \in I = [a, b]$, $a, b \in \mathbb{R}$.

Furthermore, we focus on floating-point arithmetic, which is a common choice for many finite-precision programs, and for which several tools now exist that compute absolute errors fully automatically for nonlinear straight-line code [1]–[4].

Absolute errors are, however, not always an adequate measure of result quality. Consider for instance an absolute error of 0.1. Whether this error is small and thus acceptable for a computation depends on the application as well as the magnitude of the result's value: if $|f(x)| \gg 0.1$, then the error may be acceptable, while if $|f(x)| \approx 0.1$ we should probably revise the computation or increase its precision. *Relative error* captures this relationship:

$$err_{\text{rel}} = \max_{x \in I} \left| \frac{f(x) - \hat{f}(\hat{x})}{f(x)} \right| \tag{2}$$

Note that the input domain needs to be restricted also for relative errors.

Today's static analysis tools usually report absolute as well as relative errors. The latter is, however, computed via absolute errors. That is, the tools first compute the absolute error and then divide it by the largest function value:

$$err_{\text{rel\_approx}} = \frac{\max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right|}{\min_{x \in I} |f(x)|} \tag{3}$$

Clearly, Equation 2 and Equation 3 both compute sound relative error bounds, but $err_{\text{rel\_approx}}$ is an over-approximation due to the loss of correlation between the nominator and denominator. Whether this loss of correlation leads to coarse error bounds in practice has, perhaps surprisingly, not been studied yet in the context of automated sound error estimation.

Beyond curiosity, we are interested in the automated computation of relative errors for several reasons. First, relative errors are more informative and often also more natural for user specifications. Secondly, when computing sound error bounds, we necessarily compute over-approximations. For absolute errors, the over-approximations become bigger for larger input ranges, i.e. the error bounds are less tight. Since relative errors consider the range of the expression, we expect these over-approximations to be smaller, thus making relative errors more suitable for modular verification.

One may think that computing relative errors is no more challenging than computing absolute errors; this is not the case for two reasons. First, the complexity of computing relative errors is higher (compare Equation 1 and Equation 2) and due to the division, the expression is now nonlinear even for linear f. Both complexity and nonlinearity have already

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

been challenging for absolute errors computed by automated tools, usually leading to coarser error bounds. Furthermore, whenever the range of f includes zero, we face an inherent division by zero. Indeed, today's static analysis tools report no relative error for most standard benchmarks for this reason.

Today's static analysis tools employ a variety of different strategies (some orthogonal) for dealing with the over-approximation of worst-case absolute roundoff errors due to nonlinear arithmetic: the tool Rosa uses a forward dataflow analysis with a linear abstract domain combined with a nonlinear decision procedure [3], Fluctuat augments a similar linear analysis with interval subdivision [1], and FPTaylor chooses an optimization-based approach [2] backed by a branch-and-bound algorithm.

In this paper, we investigate how today's methods can be used, extended and combined for the computation of relative errors. To the best of our knowledge, this is the first systematic study of fully automated techniques for the computation of relative errors. We mainly focus on the issue of computing tight relative error bounds and for this extend the optimization based approach for computing absolute errors to computing relative errors *directly* and show experimentally that it often results in tighter error bounds, sometimes by up to six orders of magnitude. We furthermore combine it with interval subdivision (we are not aware of interval subdivision being applied to this approach before), however, we note that, perhaps surprisingly, the benefits are rather modest.

We compare this direct error computation to forward analysis which computes relative errors via absolute errors on a standard benchmark set, and note that the latter outperforms direct relative error computation only on a single univariate benchmark. On the other hand, this approach clearly benefits from interval subdivision.

We also observe that interval subdivision is beneficial for dealing with the inherent division by zero issue in relative error computations. We propose a practical (and preliminary) solution, which reduces the impact of potential division-by-zero's to small subdomains, allowing our tool to compute relative errors for the remainder of the domain. We demonstrate on our benchmarks that this approach allows our tool to provide more useful information than state-of-the-art tools.

Contributions:

- We extend an optimization-based approach [2] for bounding absolute errors to relative errors and thus provide the first feasible and fully automated approach for computing relative errors *directly*.
- We perform the first experimental comparison of different techniques for automated computation of sound relative error bounds.
- We show that interval subdivision is beneficial for reducing the over-approximation in absolute error computations, but less so for relative errors computed directly.
- We demonstrate that interval subdivision provides a practical solution to the division by zero challenge of relative error computations for certain benchmarks.

We have implemented all techniques within the tool Daisy [5], which is available at https://github.com/malyzajko/daisy.

II. Background

We first give a brief overview over floating-point arithmetic as well as state-of-the-art techniques for automated sound worst-case absolute roundoff error estimation.

A. Floating-Point Arithmetic

The error definitions in section I include a finite-precision function $\hat{f}(\hat{x})$ which is highly irregular and discontinuous and thus unsuitable for automated analysis. We abstract it following the floating-point IEEE 754 standard [6], by replacing every floating-point variable, constant and operation by:

$$\begin{gathered} x \odot y = (x \circ y)(1+e) + d, \\ \tilde{x} = x(1+e) + d \qquad \tilde{\sqrt{x}} = \sqrt{x}(1+e) + d \end{gathered} \tag{4}$$

where $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ and $\circ \in \{+, -, \times, /\}$ are floating-point and real arithmetic operations, respectively. e is the relative error introduced by rounding at each operation and is bounded by the so-called machine epsilon $|e| \leq \epsilon_M$. Denormals (or subnormals) are values with a special representation which provide gradual underflow. For these, the roundoff error is expressed as an absolute error d and is bounded by δ_M, (for addition and subtraction $d = 0$). This abstraction is valid in the absence of overflow and invalid operations resulting in Not a Number (NaN) values. These values are detected separately and reported as errors. In this paper, we consider double precision floating-point arithmetic with $\epsilon_M = 2^{-53}$ and $\delta_M = 2^{-1075}$. Our approach is parametric in the precision, and thus applicable to other floating-point types as well.

Using this abstraction we replace $\hat{f}(\hat{x})$ with a function $\tilde{f}(x, e, d)$, where x are the input variables and e and d the roundoff errors introduced for each floating-point operation. In general, x, e and d are vector-valued, but for ease of reading we will write them without vector notation. Note that our floating-point abstraction is real-valued. With this abstraction, we and all state-of-the-art analysis tools approximate absolute errors as:

$$err_{abs} \leq \max_{x \in I, |e_i| \leq \epsilon_M, |d_i| \leq \delta} \left| f(x) - \tilde{f}(x, e, d) \right| \tag{5}$$

B. State-of-the-art in Absolute Error Estimation

When reviewing existing automated tools for absolute roundoff error estimation, we focus on their techniques for reducing over-approximations due to nonlinear arithmetic in order to compute tight error bounds.

Rosa [3] computes absolute error bounds using a forward data-flow analysis and a combination of abstract domains. Note that the magnitude of the absolute roundoff error at an arithmetic operation depends on the magnitude of the operation's value (this can easily be seen from Equation 4), and these are in turn determined by the input parameter ranges. Thus, Rosa tracks two values for each intermediate abstract syntax tree node: a sound approximation of the range and the worst-case absolute error. The transfer function for errors

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

uses the ranges to propagate errors from subexpressions and to compute the new roundoff error committed by the arithmetic operation in question.

One may think that evaluating an expression in interval arithmetic [7] and interpreting the width of the resulting interval as the error bound would be sufficient. While this is sound, it computes too pessimistic error bounds, especially if we consider relatively large ranges on inputs: we cannot distinguish which part of the interval width is due to the input interval or due to accumulated roundoff errors. Hence, we need to compute ranges and errors separately.

Rosa implements different range arithmetics with different accuracy-efficiency tradeoffs for bounding ranges and errors. To compute ranges, Rosa offers a choice between interval arithmetic, affine arithmetic [8] (which tracks linear correlations between variables) and a combination of interval arithmetic with a nonlinear arithmetic decision procedure. The latter procedure first computes the range of an expression in standard interval arithmetic and then refines, i.e. tightens, it using calls to the nlsat [9] decision procedure within the Z3 SMT solver [10]. For tracking errors, Rosa uses affine arithmetic; since roundoff errors are in general small, tracking linear correlations is in general sufficient.

Fluctuat [1] is an abstract interpreter which separates errors similarly to Rosa and which uses affine arithmetic for computing both the ranges of variables and for the error bounds. In order to reduce the over-approximations introduced by affine arithmetic for nonlinear operations, Fluctuat uses interval subdivision. The user can designate up to two variables in the program whose input ranges will be subdivided into intervals of equal width. The analysis is performed separately and the overall error is then the maximum error over all subintervals. Interval subdivision increases the runtime of the analysis significantly, especially for multivariate functions, and the choice of which variables to subdivide and by how much is usually not straight-forward.

FPTaylor, unlike Daisy and Fluctuat, formulates the roundoff error bounds computation as an optimization problem, where the absolute error expression from Equation 1 is to be maximized, subject to interval constraints on its parameters. Due to the discrete nature of floating-point arithmetic, FPTaylor optimizes the continuous, real-valued abstraction (Equation 5). However, this expression is still too complex and features too many variables for optimization procedures in practice, resulting in bad performance as well as bounds which are too coarse to be useful (see subsection V-A for our own experiments). FPTaylor introduces the Symbolic Taylor approach, where the objective function of Equation 5 is simplified using a first order Taylor approximation with respect to e and d:

$$\tilde{f}(x, e, d) = \tilde{f}(x, 0, 0) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) e_i + R(x, e, d), \quad (6)$$

$$R(x, e, d) = \frac{1}{2} \sum_{i,j=1}^{2k} \frac{\partial^2 \tilde{f}}{\partial y_i \partial y_j}(x, p) y_i y_j + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial d_i}(x, 0, 0) d_i$$

where $y_1 = e_1, \ldots, y_k = e_k, y_{k+1} = d_1, \ldots, y_{2k} = d_k$ and $p \in \mathbb{R}^{2k}$ such that $|p_i| \le \epsilon_M$ for $i = 1 \ldots k$ and $|p_i| \le \delta$ for $i = k{+}1 \ldots 2k$. The remainder term R bounds all higher order terms and ensures soundness of the computed error bounds.

The approach is symbolic in the sense that the Taylor approximation is taken wrt. e and d only and x is a symbolic argument. Thus, $f(x, 0, 0)$ represents the function point where all inputs x remain symbolic and no roundoff errors are present, i.e. $e = d = 0$ and $f(x, 0, 0) = f(x)$. Choosing $e = d = 0$ as the point at which to perform the Taylor approximation and replacing e_i with its upper bound ϵ_M reduces the initial optimization problem to:

$$err_{abs} \le \epsilon_M \max_{x \in I} \sum_{i=1}^{k} \left| \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) \right| + M_R \quad (7)$$

where M_R is an upper bound for the error term $R(x, e, d)$ (more details can be found in [2]). FPTaylor uses interval arithmetic to estimate the value of M_R as the term is second order and thus small in general.

To solve the optimization problem in Equation 7, FPTaylor uses rigorous branch-and-bound optimization. Branch-and-bound is also used to compute the resulting real function $f(x)$ range, which is needed for instance to compute relative errors. Real2Float [4], another tool, takes the same optimization-based approach, but uses semi-definite programming for the optimization itself.

III. Bounding Relative Errors

The main goal of this paper is to investigate how today's sound approaches for computing absolute errors fare for bounding relative errors and whether it is possible and advantageous to compute relative errors directly (and not via absolute errors). In this section, we first concentrate on obtaining tight bounds in the presence of nonlinear arithmetic, and postpone a discussion of the orthogonal issue of division by zero to the next section. Thus, we assume for now that the range of the function for which we want to bound relative errors does not include zero, i.e. $0 \notin f(x)$ and $0 \notin \tilde{f}(\tilde{x})$, for $x, \tilde{x}$ within some given input domain. We furthermore consider f to be a straight-line arithmetic expression. Conditionals and loops have been shown to be challenging [11] even for absolute errors and we thus leave their proper treatment for future work. We consider function calls to be an orthogonal issue; they can be handled by inlining, thus reducing to straight-line code, or require suitable summaries in postconditions, which is also one of the motivations for this work.

The forward dataflow analysis approach of Rosa and Fluctuat does not easily generalize to relative errors, as it requires intertwining the range and error computation. Instead, we extend FPTaylor's approach to computing relative errors directly (subsection III-A). We furthermore implement interval subdivision (subsection III-B), which is an orthogonal measure to reduce over-approximation and experimentally evaluate different combinations of techniques on a set of standard benchmarks (subsection V-A).

A. Bounding Relative Errors Directly

The first strategy we explore is to compute relative errors directly, without taking the intermediate step through absolute errors. That is, we extend FPTaylor's optimization based approach and maximize the relative error expression using the floating-point abstraction from Equation 4:

$$\max |\tilde{g}(x,e,d)| = \max_{x\in I, |e_i|\le \epsilon_M, |d_i|\le\delta} \left| \frac{f(x) - \tilde{f}(x,e,d)}{f(x)} \right| \quad (8)$$

The hope is to preserve more correlations between variables in the nominator and denominator and thus obtain tighter and more informative relative error bounds.

We call the optimization of Equation 8 without simplifications the *naive approach*. While it has been mentioned previously that this approach does not scale well [2], we include it in our experiments (in subsection V-A) nonetheless, as we are not aware of any concrete results actually being reported. As expected, the naive approach returns error bounds which are so large that they are essentially useless.

We thus simplify $\tilde{g}(x,e,d)$ by applying the Symbolic Taylor approach introduced by FPTaylor [2]. As before, we take the Taylor approximation around the point $(x,0,0)$, so that the first term of the approximation is zero as before: $\tilde{g}(x,0,0) = 0$. We obtain the following optimization problem:

$$\max_{x\in I, |e_i|\le \epsilon_M, |d_i|\le\delta} \sum_{i=1}^{k} \left| \frac{\partial \tilde{g}}{\partial e_i}(x,0,0)e_i \right| + M_R$$

where M_R is an upper bound for the remainder term $R(x,e,d)$. Unlike in Equation 7 we do not pull the factor e_i for each term out of the absolute value, as we plan to compute tight bounds for mixed-precision in the future, where the upper bounds on all e_i are not all the same (this change does not affect the accuracy for uniform precision computations).

Computing Upper Bounds: The second order remainder R is still expected to be small, so that we use interval arithmetic to compute a sound bound on M_R (in our experiments R is indeed small for all benchmarks except 'doppler'). To bound the first order terms $\frac{\partial \tilde{g}}{\partial e_i}$ we need a sound optimization procedure to maintain overall soundness, which limits the available choices significantly.

FPTaylor uses the global optimization tool Gelpia [12], which internally uses a branch-and-bound based algorithm. Unfortunately, we found it difficult to integrate because of its custom interface. Furthermore, we observed unpredictable behavior in our experiments (e.g. nondeterministic crashes and substantially varying running times for repeated runs on the same expression).

Instead, we use Rosa's approach which combines interval arithmetic with a solver-based refinement. Our approach is parametric in the solver and we experiment with Z3 [10] and dReal [13]. Both support the SMT-lib interface, provide rigorous results, but are based on fundamentally different techniques. Z3 implements a complete decision procedure for nonlinear arithmetic [9], whereas dReal implements the framework of δ-complete decision procedures. Internally, it is based on a branch-and-bound algorithm and is thus in principle similar to Gelpia's optimization-based approach.

Note that the queries we send to both solvers are (un)satisfiability queries, and not optimization queries. For the nonlinear decision procedure this is necessary as it is not suitable for direct optimization, but the branch-and-bound algorithm in dReal is performing optimization internally. The reason for our roundabout approach for dReal is that while the tool has an optimization interface, it uses a custom input format and is difficult to integrate. We expect this approach to affect mostly performance, however, and not accuracy.

B. Interval Subdivision

The over-approximation committed by static analysis techniques grows in general with the width of the input intervals, and thus with the width of all intermediate ranges. Intuitively, the worst-case error which we consider is usually achieved only for a small part of the domain, over-approximating the error for the remaining inputs. Additionally the domain where worst-case errors are obtained may be different at each arithmetic operation. Thus, by subdividing the input domain we can usually obtain tighter error bounds. Note that interval subdivision can be applied to any error estimation approach. Fluctuat has applied interval subdivision to absolute error estimation, but we are not aware of a combination with the optimization-based approach, nor about a study of its effectiveness for relative errors.

We apply subdivision to input variables and thus compute:

$$\max_{k\in[1\ldots m]} \left(\max_{x_j \in I_{jk}} |\tilde{g}(x,e,d)| \right) \quad (9)$$

where m is an number of subdivisions for each input interval. That is, for multivariate functions, we subdivide the input interval for each variable and maximize the error over the Cartesian product. Clearly, the analysis running time is exponential in the number of variables. While Fluctuat limits subdivisions to two user-designated variables and a user-defined number of subdivisions each, we choose to limit the total number of analysis runs by a user-specified parameter p. That is, given p, m (the desired number of subdivisions for each variable), and n (number of input variables), the first $\lfloor \log_m(p-n) \rfloor$ variables' domains are subdivided m times, with larger input domains subdivided first. The remaining variable ranges remain undivided.

C. Implementation

We implement all the described techniques in the tool Daisy [5]. Daisy, a successor of Rosa [3], is a source-to-source compiler which generates floating-point implementations from real-valued specifications given in the following format:

```
def bspline3(u: Real): Real = {
  require(0 <= u && u <= 1)
    - u * u * u / 6.0
}
```

Daisy is parametric in the *approach* (naive, forward dataflow analysis or optimization-based), the *solver* used (Z3 or dReal)

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

and the number of *subdivisions* (including none). Any combination of these three orthogonal choices can be run by simply changing Daisy's input parameters.

Furthermore, Daisy simplifies the derivative expressions in the optimization-based approach ($x + 0 = x, x \times 1 = x$, etc.). Unsimplified expressions may affect the running time of the solvers (and thus also the accuracy of the error bounds), as we observed that the solvers do not necessarily perform these otherwise straight-forward simplifications themselves.

Finally, to maintain soundness, we need to make sure that we do not introduce internal roundoff errors during the computation of error bounds. For this purpose we implement all internal computations in Daisy using infinite-precision rationals.

IV. Handling Division by Zero

An important challenge arising while computing relative errors is how to handle potential divisions by zero. State-of-the-art tools today simply do not report any error at all whenever the function range contains zero. Unfortunately, this is not a rare occurrence; on a standard benchmark set for floating-point verification, many functions exhibit division by zero (see Table III for our experiments).

Note that these divisions by zero are *inherent* to the expression which we consider and are usually not due to over-approximations in the analysis. Thus, we can only *reduce* the effect of division by zero, but we cannot eliminate it entirely. Here, we aim to reduce the domain for which we cannot compute relative errors. Similar to how relative and absolute errors are combined in the IEEE 754 floating-point model (Equation 4), we want to identify parts of the input domain on which relative error computation is not possible due to division by zero and compute absolute errors. For the remainder of the input domain, we compute relative errors as before.

We use interval subdivision (subsection III-B), attempting to compute relative errors (with one of the methods described before) on every subdomain. Where the computation fails due to (potential) division by zero, we compute the absolute error and report both to the user:

```
relError: 6.6614143807162e-16
On several sub-intervals relative error cannot be computed.
Computing absolute error on these sub-intervals.
For intervals (u -> [0.875,1.0]), absError: 9.66746937132909e-19
```

While the reported relative error bound is only sound for parts of the domain, we believe that this information is nonetheless more informative than either no result at all or only an absolute error bound, which today's tools report and which may suffer from unnecessary over-approximations.

We realize that while this approach provides a practical solution, it is still preliminary and can be improved in several ways. First, a smarter subdivision strategy would be beneficial. Currently, we divide the domain into equal-width intervals, and vary only their number. The more fine-grained the subdivision, the bigger part of the domain can be covered by relative error computations, however the running time increases correspondingly. Ideally, we could exclude from the relative error computation only a small domain around the zeros of the function f. While for univariate functions, this approach is straight-forward (zeros can be, for instance, obtained with a nonlinear decision procedure), for multivariate functions this is challenging, as the zeros are not simple points but curves. Secondly, subdivision could only be used for determining which sub-domains exhibit potential division by zero. The actual relative error bound computation can then be performed on the remainder of the input domain without subdividing it. This would lead to performance improvements, even though the 'guaranteed-no-zero' domain can still consist of several disconnected parts. Again, for univariate functions this is a straight-forward extension, but non-trivial for multivariate ones. Finally, we could compute $\max_{x_j \in I_{jk}} \left| \frac{f(x) - \tilde{f}(x,e,d)}{f(x)+\epsilon} \right|$. for some small ϵ, which is a standard approach in scientific computing. It is not sound, however, so that we do not consider it here.

V. Experimental Evaluation

We compare the different strategies for relative error computation on a set of standard benchmarks with FPTaylor and the forward dataflow analysis approach from Rosa (now implemented in Daisy) as representatives of state-of-the-art tools. We do not compare to Fluctuat directly as the underlying error estimation technique based on forward analysis with affine arithmetic is very similar to Daisy's. Indeed experiments performed previously [2], [11] show only small differences in computed error bounds. We rather choose to implement interval subdivision within Daisy.

All experiments are performed in double floating-point precision (the precision FPTaylor supports), although all techniques in Daisy are parametric in the precision. The experiments were performed on a desktop computer running Debian GNU/Linux 8 64-bit with a 3.40GHz i5 CPU and 7.8GB RAM. The benchmarks bsplines, doppler, jetEngine, rigidBody, sine, sqrt and turbine are nonlinear functions from [3]; invertedPendulum and the traincar benchmarks are linear embedded examples from [14]; and himmilbeau and kepler are nonlinear examples from the Real2Float project [4].

A. Comparing Techniques for Relative Error Bounds

To evaluate the accuracy and performance of the different approaches for the case when no division by zero occurs, we modify the standard input domains of the benchmarks whenever necessary such that the function ranges do not contain zero and all tools can thus compute a non-trivial relative error bound.

Table I shows the relative error bounds computed with the different techniques and tools, and Table II the corresponding analysis times. Bold marks the best result, i.e. tightest computed error bound, for each benchmark. The column 'Under-approx' gives an (unsound) relative error bound obtained with dynamic evaluation on 100000 inputs; these values provide an idea of the true relative errors. The 'Naive approach' column confirms that simplifications of the relative error expression are indeed necessary (note the exponents of the computed bounds).

TABLE I
RELATIVE ERROR BOUNDS COMPUTED BY DIFFERENT TECHNIQUES

	Bench-mark	Under-approx	Daisy	FPTaylor	Naive approach	Daisy + subdiv	DaisyOPT			
							Z3	dReal	Z3+subdiv	dReal+subdiv
Univariate	bspline0	1.46e-15	4.12e-13	4.26e-13	5.11e+02	7.44e-14	**3.00e-15**	**3.00e-15**	**3.00e-15**	**3.00e-15**
	bspline1	7.91e-16	**2.54e-15**	3.32e-15	4.16e-01	5.32e-15	3.22e-15	3.22e-15	3.22e-15	3.22e-15
	bspline2	2.74e-16	1.11e-15	1.16e-15	5.22e-01	1.61e-15	**8.92e-16**	9.76e-16	**8.92e-16**	**8.92e-16**
	bspline3	5.49e-16	2.46e-10	3.07e-10	5.12e+05	5.23e-11	**6.66e-16**	**6.66e-16**	**6.66e-16**	**6.66e-16**
	sine	2.84e-16	8.94e-16	8.27e-16	4.45e-01	1.39e-15	**7.66e-16**	**7.66e-16**	**7.66e-16**	**7.66e-16**
	sineOrder3	3.65e-16	1.04e-15	1.10e-15	1.39e-01	1.99e-15	**8.94e-16**	**8.94e-16**	**8.94e-16**	**8.94e-16**
	sqroot	4.01e-16	1.04e-15	1.21e-15	1.02e+00	2.20e-15	**1.02e-15**	**1.02e-15**	**1.02e-15**	**1.02e-15**
Multivariate	doppler	1.06e-15	2.08e-04	6.13e-07	2.09e+08	2.60e-05	**1.93e-13**	**1.94e-13**	**1.93e-13**	**1.94e-13**
	himmilbeau	8.46e-16	6.55e-13	7.89e-13	6.69e+02	9.81e-15	6.54e-13	1.98e-15	7.05e-15	**1.99e-15**
	invPendulum	3.74e-16	2.09e-11	2.48e-11	1.64e+00	1.22e-11	**1.21e-15**	**1.35e-15**	**1.21e-15**	**1.52e-15**
	jet	1.45e-15	9.26e-15	7.53e-15	3.87e+00	1.40e-13	**4.47e-15**	5.12e-15	6.03e-15	6.51e-15
	kepler0	4.39e-16	1.31e-12	1.64e-12	2.16e+03	3.63e-12	3.97e-12	2.39e-15	**1.63e-15**	2.64e-15
	kepler1	7.22e-16	2.17e-11	2.59e-11	7.93e+04	8.70e-13	3.80e-11	**1.29e-15**	2.85e-13	1.71e-15
	kepler2	5.28e-16	4.01e-10	5.65e-15	4.09e+05	1.35e-11	4.56e-10	2.42e-15	8.58e-12	**2.26e-15**
	rigidBody1	4.49e-16	8.77e-11	1.14e-10	1.55e+00	2.50e-11	**9.78e-16**	1.27e-15	**9.78e-16**	1.46e-15
	rigidBody2	5.48e-16	3.91e-12	4.73e-12	5.14e+03	1.77e-12	**2.21e-15**	2.33e-15	**2.21e-15**	2.96e-15
	traincar_state8	2.72e-15	2.16e-13	2.69e-13	2.91e+02	2.16e-13	**7.67e-14**	2.72e-13	**7.67e-14**	2.50e-13
	traincar_state9	8.11e-16	3.44e-13	4.31e-13	3.47e+02	1.91e-13	**3.45e-14**	4.15e-13	**3.45e-14**	2.38e-13
	turbine1	5.79e-16	6.47e-13	1.48e-13	4.16e+02	6.81e-13	**2.06e-15**	3.07e-15	**2.06e-15**	3.90e-15
	turbine2	1.03e-15	5.26e-15	4.25e-15	4.81e+00	1.66e-13	**4.12e-15**	4.30e-15	**4.12e-15**	4.33e-15
	turbine3	7.41e-16	3.52e-13	7.43e-14	2.13e+02	3.91e-13	**1.91e-14**	**1.92e-14**	**1.91e-14**	**1.93e-14**

TABLE II
ANALYSIS TIME OF DIFFERENT TECHNIQUES FOR RELATIVE ERRORS ON BENCHMARKS WITHOUT DIVISION BY ZERO

Benchmark	Daisy	FPTaylor	Naive approach	Daisy + subdiv	DaisyOPT			
					Z3	dReal	Z3 + subdiv	dReal + subdiv
bsplines	6s	13s	13m 25s	0.34s	20s	25s	27s	30s
sines	5s	8s	13m 45s	0.42s	1m 4s	1m 21s	1m 8s	1m 9s
sqroot	3s	6s	6m 4s	0.15s	14s	12s	14s	14s
doppler	5s	2m 11s	2m 14s	1s	1m 59s	2m 35s	2m 58s	7m 28s
himmilbeau	9s	4s	5m 30s	0.36s	1m 50s	1m 16s	6m 15s	8m 5s
invPendulum	3s	5s	1m 31s	0.15s	7s	37s	25s	3m 54s
jet	20s	17s	19m 35s	7s	30m 40s	32m 24s	45m 31s	2 h 20m 49s
kepler	37s	39s	14m 41s	1s	3m 27s	16m 29s	12m 20s	27m 56s
rigidBody	11s	8s	10m 4s	0.39s	30s	1m 18s	1m 26s	8m 37s
traincar	10s	42s	8m 15s	1s	1m 1s	10m 43s	4m 7s	18m 35s
turbine	11s	28s	17m 25s	2s	5m 29s	11m 28s	12m 30s	42m 36s
total	1m 60s	5m 1s	1h 52m 28s	13s	46m 42s	1h 18m 45s	1h 27m 22s	4h 19m 53s

The last four columns show the error bounds when relative errors are computed directly using the optimization based approach (denoted 'DaisyOPT') from subsection III-A, with the two solvers and with and without subdivisions. For subdivisions, we use $m = 2$ for univariate benchmarks, $m = 8$ for multivariate and $p = 50$ for both as in our experiments these parameters showed a good trade-off between performance and accuracy. For most of the benchmarks we find that direct evaluation of relative errors computes tightest error bounds with acceptable analysis times. Furthermore, for most benchmarks Z3, resp. its nonlinear decision procedure, is able to compute slightly tighter error bounds, but for three of our benchmarks dReal performs significantly better, while the running times are comparable.

Somewhat surprisingly, we note that interval subdivision has limited effect on accuracy when combined with direct relative error computation, while also increasing the running time significantly.

Comparing against state-of-the-art techniques (columns Daisy and FPTaylor), which compute relative errors via absolute errors, we notice that the results are sometimes several orders of magnitude less accurate than direct relative error computation (e.g. six orders of magnitude for the bspline3 and doppler benchmarks).

The column 'Daisy+subdiv' shows relative errors computed via absolute errors, using the forward analysis with subdivision (with the same parameters as above). Here we observe that unlike for the directly computed relative errors, interval subdivision is mostly beneficial.

Finally, for the experiments in Table I, we use as large input domains as possible, without introducing result ranges which include zero. When comparing relative error bounds computed for smaller and larger input domains, where a small input domain means that the input intervals have smaller width, we observe that relative errors computed directly usually scale better than relative errors computed via absolute errors, i.e. the over-approximation committed is smaller. For example, (for space reasons only) for the doppler benchmark we obtain the following relative errors:

	Daisy (via absolute)	relative err. directly
small input domain	1.48e-11	1.26e-15
large input domain	2.08e-04	1.93e-13

B. Handling Division by Zero

To evaluate whether interval subdivision is helpful when dealing with the inherent division by zero challenge, we now consider the standard benchmark set, with standard input domains. Table III summarizes our results. We first note that division by zero indeed occurs quite often, as the missing results in the Daisy and FPtaylor columns show.

The last three columns show our results when using interval subdivision. Note that to obtain results on as many benchmarks as possible we had to change the parameters for subdivision to $m = 8$ and $p = 50$ for univariate and $m = 4, p = 100$ for multivariate benchmarks. The result consists of three values: the first value is the maximum relative error computed over the sub-domains where relative error was possible to compute; in the brackets we report the maximum absolute error for the sub-domains where relative error computation is not possible, and the integer is the amount of these sub-domains where absolute errors were computed. We only report a result if the number of sub-domains with division by zero is less than 80% of the total amount of subdomains, as larger numbers would probably be impractical to be used within, e.g. modular verification techniques. Whenever we report '-' in the table, this means that division by zero occurred on too many or all subdomains.

We observe that while interval subdivision does not provide us with a result for all benchmarks, it nonetheless computes information for more benchmarks than state-of-the-art techniques.

VI. Related work

The goal of this work is an automated and sound static analysis technique for computing tight relative error bounds for floating-point arithmetic. Most related are current static analysis tools for computing *absolute* roundoff error bounds [1]–[4].

Another closely related tool is Gappa [15], which computes both absolute and relative error bounds in Coq. It appears relative errors can be computed both directly and via absolute errors. The *automated* error computation in Gappa uses intervals, thus, a computation via absolute errors will be less accurate than Daisy performs. The direct computation amounts to the naive approach, which we have shown to work poorly.

The direct relative error computation was also used in the context of verifying computations which mix floating-point arithmetic and bit-level operations [16]. Roundoff errors are computed using an optimization based approach similar to FPTaylor's. Their approach is targeted to specific low-level operations including only polynomials, and the authors do not use Taylor's theorem. However, tight error estimates are not the focus of the paper, and the authors only report that they use whichever bound (absolute or relative) is better. we are not aware of any systematic evaluation of different approaches for sound relative error bounds.

More broadly related are abstract interpretation-based static analyses which are sound wrt. floating-point arithmetic [17], [18], some of which have been formalized in Coq [19] These domains, however, do not quantify the difference between the real-valued and the finite-precision semantics and can only show the absence of runtime errors such division-by-zero or overflow.

Floating-point arithmetic has also been formalized in an SMT-lib [20] theory and solvers exist which include floating-point decision procedures [20], [21]. These are, however, not suitable for roundoff error quantification, as a combination with the theory of reals would be at the propositional level only and thus not lead to useful results.

Floating-point arithmetic has also been formalized in theorem provers such as Coq [22] and HOL Light [23], and some automation support exists in the form of verification condition generation and reasoning about ranges [24], [25]. Entire numerical programs have been proven correct and accurate within these [26], [27]. While very tight error bounds can be proven for specific computations [28], these verification efforts are to a large part manual and require substantial user expertise in both floating-point arithmetic as well as theorem proving. Our work focuses on a different trade-off between accuracy, automation and generality.

Another common theme is to run a higher-precision program alongside the original one to obtain error bounds by testing [29]–[32]. Testing has also been used as a verification method for optimizing mixed-precision computations [33], [34]. These approaches based on testing, however, only consider a limited number of program executions and thus cannot prove sound error bounds.

VII. Conclusion

We have presented the first experimental investigation into the suitability of different static analysis techniques for sound accurate relative error estimation. Provided that the function range does not include zero, computing relative errors *directly* usually yields error bounds which are (orders of magnitude) more accurate than if relative errors are computed via absolute errors (as is current state-of-the-art). Surprising to us, while interval subdivision is beneficial for absolute error estimation, when applied to direct relative error computation it most often does not have a significant effect on accuracy.

We furthermore note that today's rigorous optimization tools could be improved in terms of reliability as well as scalability. Finally, while interval subdivision can help to alleviate the effect of the inherent division by zero issue in relative error computation, it still remains an open challenge.

TABLE III
RELATIVE ERROR BOUNDS COMPUTED BY DIFFERENT TECHNIQUES ON STANDARD BENCHMARKS (WITH POTENTIAL DIVISION BY ZERO)

Benchmark	**Daisy**	**FPTaylor**	**Daisy + subdiv**	**DaisyOPT Z3 + subdiv**	**DaisyOPT dReal + subdiv**
bspline0	-	-	1.58e-01 (1.08e-18, 1)	3.00e-15 (1.08e-18, 1)	3.00e-15 (1.08e-18, 1)
bspline1	-	3.32e-15	2.80e-13	3.22e-15	3.22e-15
bspline2	-	3.50e-15	9.20e-16	8.92e-16	8.92e-16
bspline3	-	-	1.31e-14 (9.67e-19, 1)	6.66e-16 (9.67e-19, 1)	6.66e-16 (9.67e-19, 1)
sine	-	-	1.07e-15 (2.00e-16, 2)	7.02e-16 (2.02e-16, 2)	7.02e-16 (2.02e-16, 2)
sineOrder3	-	-	2.29e-15 (3.10e-16, 2)	8.94e-16 (3.17e-16, 2)	8.94e-16 (3.17e-16, 2)
sqroot	-	-	7.09e-15 (2.83e-14, 3)	1.92e-15 (3.11e-14, 3)	1.92e-15 (3.11e-14, 3)
doppler	1.48e-11	4.99e-12	8.95e-13	1.26e-15	1.35e-15
himmilbeau	-	-	3.75e-14 (1.00e-12, 12)	2.57e-14 (1.00e-12, 12)	2.84e-15 (1.00e-12, 12)
invPendulum	-	-	4.94e-15 (2.60e-14, 32)	2.82e-15 (2.60e-14, 32)	3.08e-15 (2.60e-14, 32)
jet	-	-	-	-	-
kepler0	4.35e-15	4.57e-15	2.38e-13 (8.08e-14, 49)	2.16e-15 (7.92e-14, 49)	3.88e-15 (8.32e-14, 49)
kepler1	1.33e-14	1.17e-14	-	-	-
kepler2	-	4.21e-14	-	-	-
rigidBody1	-	-	2.29e-14 (2.16e-13, 46)	1.07e-15 (2.16e-13, 46)	1.78e-15 (2.16e-13, 46)
rigidBody2	-	-	2.65e-12 (3.51e-11, 50)	1.67e-15 (3.65e-11, 50)	3.80e-15 (3.65e-11, 50)
traincar_state8	-	-	-	-	-
traincar_state9	-	-	-	-	-
turbine1	6.12e-14	1.18e-14	6.03e-15	1.75e-15	5.21e-15
turbine2	-	-	5.64e-14 (3.65e-14, 25)	2.74e-15 (1.20e-13, 25)	6.97e-14 (3.90e-14, 25)
turbine3	1.52e-13	2.21e-14	2.77e-14	6.50e-15	6.71e-15

REFERENCES

[1] E. Goubault and S. Putot, “Static Analysis of Finite Precision Computations,” in *VMCAI*, 2011.

[2] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, “Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions,” in *FM*, 2015.

[3] E. Darulova and V. Kuncak, “Sound Compilation of Reals,” in *POPL*, 2014.

[4] V. Magron, G. A. Constantinides, and A. F. Donaldson, “Certified Roundoff Error Bounds Using Semidefinite Programming,” *CoRR*, vol. abs/1507.03331, 2015.

[5] MPI-SWS, “Daisy - a framework for accuracy analysis and synthesis of numerical programs,” 2017, https://github.com/malyzajko/daisy.

[6] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, Aug 2008.

[7] R. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[8] L. H. de Figueiredo and J. Stolfi, “Affine Arithmetic: Concepts and Applications,” *Numerical Algorithms*, vol. 37, no. 1-4, 2004.

[9] D. Jovanović and L. de Moura, “Solving non-linear arithmetic,” in *IJCAR*, 2012.

[10] L. De Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *TACAS*, 2008.

[11] E. Darulova and V. Kuncak, “Towards a Compiler for Reals,” *ACM TOPLAS*, vol. 39, no. 2, 2017.

[12] M. S. Baranowski and I. Briggs, “Gelpia - global extrema locator parallelization for interval arithmetic,” https://github.com/soarlab/gelpia.

[13] S. Gao, S. Kong, and E. M. Clarke, “dReal: An SMT Solver for Nonlinear Theories over the Reals,” in *CADE*, 2013.

[14] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, “Synthesis of Fixed-point Programs,” in *EMSOFT*, 2013.

[15] M. Daumas and G. Melquiond, “Certification of bounds on expressions involving rounded operators,” *ACM Trans. Math. Softw.*, vol. 37, no. 1, pp. 2:1–2:20, Jan. 2010.

[16] W. Lee, R. S. 0001, and A. Aiken, “Verifying Bit-Manipulations of Floating-Point.” *PLDI*, 2016.

[17] L. Chen, A. Miné, and P. Cousot, “A Sound Floating-Point Polyhedra Abstract Domain,” in *APLAS*, 2008.

[18] B. Jeannet and A. Miné, “Apron: A Library of Numerical Abstract Domains for Static Analysis,” in *CAV*, 2009.

[19] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, “A Formally-Verified C Static Analyzer,” in *POPL*, 2015.

[20] M. Brain, C. Tinelli, P. Ruemmer, and T. Wahl, “An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic,” Technical Report, http://smt-lib.org/papers/BTRW15.pdf, 2015.

[21] M. Brain, V. D’Silva, A. Griggio, L. Haller, and D. Kroening, “Deciding floating-point logic with abstract conflict driven clause learning,” *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, Dec. 2013.

[22] S. Boldo and G. Melquiond, “Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq,” in *ARITH*, 2011.

[23] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan, “A Parameterized Floating-Point Formalizaton in HOL Light,” *Electronic Notes in Theoretical Computer Science*, vol. 317, pp. 101–107, 2015.

[24] 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 *CGO*, 2010.

[25] A. Ayad and C. Marché, “Multi-prover verification of floating-point programs,” in *IJCAR*, 2010.

[26] 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*, vol. 50, no. 4, pp. 423–456, 2013.

[27] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, “A Unified Coq Framework for Verifying C Programs with Floating-Point Computations,” in *CPP*, 2016.

[28] S. Graillat, V. Lefèvre, and J.-M. Muller, “On the maximum relative error when computing x^n in floating-point arithmetic,” Université Pierre et Marie Curie Paris 6 ; CNRS, Research Report, ensl-00945033, 2014.

[29] F. Benz, A. Hildebrandt, and S. Hack, “A Dynamic Program Analysis to Find Floating-point Accuracy Problems,” in *PLDI*, 2012.

[30] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, “Efficient Search for Inputs Causing High Floating-point Errors,” in *PPoPP*, 2014.

[31] M. O. Lam, J. K. Hollingsworth, and G. Stewart, “Dynamic Floating-point Cancellation Detection,” *Parallel Computing*, vol. 39, no. 3, 2013.

[32] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, “Automatically Improving Accuracy for Floating Point Expressions,” in *PLDI*, 2015.

[33] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, “Precimonious: Tuning Assistant for Floating-point Precision,” in *SC*, 2013.

[34] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, “Automatically Adapting Programs for Mixed-precision Floating-point Computation,” in *ICS*, 2013.

Column-Wise Verification of Multipliers Using Computer Algebra

Daniela Ritirc Armin Biere Manuel Kauers
Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
daniela.ritirc@jku.at armin.biere@jku.at manuel.kauers@jku.at

***Abstract*—Verifying arithmetic circuits, and most prominently multipliers, is an important problem but in practice still requires substantial manual effort. Recent work tries to solve this issue using techniques from computer algebra. The most effective approach uses polynomial reasoning over pseudo boolean polynomials. In this paper we give a rigorous formalization of this approach and present a new column-wise verification technique for the correctness of gate-level multipliers which does not require the reduction of a full word-level specification. We formally prove soundness and completeness of our technique, making use of our precise formalization. Our experiments show that simple multipliers can be verified efficiently by using off-the-shelf computer algebra tools, while more complex and optimized multipliers require more sophisticated techniques. Further, our paper independently confirms the effectiveness of previous related work. We make all benchmarks and tools publicly available.**

I. Introduction

Formal verification of arithmetic circuits is motivated by the necessity to avoid issues like the famous Pentium FDIV bug, which is reported to have cost Intel almost half a billion dollar. There have been many attempts since then to verify such circuits, but even today verifying designs with arithmetic parts is not considered to be fully automated. For instance, a common approach is to black-box multipliers and then verify them separately. This might also require insight into the multiplier design, which has to be communicated to the verification tool. Commercial tools can not fully automatically handle full-sized multipliers [24] or huge multipliers occurring in cryptographic circuits. In this paper we will focus, as a first step, on the simplest but also most important arithmetic circuit verification problem of verifying multipliers.

This lack of automation was a common conclusion in three plenary talks at the joint FMCAD'15 and SAT'15 conferences in Austin in 2015, by Anna Slobodova on formal verification of processors, Aaron Tomb on verifying cryptographic circuits, and, from the academic side, Priyank Kalla on methods for data path verification. In order to stimulate research into this direction, particularly the development of fast SAT solving techniques for arithmetic circuit verification, we collected a large set of such benchmarks, generated and submitted CNF encodings of these problems to the SAT 2016 competition and made them publicly available [4]. The competition results confirmed that miters of even small multipliers pose a real challenge to SAT solvers.

The weak performance of SAT solvers on these benchmarks lead to the conjecture that verifying miters of multipliers and other ring properties after encoding them into CNF needs exponential sized resolution proofs [5], which would imply exponential run-time of CDCL SAT solvers. Surprisingly, however, this conjecture was recently answered negatively [2]. Such ring properties do admit polynomial resolution proofs. However, proof search is non-deterministic. Thus this theoretical result still needs to be transferred into practical SAT solving. The complexity bounds on proof size given in [2] involve polynomials of high degree too.

The first technique which was shown to be able to have prevented the Pentium bug was based on decision diagrams, precisely on binary moment diagrams (BMDs) [10] and variants [11]. While common (gate-level) BDDs are exponential in size for multipliers [6], BMDs remain linear in the number of the input bits of a multiplier (using edge weights). However, the BMD approach is not robust, in the sense that it still requires structural knowledge of the multipliers to determine the order in which BMDs are built, which has tremendous influence on performance. Actually only a row-wise backward substitution approach seems to be feasibly [9], which in addition assumes a simple carry-save-adder (CSA) design.

Recent algebraically inspired techniques [12], [28] based on so-called function-extraction also fail for even slightly optimized multiplier designs. On the positive side, this technique is able to handle very large clean multipliers.

In even more recent work [24] substantial progress was made. The authors use a dedicated polynomial reduction engine and also gave various optimizations (discussed further down), which made their algebraic technique scale to large non-trivial multiplier designs of various architectures [16] (called AOKI benchmarks in the following) even with and without Booth reencoding. It is still unclear however, whether their technique is robust under synthesis or technology mapping. Their arguments for soundness and completeness are rather imprecise. Their tool is not available, nor details about the experiments. Benchmarks have not been published either.

There is a substantial amount of previous work for arithmetic circuit verification. We focus on comparing our approach to the currently most successful techniques for verifying multipliers, which all are using some form of algebraic reasoning [28], [24]. For an up-to-date discussion of related work and a more comprehensive list see the recent article [28].

Supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), Y464-N18, SFB F5004.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

II. Algebra

Following [21], [23], [12], [28], we model the behavior of circuits using multivariate polynomials. There will be a variable for every input and every output of each gate, and the specification of each gate is translated into a polynomial relation among these variables. All these polynomials together form a description of the circuit, and we will prove the correctness of a given circuit by showing that the desired relation between input and output is implied by the polynomials that describe the circuit on the gate level.

The appropriate formalism for such a reasoning is the theory of Gröbner bases [8], [13]. Basic facts are:

- $\mathbb{Q}[X] = \mathbb{Q}[x_1, \dots, x_n]$ denotes the ring of polynomials in variables $x_1, \dots, x_n$ with coefficients in the field $\mathbb{Q}$.
- A *term* (or *power product*) is a polynomial of the form $x_1^{e_1} \cdots x_n^{e_n}$ for certain $e_1, \dots, e_n \in \mathbb{N}$. A *monomial* is a constant multiple of a term.
- Fix an order $\leq$ on the set of terms such that for all terms τ, σ_1, σ_2 we have $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$.
- Every polynomial $p \neq 0$ contains only finitely many terms, the largest of which (w.r.t. the chosen order $\leq$) is called the *leading term* and denoted by $\operatorname{lt}(p)$.
- If $p = c\tau + \cdots$ and $\operatorname{lt}(p) = \tau$, then $\operatorname{lc}(p) = c$ is called the *leading coefficient* and $\operatorname{lm}(p) = c\tau$ is called the *leading monomial* of p.
- A nonempty subset $I \subseteq \mathbb{Q}[X]$ is called an *ideal* if $\forall\, p, q \in I : p + q \in I$ and $\forall\, p \in \mathbb{Q}[X]\ \forall\, q \in I : pq \in I$.
- If $I \subseteq \mathbb{Q}[X]$ is an ideal, then a set $\{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$ is called a *basis* of I if $I = \{q_1 p_1 + \cdots + q_m p_m \mid q_1, \dots, q_m \in \mathbb{Q}[X]\}$, i.e., if I consists of all the linear combinations of the p_i with polynomial coefficients.
- A basis $\{g_1, \dots, g_n\}$ of an ideal $I \subseteq \mathbb{Q}[X]$ is called a *Gröbner basis* (w.r.t. the fixed order $\leq$) if the leading term of every nonzero element of I is a multiple of (at least) one of the leading terms $\operatorname{lt}(g_1), \dots, \operatorname{lt}(g_n)$.
- Every ideal of $\mathbb{Q}[X]$ has a Gröbner basis, and there is an algorithm which, given an arbitrary basis of an ideal, computes a Gröbner basis of it.

The theory of Gröbner bases offers a decision procedure for the ideal membership problem: given a polynomial $q \in \mathbb{Q}[X]$ and a basis $\{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$, it is a priori not obvious how to check whether q belongs to the ideal generated by $p_1, \dots, p_m$. However, if $\{p_1, \dots, p_m\}$ is a Gröbner basis, then the question can be answered using a multivariate version of polynomial division with remainder. It can be shown that when G is a Gröbner basis, then q belongs to the ideal generated by G iff the remainder of division of q by G is zero.

Example 1.

1) Consider $q = x^2 + 4x + 3$, $p = x + 1 \in \mathbb{Q}[x]$. Since $x^2 + 4x + 3 = (x+3)(x+1) + 0$, it follows that q belongs to the ideal I generated by $x + 1$ in $\mathbb{Q}[x]$. On the other hand, taking $\tilde{q} = x^2 + 4x + 5$, division with remainder gives $x^2 + 4x + 5 = (x + 3)(x + 1) + 2$, and thus $\tilde{q} \notin I$.

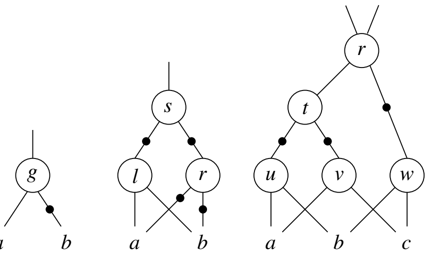


Fig. 1. AIGs [20] used in Example 1 and Sect. IV.

2) For the AIG [20] on the left of Fig. 1, we have the relation $g = a(1 - b)$ for all $a, b, g \in \{0, 1\}$. Furthermore, we always have $g(g-1) = a(a-1) = b(b-1) = 0$ for all $a, b, g \in \{0, 1\}$. To show that we always have $gb = 0$, it is therefore enough to check if the polynomial $gb \in \mathbb{Q}[g, a, b]$ belongs to the ideal $I \subseteq \mathbb{Q}[g, a, b]$ generated by

$$\{-g + a(1 - b), g(g - 1), a(a - 1), b(b - 1)\}.$$

Multivariate polynomial division yields

$$gb = (-b)\,(-g + a(1 - b)) + (-a)\, b(b - 1) + \overset{\overset{\text{remainder}}{\downarrow}}{0},$$

therefore $gb \in I$ and thus $gb = 0$ in the left AIG of Fig. 1.

As illustrated in the second example, we can view an ideal $I \subseteq \mathbb{Q}[X]$ as an equational theory, with a basis $\{p_1, \dots, p_m\}$ as its set of axioms. Indeed, the ideal I generated by $p_1, \dots, p_m$ contains exactly those polynomials q for which the equation "$q = 0$" can be deduced from the assumptions "$p_1 = \cdots = p_m = 0$" through repeated application of the rules $u = 0 \wedge v = 0 \Rightarrow u + v = 0$ and $u = 0 \Rightarrow uw = 0$ (compare the two defining properties for ideals quoted above).

We will need a few more facts about Gröbner bases and multivariate polynomial division.

Lemma 1.

1) Let $q \in \mathbb{Q}[X]$ and $P = \{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$. The remainder r of the division of q by P is a polynomial such that $q - r$ is in the ideal generated by P and r is *reduced* w.r.t. P, which means it does not contain any term that is a multiple of one of the leading terms $\operatorname{lt}(p_1), \dots, \operatorname{lt}(p_m)$.

2) Let $G \subseteq \mathbb{Q}[X] \setminus \{0\}$, and define

$$\operatorname{spol}(p, q) := \operatorname{lcm}(\operatorname{lt}(p), \operatorname{lt}(q)) \left(\frac{p}{\operatorname{lm}(p)} - \frac{q}{\operatorname{lm}(q)} \right)$$

for all $p, q \in \mathbb{Q}[X] \setminus \{0\}$, with lcm the least common multiple. Then G is a Gröbner basis if and only if the remainder of the division of $\operatorname{spol}(p, q)$ by G is zero for all pairs $(p, q) \in G \times G$.

3) If $p, q \in \mathbb{Q}[X] \setminus \{0\}$ are such that their leading terms $\operatorname{lt}(p), \operatorname{lt}(q)$ have no variables in common, then the division of $\operatorname{spol}(p, q)$ with $\{p, q\}$ has remainder zero.

Proof. 1) is Prop. 1 in Chap. 2 §6 of [13]; 2) is Thm. 6 in Chap. 2 §6 of [13]; 3) is Prop. 1 in Chap. 2 §10 of [13]. □

III. Ideals associated to circuits

We consider circuits with $2n$ inputs $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$, $2n$ outputs $s_0, \ldots, s_{2n-1}$, and a number of logical gates. The output of some gate may be input to some other gate, but cycles are not allowed. In addition to the variables for input and output, we also associate one variable to each internal edge of the circuit, say $g_1, \ldots, g_k$. By R we denote the ring $\mathbb{Q}[a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, g_1, \ldots, g_k, s_0, \ldots, s_{2n-1}]$.

The semantics of the circuit gates imply polynomial relations among these variables, such as the following ones:

$$\begin{array}{lll} u = \neg v & \text{implies} & 0 = -u + 1 - v \\ u = v \wedge w & \text{implies} & 0 = -u + vw \\ u = v \vee w & \text{implies} & 0 = -u + v + w - vw \\ u = v \oplus w & \text{implies} & 0 = -u + v + w - 2vw. \end{array} \quad (1)$$

We also have the relations $u(u-1) = 0$ for each variable u, because the circuit operates with boolean values.

Since logical gates are functional, the values of all the variables $g_1, \ldots, g_k, s_0, \ldots, s_{2n-1}$ in a circuit are uniquely determined as soon as $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \in \{0,1\}$ are fixed. This motivates the following definition.

Definition 1. Let C be a circuit.

1) A polynomial $p \in R$ is called a *polynomial circuit constraint (PCC)* for C if for every choice of

$$(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}) \in \{0,1\}^{2n}$$

and resulting values $g_1, \ldots, g_k, s_0, \ldots, s_{2n-1}$ implied by the gates of C the substitution of these values into p gives zero.

2) The set of all PCCs for C is denoted by $I(C)$.

It is easy to see that $I(C)$ is in fact an ideal of R. By definition, this ideal contains all the relations that hold among the values at the different points in the circuit. In particular, it "knows" everything about how input and output are related. Therefore, the circuit fulfills a certain specification if and only if the polynomial relation corresponding to the specification is contained in $I(C)$. This motivates the next definition.

Definition 2. A circuit C is called a *multiplier* if

$$\sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \in I(C).$$

Checking whether a given circuit C is a multiplier thus reduces to an ideal membership test. Definition 1 does not provide us with a basis of $I(C)$, so Gröbner basis technology is not directly applicable. However, we can deduce at least some elements of $I(C)$ from the semantics of circuit gates.

Definition 3. Let C be a circuit. Let $G \subseteq R$ be the set which contains for each gate of C the corresponding polynomial of (1) (with u, v, w replaced by the variables of the edges attached to the gate), as well as the polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ for $0 \leq i < n$, called *input field polynomials*. Then the ideal generated by G in R is denoted by $J(C)$.

As a basis of $J(C)$ is explicitly known, we can decide membership using Gröbner bases. Consider a verifier for circuits which checks for a given C and a given specification polynomial p whether p belongs to $J(C)$. Because of $J(C) \subseteq I(C)$, such a verifier is certainly sound. In order to prove that it is also complete, we need to show $J(C) \supseteq I(C)$. For doing so, we recall a crucial observation which for instance already appears in [26], [21].

Theorem 1. Let C be a circuit, and let G be as in Def. 3. Let $\leq$ be a lexicographic term order for a variable order such that the variable attached to the output edge of a gate is always greater than the variables attached to the input edges of that gate. Then G is a Gröbner basis with respect to $\leq$.

Proof. By the constraint on the term order and the form of the equations (1), the leading term of each gate polynomial is simply the output variable of the corresponding gate. Further, the leading terms of the polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ are a_i^2 and b_i^2. Therefore, by part 3 of Lemma 1, division of $\operatorname{spol}(p, q)$ by $\{p, q\}$ gives the remainder zero for any choice $p, q \in G$. Then, since $\{p, q\} \subseteq G$ for all $p, q \in G$, also division of $\operatorname{spol}(p, q)$ by G gives the remainder zero for all $p, q \in G$, and then, by part 2 of Lemma 1, the claim follows. $\square$

Theorem 2. For all acyclic circuits C, we have $J(C) = I(C)$.

Proof. "$\subseteq$" (soundness) Clear by definition of $J(C)$.

"$\supseteq$" (completeness) Let $p \in I(C)$. We have to show that $p \in J(C)$. Since C is acyclic, there is a way to order the variables such as to meet the requirement of Thm. 1. Let r be the remainder of the division of p by G, where G is the Gröbner basis of Thm. 1. Then $p - r \in J(C)$ by part 1 of Lemma 1, so $r \in J(C) \iff p \in J(C)$. Furthermore, $p \in I(C)$ and $p - r \in J(C) \subseteq I(C)$ implies $r \in I(C)$. It is therefore sufficient to show that $r \in J(C)$.

By the choice of the term order and the observations made in the previous proof about the leading terms in G, part 1 of Lemma 1 also implies that r only contains input variables $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$, and none of them appears with degree greater than one. At the same time, since $r \in I(C)$, all the evaluations of r for all choices $a_i, b_j \in \{0,1\}$ are zero.

We show that $r = 0$, and thus $r \in J(C)$. Suppose $r \neq 0$. Let m be a monomial of r with a minimal number of variables, which includes the case where m is constant. Since exponents are at most one, the set of variables of monomials in r differ by at least one variable. Now choose a_i (b_j) to evaluate to 1 iff $a_i \in m$ ($b_j \in m$). By this choice all monomials of r except m vanish (evaluate to zero). Thus r evaluates to the (non-zero) coefficient of m, in contradiction to $r \in I(C)$. $\square$

Let us conclude the theoretical part of this paper with the following simple but important observations.

First, $I(C)$ is by definition a so-called vanishing ideal. Therefore, the theorem implies that $J(C)$ is a radical ideal. This explains why ideal membership is sufficient for our purpose, and there is no need to use the stronger radical membership test (cf. Chap. 4 §2 of [13]).

Second, note that $I(C) = J(C)$ contains the set F of all field polynomials $x(x-1)$ for all variables x, not only for inputs, thus we may add them to G.

Third, in the standard Gröbner basis for gate-level circuits defined above in Def. 3 using Eqn. (1) all polynomials have a leading coefficient of ± 1 and thus during reduction never introduce any coefficient outside of $\mathbb{Z}$ (with non-trivial denominator). So all coefficient computations actually remain in $\mathbb{Z}$. This formally proves that dedicated implementations, e.g., those from [28], [24], used for determining ideal membership to verify properties of gate-level circuits, can rely on computation in $\mathbb{Z}$ only without loosing soundness nor completeness (assuming the same term order as in Thm. 1 is used).

Fourth, from a technical point of view, we do not need to use $\mathbb{Z}$ as coefficient ring if we employ computer algebra systems, but can simply use any field containing $\mathbb{Z}$, e.g., $\mathbb{Q}$. This actually speeds up the computation, since computer algebra systems are optimized for this case. In our experiments, using rational coefficients made a huge difference for Singular [14] (but did not have any effect in Mathematica [27]).

Fifth, given circuit C, checking that there exists an assignment to the inputs which yields a certain value, at an output is of course the same as (circuit) SAT, and thus NP complete:

Corollary 1. Checking ideal membership over $\mathbb{Q}[X]$ even in terms of a given Gröbner basis is co-NP-hard.

Similar results but for $\mathbb{Z}_2$ and $\mathbb{Z}$ instead of $\mathbb{Q}$ and without assuming a Gröbner basis can be found in [1], [18].

Finally, the last part in the proof of Thm. 2 allows us to determine a concrete input evaluation in case a polynomial g fails the membership test, e.g., an evaluation for which g does not vanish. In our application of multiplier verification these evaluations provide counter-examples, in case a circuit is determined not to be a multiplier (Alg. 1 returns *false*).

We claim that this section is a first simple and precise mathematical characterization of recent successful algebraic approaches [24], [28] to the verification of gate-level integer multipliers (without overflow), where we formally prove not only soundness but also completeness. Soundness corresponds to $I \subseteq J$ and completeness to $I \supseteq J$ in Thm. 2.

In previous work soundness and completeness was formally proven too but only for other polynomial rings, i.e., over $\mathbb{F}_{2^q}$ to model circuits implementing Galois-field multipliers [21], [23], or for polynomial rings over $\mathbb{Z}_{2^q}$ to model arithmetic circuit verification with overflow semantics [26].

In [28] soundness and completeness is discussed too, but instead of giving proofs only refers to [21], [23] which as discussed above uses coefficients in $\mathbb{F}_{2^q}$ and not $\mathbb{Z}$, the coefficient ring the approach [28] is actually working with.

IV. Optimizations

Following the argument of Cor. 1 in the previous section, simply reducing the specification in the constructed Gröbner basis may lead and in general has to lead (unless P = NP) to an exponential number of monomials in intermediate results. Thus in practice to use polynomial reduction to verify specific circuits tailored heuristics become very important.

To reduce the number of monomials in [24] a logic reduction rewriting scheme consisting of *XOR-Rewriting* and *Common-Rewriting* is proposed. It is further combined with eliminating monomials which fulfill certain *Vanishing-Constraints*. In the following we show how these techniques can be applied to computer algebra systems.

The technique of *XOR-Rewriting* [24] ensures that in the Gröbner basis all variables which do not correspond to an output nor input of an XOR-gate, nor primary input, nor output of the circuit, are eliminated from the Gröbner basis up-front.

We adopt this rewriting for AIGs by matching XOR patterns in the AIG which represents an XOR or XNOR, e.g., we find nodes of the form $s = \overline{(a \wedge b)} \wedge \overline{(\bar{a} \wedge \bar{b})}$. We then define the polynomial of the parent in terms of the grandchildren instead of the immediate children. For instance, in order to apply XOR-Rewriting in the middle AIG in Fig. 1 we only use the polynomial $-s + a + b - 2ab$ as definition for the XOR output instead of all the polynomials $-l + ab$, as well as $-r + (1-a)(1-b)$, and $-s + (1-l)(1-r)$. This removes defining polynomials for all children of XOR gates.

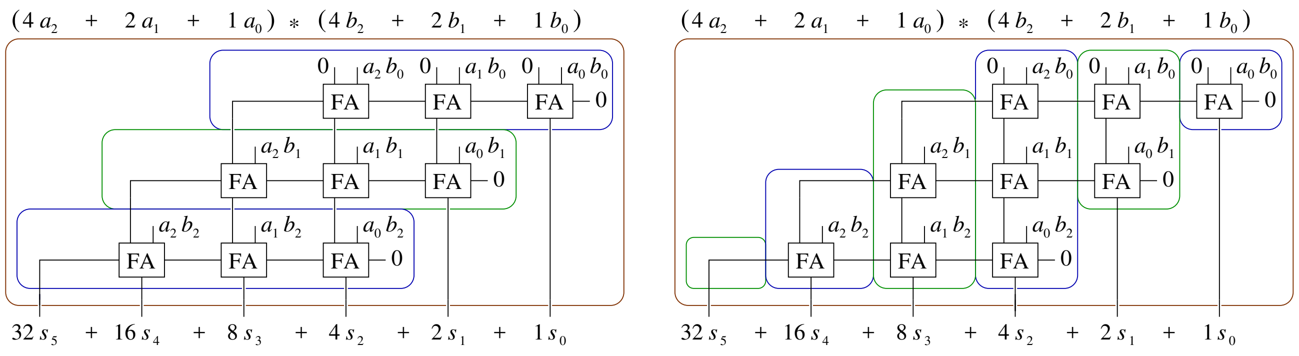
The technique of *Common-Rewriting* [24] eliminates all nodes which have exactly one parent. In the right AIG of Fig. 1 Common-Rewriting eliminates gates t, u, v, and w, assuming r occurs twice, but t, u, v and w only once. Thus r is directly expressed in terms of a, b, c. This technique is actually similar to what bounded variable elimination in SAT would do [15] after encoding a circuit to CNF by say Tseitin encoding. It would also eliminate all variables in the CNF representing gates in the circuit with only one parent [17].

In [24] an important optimization was a specific "vanishing rule", called *XOR-AND Vanishing Rule*. This rule can be derived from the middle AIG in Fig. 1, a half adder, where l represents the carry (AND) and s represents the sum (XOR) of the two inputs. In a half adder both the carry bit l and the sum bit s can never be 1 at the same time. Thus $sl = 0$, and [24] suggests to remove monomials containing s and l immediately. We simulate the effect of this rule by searching for (negated) children or grand-children of certain AND-gates and adding appropriate polynomial constraints to our reduction basis.

V. Order

According to Thm. 1 the choice of the reverse topological term order does not influence the correctness of the procedure. However in [28] it is shown empirically that the number of monomials during the reduction process varies substantially for different reverse topological orders.

Given the planar two dimensional "shape" of multipliers, two approaches of ordering are quite natural, namely a row-wise approach and a column-wise approach. Basically the idea is to partition the gates into *slices*, which are totally ordered, i.e., row-wise or column-wise, and then order the gates within a slice (row or column) topologically. The combined total order has to be topological, which then gives a valid term order and thus a Gröbner basis according to Thm. 1.



$(4\,a_2 + 2\,a_1 + 1\,a_0) * (4\,b_2 + 2\,b_1 + 1\,b_0)$
FA
$32\,s_5 + 16\,s_4 + 8\,s_3 + 4\,s_2 + 2\,s_1 + 1\,s_0$

Fig. 2. Classical row-wise slicing (left) versus our column-wise slicing approach (right) for clean 3-bit input (6-bit output) CSA multiplier.

The idea of the row-wise approach is to order the gates according to their backward level. The intuition of row-wise slicing is outlined in the left side of Fig. 2. It shows how full adders are partitioned in a "clean" (CSA) multiplier. Informally, we call a multiplier without gate synthesis, nor mapping and where the XOR-gates and the half/full adders can easily be identified, as *clean*. If a multiplier is not clean, it is called *dirty*. Thus the AOKI benchmarks [16], [24] already discussed in the introduction are considered to be dirty.

Previous papers [28], [10] use a row-wise approach. In [28] gates are ordered by the logic level seen from the circuit inputs. In [10] the order is only given for clean CSA multipliers, such that a word-level spec for a CSA step can be given. It is unclear how to apply this order to dirty multipliers, like the AOKI benchmarks. Unfortunately, the description of the order in [24] stays on a very high level. The tool is not available.

In the column-wise approach, cf. right side of Fig. 2, the multiplier is partitioned vertically, where each slice contains exactly one output bit. Our proposal is to use a column-wise order which gives a more robust incremental checking method.

VI. Column-Wise Checking

The goal of using a column-wise term order is to divide the problem into smaller more manageable sub-problems, which can be verified incrementally.

Definition 4. Let C be a circuit (as in Sect. III).

1) A sequence of $2n+1$ polynomials $C_0, \ldots, C_{2n}$ over the variables of C is called a *carry sequence* of *carry polynomials*.

2) For column i with $0 \leq i < 2n$ let $P_i = \sum_{k+l=i} a_k b_l$ be the *partial product sum* (of column i).

3) For $0 \leq i < 2n$, carry polynomial C_i and output s_i let

$$-C_i + 2C_{i+1} + s_i - P_i$$

denote the *carry recurrence relation* R_i for column i.

4) Then R_i *holds* on C if it vanishes in $I(C)$, i.e., $R_i \in I(C)$.

With these definitions we obtain an abstract theorem which can be used to verify multipliers independent how the carry sequence is actually constructed.

Theorem 3. Let C be a circuit where all carry recurrence relations hold as defined in Def. 4. Then C is a multiplier in the sense of Def. 2, if and only if $C_0 - 2^{2n}C_{2n} \in I(C)$.

Proof. By the condition of Def. 4, we have (modulo $I(C)$)

$$\begin{aligned}\sum_{i=0}^{2n-1} 2^i s_i &= \sum_{i=0}^{2n-1} 2^i(P_i + C_i - 2C_{i+1}) \\ &= \sum_{i=0}^{2n-1} 2^i P_i + \underbrace{\sum_{i=0}^{2n-1} (2^i C_i - 2^{i+1} C_{i+1})}_{C_0 - 2^{2n} C_{2n}}.\end{aligned}$$

It remains to show $\sum_{i=0}^{2n-1} 2^i P_i = (\sum_{i=0}^{n-1} 2^i a_i)(\sum_{i=0}^{n-1} 2^i b_i)$, which is a rather straight forward calculation. □

To obtain our column-wise checking algorithm we define slices incrementally. For each output bit s_i we determine its input cone, namely the gates which s_i depends on (cf. Fig. 3):

$$I_i := \{\text{gate } g \mid g \text{ is in input cone of output } s_i\}$$

We define slices S_i as the difference of consecutive cones I_i:

$$S_0 := I_0 \qquad S_{i+1} := I_{i+1} \setminus \bigcup_{j=0}^{i} S_j$$

Definition 5 (Sliced Gröbner Bases)**.** Let G_i be the set of polynomial representations of the gates in slice S_i, cf. Eqn. 1.

Algorithm 1: Multiplier Checking Algorithm

```
Input  : Circuit $C$ with sliced Gröbner bases $G_i$
Output: Determine whether $C$ is a multiplier
1 $C_{2n} \leftarrow 0$;
2 for $i \leftarrow 2n-1$ to $0$ do
3     $C_i \leftarrow$ Remainder $(2C_{i+1} + s_i - P_i,\ \ G_i \cup F\ )$
4 end
5 return $C_0 = 0$
```

In Alg. 1 we start at the last output bit s_i with $i = 2n-1$. Then C_i is computed recursively by taking the remainder of $2C_{i+1} + s_i - P_i$ modulo the sliced Gröbner basis G_i and (all) field polynomials F in order to make sure that the carry

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

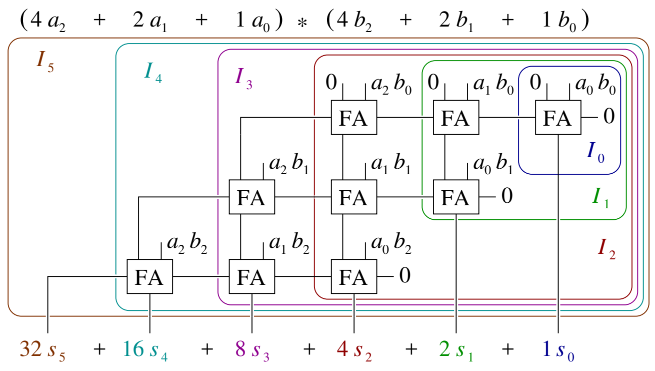

(4 a2 + 2 a1 + 1 a0) * (4 b2 + 2 b1 + 1 b0)
I5
I4
I3
I2
I1
I0
FA
0
a2 b0
a1 b0
a0 b0
a2 b1
a1 b1
a0 b1
a2 b2
a1 b2
a0 b2
32 s5 + 16 s4 + 8 s3 + 4 s2 + 2 s1 + 1 s0

Fig. 3. Input cones of outputs to determine column slices.

recurrence relation R_i holds. Thus C_i is uniquely defined given the sum of the partial products P_i of column i, the output bit s_i and the previous carry polynomial C_{i+1}. It remains to fix the boundary carry polynomial C_{2n}. In our algorithm we actually always simply use $C_{2n} = 0$.

Theorem 4. Algorithm 1 returns *true* iff C is a multiplier.

Proof. By definition $R_i := -C_i + 2C_{i+1} + s_i - P_i$ vanishes on the ideal generated by $G_i \cup F$ which is a subset of the ideal generated by $G \cup F$ since $G_i \subseteq G$. Thus $R_i \in J(C) = I(C)$.

We can show inductively that C_i is reduced w.r.t. H_i with $H_i := \bigcup_{j \geq i}(G_j \cup F)$. This induction requires that s_i and P_i are reduced w.r.t. to H_{i+1} which holds due to the construction of the sliced Gröbner bases. With $H_0 = G \cup F$ we then get C_0 is reduced w.r.t. $G \cup F$ thus $C_0 = C_0 - 2^{2n} C_{2n} \in I(C) = J(C)$ iff $C_0 = 0$, which concludes the proof using Thm. 3. □

For incorrect multipliers Alg. 1 returns *false*, i.e., $C_0 \neq 0$. As described after Cor. 1 this easily yields a concrete counter-example. In this case it might further be possible to abort the algorithm earlier if partial products $a_k b_l$ of higher slices $k + l > i$ not occurring in S_j with $i < j$ remain in C_i.

VII. Engineering

Our tool AIGMULTOPOLY takes an AIG describing a circuit as input and produces output which can be passed to the computer algebra systems Mathematica [27] and Singular [14].

Algorithm 2: Outline of AIGMULTOPOLY

```
Input  : Circuit in AIG format
Output: File $f$ for computer algebra system
1 for $i \leftarrow 0$ to $2n - 1$ do
2     Define-Cones-of-Influence ();
3     Merge ($S_i$);
4     Promote ($S_i$);
5     Levelize ($S_i$);
6     Search-for-Common-Rewriting ($S_i$);
7     Identify-Vanishing-Constraints ($S_i$);
8 end
9 $f \leftarrow$ Print to file;
```

For dirty multipliers slicing based on input cones, (Sect. VI), is not precise enough. It regularly happens, that gates are allocated to later slices, if they are not used to compute the output value of the slice. This frequently happens for carry outputs of full/half-adders (or combined carry outputs) and results in larger carry polynomials C_i than necessary.

To avoid this performance issue we eagerly move gates between slices, in a kind of peephole optimization, which makes sure that the overall number of carries decreases:

Definition 6. We define those gates in S_j used as children of gates in slice S_i with $i > j$ as *carries* of S_j.

The following technique reduces the support of carry polynomials increasing the chances for cancellation of monomials.

Merge: Whenever we find an AND-gate g (not matched to be an XOR- gate) in slice S_i with children l, r in smaller slices S_j and S_k, we move g back to S_l with $l = \max(j, k)$. The procedure is depicted on the left side of Fig. 4. Thus after merging g, the gates l, r are less likely to be carry variables any more. We apply merging repeatedly until completion and S_l and S_i are updated after each application.

In some multipliers it happens that a gate g in the carry depends on two other gates in the carry. We decrease the number of carries by promoting g to the next bigger slice:

Promote: We search for gates g in slice S_{i-1} with again exactly one parent, which in addition is required to be part of some larger slice S_j where $j \geq i$. Furthermore the children of g also have to be in slice S_{i-1} and have at least one parent in some later slice S_j with $j \geq i$. We decrease the number of carries by promoting g to slice S_i, cf. right side of Fig. 4.

A gate g which is merged can not be promoted back to its original slice, because the requirements for the children of g differ. This prevents cyclic rule applications. After merging and promoting, the association of gates to slices is fixed. We order the gates in a slice by levelization from inputs.

In order to simulate Common-Rewriting, we factor out from S_i the set U_i of "unique gates", i.e., all gates g of S_i not used in another slice with exactly one parent in slice S_i. Polynomials of gates which remain in S_i and depend on gates in U_i are reduced first by polynomials of gates in U_i and field polynomials F before computing the remainder in Alg. 1.

As last step we search for Vanishing Constraints in S_i, namely gate products which always evaluate to zero, e.g., Example 1. We store such constraints in a corresponding set V_i and during remainder computation reduce against elements of V_i too. Because of Thm. 2, we can add these polynomials to the ideal without violating the Gröbner basis property.

Finally, in AIGMULTOPOLY the optimization of "XOR-Rewriting" is handled implicitly during printing by producing polynomials for XOR-gates instead of AND-gates.

All optimizations either maintain the crictical criteria of keeping the reduction order topologically sorted, add vanishing constraints of the circuit ideal, or are standard techniques used in computer algebra, e.g., autoreduction, and thus do not affect correctness of our claims.

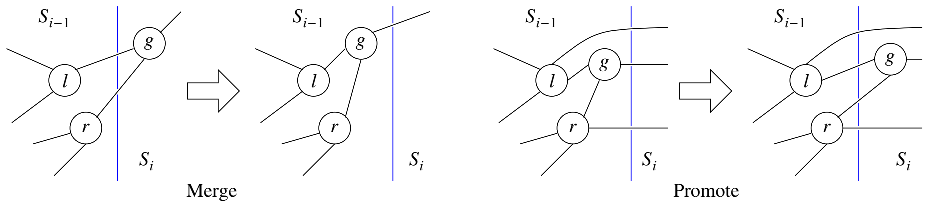


Fig. 4. Locally optimizing number of carries (gates used in later slices) by moving gates backward (Merge) and forward (Promote). Inputs are on the left, outputs on the right of AND-gates, which is the more common order used in visualizing circuits, but reversed compared to the column order in Fig. 2.

VIII. Experiments

As in previous work we focus on (integer) multipliers with two n-bit vectors as inputs and $2n$ output bits. In [28], [12] the authors used clean CSA multipliers, handcrafted from [19], for verifying their results. In [24] several architectures from the AOKI benchmarks are used in their experiments. In our experiments we use the multiplier types "btor", "sp-ar-rc" and "abc". The "btor" benchmarks are generated from Boolector [22] and can be considered as clean multipliers. The "sp-ar-rc" multipliers are part of the AOKI benchmarks [16] and can be considered as dirty multipliers. The "abc" benchmarks are generated with ABC [3]. The different versions of synthesis and technology mapping should be the same as in [28], [12].

We used a standard Ubuntu 16.04 Desktop machine with Intel i7-2600 3.40GHz CPU and 16 GB of main memory. The (wall-clock) time limit was set to 1200 seconds and main memory was limited to 14GB. An extended set of experimental data, as well as source code, benchmarks, and scripts are available at http://fmv.jku.at/cwmulverca. Beside those benchmarks used in our experiments we also include the AIGs we derived for other multipliers used in [28], [24]. More information on the structure of the multipliers used in our experiments can be found in [16], [28], [12] and the README files which come with the experimental data.

In all our experiments the times are listed in seconds (wall-clock time). We measure the time from starting our tool until Mathematica resp. Singular are finished or we reach the time limit (TO), the memory limit (MO), or reach an error state (EE). An error state occurs in Singular when more than 32767 ring variables are allocated. Our results include the time which our tool AigMulToPoly needs to generate the files for the computer algebra system. This time is in the worst case around 3 seconds for 128 bit multipliers. The results also include the time to launch Mathematica resp. Singular.

In Table I we compare our incremental column-wise reduction, outlined in Alg. 1 against the non-incremental approach, where the word-level specification of Def. 2 is reduced against the whole circuit. We apply the non-incremental reduction for column-wise and row-wise ordering. All optimizations (XOR-Rewriting, Common-Rewriting, Vanishing Constraints, Merge, Promote) are enabled. The results in Table I show that in Mathematica and Singular our approach is faster and needs less memory than any non-incremental approach. In the non-incremental experiments, the results between column-wise and row-wise do not really differ. Generally Singular is faster than Mathematica, but it also needs more memory than Mathematica. For multiplier "btor-128" we get an error state.

TABLE I
Incremental (+inc) vs. column- and row-wise non-incr. (-inc).

mult	n	Mathematica			Singular		
		+inc	-inc		+inc	-inc	
			col	row		col	row
btor	16	4	12	12	1	2	2
btor	32	35	531	491	16	53	58
btor	64	409	MO	MO	MO	MO	MO
btor	128	TO	TO	TO	EE	EE	EE
sp-ar-rc	16	7	TO	TO	1	TO	TO
sp-ar-rc	32	67	TO	TO	39	TO	TO
sp-ar-rc	64	841	MO	MO	MO	MO	MO

TABLE II
Effect of turning off optimizations.

mult	n	Mathematica				Singular			
		+inc	-xor	-com	-cs	+inc	-xor	-com	-cs
btor	16	4	TO	1	TO	1	2	1	1
btor	32	35	TO	7	TO	16	64	6	19
btor	64	409	TO	65	TO	MO	MO	MO	MO
btor	128	TO	TO	823	TO	EE	EE	EE	EE
sp-ar-rc	16	7	30	TO	7	1	7	TO	2
sp-ar-rc	32	67	373	TO	64	39	266	TO	34
sp-ar-rc	64	841	TO	TO	805	MO	EE	MO	MO

In the experiments shown in Table II we investigate the effects of turning off optimizations in our column-wise approach and compare these variants to the "+inc" columns of Table I. The results differ for clean and dirty multipliers. For the clean "btor" multipliers turning off Common-Rewriting surprisingly improves the reduction. In this case there are only few gates outside of XORs with only one parent, and splitting remainder computation just increases run-time and space usage. In dirty multipliers, structures like carry trees containing gates with only one parent occur much more frequently. If we turn off Common-Rewriting remainder computation slows down a lot in this case. Turning off XOR-Rewriting influences both clean and dirty multipliers and slows down the reduction (especially in Mathematica), whereas turning off Vanishing Constraints has only a bad effect for clean multipliers in Mathematica, in Singular the results are nearly the same. In summary, the optimizations described in [24] have both positive and negative effects in our experiments, depending on the type of multiplier considered and the computer algebra system used.

TABLE III
DIFFERENCE OF TURNING OFF MERGE AND PROMOTE

mult	n	Mathematica			Singular		
		+inc	-merge	-prom	+inc	-merge	-prom
sp-ar-rc	16	7	8	TO	1	1	TO
sp-ar-rc	32	67	72	TO	39	42	MO
sp-ar-rc	64	841	912	TO	MO	MO	MO

TABLE IV
DIRTY SYNTHESIZED AND MAPPED MULTIPLIERS

mult	n	Mathematica	Singular
abc	8	2	1
abc	16	4	1
abc-resyn3-no-comp	8	351	3
abc-resyn3-no-comp	16	TO	TO
abc-resyn3-comp	8	TO	TO

The experiments shown in Table III compare the effects of turning off our Merge and Promote optimizations on dirty multipliers. In clean multipliers (such as "btor") no gates are merged nor promoted. The running times of Merge enabled or disabled can be considered to be the same. The difference is the size of the carry polynomials, e.g., in sp-ar-rc-8 the carry polynomials have up to 38 monomials with Merge disabled. In our default setting with Merge enabled the biggest carry polynomial contains only 8 monomials and is linear.

In Table IV we also consider synthesized and mapped versions of multipliers. Synthesizing a circuit makes it very hard to verify. When complex mapping is applied it gets even harder and the 8-bit version cannot be verified any more, neither in Mathematica nor Singular confirming [12], [28].

IX. CONCLUSION

We give a simple and precise mathematical formalization of recent successful algebraic approaches to the verification of multiplier circuits, including rigorous proofs of soundness and completeness. We further show how to effectively make use of computer algebra systems. Our main technical contribution is a new incremental column-based verification approach to multipliers, which is an order of magnitude faster than previous row-based approaches relying on reducing a global spec. We further confirm the effectiveness of the algebraic approach and make all data, benchmarks and tools publicly available.

As future work, we want to analyze complexity of previous and our new column-wise approach similar to [7] and [2] and extend our methods to floating-points (following [25]) and other word-level operators. We also want to consider overflow-semantics and negative numbers. An experimental comparison with BMD based techniques should also be performed.

We would like to thank Paul Beame for sharing drafts of [2], Mathias Soeken helping to synthesize AOKI multipliers [16] used in their DATE'16 paper [24], Naofumi Homma sending 128-bit versions of these benchmarks, Maciej Ciesielski explaining the experimental set-up in [12], [28], and finally Deepak Kapur for pointing us to related work [1], [18].

REFERENCES

[1] S. Agnarsson, A. Kandri-Rody, D. Kapur, P. Narendran, and B. Saunders. Complexity of testing whether a polynomial ideal is nontrivial. In *Third MACSYMA User's Conference*, pages 452–458, 1984.

[2] P. Beame and V. Liew. Towards verifying nonlinear integer arithmetic. In *CAV*, volume 10427 of *LNCS*, pages 238–258. Springer, 2017.

[3] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/. Bitbucket Version 1.01, last change Feb. 27, 2017.

[4] A. Biere. Collection of combinational arithmetic miters submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. Univ. Helsinki, 2016.

[5] A. Biere. Weaknesses of CDCL solvers, August 2016. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.

[6] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[7] R. Bryant and Y. Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.

[8] B. Buchberger and M. Kauers. Gröbner basis. *Scholarpedia*, 5(10):7763, 2010. http://www.scholarpedia.org/article/Groebner_basis.

[9] J. Chen and Y. Chen. Equivalence checking of integer multipliers. In S. Goto, editor, *ASP-DAC 2001*, pages 169–174, 2001.

[10] Y. Chen and R. Bryant. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541, 1995.

[11] Y. Chen, E. Clarke, P. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *FMCAD*, volume 1166 of *LNCS*, pages 19–33. Springer, 1996.

[12] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi. Verification of gate-level arithmetic circuits by function extraction. In *DAC*, pages 52:1–52:6. ACM, 2015.

[13] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.

[14] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-1-0. http://www.singular.uni-kl.de, 2016.

[15] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[16] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.

[17] M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.

[18] A. Kandri-Rody, D. Kapur, and P. Narendran. An ideal-theoretic approach to work problems and unification problems over finitely presented commutative algebras. In *RTA*, volume 202 of *LNCS*, pages 345–364. Springer, 1985.

[19] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2001.

[20] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.

[21] J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.

[22] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *JSAT*, 9:53–58, 2014 (published 2015).

[23] T. Pruss, P. Kalla, and F. Enescu. Equivalence verification of large Galois field arithmetic circuits using word-level abstraction via Gröbner bases. In *DAC*, pages 152:1–152:6. ACM, 2014.

[24] 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 *DATE*, pages 1048–1053. IEEE, 2016.

[25] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176. IEEE, 2016.

[26] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel. An algebraic approach for proving data correctness in arithmetic data paths. In *CAV*, volume 5123 of *LNCS*, pages 473–486. Springer, 2008.

[27] Wolfram Research, Inc. Mathematica, 2016. Version 10.4.

[28] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *IEEE TCAD*, 35(12):2131–2142, 2016.

Efficient Generation of All Minimal Inductive Validity Cores

Elaheh Ghassabani, Michael Whalen
Department of Computer Science & Engineering
University of Minnesota
MN, USA
ghass013, mwwhalen@umn.edu

Andrew Gacek
Rockwell Collins
Advanced Technology Center
IA, USA
andrew.gacek@rockwellcollins.com

***Abstract*—Symbolic model checkers can construct proofs of safety properties over complex models, but when a proof succeeds, the results do not generally provide much insight to the user. Recently, proof cores (alternately, for inductive model checkers, Inductive Validity Cores (IVCs)) were introduced to trace a property to a minimal set of model elements necessary for proof. Minimal IVCs facilitate several engineering tasks, including performing traceability and analyzing requirements completeness, that usually rely on the minimality of IVCs. However, existing algorithms for generating an IVC are either expensive or only able to find an approximately minimal IVC.**

Besides minimality, computing *all* minimal IVCs of a given property is an interesting problem that provides several useful analyses, including regression analysis for testing/proof, determination of the minimum (as opposed to minimal) number of model elements necessary for proof, the diversity examination of model elements leading to proof, and analyzing fault tolerance.

This paper proposes an efficient method for finding *all minimal* IVCs of a given property proving its correctness and completeness. We benchmark our algorithm against existing IVC-generating algorithms and show, in many cases, the cost of finding all minimal IVCs by our technique is similar to finding a single minimal IVC using existing algorithms.

***Keywords*-Inductive Validity Cores; UNSAT-core generation; SMT-based model checking; Inductive proofs;**

I. Introduction

Most modern sequential model checking techniques for safety properties, including IC3/PDR [1] and k-induction [2], use a form of induction to establish proof. These techniques are very powerful, and can often reason successfully over very large or even infinite state spaces. The proofs provided by these tools can provide rigorous evidence that a software or hardware system works as intended.

On the other hand, there are many situations in which properties can be proved, but systems still will not perform as intended. Issues such as vacuity [3], incorrect environmental assumptions [4], and errors either in English language requirements or formalization [5] can all lead to failures of "proved" systems. Thus, even if proofs are established, one must approach verification with skepticism.

Recently, *proof cores* [6] have been proposed as a mechanism to determine which elements of a model are used when constructing a proof. This idea is formalized by Ghassabani et al. for inductive model checkers in [7] as *Inductive Validity Cores* (IVCs). IVCs offer proof explanation as to why a property is satisfied by a model in a formal and human-understandable way. The idea lifts UNSAT cores [8] to the level of sequential model checking algorithms using induction. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof such that if any constraint is removed, the property is no longer valid. IVCs and MIVCs can be used for several purposes, including performing traceability between specification and design elements [9], assessing model coverage [10], and explaining unsatisfiable test obligations when using model checkers for test case generation. Ghassabani et al. [7] presented two algorithms: one that computes an approximately minimal IVC using UNSAT cores (`IVC_UC`) that is computationally inexpensive, and a more accurate algorithm that usually produces a minimal IVC using a brute-force post-processing step (`IVC_UCBF`) that is considerably more expensive to compute.[1]

The IVC and proof core ideas share many similarities with approaches for computing minimal invariant sets for inductive proofs (such as is performed for inductive proof certificates [11], [12]), and in fact the `IVC_UC` algorithm performs a minimal lemma set computation. However, there is a substantive difference: to find a minimal set of constraints, it is usually necessary to find new proofs involving *new lemmas* not used in the original proof, which accounts for the expense of the `IVC_UCBF` algorithm.

It is often the case that there are multiple MIVCs for a given property. In this case, computing a single IVC provides, at best, an incomplete picture of the traceability information associated with the proof. Depending on the model and property to be analyzed, there is often substantial diversity between the IVCs used for proof, and there can also be a substantive difference in the size of a *minimal* IVC and a *minimum* IVC, which is the (not necessarily unique) smallest MIVC. If *all* MIVCs can be found, then several additional analyses can be performed:

- Coverage Analysis: MIVCs can be used to define cov-

[1]In [7] it is shown that minimization is as hard as model checking, so for model checking problems that generally undecidable, the minimization process is also generally undecidable, so the `IVC_UCBF` algorithm may time out and return an approximate result.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

erage metrics by examining the percentage of model elements required for a proof. However, since MIVCs are not unique, there are multiple, equally legitimate coverage scores possible. Having *all* MIVCs allows one to define additional metrics: coverage of MAY elements, coverage of MUST elements, as well as policies for the existing MIVC metric: e.g., choose the smallest MIVC [10].

- Optimizing Logic Synthesis: synthesis tools can benefit from MIVCs in the process of transforming an abstract behavior into a design implementation. A practical way of calculating all MIVCs allows to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.
- Impact Analysis: Given all MIVCs, it is possible to determine which requirements may be falsified by changes to the model. This analysis allows for selective regression verification of tests and proofs: if there are alternate proof paths that do not require the modified portions of the model, then the requirement does not need to be re-verified [9].
- Robustness Analysis: It is possible to partition the model elements into MUST and MAY sets based on whether they are in every MIVC or only some MIVCs, respectively. This may allow insight into the relative importance of different model elements for the property. For example, if the MUST set is empty, then the requirement has been implemented in multiple ways, such as would be expected in a fault-tolerant system [9].

In addition, the Requirements Engineering community is keenly interested in approaches to manage requirements traceability. In most cases, it is assumed that there is a single "golden" set of trace links that describes how requirements are implemented in software [13]–[15]. However, if there are multiple MIVCs, then it is possible that there are several equally valid sets of trace links. Examining the diversity of all MIVCs could lead to changes in how traceability is performed for critical systems.

In this paper, we propose a new method for computing *all* IVCs that is *always* minimal for decidable model checking problems and *usually* (and detectably) minimal for model-checking problems that are generally undecidable. In recent years, a number of efficient algorithms for extracting minimal UNSAT subformulae (MUSes) have been proposed [16], most of which are focused on computing a single MUS [17]–[21]. In this paper, we adapt the recent work by Liffiton et al. [22] from the generation of MUSes from UNSAT-cores to all IVCs for inductive model checking. This requires changing the underlying mechanisms that are used to construct candidate solutions and also changing the structure of the proof of correctness. In addition, we demonstrate that the approach can terminate with all minimal IVCs even if the witness generator only generates approximately minimal IVCs (utilizing the "fast" `IVC_UC` algorithm from [7]). In our empirical results, this allows our algorithm to be quite efficient to the extent

```
node asw (alt1, alt2: int) returns (doi_on: bool);
   var
       a1_below, a2_below, a1_above, a2_above,
       one_below, both_above, on_p : bool;
   let
(1)    a1_below = (alt1 < THRESHOLD);
(2)    a2_below = (alt2 < THRESHOLD);
(3)    a1_above = (alt1 >= T_HYST);
(4)    a2_above = (alt2 >= T_HYST);
(5)    one_below = a1_below or a2_below;
(6)    both_above = a1_above and a2_above;
(7)    doi_on = if one_below then true
               else if both_above then false
               else (false -> pre(doi_on));
(8)    on_p = ((alt1 < THRESHOLD) and
              (alt2 < THRESHOLD)) => doi_on;
   tel;
```

Fig. 1. Altitude Switch Model

that in many cases, the cost of extracting all minimal IVCs is similar to the cost of finding a single guaranteed-minimal IVC, and on average is approximately 1.6x the cost of determining a single minimal IVC. The contributions of the work are therefore as follows:

- An algorithm for computing all minimal IVCs.
- A proof of correctness and completeness of the algorithm.
- An evaluation of the algorithm for performance and diversity of result sets against a benchmark suite.

Several commercial tools produce *proof-cores* [6], [23], which we believe to be similar to IVCs/MIVCs, but are not presented at a level of formality to perform a precise comparison. However, to the best of our knowledge, none of these tools offer to calculate *all* proof-cores. Our work can also be useful towards the support of this capability in future editions of these tools.

The rest of the paper is organized as follows. Section II introduces a running example used to illustrate concepts and our method. Section III covers the formal preliminaries for the approach. In Section IV, we present our method for enumerating all minimal IVCs, which is illustrated in Section V. In Sections VI and VII we talk about implementation and evaluation of our method. Finally, Section VIII mentions conclusions and future work.

II. Running Example

We will use a very simple system from the avionics domain to illustrate our approach. An Altitude Switch (ASW) is a hypothetical device that turns power on to another subsystem, the Device of Interest (DOI), when the aircraft descends below a threshold altitude, and turns the power off again after the aircraft ascends over the threshold plus some hysteresis factor. An implementation of an ASW containing two altimeters written in the Lustre language (simplified and adapted from [24]) is shown in Fig. 1. If either altimeter is below the constant `THRESHOLD`, then it turns on the DOI; else, if the system is inhibited or both altimeters are above the threshold plus the hysteresis factor `T_HYST`, then the DOI is turned off, and if neither condition holds, then in the initial computation it is false and thereafter retains its previous value. The notation `(false -> pre(doi_on))` in equation (7) describes an initialized register in Lustre: in the first step, the expression is `false`, and thereafter it is the previous value of `doi_on`. A simple property `on_p` states that if both altimeters are under the threshold, then the DOI is turned on. This property can easily be proved over the model using a k-induction based verifier such as `JKind` [25].

III. Preliminaries

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \to \mathit{bool}$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \to \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$. We assume the transition relation has the structure of a top-level conjunction. Given $T(u, u') = T_1(u, u') \wedge \cdots \wedge T_n(u, u')$ we will write $T = \bigwedge_{i=1..n} T_i$ for short. By further abuse of notation, T is identified with the set of its top-level conjuncts. Thus, $T_i \in T$ means that T_i is a top-level conjunct of T, and $S \subseteq T$ means all top-level conjuncts of S are top-level conjuncts of T. When a top-level conjunct T_i is removed from T, we write $T \setminus \{T_i\}$. Such a transition system can easily encode our example model in Section II, where each equation defines a conjunct within T that we will denote by the variable assigned; so, $T = \{$ `a1_below, a2_below, a1_above, a2_above, one_below, both_above, doi_on, on_p` $\}$.

The idea behind finding an IVC for a given property P [7] is based on inductive proof methods used in SMT-based model checking, such as K-induction and IC3/PDR [1], [26], [27]. Generally, an IVC computation technique aims to determine, for any subset $S \subseteq T$, whether P is provable by S. Then, a minimal subset that satisfies P is seen as a minimal proof explanation called a minimal Inductive Validity Core. Theorem 1 demonstrates that the minimization process is as hard as model checking, so finding a minimal inductive validity core may not be possible for some model checking problems.

Definition 1. Inductive Validity Core (*IVC*) [7]: *$S \subseteq T$ for $(I, T) \vdash P$ is an Inductive Validity Core, denoted by IVC(P, S), iff $(I, S) \vdash P$.*

Definition 2. Minimal Inductive Validity Core (*MIVC*) [7]: *$S \subseteq T$ is a minimal Inductive Validity Core, denoted by MIVC(P, S), iff IVC$(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \nvdash P$.*

Theorem 1. *Determining if an IVC is minimal is as hard as model checking.*
Proof: see [7]. □

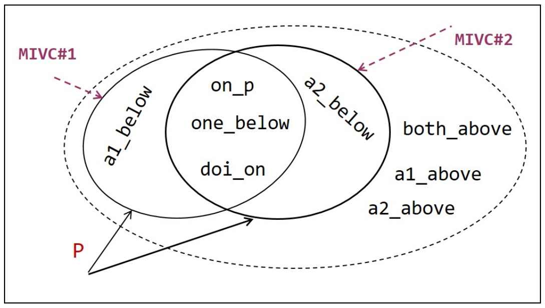


Fig. 2. Graphical representation of *MIVC*s for the model in Fig. 1 with $P = (\texttt{on\_p})$

Note that, given $(I, T) \vdash P$, P always has at least one *MIVC*, and it may also have many distinct *MIVC*s corresponding to different proof paths. To capture the latter, the *all MIVCs ($AIVC$)* relation has been introduced in [9].

Definition 3. All *MIVC*s ($AIVC$): *Given $(I, T) \vdash P$, $AIVC(P)$ is an association to all MIVCs for P:*

$$AIVC(P) \equiv \{\ S \mid S \subseteq T \wedge MIVC(P, S)\}$$

Fig. 2 illustrates these notions by a graphical representation of IVCs for property $P = (\texttt{on\_p})$ in the example presented in Section II. As shown in the picture, this property has two distinct *MIVC*s, which means the model satisfies P in two different ways: $\{\{$`a1_below, one_below, doi_on, on_p`$\}$, $\{$`a2_below, one_below, doi_on, on_p`$\}\}$, This is because in the implementation, the DOI is turned on when either of the altimeters is below the threshold, while our property states that they both must be below. Note that there is a subset of model elements, $\{$`a1_above, a2_above, both_above`$\}$, that does not show up in $AIVC(P)$. Elements in such a subset do not affect the satisfaction of P. In the complete ASW model in [24] there are additional properties that use these elements, but they are not necessary for the discussion in this paper.

IV. Method

Considering the definition of a *MIVC*, a brute-force technique for enumerating all *MIVC*s would be the same as exploring the power set of T (denoted by $\mathcal{P}(T)$). Basically, the algorithm needs to explore the provability of a given property by any subset of T, which would be computationally expensive. Our approach is an adaptation of the the work of MARCO for generating all minimal unsatisfiable subsets (MUSes) in [22], and only needs to explore a (small) portion of $\mathcal{P}(T)$ in order to compute $AIVC$. In fact, it can be viewed as an instantiation of the MARCO proof schema for the richer theory of sequential model checking. We begin by introducing several additional notions and definitions, most of which are analogous or equivalent to those in [22].

Definition 4. Maximal Inadequate Set (*MIS*): *$S \subset T$ for $(I, T) \vdash P$ is a Maximal Inadequate Set (MIS) iff $(I, S) \nvdash P$ and $\forall T_i \in T \setminus S.\ (I, S \cup \{T_i\}) \vdash P$.*

Given $(I, T) \vdash P$, for every $S \in \mathcal{P}(T)$, we have either $(I, S) \vdash P$ or $(I, S) \nvdash P$. In the former case, we say S is **adequate** for P; in the latter, we say that S is **inadequate** for the proof of P. Note that every *IVC* is an adequate set for P, and every *MIS* is an inadequate set.

Lemma 1. *For $(I, T) \vdash P$, if $S \subseteq T$ is adequate for property P, then all of its supersets are adequate for P as well:*

$$\forall S_1 \subseteq S_2 \subseteq T.\ (I, S_1) \vdash P \Rightarrow (I, S_2) \vdash P$$

Proof: From $S_1 \subseteq S_2$ we have $S_2 \Rightarrow S_1$. Thus the reachable states of (I, S_2) are a subset of the reachable states of (I, S_1). ∎

Corollary 1. *For $(I, T) \vdash P$, if a given subset S is inadequate, then all of its subsets are inadequate as well:*

$$\forall S_1 \subseteq S_2 \subseteq T.\ (I, S_2) \nvdash P \Rightarrow (I, S_1) \nvdash P$$

Proof: Immediate from Lemma 1. ∎

The basic idea behind an algorithm for computing $AIVC(P)$ is the same as exploration of $\mathcal{P}(T)$, with two major performance improvements. First, Lemma 1 and Corollary 1 are used to block large portions of $\mathcal{P}(T)$ from consideration. For example, if a set $S \in \mathcal{P}(T)$ is found to be inadequate, then all subsets of S are also inadequate and do not need to be explicitly considered. Second, if a set $S \in \mathcal{P}(T)$ is found to be adequate, then a fast algorithm (such as `IVC_UC` from [7]) is used to find a smaller $S' \subseteq S$ which is still adequate. This feeds into the first optimization since now all supersets of S' rather than S are blocked from future consideration.

To guide our algorithm, we now introduce a way of exploring $\mathcal{P}(T)$ which allows us to eliminate all subsets or supersets of any given set. We use a Boolean expression called map, which is in conjunctive normal form (CNF) and built gradually as the algorithm proceeds. Satisfying assignments for map correspond to elements of $\mathcal{P}(T)$. For each $S \in \mathcal{P}(T)$ that the algorithm determines to be adequate or inadequate, a corresponding clause is added to map which blocks S and all supersets or subsets, respectively, from consideration. When a clause is added to map, the corresponding $S \in \mathcal{P}(T)$ is called *explored*. The supersets or subsets of S which are blocked from consideration are called *excluded*. The remaining elements of $\mathcal{P}(T)$ are *unexplored*.

More precisely, given T with n top-level conjuncts, we define an ordered set of activation literals $\mathcal{A} = \{a_1, \ldots, a_n\}$, where each a_i has type Boolean. We assume the function $\textsc{ActLit} : T \rightarrow \mathcal{A}$ is a bijection assigning every $T_i \in T$ to an $a_i \in \mathcal{A}$ and vice versa. Then, a map for $AIVC(P)$ is a CNF formula built over the elements of $\mathcal{A}$ such that:

- Initially map is $\top$ since all of $\mathcal{P}(T)$ is unexplored.
- When map is satisfiable, a model of it is a set $M \in \mathcal{P}(\mathcal{A})$ consisting of those $a \in \mathcal{A}$ which are assigned $true$.
- Every model M of map corresponds to a set $S \in \mathcal{P}(T)$ such that $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ and $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$.
- For every explored set $S \in \mathcal{P}(T)$:
  - if S is adequate for P, then map contains a clause $\bigvee_{T_i \in S} \neg\textsc{ActLit}(T_i)$. This clause blocks all supersets of S from future consideration which is consistent with Lemma 1.
  - if S is inadequate for P, then map contains a clause $\bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$. This clause blocks all subsets of S from future consideration which is consistent with Corollary 1.

Lemma 2. *When map is satisfiable with model M, set $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ is not equal to any adequate or inadequate explored set, nor a subset (superset) of any inadequate (adequate) explored set in $\mathcal{P}(T)$.*

Proof: Proof by contradiction. Case 1: Suppose there is an adequate set $Ex \subseteq S$ that has been already explored. Therefore, according to the definition, map contains a clause $C = \bigvee_{T_i \in Ex} \neg\textsc{ActLit}(T_i)$, and since $Ex \subseteq S$, it is impossible for the model $M = \bigcup_{T_i \in Ex} \textsc{ActLit}(T_i)$ to satisfy C; hence, the assumption is false.

Case 2: Suppose there is an inadequate set Ex such that $S \subseteq Ex$ and Ex has been already explored. Therefore, according to the definition, map contains a clause $C = \bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$, and since $S \subseteq Ex$, it is impossible for the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$ to satisfy C; so, the assumption is false.

From Case 1 and Case 2, there is no model of map whose corresponding set in $\mathcal{P}(T)$ is a non-strict subset (superset) of any inadequate (adequate) explored set. ∎

Lemma 3. *For $(I, T) \vdash P$, map is satisfiable iff at least one $S \in AIVC(P)$ or one MIS of T is unexplored.*

Proof: Let map is satisfiable with a model M, and let $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ be the corresponding set of $\mathcal{P}(T)$. If S is adequate, then it contains a *MIVC*. That *MIVC* must not be explored since otherwise S would have been blocked from consideration. The *MIVC* must not be excluded since it is not a strict superset of any adequate set (by minimality) nor a subset of any inadequate set (by Corollary 1). Thus the *MIVC* must be unexplored. The case where S is inadequate is symmetric.

In the other direction, let $S \subseteq T$ be an unexplored *MIVC*. Then consider the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$. We will show that each clause of map is satisfied by M. There are two types of clauses to consider. A clause $\bigvee_{T_i \in S'} \neg\textsc{ActLit}(T_i)$ is in map only if S' is adequate. M would falsify this clause only if $S' \subseteq S$ which is impossible by minimality of S. A clause $\bigvee_{T_i \in (T \setminus S')} \textsc{ActLit}(T_i)$ is in map only if S' is inadequate. M would falsify this clause only if $S \subseteq S'$ which is imposssible by Corollary 1. Thus M is a model for map. The case for an unexplored *MIS* is symmetric. ∎

Corollary 2. *For $(I, T) \vdash P$, map is unsatisfiable iff every $S \in \mathcal{P}(T)$ has been explored or excluded.*

Proof: Immediate from the definition of map and Lemma 3. ∎

Algorithm 1 shows the process of capturing all *MIVC*s, which are kept in set A, along with a warning flag, explained below. In line 2, we create the set of activation literals used by function ACTLIT. Line 3 initializes map with $\top$ over the set of literals we have. The main loop of state exploration starts at line 4 and continues until map becomes UNSAT which means all the *MIVC*s have been found. We assume we have a function CHECKSAT that determines if an existentially quantified formula is satisfiable or not.[2] As long as map is satisfiable, the algorithm computes a *maximal* SAT model for it (line 5). In this context, a maximal SAT model is a model with as many $true$ assignment as possible without violating a clause; this problem, is equivalent to the MaxSAT problem, which has been well studied in the literature [29], [30].[3] So, we assume there is a method by which we are able to have a maximal model of map. Line 6 extracts a set $M \in \mathcal{P}(\mathcal{A})$ of literals assigned to $true$ in the model. Then, we need to obtain the corresponding set of S in $\mathcal{P}(T)$, which is done with function ACTLIT^{-1} in line 7.

We also assume there is a function CHECKADQ that checks whether or not P is provable by a given subset of T. Note that from Theorem 1, finding a minimal is undecidable if the original checking problem is undecidable. Thus, for undecidable model checking problems, CHECKADQ can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set S, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a warning flag w to the user that the results may be approximate. if S is adequate, a *MIVC* is computed by GETIVC and added to set A (lines 10-11).[4] In this case map is constrained by a new clause in a way described before and shown in line 12. However, in the case that S is inadequate or unknown, map is constrained by the corresponding literals from $T \setminus S$ in line 14. Finally, if S is unknown, the warning flag w is set to true, as the results may be approximate (lines 15-16).

Theorem 2. *Algorithm 1 will terminate.*

Proof: We assume that CHECKADQ has a finite timeout, so all operations within the loop require finite time. Each iteration of the while loop in Algorithm 1 blocks at least one element of $\mathcal{P}(T)$ which was not previously blocked. Since $\mathcal{P}(T)$ is finite, the algorithm terminates. ■

Theorem 3. *If no approximation warning is returned (w is* FALSE*), Algorithm 1 enumerates all MISes and MIVCs.*

Algorithm 1: Algorithm `All_IVCs` for computing $AIVC$

```
input  : $(I,T) \vdash P$
output: $AIVC(P)$, Approximation warning flag $w$
1  $A \leftarrow \varnothing$; $w \leftarrow$ FALSE
2  Create activation literals $\{a_1, \ldots, a_n\}$
3  $map \leftarrow \top$
4  while CHECKSAT($map$) do
5      $model \leftarrow$ build a maximal model of $map$
6      $M \leftarrow$ extract the set of variables assigned $true$ in
       $model$
7      $S \leftarrow \bigcup_{a_i \in M} \text{ACTLIT}^{-1}(a_i)$
8      $res \leftarrow$ CHECKADQ($P, S$)
9      if $res =$ ADEQUATE then
10         $S' \leftarrow$ GETIVC($P, S$)
11         $A \leftarrow A \cup \{S'\}$
12         $map \leftarrow map \wedge (\bigvee_{T_i \in S'} \neg\text{ACTLIT}(T_i))$
13     else
14         $map \leftarrow map \wedge (\bigvee_{T_i \in (T \setminus S)} \text{ACTLIT}(T_i))$
15         if $res =$ UNKNOWN then
16             $w \leftarrow$ TRUE
17 return $A, w$
```

Proof: By Theorem 2 the algorithm terminates. This means map is eventually unsatisfiable. If $w =$ FALSE then all model checking problems are solved definitively (no UNKNOWN results), so by Lemma 3, all *MIS*es and *MIVC*s are either explored or excluded. However, by maximality and Lemma 1, an *MIS* can never be excluded. Similarily, by minimality and Corollary 1, a *MIVC* can never be excluded. Thus all *MIS*es and *MIVC*s are explored and are elements of A by the end of the algorithm. ■

Note that none of the proofs above require that GETIVC returns a minimal IVC. From [7], it is computationally cheap to find an approximately minimal *IVC* using the algorithm `IVC_UC`; however, using the better, usually minimal *IVC* using the `IVC_UCBF` algorithm is computationally expensive. For efficiency reasons, it is much better to use the approximate `IVC_UC` algorithm to compute the set of all *MIVC*s. The `IVC_UCBF` algorithm attempts to repeatedly prove the property by brute-force removing elements (BF = "brute force"), so does much of the work of Algorithm 1 in a way that is not effective towards finding other IVCs. The overhead of the `IVC_UC` algorithm is on average 10% over the baseline proof, as opposed to 882% for the `IVC_UCBF` algorithm. In addition, the average increase in size of IVCs returned by `IVC_UC` is approximately 10% of the `IVC_UCBF` algorithm.

On the other hand, if GETIVC does not return minimal adequate sets, at the end of the process, set A may contain both *MIVC*s and some supersets of *MIVC*s. To make sure that the algorithm only returns the minimal adequate sets (*MIVC*s), all we need is to remove any supersets of other sets in A. We

[2]We assume readers are familiar with the Boolean satisfiability problem, which is the problem of determining whether there exists an assignment that satisfies a given propositional formula. For more information, refer to [28].

[3]MaxSAT is defined as the problem of satisfying as many (weighted) clauses as possible in a SAT instance. For N variables, similar to the MaxSAT problem, each clauses is weighted at $N+1$ and extra unit-weight clauses are added forcing each variable to 1.

[4]Note that CHECKADQ can be any method that verifies a safety property, such as K-induction, and the GETIVC function can be any function that returns an (approximately) minimal IVC, such as the `IVC_UC` or `IVC_UCBF` algorithms from [7]. The only requirement is that it follows the definition of an inductive validity core, that is: $S' \leftarrow \text{GETIVC}(P, S)$ implies that $S' \subseteq S$ and $(I, S') \vdash P$.

can do this "on the fly" by changing line 11 to the following: $A \leftarrow A \cup \{S'\} \setminus \{S \mid S \in A \wedge S' \subset S\}$. Obviously, the closer to minimal the results of GETIVC are, the fewer iterations are required for Algorithm 1 to terminate. Each non-minimal adequate set returned by GETIVC will induce an additional iteration for Algorithm 1.

V. ILLUSTRATION

To illustrate the `All_IVCs` algorithm we use the example presented in Section II with $P = (\texttt{on\_p})$. For better description, we view T as an ordered set of its top-level conjuncts; i.e. $T = \{$ `a1_below`, `a2_below`, `a1_above`, `a2_above`, `one_below`, `both_above`, `doi_on`, `on_p` $\}$. The algorithm starts with creating activation literals for each $T_i \in T$. Let the ordered set of Boolean variables $\{a_1, \ldots, a_8\}$ be the corresponding literals to the elements of T (e.g. ACTLIT(`a1_below`) $= a_1$ and ACTLIT(`on_p`) $= a_8$). Then, line 3 initializes *map* with $\top$.

In the first iteration of the `while` loop, since *map* is empty, it is satisfiable, and a model for it can be any subset of literals. So obviously, the first maximal model of *map* contains all the literals, which means, in line 6, $M = \{a_1, \ldots, a_8\}$, and in line 7, $S = T$. Since S is adequate for P, the GETIVC module is called in line 10. Suppose the returned *MIVC* by this function is $S' = \{$`a1_below`, `one_below`, `doi_on`, `on_p`$\}$; this set is added to A in line 11, and thus it comes to adding a new clause to *map* (line 12), which makes $map = (\neg a_1 \vee \neg a_5 \vee \neg a_7 \vee \neg a_8)$. As discussed, this constraint marks all the supersets of S' as blocked and prunes them off the search space.

For the second iteration, *map* is still satisfiable, so the algorithm gets to find a maximal model of it in line 5. Suppose this time, the maximal model makes $M = \{a_1, \ldots, a_7\}$, which leads to $S = T \setminus \{\texttt{on\_p}\}$ in line 7. Since S is inadequate for P, the algorithm jumps to line 12 updating *map* as $map \leftarrow map \wedge a_8$. Adding this new clause removes all the subsets of $T \setminus \{\texttt{on\_p}\}$ from the search space. Similarly, in the third iteration, if the maximal model of *map* yields $M = \{a_1, \ldots, a_4, a_6, \ldots, a_8\}$, then $S = T \setminus \{\texttt{one\_below}\}$ will be another inadequate set that makes *map* become $map \leftarrow map \wedge a_5$ in line 14.

Suppose, in the fourth iteration, the maximal model leads to $M = \{a_2, \ldots, a_8\}$ and $S = T \setminus \{\texttt{a1\_below}\}$ in lines 6 and 7. Since this S is adequate for P, GETIVC computes a new *MIVC* in line 10. Let the new *MIVC* be $S' = \{$`a2_below`, `one_below`, `doi_on`, `on_p`$\}$; after adding this set to A, it is time to constrain *map* by a new clause in line 11, which results in $map \leftarrow map \wedge (\neg a_2 \vee \neg a_5 \vee \neg a_7 \vee \neg a_8)$.

After these iterations, *map* is still satisfiable, and the maximal model is $S = T \setminus \{\texttt{a1\_below}, \texttt{a2\_below}\}$ in line 7. In this case, S is inadequate, so we update *map* as $map \leftarrow map \wedge (a_1 \vee a_2)$ (line 14). After adding this new clause to *map*, all the subsets of $T \setminus \{\texttt{a1\_below}, \texttt{a2\_below}\}$ will be blocked. The algorithm continues similar to the forth iteration leading to S (in line 7) and *map* (in line 14) to be as $S = T \setminus \{\texttt{doi\_on}\}$ and $map \leftarrow map \wedge a_7$.

Finally, after the sixth iteration, *map* becomes `UNSAT` and the algorithm terminates. Note that MISes and IVCs may be discovered in different orders from what explained here. The order by which sets are explored is quite dependent on the maximal model returned in line 5 as well as the *MIVC*s returned in line 10 because there could be several distinct maximal models (*MIS*es) and *MIVC*s. For this example with a $|T| = 8$ and $|\mathcal{P}(T)| = 2^8$, a brute force approach of power set exploration needs to look into 256 cases. However, the `All_IVCs` algorithm only explored 6 cases to cover the entire power set.

VI. IMPLEMENTATION

We have implemented the `All_IVCs` algorithm in an industrial model checker called `JKind` [25], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [31] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [32]. Verification is supported by multiple "proof engines" that execute in parallel, including K-induction, property directed reachability (PDR), and lemma generation engines that attempt to prove multiple properties in parallel. To implement the engines, `JKind` emits SMT problems using the theories of linear integer and real arithmetic. `JKind` supports the `Z3`, `Yices`, `MathSAT`, `SMTInterpol`, and `CVC4` SMT solvers as back-ends. When a property is proved and IVC generation is enabled, an additional parallel engine executes the `IVC_UC` algorithm [7] to generate an (approximately) minimal IVC. To implement our method, we have extended `JKind` with a new engine that implements Algorithm 1 on top of `Z3`. We use the `JKind` IVC generation engine to implement the GETIVC procedure in Algorithm 1.

As mentioned in Section IV the CHECKADQ procedure may not terminate. In our implementation, we measure the time required to prove the property and the initial given the full model (*proof-time*), and the time required to calculate the first (approximate) IVC using `IVC_UC` (*`IVC_UC`-time*). We then set a timeout for each iteration of the `All_IVCs` algorithm to (30 sec + 5 × (*proof-time* + *`IVC_UC`-time*)). In almost all cases in our experiment and our use of the tools, this timeout is sufficient to ensure exact results. In the experiment, only 15 of 475 models (3%) had potentially approximate results. It is important to note that by increasing the timeout, it is possible that in some cases smaller IVCs can be generated, but the general problem will remain due to the undecidability of the model checking problem.

VII. EXPERIMENT

We are interested in examining the *efficacy* and *efficiency* of generating all minimal IVCs, as compared to algorithms for computing a *single approximately minimal* IVC, and a *minimal IVC* as implemented in [7] using the `IVC_UC` and `IVC_UCBF` algorithms, respectively. We would also like to know how performance is affected by the size of models and number of minimal IVCs. Finally, we are also interested in determining whether the `All_IVCs` algorithm generates *smaller* cores than

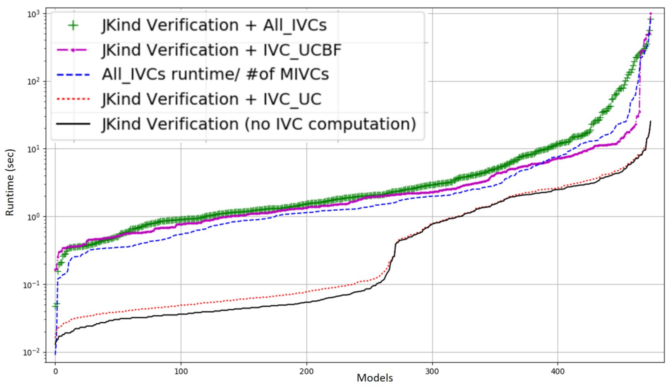


Fig. 3. Runtime of `All_IVCs`, `IVC_UCBF`, and `IVC_UC` algorithms

are generated by the `IVC_UCBF` algorithm that generates a single MIVC. Therefore, we investigate the following research questions:

- **RQ1:** How expensive is it to compute the `All_IVCs` algorithm for determining all minimal IVCs when compared to the `IVC_UC` and `IVC_UCBF` algorithms, which find a single approximately minimal and guaranteed minimal IVC?
- **RQ2:** How is the verification time of the `All_IVCs` algorithm affected by the baseline proof time and the number of IVCs that can be found for a property?
- **RQ3:** How large are the IVCs produced by the `All_IVCs` algorithm compared to those of `IVC_UC` and `IVC_UCBF`?

A. Experimental Setup

The benchmark contains 475 Lustre models, 395 from [32] and 80 industrial models derived from [33] and other sources. Most of the benchmark models from [32] are small (10kB or less, with 6-40 equations) and include a range of hardware benchmarks and software problems involving counters that are difficult to solve inductively. The 80 industrial models each contain over 600 equations and are each $\geq$80kB in size.

We selected only benchmark problems consisting of a Lustre model with properties that `JKind` could prove with an hour timeout. For each test model, we computed `All_IVCs`, `IVC_UC`, and `IVC_UCBF` algorithms in a configuration with the `Z3` solver and the "fastest" mode of `JKind` (which involves running the k-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on an Intel(R) i5-4690, 3.50GHz, 16 GB memory machine running Linux, and are available at [34].

B. Experimental Results

In this section, we examine our experimental results to address the research questions defined in the experiment.

1) RQ1: To address RQ1, we measured the performance overhead of the various IVC algorithms against the baseline time necessary to find a proof using inductive model checking. Fig. 3 provides an overview of the overhead of the `All_IVCs` algorithm in comparison with the `IVC_UC` and `IVC_UCBF` algorithms. In the figure, each curve is ranked along the x-axis according to the time required for the algorithm to terminate for each analysis problem. Table I provides a summary of the computation time and the overhead of different algorithms. The `IVC_UC` algorithm imposes a 1.25x overhead to the baseline proof time, whereas both the `IVC_UCBF` and `All_IVCs` algorithms add a substantial time penalty: `IVC_UCBF` and `All_IVCs` add a (mean) 18.8x and 31.3x overhead, respectively, to the proof time. For small models, much of this penalty is due to starting many instances of the SMT solver; if we examine models that require $\geq 1s$ of analysis time, the mean overhead of `All_IVCs` over the baseline analysis drops from 31.3x to 9.7x.

TABLE I
RUNTIME AND OVERHEAD OF DIFFERENT COMPUTATIONS

runtime (sec)	min	max	mean	stdev
proof-time	0.016	25.489	1.250	2.381
`All_IVCs`	0.009	792.01	16.457	64.491
`IVC_UCBF`	0.163	996.734	11.987	68.525
`IVC_UC`	0.003	1.126	0.078	0.158

2) RQ2: For this question, we examine how the proof time of the original model and the number of MIVCs associated with the property affects the analysis time of the `All_IVCs` algorithm. Fig. 4 provides an overview of this data. The data in Fig. 4 is sorted twice along the x-axis: the major axis is the number of MIVCs that exist for the model, and the minor axis is the analysis time of the baseline model. In this graph, the graph shows how both factors effect the performance of the `All_IVCs` algorithm. Note that there are two scales for the y-axis: the scale on the left is a logarithmic scale of performance in terms of the run time; the scale on the right is a linear scale based on the number of minimal IVCs discovered.

Fig. 4 shows two distinct trends. First, for models whose baseline proofs are inexpensive and that only have a single MIVC, the `All_IVCs` is roughly equivalent in performance to the `IVC_UCBF`. However, as proofs become more expensive for a single MIVC, the `All_IVCs` begins to underperform the `IVC_UCBF`, this is the case for the properties with one MIVC. In the cases where several MIVCs are found, the performance of the `All_IVCs` is driven to a large degree by the number of MIVCs that exist: the more MIVCs associated with a property, the higher the expense of `All_IVCs` as compared to the `IVC_UCBF` algorithm.

3) RQ3: For this research question, we analyzed the minimality of the discovered IVC by each algorithm (Figure 5). Since 394 of the models had only one MIVC, for these models, the size of the minimum model produced by the `All_IVCs` algorithm should be the same as the `IVC_UCBF` algorithm. For the remainder, even when multiple MIVCs were produced, in only 12 cases did the `All_IVCs` produce smaller minimal IVCs. For these 12 models, the smallest MIVC was 16% the size of the MIVC produced by `IVC_UCBF`, and in the most dramatic case, the number of elements shrank from 30 to 5.

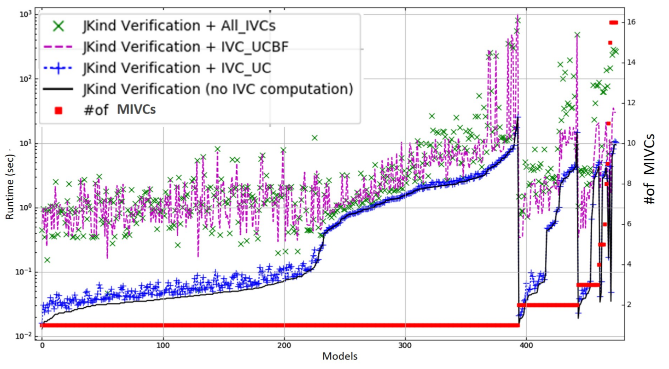


Fig. 4. Runtime of different computations along with the number of MIVCs

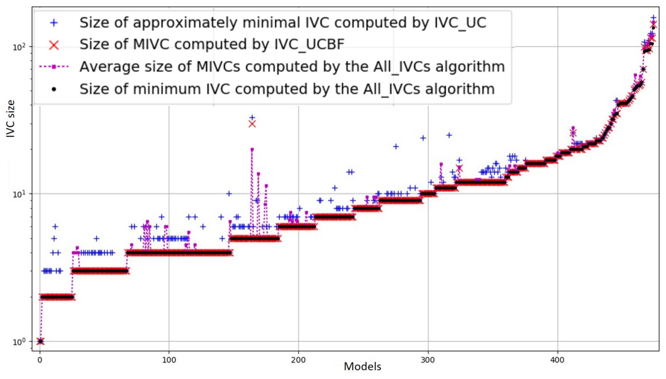


Fig. 5. Size of the IVC sets produced by different algorithms

VIII. Conclusions & Future Work

The idea of extracting a minimal IVC for a given property and its applications was recently introduced in [7]. However, a single IVC often does not provide a complete picture of the traceability from a property to a model. In this paper, we have addressed the problem of extracting *all minimal* IVCs. We have shown the correctness and completeness of our method and algorithm. In addition, we have a substantial evaluation that shows that the practicality and efficiency of our technique.

Our method is inspired by a recent work in the domain of satisfiability analysis [22]. One interesting future direction is to devise similar MIVC enumeration algorithms based on other studies on MUSes extraction such as [21]. We are also looking into improving our implementation by using more efficient methods for the CheckAdq and GetIVC modules used by our algorithm. Another interesting direction is to parallelize the enumeration process: it is certainly possible to ask for multiple distinct maximal models to be solved in parallel.

We also plan to investigate additional applications of the idea. When performing *compositional verification*, the All-IVCs technique may be able to determine *minimal component sets* within an architecture that can satisfy a given set of requirements, which may be helpful for design-space exploration and synthesis. Finally, we are interested in adapting the notion of (all) validity cores for *bounded* model checking for quantifying how much of models have been explored by bounded analysis.

ACKNOWLEDGMENTS This work was carried out within the HACMS and SOSITE Phase II grants (DARPA FA8750-12-9-0179 and FA8650-16-C-7656).

References

[1] N. Een *et al.*, "Efficient implementation of property directed reachability," in *FMCAD'11*.
[2] M. Sheeran *et al.*, "Checking safety properties using induction and a SAT-solver," in *FMCAD'02*, 2000.
[3] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *STTT*, 2003.
[4] M. Whalen *et al.*, "Integration of formal analysis into a model-based software development process," in *FMICS*, 2007.
[5] L. Pike, "A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms"," *TSE*, 2006.
[6] "Cadence JasperGold Formal Verification Platform," https://www.cadence.com/.
[7] E. Ghassabani *et al.*, "Efficient generation of inductive validity cores for safety properties," in *FSE'16*, 2016.
[8] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula," in *SAT'03*.
[9] A. Murugesan *et al.*, "Complete traceability for requirements in satisfaction arguments," in *RE'16 (RE@Next! Track)*, 2016.
[10] E. Ghassabani *et al.*, "Proof-based coverage metrics for formal verification," in *ASE'17*, 2017.
[11] A. Mebsout and C. Tinelli, "Proof certificates for smt-based model checkers for infinite-state systems," in *FMCAD'16*, 2016.
[12] A. Ivrii *et al.*, "Small inductive safe invariants," in *FMCAD'14*, 2014.
[13] "Center of Excellence for Software Traceability," http://www.coest.org, 2016.
[14] J. H. Hayes *et al.*, "Improving requirements tracing via information retrieval," in *RE'03*, 2003.
[15] J. Cleland-Huang *et al.*, "Best practices for automated traceability," *Computer*, 2007.
[16] M. H. Liffiton *et al.*, "From MaxSAT to MinUNSAT: Insights and applications," *Ann Arbor*, 2005.
[17] F. Bacchus and G. Katsirelos, "Using minimal correction sets to more efficiently compute minimal unsatisfiable sets," in *CAV'15*, 2015.
[18] A. Belov and J. Marques-Silva, "Muser2: An efficient mus extractor," *JSAT journal*, 2012.
[19] A. Belov *et al.*, "Core minimization in sat-based abstraction," in *DATE'13*, 2013.
[20] A. Belov *et al.*, "Towards efficient MUS extraction," *AI Communications*, 2012.
[21] A. Nadel *et al.*, "Accelerated deletion-based extraction of minimal unsatisfiable cores," *JSAT journal*, 2014.
[22] M. Liffiton *et al.*, "Fast, flexible MUS enumeration," *Constraints*, 2016.
[23] Z. Hanna *et al.*, "Formal verification coverage metrics for circuit design properties," 2015. [Online]. Available: https://www.google.com/patents/US20150135150
[24] M. Heimdahl *et al.*, "Deviation analysis via model checking," in *ASE'02*, 2002.
[25] "JKind," http://loonwerks.com/tools/jkind.html.
[26] T. Kahsai *et al.*, "Incremental verification with mode variable invariants in state machines," in *NFM'12*, 2012.
[27] N. Amla *et al.*, "An analysis of sat-based model checking techniques in an industrial environment," in *CHARME'05*, 2005.
[28] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, 1971.
[29] J. Davies and F. Bacchus, "Solving MAXSAT by solving a sequence of simpler sat instances," in *CP'11*, 2011.
[30] A. Morgado *et al.*, "Iterative and core-guided MaxSAT solving: A survey and assessment," *Constraints*, 2013.
[31] N. Halbwachs *et al.*, "The synchronous dataflow programming language Lustre," *Proceedings of the IEEE*, 1991.
[32] G. Hagen and C. Tinelli, "Scaling up the formal verification of lustre programs with smt-based techniques," in *FMCAD'08*, 2008.
[33] A. Murugesan *et al.*, "Compositional verification of a medical device system," in *HILT'13*, 2013.
[34] "All IVCs repository," https://github.com/elaghs/Working/tree/master/all_ivcs/experiments.

Duality-Based Interpolation for Quantifier-Free Equalities and Uninterpreted Functions

Leonardo Alt, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina
Università della Svizzera italiana, Lugano, Switzerland
Email:firstname.lastname@usi.ch

Abstract—**Interpolating, i.e., computing safe over-approximations for a system represented by a logical formula, is at the core of symbolic model-checking. One of the central tools in modeling programs is the use of the equality logic and uninterpreted functions (EUF), but certain aspects of its interpolation, such as size and the logical strength, are still relatively little studied. In this paper we present a solid framework for building compact, strength-controlled interpolants, prove its strength and size properties on EUF, implement and combine it with a propositional interpolation system and integrate the implementation into a model checker. We report encouraging results on using the interpolants both in a controlled setting and in the model checker. Based on the experimentation the presented techniques have potentially a big impact on the final interpolant size and the number of counter-example-guided refinements.**

I. Introduction

An important skill in constructing mathematical proofs is to identify the aspects of the problem that are relevant. When applied to formal reasoning about the correctness of software this means ignoring the parts of the system that play no role in its correctness. One such approach that works well in automated software verification based on satisfiability modulo theories (SMT) engines (see, e.g, [1]) is to employ the Equality Logic and Uninterpreted Functions (EUF) when applicable: in some cases it suffices to assume that a given function returns the same value when invoked with the same arguments. This technique is particularly useful, for example, when modeling memory or arrays [2], proving program equivalence [3], or as a technique for avoiding flattening in solving bit-vector problems [4], [5].

Generalizing a formula over the states reachable by a program is a natural subtask when summarizing the behavior of a procedure [6], or computing a fixed-point of a transition function [7], [8]. These techniques are now popular in software model-checking [9], [10], and together with the theory-based abstraction result in a growing interest in an over-approximation technique known as *interpolation*.

In this paper we present the *EUF-interpolation system* which aims at specializing and tailoring interpolants for the needs of interpolation-based model-checking. The paper contributes to the state-of-the-art by (i) providing the first approach for controlling the strengths of EUF interpolants; (ii) identifying a strength lattice of interpolation algorithms; and (iii) proving under certain assumptions the size order for the interpolants produced by the system. In addition we (iv) provide an implementation of the system; (v) integrate and experiment with the system on a model checker; and (vi) study the combination of labeled interpolation systems for EUF and propositional logic. The EUF-interpolation operates on the proof of unsatisfiability in EUF based on a recursive algorithm for building a final interpolant from partial interpolants and uses *duality* of interpolants, a logical relation between an interpolant and its negation discussed below, to control the strength of the constructed partial and final interpolants.

The system is implemented in the SMT solver OpenSMT2 [11], and used in a model-checking algorithm based on the interpolating incremental C verifier HiFrog [6]. This gives us the advantage of making a direct connection between the theoretical contributions and practice. We evaluate the efficiency of the EUF-interpolation system with two major experiments. In the first experiment we verify a set of C software verification problems produced by HiFrog, and in the second experiment we study different combinations of propositional and EUF interpolation algorithms on a set of instances from the SMT-LIB benchmark collection. Based on the results the system has a big impact on the generated interpolants, and the interpolants seem to be very useful in our application to model-checking. To the best of our knowledge our work is the first to consider the duality of interpolants in constructing EUF interpolants recursively, and to report experiments with EUF interpolation together with incremental verification.

a) Related work : Recent work on *labeled interpolation systems* (LIS) addresses interpolation in propositional logic [12], [13], [14], [15] by providing control over fitting the interpolant strength and size to particular model-checking applications. Our approach extends the work on propositional interpolation to SMT theories and in particular to EUF. Interpolation procedures for EUF have been introduced in [16], [17]. The interpolation procedure given in [16] provides a way of computing a single interpolant from a given proof. The technique is extended in [17] to allow construction of several interpolants through the coloring of congruence graphs edges. Our work differs fundamentally from both these approaches by using duality for controlling the interpolant strength, a feature not available in earlier formalizations.

The parametric interpolation frameworks presented in [18] and [19] generalize first-order interpolation procedures. The former provides labeled interpolation systems for hyper-resolution proofs which are then extended to first order in-

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

terpolation systems for local proofs; the latter generalizes the former further to non-local proofs. Both of these techniques provide control on the propositional level. Unlike ours, they are not specialized and optimized for EUF and, to the best of our knowledge, have not been implemented.

Other orthogonal procedures exist for the quantifier-free fragments of the theories of linear integer arithmetics [20], [21], linear real arithmetics [16], [22], [23], and Arrays [24], while [25] provides a labeled interpolation system for Non-linear Real Arithmetics. On a high level, we believe that the duality-based approach followed in this work can be applied also in these fields.

This paper is organized as follows: Sec. II presents a general algorithmic framework for interpolation as a preliminary for the EUF-interpolation system. The main result on the EUF-interpolation system is presented in Sec. III The experiments are reported in Sec. IV, and the paper concludes in Sec. V. For lack of space the proofs are available in the extended version of the paper, available with the implementation and more experimental results at http://verify.inf.usi.ch/euf-interpolation.

II. Preliminaries

This paper considers the extension of propositional logic to Boolean variables that are interpreted as equalities over uninterpreted functions. Following [26], we call this extension the theory of equality logic and uninterpreted functions (EUF). For example $\neg(a = b) \vee f(a) = f(b)$ is an EUF formula containing the uninterpreted functions a, b, and f, embedded in a Boolean structure. Given an EUF formula F, we call the equality ($=$), and the Boolean connectives (e.g. $\neg, \wedge, \vee$) the *logical symbols*, while the Boolean variables and uninterpreted functions are its *non-logical symbols*, denoted by $\mathit{Vars}(F)$.

Given an unsatisfiable conjunction $A \wedge B$ of EUF formulas A and B, an *interpolation instance* is a pair (A, B), and an *interpolant* for (A, B) is a formula $I(A, B)$ such that *(i)* $A \rightarrow I(A, B)$, *(ii)* $I(A, B) \wedge B$ is unsatisfiable, and *(iii)* $\mathit{Vars}(I(A, B)) \subseteq \mathit{Vars}(A) \cap \mathit{Vars}(B)$. When B is clear from context, we refer to $I(A, B)$ as an *interpolant for A*. In general several interpolants can be computed for an instance (A, B). We denote an algorithm computing an interpolant $I(A, B)$ by $\mathit{Itp}(A, B)$, and, with a slight abuse of the notation, use $\mathit{Itp}(A, B)$ to denote the interpolant $I(A, B)$ when the interpolation algorithm needs to be specified. A central concept to this paper is the duality between interpolation algorithms: Given an interpolation algorithm $\mathit{Itp}(A, B)$, also the algorithm $\mathit{Itp}^{-}(A, B)$ returning $\neg \mathit{Itp}(B, A)$ computes an interpolant for (A, B), as can be seen from the following reasoning: By definition, $\mathit{Itp}^{-}(A, B) = \neg \mathit{Itp}(B, A)$. $\mathit{Itp}(B, A)$ satisfies (i) $B \rightarrow \mathit{Itp}(B, A)$; (ii) $\mathit{Itp}(B, A) \rightarrow \neg A$; and (iii) $\mathit{Vars}(\mathit{Itp}(B, A)) \subseteq \mathit{Vars}(A) \cap \mathit{Vars}(B)$. By rewriting, from (ii) follows that (iv) $A \rightarrow \neg \mathit{Itp}(B, A)$, and from (i) that (v) $\neg \mathit{Itp}(B, A) \rightarrow \neg B$. From (iii), commutativity of intersection, and definition of non-logical symbols, follows (vi) $\mathit{Vars}(\neg \mathit{Itp}(B, A)) \subseteq \mathit{Vars}(B) \cap \mathit{Vars}(A)$.

In this work we consider algorithms that build interpolants based on the unsatisfiability proof of $A \wedge B$. We make this explicit by denoting the interpolation algorithm (and the resulting interpolant) by $\mathit{Itp}(A, B, R)$, where R is the *refutation* representing the proof of unsatisfiability. In this work we are particularly interested in ordering interpolation algorithms with respect to the strength of the interpolants they compute. An interpolant I is *stronger* than an interpolant I' if $I \rightarrow I'$. We extend the strength relation to interpolation algorithms: if $\mathit{Itp}^{s}(A, B, R) \rightarrow \mathit{Itp}^{w}(A, B, R)$ for algorithms Itp^{s} and Itp^{w} for all interpolation instances (A, B), then Itp^{s} is stronger than Itp^{w}. If the strength relation can be established between the algorithms Itp and Itp^{-}, we call the algorithm computing the stronger interpolant the *base* and the weaker the *dual interpolation algorithm* and denote them by Itp and Itp', respectively.

Algorithm 1 Congruence closure

```
1: procedure CongruenceClosure($T, Eq$)
2:     Initialize $E \leftarrow \emptyset$ and $G \leftarrow (T, E)$
3:     repeat pick $x, y \in T$ such that $(x \not\sim y)$
4:         if (a) $(x = y) \in Eq$ or
5:         (b) $x$ is $f(x_1, \ldots, x_k)$, $y$ is $f(y_1, \ldots, y_k)$, and
6:         $(x_1 \sim y_1), \ldots, (x_k \sim y_k)$ then
7:             $E \leftarrow E \cup \{(x, y)\}$
8:     until no such $x, y$ can be chosen so that $E$ would grow
9:     return $G$
```

A. EUF Preliminaries

This section describes our interpolation system for EUF. The presentation is based on [17] and uses the congruence graph as the refutation.

Many EUF solvers rely on the *congruence closure* algorithm [27] to decide the satisfiability of a set of equalities and disequalities. The algorithm, described in Alg. 1, takes as input a finite set Eq of equalities, and the subterm-closed set T over which Eq is defined. During the execution the algorithm builds an undirected *congruence graph* G using the set T as nodes. We write $(x \sim y)$ if there is a path in G connecting x and y and denote this path by $\overline{xy}$.

Theorem 1 (c.f. [27]): Let S be a set of EUF disequalities $x \neq y$ over the terms T. The set $S \cup Eq$ is satisfiable if and only if the congruence graph G constructed by CongruenceClosure(T, Eq) has no path $(x \sim y)$ such that $(x \neq y) \in S$.

During the creation of G, an edge (x, y) is added only if $(x \sim y)$ does not hold, which ensures that G is acyclic. Therefore, for any pair of terms x and y such that $(x \sim y)$ holds in G, the path $\overline{xy}$ connecting these terms is unique. The path $\overline{xx}$ is called an *empty path.* For an arbitrary path π, we use the notation $[\![\pi]\!]$ to represent the equality of the terms that π connects. If, for example, $\pi = \overline{xy}$, then $[\![\pi]\!] := (x = y)$. We also extend this notation over sets of paths P so that $[\![P]\!] := \bigwedge_{\sigma \in P} [\![\sigma]\!]$.

An edge may be added to a congruence graph G because of two different reasons in Alg. 1 at line 7. Edges added because of Condition (a) are called *basic*, while edges added because of Condition (b) are called *derived.* Let e be a derived edge

$(f(x_1, \ldots, x_k), f(y_1, \ldots, y_k))$. The k *parent paths* of e are $\overline{x_1y_1}, \ldots, \overline{x_ky_k}$. Given a congruence graph G and two terms x, y such that $x \sim y$ we denote by $G[\overline{xy}]$ the congruence graph obtained from the graph G by including the edges and terms that appear on the path $\overline{xy}$ and recursively all its parent paths.

To compute an interpolant for (A, B), the congruence graph needs to be annotated with the information on which equalities and terms belong to A and which to B. This information is encoded using *colors*. Let F be a set of equalities and disequalities, $A \cup B$ a partition of F, and $(x \bowtie y) \in F$ an equality or a disequality over the terms x and y (i.e., $\bowtie \in \{=, \neq\}$). A term is a-colorable if all its non-logical symbols occur in A; b-colorable if all its non-logical symbols occur in B; and ab-colorable if both a and b-colorable. Given a set of edges E of a congruence graph, a coloring $C : E \to \{a, b\}$ assigns a color a or b to each edge in E considering two restrictions: *(i)* basic edges $e = (x, y)$ must be colored a if $(x = y) \in A$ and b if $(x = y) \in B$; and *(ii)* if an edge (x, y) has color $\kappa \in \{a, b\}$, both x and y must be κ-colorable. In particular a derived or basic edge $e = (x, y)$ such that both x and y are ab-colorable can be coloured arbitrarily. A path in a congruence graph is colorable if all its edges are colorable, and a congruence graph is colorable if all its edges are colorable.

While it is possible to construct a non-colorable congruence graph, the following lemma and its constructive proof in [17] state that we may assume without loss of generality that congruence graphs are colorable.

Lemma 1 (c.f. [17]): Let (A, B) be an interpolation instance over EUF. If x and y are colorable terms and if $A, B \models (x = y)$, then there exist a term set T and a colorable congruence graph over the equalities contained in $A \cup B \cup T$ in which $(x \sim y)$.

We denote a congruence graph G colored with a function C by G^C. A path is called an *a-path* if all its edges are colored a, and a *b-path* if all its edges are colored b. A *factor* of a path in G^C is a maximal subpath such that all its edges have the same color. Notice that every path is uniquely represented as a concatenation of the consecutive factors of opposite colors.

Example 1: Let $A := \{(v_1 = f(y_1)), (f(y_2) = v_2), (y_1 = t_1), (t_2 = y_2), (s_1 = f(r_1)), (f(r_2) = s_2), (r_1 = u_1), (u_2 = r_2)\}$ and $B := \{(x_1 = v_1), (v_2 = x_2), (t_1 = f(z_1)), (f(z_2) = t_2), (z_1 = s_1), (s_2 = z_2), (u_1 = u_2), (x_1 \neq x_2)\}$. Figure 1 shows a colored congruence graph G^C built while proving the unsatisfiability of A and B with Alg. 1. The curvy edges with the labels *s* or *w* in G^C are not relevant for this example and are used later in Section III. The congruence graph G^C demonstrates the joint unsatisfiability of A and B, since it proves $(x_1 = x_2)$ and $(x_1 \neq x_2)$ is an original term. Edges are represented by thick lines, and dotted arrows point to the parents of derived edges. We present a-colorable nodes (terms) and a-colored edges by black circles and solid lines, b-colorable nodes and b-colored edges by white circles and dashed lines, and ab-colorable nodes by gray circles. In the first (top) path of G^C, we see that basic edges (original equalities from $A \cup B$) are used to prove $(r_1 = r_2)$. This fact is used to infer $(f(r_1) = f(r_2))$, which is in turn used as a derived edge in the path below, proving $(z_1 = z_2)$. The equality $(f(z_1) = f(z_2))$ is then inferred and used to prove $(y_1 = y_2)$ in the path below. In the last (bottom) path of G^C, the derived edge representing $(f(y_1) = f(y_2))$ is created and finally $(x_1 = x_2)$ is proved.

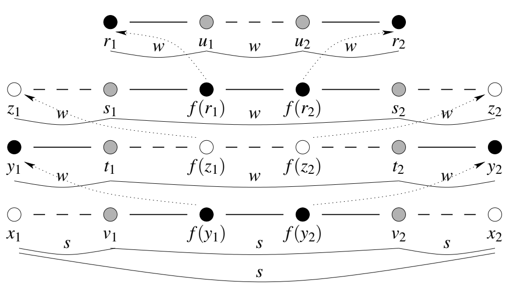


Figure 1. Congruence graph G^C that proves the unsatisfiability of $A \cup B$

III. The EUF Interpolation System

In this section we present the EUF-interpolation system which extends the approach described in [17] with a modular use of dual interpolants. Our main novelty is the control over the interpolant strength. Due to lack of space all the proofs of the theorems in this section are presented in Appendix **??**.

Intuitively, the approach computes partial interpolants with either a base or a dual interpolation algorithm using the structure of a congruence graph. We show that while interpolating on a fixed congruence graph the liberty in choosing between the two interpolation algorithms allows computing several interpolants that can be partially ordered with respect to their strength. To make this choice explicit we introduce the *labeling functions* L for the EUF-interpolation system, and the algorithm Itp_L for computing the interpolants.

Definition 1 (Labeling function): Let $G[\overline{xy}]^C$ be a colored congruence graph and W its factors. A *labeling function* $L : W \cup \{\overline{xy}\} \to \{s, w\}$ labels the factors and the path corresponding to the conflict $x \neq y$ as s or w.

We emphasize that colors, described in Sec. II-A, and labels are different concepts. The colors a, b tell if an edge belongs to A or B, whereas labels s, w determine whether to use the primal or the dual interpolant.

Given an (unsatisfiable) interpolation instance (A, B), an EUF interpolation algorithm $Itp_L(A, B, G[\overline{xy}]^C)$ computes an interpolant for (A, B); $G[\overline{xy}]^C$ is a congruence graph with coloring C; $\overline{xy}$ a path such that $(x \sim y)$ is in G and the disequality $(x \neq y)$ exists in $A \cup B$; and L is a labeling function. We omit A, B, G^C and L when they are clear from the context, referring to the interpolation algorithm and the corresponding interpolant as $Itp(\overline{xy})$. Given an arbitrary path σ we define separately two constant labeling functions $L_s(\sigma) = L_s = s$ and $L_w(\sigma) = L_w = w$ that will be useful in the following analysis.

The interpolation algorithms in [16] and [17] essentially compute an interpolant by collecting the A-factors that prove $(x = y)$ in G^C. To maintain the unsatisfiability with the B

part of the problem, the A factors will then be implied by their B*-premise set*. A *premise set* for factor of a given color is the set of equalities of the opposite color justifying the existence of the factor's parent edges. More technically, the B-premise set $\mathcal{B}$ for a path π is

$$\mathcal{B}(\pi) := \begin{cases} \bigcup\{\mathcal{B}(\sigma) | \sigma \text{ is a factor of } \pi\}, \text{if } \pi \text{ has } \geq 2 \text{ factors;} \\ \{\pi\}, \text{if } \pi \text{ is a } B\text{-path; and} \\ \bigcup\{\mathcal{B}(\sigma) | \sigma \text{ is a parent path of an edge of } \pi\}, \\ \quad \text{if } \pi \text{ is an } A\text{-path.} \end{cases} \tag{1}$$

As stated in Sec. II, it is also possible to compute a dual interpolant for A as the negation of an interpolant for B. To compute the dual interpolant we similarly collect the B-factors that prove $(x = y)$ in G^C, implied by their A-premise set. The A-premise set $\mathcal{A}$ for a path π is defined as

$$\mathcal{A}(\pi) := \begin{cases} \bigcup\{\mathcal{A}(\sigma) | \sigma \text{ is a factor of } \pi\}, \text{if } \pi \text{ has } \geq 2 \text{ factors;} \\ \{\pi\}, \text{if } \pi \text{ is an } A\text{-path; and} \\ \bigcup\{\mathcal{A}(\sigma) | \sigma \text{ is a parent path of an edge of } \pi\}, \\ \quad \text{if } \pi \text{ is a } B\text{-path.} \end{cases} \tag{2}$$

We extend the notation of $\mathcal{A}$ and $\mathcal{B}$ over a set S of paths as $\mathcal{A}(S) := \bigcup_{\sigma \in S} \mathcal{A}(\sigma)$ and $\mathcal{B}(S) := \bigcup_{\sigma \in S} \mathcal{B}(\sigma)$. We also write $\mathcal{AB}(\pi) = \mathcal{A}(\mathcal{B}(\pi))$ etc. to denote compositions of operators. The functions J_A and J_B give, respectively, the contribution of an individual A-factor and an individual B-factor to the interpolants.

$$J_A(\pi) := [\![\mathcal{B}(\pi)]\!] \rightarrow [\![\pi]\!] \tag{3}$$

$$J_B(\pi) := [\![\mathcal{A}(\pi)]\!] \rightarrow [\![\pi]\!] \tag{4}$$

Let S be a set of factors. $S|_\nu$ is the subset of S containing the factors σ such that $L(\sigma) = \nu$ for $\nu \in \{s, w\}$. Let (A, B) be an EUF interpolation instance, G the corresponding congruence graph, and $x \neq y \in A \cup B$ that is in conflict with G. Let $P = (A, B, G[\overline{xy}]^C)$. The algorithm $\mathit{Itp}_L(P)$ computes the EUF interpolant over A for a path $\overline{xy}$. It is defined using four sub-procedures I_A, I'_A, I_B, and I'_B that map congruence graphs to partial interpolants, and are invoked depending on which partition the conflict $x \neq y$ belongs to and what label the path $\overline{xy}$ has:

$$\mathit{Itp}_L(P) := \begin{cases} I_A(\overline{xy}) & \text{if } (x \neq y) \in B \text{ and } L(\overline{xy}) = s, \\ I'_A(\overline{xy}) & \text{if } (x \neq y) \in A \text{ and } L(\overline{xy}) = s, \\ \neg I_B(\overline{xy}) & \text{if } (x \neq y) \in A \text{ and } L(\overline{xy}) = w, \text{ and} \\ \neg I'_B(\overline{xy}) & \text{if } (x \neq y) \in B \text{ and } L(\overline{xy}) = w. \end{cases} \tag{5}$$

The sub-procedures for I_A and I_B are defined as

$$I_A(\pi) := \bigwedge_{\sigma \in \mathcal{A}(\pi)} J_A(\sigma) \wedge \bigwedge_{\sigma \in \mathcal{BA}(\pi)|_s} I_A(\sigma) \wedge \bigwedge_{\sigma \in \mathcal{BA}(\pi)|_w} \neg I'_B(\sigma) \tag{6}$$

and

$$I_B(\pi) := \bigwedge_{\sigma \in \mathcal{B}(\pi)} J_B(\sigma) \wedge \bigwedge_{\sigma \in \mathcal{AB}(\pi)|_w} I_B(\sigma) \wedge \bigwedge_{\sigma \in \mathcal{AB}(\pi)|_s} \neg I'_A(\sigma). \tag{7}$$

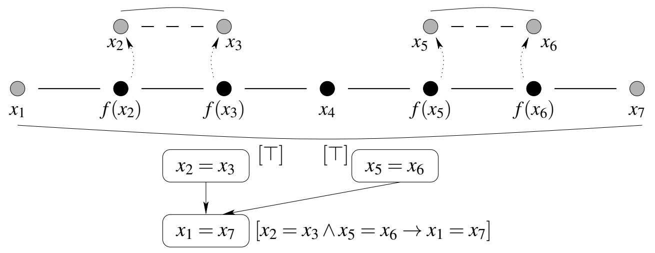


Figure 2. Computing partial interpolants for the EUF-interpolation system.

For the cases where either the conflict $x \neq y \in A$ and $L(\overline{xy}) = s$, or the conflict $x \neq y \in B$ and $L(\overline{xy}) = w$, the path $\overline{xy} = \pi$ needs to be decomposed for computing the partial interpolant as $\pi_1\theta_b\pi_2$ or $\pi_1\theta_a\pi_2$, where θ_κ is the longest subpath of π with κ-colorable endpoints. Hence, I'_A and I'_B are

$$\begin{aligned} I'_A(\pi) := I_A(\theta_b) \wedge & \bigwedge_{\sigma \in \mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)} I_A(\sigma) \\ & \wedge ([\![\mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)]\!] \rightarrow \neg[\![\theta_b]\!]), \end{aligned} \tag{8}$$

and

$$\begin{aligned} I'_B(\pi) := I_B(\theta_a) \wedge (& \bigwedge_{\sigma \in \mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)} I_B(\sigma)) \\ & \wedge([\![\mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)]\!] \rightarrow \neg[\![\theta_a]\!]). \end{aligned} \tag{9}$$

Theorem 2: Given two sets of equalities and disequalities A and B such that $A \cup B$ is unsatisfiable, a colored congruence graph G^C containing a path $\pi := \overline{xy}$ such that $(x \neq y) \in A \cup B$, and a labeling function L, Eq. (5) computes a valid interpolant for A using L over G^C.

The following example shows how Eq. (5) can be used to compute the interpolants from [17].

Example 2: Let $A := \{(x_1 = f(x_2)), (f(x_3) = x_4), (x_4 = f(x_5)), (f(x_6) = x_7)\}$ and $B := \{(x_2 = x_3), (x_5 = x_6), (x_1 \neq x_7)\}$. Figure 2 shows a possible congruence graph G^C that proves the joint unsatisfiability of A and B by proving $(x_1 = x_7)$ such that $(x_1 \neq x_7) \in A \cup B$. We denote the proof as a tree with each node annotated by its partial interpolant. In this example we use the constant labeling function $L_s = s$. From Eq. (5) we have that $\mathit{Itp}(\overline{x_1x_7}) = I_A(\overline{x_1x_7})$, because $L_s(\overline{x_1x_7}) = s$ and $(x_1 \neq x_7) \in B$. The call to $I_A(\overline{x_1x_7})$ is represented by the root node in the tree in Fig. 2. First we compute $\mathcal{A}(\overline{x_1x_7}) = \{\overline{x_1x_7}\}$ and $\mathcal{BA}(\overline{x_1x_7}) = \{\overline{x_2x_3}, \overline{x_5x_6}\}$. Then from Eq. (6) we have that $I_A(\overline{x_1x_7}) = J_A(\overline{x_1x_7}) \wedge I_A(\overline{x_2x_3}) \wedge I_A(\overline{x_5x_6})$. The calls to $I_A(\overline{x_2x_3})$ and $I_A(\overline{x_5x_6})$ are represented by the edges from the leaf nodes to the root in the tree in Fig. 2. We then proceed computing $\mathcal{A}(\overline{x_2x_3}) = \emptyset$ and $\mathcal{BA}(\overline{x_2x_3}) = \emptyset$ which lead to $I_A(\overline{x_2x_3}) = \top$; and $\mathcal{A}(\overline{x_5x_6}) = \emptyset$ and $\mathcal{BA}(\overline{x_5x_6}) = \emptyset$ which lead to $I_A(\overline{x_5x_6}) = \top$, the partial interpolants of the leaf nodes. Finally we have that $I_A(\overline{x_1x_7}) = ((x_2 = x_3) \wedge (x_5 = x_6)) \rightarrow (x_1 = x_7)$ is the partial interpolant of the root node, representing the final interpolant for A.

A. The Interpolant Strength

Let $P = (A, B, G[\pi]^C)$ and L_s and L_w the strong and the weak labeling functions. We will show in Th. 3 that

$Itp_{L_s}(P) \to Itp_{L_w}(P)$, and then in Ex. 3 that there are cases where the strength relation is strict in the sense that there are models that satisfy $Itp_{L_w}(P)$ but do not satisfy $Itp_{L_s}(P)$. Theorem 3 needs Lemma 4 which in turn is a generalization of Lemma 2. We then show our main result on EUF in Theorem 4 on comparing the strength of interpolants based on the labeling functions used.

Lemma 2: Let G^C be a congruence graph with coloring C, and ω a factor from G. Then $I_A(\omega) \wedge I_B(\omega) \to [\![\omega]\!]$.

Lemma 3: Let π be an arbitrary path in the congruence graph, and $\phi(\pi)$ the set of all factors in π. Then $I_A(\pi) = \bigwedge_{\sigma\in\phi(\pi)} I_A(\sigma)$ and $I_B(\pi) = \bigwedge_{\sigma\in\phi(\pi)} I_B(\sigma)$.

Lemma 4: Lemma 2 holds when ω is a path containing multiple factors.

Theorem 3: For fixed A, B, and $G[\overline{xy}]^C$, for the corresponding interpolants defined in Eq. (5) it holds that $Itp_{L_s}(A, B, G[\overline{xy}]^C) \to Itp_{L_w}(A, B, G[\overline{xy}]^C)$.

We demonstrate that the implication is not trivial in general by constructing three different labeling functions for the congruence graph from Ex. 1 that result in three pairwise unequal interpolants.

Example 3: Consider again the sets A and B and the congruence graph G^C from Ex. 1 and Fig. 1. Let L_c be a custom labeling function mapping the paths to labels as $\{\overline{x_1x_2} \mapsto s, \overline{x_1v_1} \mapsto s, \overline{v_1v_2} \mapsto s, \overline{v_2x_2} \mapsto s, \overline{y_1t_1} \mapsto w, \overline{t_1t_2} \mapsto w, \overline{t_2y_2} \mapsto w, \overline{z_1s_1} \mapsto w, \overline{s_1s_2} \mapsto w, \overline{s_2z_2} \mapsto w, \overline{r_1u_1} \mapsto w, \overline{u_1u_2} \mapsto w, \overline{u_2r_2} \mapsto w\}$. We recall that the labeling function only needs to be defined on the factors and the path that contradicts the original disequality, in this case $\overline{x_1x_2}$. The labels are shown over curves representing which path is being labeled. The labeling function L_c represents the intent of generating stronger partial interpolants closer to $(x_1 = x_2)$, and weaker partial interpolants in the inner explanations. Let Itp_s, Itp_w and Itp_c be, respectively, the interpolants generated by Eq. (5) by using the labeling functions L_s, L_w and L_c. The computed interpolants are $Itp_s = ((t_1 = t_2) \to (v_1 = v_2)) \wedge ((u_1 = u_2) \to (s_1 = s_2))$, $Itp_w = \neg((u_1 = u_2) \wedge ((s_1 = s_2) \to (t_1 = t_2)) \wedge \neg(v_1 = v_2))$, and $Itp_c = ((t_1 = t_2) \to (v_1 = v_2)) \wedge \neg(((s_1 = s_2) \to (t_1 = t_2)) \wedge (u_1 = u_2) \wedge \neg(t_1 = t_2))$. The reader is welcome to verify that $Itp_s \to Itp_c \to Itp_w$, and none of them is equivalent to another.

Finally we present our main result providing a way to partially order interpolation algorithms into a lattice based on their strength. From this follows that the constant labeling functions L_s and L_w give, respectively, the strongest and the weakest interpolants within this framework.

Theorem 4: Let $\sqsupseteq$ be a strength relation defined over the labels s and w such that $s \sqsupseteq s$, $w \sqsupseteq w$ and $s \sqsupseteq w$. Let (A, B) be an interpolation instance, G^C a congruence graph proving the unsatisfiability of $A \wedge B$, and L and L' two labeling functions such that $L(\sigma) \sqsupseteq L'(\sigma)$ for all the factors σ of G^C. Then $Itp_L(A, B, G^C) \to Itp_{L'}(A, B, G^C)$.

B. Interpolant Size

The EUF-interpolation system presented above introduces a way of computing interpolants of different strength by labeling the factors of a congruence graph as s or w, depending on the required strength. Each labeling function results potentially in a different interpolant, and creating meaningful labeling functions is a challenging task on its own. For the labeling functions L_s and L_w we give the following results with respect to their size.

Theorem 5: Let $P = (A, B, G[\pi]^C)$. The interpolant with the smallest number of equality occurrences over all interpolants computable with the EUF interpolation system is $Itp_{L_s}(P)$ if $\pi \in B$ and $Itp_{L_w}(P)$ if $\pi \in A$.

IV. Experiments

We integrated the EUF interpolation system together with propositional interpolation to the OpenSMT2 solver and HiFrog, an interpolation-based incremental model checker for C [6], [28]. We report experiments in two different settings in the implementation: running the approach (i) integrated in HiFrog; and (ii) over unsatisfiable EUF benchmarks from SMT-LIB (i.e., the QF_UF benchmarks). The benchmarks and the software are available at http://verify.inf.usi.ch/euf-interpolation. Before describing the experiments we give a concise explanation on how EUF and propositional interpolation are integrated.

A. Integration of Propositional and EUF Interpolation.

An SMT solver takes as input a propositional formula where some atoms are interpreted over the theory of equalities over uninterpreted functions. If a satisfying truth assignment for the propositional structure is found, a theory solver is queried to determine the consistency of its equalities. In case of inconsistency the theory solver adds a reason-entailing clause to the propositional structure. The process ends when either a theory-consistent truth assignment is found or the propositional structure becomes unsatisfiable. The SMT framework provides a natural integration for the theory and propositional interpolants. The clauses provided by the theory solver are annotated with their theory interpolant and are used as partial interpolants in the propositional interpolation system (see, e.g., [15]). Similar to EUF, the propositional interpolation algorithms control the strength of the resulting interpolant by choosing the partition for the shared variables through labeling [15]. The labeling has to be followed then by the theory interpolation algorithm to preserve interpolant soundness. In the following experiments we use instances of the propositional labeled interpolation system [29], [15] supported by OpenSMT2, and in particular the McMillan's algorithms M_s and M_w [7], the Pudlák's algorithm P [30], and the proof-sensitive algorithms PS, PS_s, and PS_w [15] that use the proof structure to optimize the labeling. Fig. 4 shows the algorithms ordered with respect to the logical strength of the interpolants they compute.

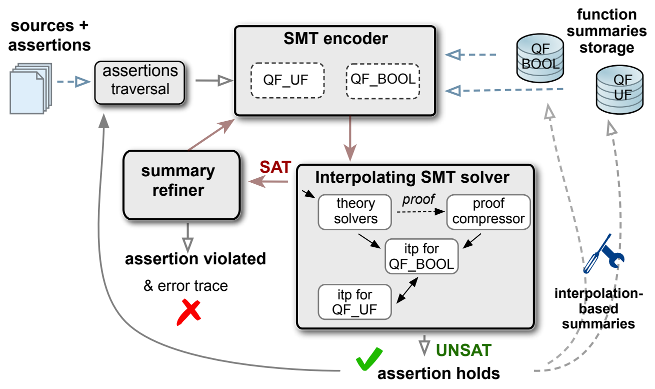


Figure 3. HiFrog overview

B. Interpolation-Based Incremental Verification

We integrated the EUF-interpolation system with the incremental model checker HiFrog as part of OpenSMT2, and used it to verify a set of C benchmarks from SV-COMP (https://sv-comp.sosy-lab.org/) and other sources. In total we checked 973 verification conditions.

We use both purely propositional logic and QF_UF to model the programs. The incremental C model checker HiFrog attempts to prove or refute the validity of a sequence of verification conditions using an SMT solver and an encoding in EUF or in bit-precise propositional logic. Figure 3 shows HiFrog's verification flow; see [28] for a more detailed description on function refinement. The problem instance is pre-processed and encoded into an SMT instance. An SMT solver computes whether the assertion holds by determining the satisfiability of the instance. If the instance is unsatisfiable, the assertion holds, and interpolation is used to extract function summaries from the proof. These summaries are then stored and used in lieu of the precise encoding of a function to incrementally verify the consequent assertions. If the instance is satisfiable, the witnessing truth assignment corresponds to an execution violating the assertion. However, due to the over-approximative nature of both EUF and the function summaries, the execution might be spurious. In this case the model checker uses the precise encoding instead of the summaries to decide the correct answer.

Table I overviews of our results. The numbers in parentheses after the names report the number of assertions in the instance. The table shows the verification time for HiFrog with propositional logic in the column *Bool*; and with EUF in the columns marked *EUF Time*. Unlike the bit-precise propositional model, the EUF model provides an over-approximation of the program behavior. If HiFrog reports that a safety property is true under EUF it is also true for the propositional model. However, if a property is reported false, it may indicate either a real or a spurious counterexample introduced by the EUF abstraction. In the spurious case the model checker should, for instance, consult the propositional encoding. The three columns under the label *EUF Results* list, from left to right, the number of correctly identified assertions using EUF encoding, the number of reachable assertions, and how many of the reachable assertions were spurious. The table reports run times for three variations

Table I
SUMMARY OF VERIFICATION RESULTS ON A SET OF C BENCHMARKS.

	EUF Results				EUF Time (s)		
Name (asrts)	**Corr**	**SAT**	**Sp**	**Bool**	**EUF**	**Sp**	**Full**
floppy1 (18)	15	3	3	69.6	8.3	34.7	34.7
floppy2 (21)	18	3	3	192.1	46.7	122.5	122.5
kbfiltr1 (10)	10	0	0	4.1	1.3	1.3	1.3
diskperf1 (14)	11	3	3	193.7	20.5	67.8	67.8
floppy3 (19)	16	4	3	76.2	9.6	36.4	43.7
kbfiltr2 (13)	13	0	0	10.2	3.0	3.1	3.1
floppy4 (22)	19	4	3	207.3	46.7	127.9	144.1
kbfiltr3 (14)	14	1	0	18.7	5.7	5.6	14.6
tcas_asrt (162)	149	145	13	86.0	16.7	21.6	100.0
cafe (115)	100	100	15	19.2	4.2	5.8	14.7
s3 (131)	123	112	8	1.5	1.7	1.8	3.0
mem (149)	146	52	3	44.6	59.9	60.0	78.5
ddv (152)	56	105	96	260.3	11.2	122.0	122.9
token (54)	54	20	0	962.3	150.6	150.6	998.6
disk (79)	62	72	17	8195.0	237.6	638.2	8151.2
total (973)	806	624	167	10340.8	623.7	1399.3	9900.7

Table II
INTERPOLATION ALGORITHM COMPARISON ON A SET OF C BENCHMARKS.

	$M_s + Itp_s$		$M_s + Itp_w$		$M_w + Itp_s$		$M_w + Itp_w$	
Name	**t**	**refs**	**t**	**refs**	**t**	**refs**	**t**	**refs**
floppy1	10.9	28672	9.8	27648	**8.3**	**24320**	12.7	32256
floppy2	58.9	**37120**	64.8	41216	**46.7**	37632	59.6	40704
kbfiltr1	1.5	**4864**	1.5	**4864**	**1.3**	**4864**	1.5	**4864**
diskperf1	30.1	45568	**20.5**	**44544**	29.7	47104	26.0	48384
floppy3	**9.6**	28928	13.6	34304	**9.6**	**26624**	10.8	29952
kbfiltr2	**3.0**	**4864**	3.1	**4864**	3.1	**4864**	**3.0**	**4864**
floppy4	57.2	41472	**46.7**	43008	48.6	**40704**	58.6	43776
kbfiltr3	**5.7**	**10240**	6.5	10496	5.6	**10240**	6.3	10496
tcas_asrt	17.2	**59648**	16.8	60160	**16.7**	**59648**	17.3	60160
cafe	**4.2**	**6656**	4.3	**6656**	**4.2**	**6656**	4.3	6656
s3	1.7	**0**	1.7	**0**	**1.6**	**0**	**1.6**	**0**
mem	60.1	**23808**	**59.9**	25088	60.7	**23808**	60.1	25088
ddv	**11.2**	**7936**	11.6	**7936**	11.6	**7936**	11.7	**7936**
token	151.4	15616	151.0	**13568**	152.8	15616	**150.6**	**13568**
disk	**237.6**	**9472**	241.3	38912	240.4	**9472**	246.4	38912
Total	660.3	324684	653.1	363264	**640.9**	**319488**	670.5	367616

of the model checker. Column *EUF* reports the time used only by the EUF check. Column *Sp* reports the time when HiFrog is allowed to query the spuriousness of the counter-example from an oracle (see [31] for heuristics for implementing such an oracle) and only needs to consult the propositional encoding if the answer is yes. Column *Full* reports the time when HiFrog needs to resort to the propositional encoding always in case of a failure to verify. Notably the use of EUF as an abstraction technique usually speeds up the solving even in the case of the full overhead.

Finally we report the effect of interpolation algorithm strength to the number of required refinements and the run time for the four combinations M_s+Itp_s, M_s+Itp_w, M_w+Itp_s and $M_w + Itp_w$ in Table II. The number of summary refinements varies sometimes considerably over the combinations, demonstrating the advantage of the flexibility our framework provides for the EUF-interpolation. The number of summary refinement shows the total number of function summaries that were used in whole verification process, did not work, and were replaced by precise encoding of functions, hence the smaller number is,

$$\begin{tikzcd}
M_s \arrow[r, phantom, "\leq"] & PS_s \arrow[r, phantom, "\leq"] \arrow[dr, phantom, "\leq"] & PS \arrow[r, phantom, "\leq"] & PS_w \arrow[r, phantom, "\leq"] & M_w \\
 & & \mathrm{P} \arrow[ur, phantom, "\leq"] & &
\end{tikzcd}$$

Figure 4. The relative strength of the propositional interpolation algorithms [15].

the more efficient is the solving process. The best-performing algorithm in this benchmark set is $M_w + Itp_s$ with both lowest total run time and the lowest total number of refinements. We note that the run time and the number of refinements do not always correlate, and that in particular the combination $M_s + Itp_s$ works very well with respect to refinements while losing nevertheless clearly in total run time. Finally, the worst approach has 15% more refinements and 5% higher run-time compared to the best approach.

Our experiments show two main results. First, using EUF to represent software instead of only Boolean formulas is beneficial, and leads to an impressive speed up in verification time. Second, it is possible to obtain further speed-up by fine tuning the interpolation algorithms used for Boolean and EUF interpolation in order to ultimately optimize convergence in the model checker.

C. Interpolation over SMT-LIB Benchmarks

We also report a more controlled set of experiments on generating interpolants of different strength and size. We computed interpolants from over 2000 benchmarks from the QF_UF category of SMT-LIB, and report here the results of 106 benchmarks that resulted in non-trivial interpolation instances having complex EUF proofs with large congruence graphs. In total this set contains over two and a half million individual EUF interpolants. Following [17], [32], we randomly split the assertions in each benchmark to partitions A and B.

a) Logical strength: The theory interpolation algorithms use three labeling functions L_s, L_w (see Sec. III), and L_r, a labeling function that labels all components randomly as either s or w. The algorithms are called, respectively, Itp_s, Itp_w, and Itp_r. We use the proof-sensitive interpolation algorithm [15] in the propositional structure. This results in three final interpolants I_s, I_w and I_r for each benchmark.

We computed the strength relationship for each theory partial interpolant as well as the final SMT interpolants. Even though the EUF interpolants are often simple, in 71% of them it was possible to generate at least two interpolants of different strength, and 5.7% resulted in all three having different strength.

After solving and interpolating, we ran extra experiments to check the strength relations of the final interpolants I_s, I_w and I_r. Since the final interpolants are much more complex, of the 106 benchmarks, 55 ran out of memory while computing the strength relations. For the remaining 51, all the three final interpolants were pairwise inequivalent, confirming that the framework is able to generate interpolants of different strength.

b) Interpolant size: Since the propositional and EUF interpolation algorithms are to a large degree independent, it is natural to ask what combination of the algorithms is

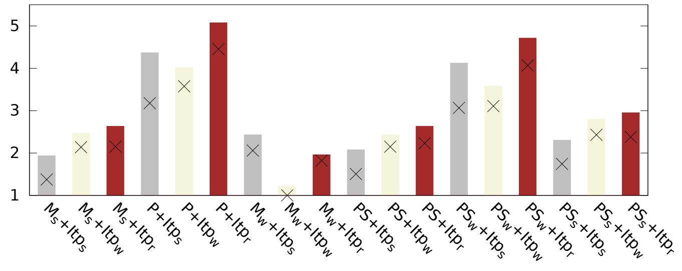


Figure 5. Comparison between interpolation combinations with respect to the number of Boolean connectives in the final interpolant

most efficient. This experiment studies the question using the interpolant size as a measure of efficiency. The six propositional and three EUF interpolation algorithms result in 18 combinations. We measure the sizes of the final interpolants both in *(i)* the number of Boolean connectives (Fig. 5); and *(ii)* the number of EUF equalities (Fig. 6). Excluding the instances where we encountered memory outs we report the results on 82 of the original 106 benchmarks. For each benchmark, we computed the smallest number of Boolean connectives or equalities in the interpolant among all the configurations (*best*) and the ratio *combination/best* for each possible combination, which shows us how much worse each combination did compared to the best combination for that benchmark. Notice that the ratio of the best combination for a benchmark is one and therefore no ratio can be less than one. The bars present the average and the crosses the median of those ratios among all the benchmarks for each combination.

In Fig. 5 the combination $M_w + Itp_w$ gives the smallest number of Boolean connectives, and $M_s + Itp_s$ appears in the second place. The median of $M_w + Itp_w$ is 1, which means that it was responsible for the smallest number of connectives in at least half of the benchmarks, and its average of 1.2 shows that even when this was not the case, the combination was still close to the optimum. On the losing side, we make two observations. The EUF interpolation algorithm Itp_r leads to a larger number of Boolean connectives, and the propositional interpolation algorithm P leads to larger interpolants.

Interestingly the combinations $PS + Itp_s$ and $PS_s + Itp_s$ have low medians and averages. This seemingly contradicts our earlier observation in [15] that PS and PS_s consistently lead to small number of connectives in the interpolant. The likely reason is the soundness restriction in integration (see Sec. IV-A), since the results gradually worsen as the proposi-

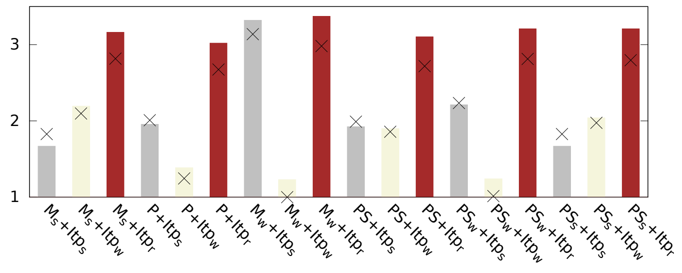


Figure 6. Comparison between interpolation combinations with respect to the number of equalities in the final interpolant

tional and the EUF interpolation algorithms disagree more on the labeling, best being $PS_s + Itp_s$ and the worst $PS_w + Itp_s$.

The same trend is seen in Fig. 6 in the number of EUF equalities. A strong propositional interpolation algorithm (M_s, PS_s) combined with Itp_s leads to smaller interpolants compared to their combination with Itp_w; and a weak propositional interpolation algorithm (M_w, PS_w) combined with Itp_w leads to smaller interpolants compared to their combination with Itp_s. Interestingly PS, a propositional interpolation algorithm that tends to balance the distribution of variables [15], leads to very similar results when combined with Itp_s and Itp_w.

Our experiments with interpolation over complex SMT benchmarks show that the interpolants generated by the EUF system presented in this work indeed have strictly different logical strength. Moreover, in the combination of Boolean and EUF interpolants, it is important to match the strength of the used interpolation algorithms in order to reduce the size of the generated interpolants.

V. Conclusions

We present and analyse a new interpolation framework for the theory of Equalities and Uninterpreted Functions, capable of generating interpolants of different strength and small size in a controlled way. The technique bases on the use of dual partial interpolants parameterized by a labeling function. We confirm the analysis with experiments and show the feasibility of generating multiple interpolants of different strengths. In addition, we report on the size of the created interpolants, comparing different combinations of propositional and EUF interpolation algorithms. Our major contribution work is the integration of a complete interpolation-based model checker to the system, and showing the significant impact the interpolant strength has on both run time and convergence.

In the future we intend to generalize the approach to be applicable to other theories, and study the effects of different labeling functions on fix-point computation in other model-checking applications.

Acknowledgements. This work was financially supported by SNF projects number 200020_163001 and 200020_166288.

References

[1] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: A theorem prover for program checking,” *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[2] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, “A decision procedure for an extensional theory of arrays,” in *Proc. LICS 2001*. IEEE, 2001, pp. 29–37.

[3] B. Godlin and O. Strichman, “Regression verification: proving the equivalence of similar programs,” *Softw. Test., Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, 2013.

[4] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, “A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems,” in *Proc. CAV 2007*, ser. LNCS, vol. 4590. Springer, 2007, pp. 547 – 560.

[5] L. Hadarean, K. Bansal, D. Jovanović, C. Barret, and C. Tinelli, “A tale of two solvers: Eager and lazy approaches to bit-vectors,” in *Proc. CAV 2014*, ser. LNCS. Springer, 2014, pp. 680 – 695.

[6] L. Alt, S. Asadi, H. Chockler, K. E. Mendoza, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, “HiFrog: SMT-based function summarization for software verification,” in *Proc. TACAS 2017*, ser. LNCS, vol. 10206. Springer, 2017, pp. 207–213.

[7] K. L. McMillan, “Interpolation and SAT-based model checking,” in *Proc. CAV 2003*, ser. LNCS, vol. 2725. Springer, 2003, pp. 1–13.

[8] A. R. Bradley, “SAT-based model checking without unrolling,” in *Proc. VMCAI 2011*, ser. LNCS, vol. 6538. Springer, 2011, pp. 70–87.

[9] D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *Proc. CAV 2011*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190.

[10] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn verification framework,” in *Proc. CAV 2015*, ser. LNCS, vol. 9206. Springer, 2015, pp. 343–361.

[11] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, “OpenSMT2: An SMT solver for multi-core and cloud computing,” in *Proc. SAT 2016*, ser. LNCS, vol. 9710. Springer, 2016, pp. 547–553.

[12] V. D’Silva, “Propositional interpolation and abstract interpretation,” in *Proc. ESOP 2010*, ser. LNCS, vol. 6012. Springer, 2010, pp. 185–204.

[13] S. F. Rollini, O. Sery, and N. Sharygina, “Leveraging interpolant strength in model checking,” in *Proc. CAV 2012*, ser. LNCS, vol. 7358. Springer, 2012, pp. 193–209.

[14] S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, “PeRIPLO: A framework for producing effective interpolants in sat-based software verification,” in *Proc. LPAR 2013*, ser. LNCS, vol. 8312. Springer, 2013, pp. 683–693.

[15] L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, “A proof-sensitive approach for small propositional interpolants,” in *Proc. VSTTE 2015*, ser. LNCS, vol. 9593. Springer, 2016, pp. 1–18.

[16] K. L. McMillan, “An interpolating theorem prover,” *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.

[17] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli, “Ground interpolation for the theory of equality,” *Logical Methods in Computer Science*, vol. 8, no. 1, 2012.

[18] G. Weissenbacher, “Interpolant strength revisited,” in *Proc. SAT 2012*, ser. LNCS, vol. 7317. Springer, 2012, pp. 312–326.

[19] L. Kovács, S. F. Rollini, and N. Sharygina, “A parametric interpolation framework for first-order theories,” in *Proc. MICAI 2013*, ser. LNCS, vol. 8265. Springer, 2013, pp. 24–40.

[20] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, “An interpolating sequent calculus for quantifier-free Presburger arithmetic,” *Journal of Automated Reasoning*, vol. 47, no. 4, pp. 341–367, 2011.

[21] A. Griggio, T. T. H. Le, and R. Sebastiani, “Efficient interpolant generation in satisfiability modulo linear integer arithmetic,” in *Proc. TACAS 2011*, ser. LNCS, vol. 6605. Springer, 2011, pp. 143–157.

[22] A. Rybalchenko and V. Sofronie-Stokkermans, “Constraint solving for interpolation,” in *Proc. VMCAI 2007*, ser. LNCS, vol. 4349. Springer, 2007, pp. 346–362.

[23] A. Albarghouthi and K. L. McMillan, “Beautiful interpolants,” in *Proc. CAV 2013*, ser. LNCS, vol. 8044. Springer, 2013, pp. 313–329.

[24] 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.

[25] S. Gao and D. Zufferey, “Interpolants in nonlinear theories over the reals,” in *Proc. TACAS 2016*, ser. LNCS, vol. 9636. Springer, 2016, pp. 625–641.

[26] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[27] R. Nieuwenhuis and A. Oliveras, “Proof-producing congruence closure,” in *Proc. RTA 2005*, ser. LNCS, vol. 3467. Springer, 2005, pp. 453–468.

[28] O. Sery, G. Fedyukovich, and N. Sharygina, “Interpolation-based function summaries in bounded model checking,” in *Proc. HVC 2011*, ser. LNCS, vol. 7261. Springer, 2012, pp. 160–175.

[29] V. D’Silva, D. Kroening, M. Purandare, and G. Weissenbacher, “Interpolant strength,” in *Proc. VMCAI 2010*, ser. LNCS, vol. 5944. Springer, 2010, pp. 129–145.

[30] P. Pudlák, “Lower bounds for resolution and cutting plane proofs and monotone computations,” *Journal of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.

[31] A. E. J. Hyvärinen, S. Asadi, K. Even-Mendoza, G. Fedyukovich, H. Chockler, and N. Sharygina, “Theory refinement for program verification,” in *Proc. SAT 2017*, ser. LNCS. Springer, 2017, to appear.

[32] A. Cimatti, A. Griggio, and R. Sebastiani, “Efficient interpolant generation in satisfiability modulo theories,” in *Proc. TACAS 2008*, ser. LNCS, vol. 4963. Springer, 2008, pp. 397–412.

Solving Linear Arithmetic with SAT-based Model Checking

Yakir Vizel
Princeton University, USA

Alexander Nadel
Intel Development Center, Haifa, Israel

Sharad Malik
Princeton University, USA

***Abstract*—We present `LIAMC`, a novel decision procedure for (quantifier-free) linear arithmetic over both integers modulo 2^N (LIA$_N$) and integers (LIA).**

There is no need to explain our motivation to design a new efficient decision procedure for the widely used LIA logic. A LIA$_N$ decision procedure can be extremely useful in the context of software (SW) verification. SW verification usually requires to reason about arithmetic constraints over finite integers. To that end, modern SW verification tools commonly use fixed-width bit-vector (BV) solvers. However, BV solvers' efficiency drops dramatically as the width increases. To solve the performance problem, LIA solvers are applied, but they are imprecise as they cannot handle integer overflow. An efficient LIA$_N$ solver would be the ideal solution in this context.

Our decision procedure `LIAMC` is based on a transformation of linear arithmetic into safety verification. We treat integers as unbounded streams of bits over time. More precisely, for each input integer, the least significant bit (LSB) corresponds to time 0 in the corresponding stream, and the k-th bit corresponds to the bit received at time k. `LIAMC` then uses SAT-based model checking (SATMC) to solve the resulting problem. In order to achieve efficiency, `LIAMC` uses two forms of generalization. First, if it finds a formula to be unsatisfiable for width N, it tries to generalize this result for all the widths. Second, if `LIAMC` finds a formula to be satisfiable for width N, it tries to "extend" and thus generalize the assignment to a wider target width.

To evaluate `LIAMC` we used the QF_LIA subset of SMT-COMP'16, and ran two sets of experiments. First, we reinterpreted the QF_LIA over fixed-width bit-vectors of varying widths and compared `LIAMC` in LIA$_N$ mode to both Boolector and Z3. `LIAMC` solved the most satisfiable instances out of the three even for the shortest width 32. Second, we compared `LIAMC` to CVC4 and Z3 on the original QF_LIA benchmarks. `LIAMC` was able to solve many instances that had not been solved by the other solvers.

I. Introduction

Nowadays, Satisfiability Modulo Theory (SMT) [1] solvers for the quantifier-free linear integer arithmetic (LIA) [2] logic are widely used, and have become highly efficient. Despite their efficiency, there is a growing demand for SMT solvers that can efficiently solve quantifier-free linear arithmetic over *integers modulo* 2^N (LIA$_N$). This paper presents a novel decision procedure, `LIAMC`, suitable for solving LIA$_N$ and arbitrary LIA instances. Our motivation for designing a decision procedure for LIA$_N$ originates in software (SW) verification.

Formal verification of SW is one of the main driving forces in SMT research. SW verification usually involves reasoning about arithmetic constraints, and in particular, linear arithmetic constraints over integers modulo 2^N for some $N \in \mathbb{N}$. This is due to the fact that SW uses a finite representation for integers. More precisely, arithmetic operations over integers are interpreted over the ring $\mathbb{Z}/2^N\mathbb{Z}$ ("machine arithmetic") rather than over the ring $\mathbb{Z}$. As a result, efficient bit-precise reasoning is highly desired.

In order to capture the semantics of linear arithmetic over $\mathbb{Z}/2^N\mathbb{Z}$ (LIA$_N$), SMT solvers for the theory of fixed-width bit-vectors (BV solvers) are often used, since LIA$_N$ is a proper subset of QF_BV. BV solvers, however, are not efficient when the bit-vectors are wide. Namely, when the value of 2^N is large (e.g. $N = 512$), solving formulas in LIA$_N$ becomes intractable for BV solvers. This inefficiency is mainly due to the way BV solvers are implemented: in most cases, the formula is reduced to a propositional formula using *bit-blasting*. Therefore, as N increases, so does the complexity of the resulting SAT formula. One way to overcome this inefficiency is by applying a LIA solver. Unlike BV solvers, LIA solvers reason about linear arithmetic over $\mathbb{Z}$. While LIA solvers are more efficient than that of BV solvers for this task, they are less precise. This imprecision comes from the different semantics between LIA and LIA$_N$. Namely, arithmetic operations over $\mathbb{Z}$ cannot result in an "overflow" (i.e. wrap-around). In the context of SW verification, this may lead to unsound results. Hence, an efficient LIA$_N$ solver, as presented in this paper, should be extremely useful for SW verification.

Our novel decision procedure for LIA$_N$ and LIA, `LIAMC`, is based on a reduction of the input formula to a safety verification problem. Namely, a formula φ in either LIA$_N$ or LIA is transformed to a transition system T such that the satisfiability of φ corresponds to whether T is SAFE or UNSAFE. The reduction treats integers as unbounded streams of bits over time. More precisely, for each input integer, the least significant bit (LSB) corresponds to time 0 in the corresponding stream, and the k-th bit corresponds to the bit received at time k. The structure of T captures the constraints between the integer variables that appear in φ. To determine if T is SAFE or UNSAFE, `LIAMC` uses SAT-based model checking (SATMC) [3].

One possible way to reason about T is by using Bounded Model Checking (BMC) [4], an efficient SATMC algorithm that can show T is UNSAFE. Considering our reduction, if BMC finds a counterexample of length N in T (T is UNSAFE), then φ is satisfiable over $\mathbb{Z}/2^N\mathbb{Z}$. If no counterexample of length N exists in T, then φ is unsatisfiable over $\mathbb{Z}/2^N\mathbb{Z}$. This can be used as a decision procedure for LIA$_N$. However, the performance of such an approach is usually not better then that of BV solvers [5]. BMC can either find a counterexample of length N, or prove that counterexample of length N does

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

not exist. In that sense, in the context of LIAMC, it can only reason about LIA$_N$ for a given N. In fact, this approach is somewhat "equivalent" to how modern eager BV solvers are implemented.

Unlike BMC, modern SATMC algorithms [6]–[8] use *generalization* in order to show that no counterexample, of any length, exists, and by that they can prove a transition system is SAFE. LIAMC takes advantage of this generalization mechanism. In case LIAMC finds φ to be unsatisfiable over $\mathbb{Z}/2^k\mathbb{Z}$, SATMC's generalization mechanism is applied to show φ is unsatisfiable over $\mathbb{Z}/2^N\mathbb{Z}$ for every $N > k$ and moreover, unsatisfiable over the integers. For the case a counterexample of length k is found, we have implemented an efficient procedure in LIAMC that tries to extend the counterexample to some target N (where $N > k$) and by that show φ is satisfiable over $\mathbb{Z}/2^N\mathbb{Z}$. In addition, LIAMC can also extend a counterexample over $\mathbb{Z}/2^N\mathbb{Z}$ to a counterexample over $\mathbb{Z}$.

We evaluated our approach on QF_LIA subset of the SMT-COMP'16 benchmark. Since LIAMC can be used for both LIA and LIA$_N$, we used two sets of experiments. For our first set of experiments we translated QF_LIA benchmarks to QF_BV using fixed-width bit-vectors of sizes 32, 64, and 128. We then compared LIAMC to Boolector [9], and Z3 [10]. LIAMC solved *the most satisfiable* instances out of the three, even for a width as low as 32. For our second set of experiments we used the LIA solvers in CVC4 [11] and Z3, and compared LIAMC against them on QF_LIA. Here too, LIAMC was able to solve instances that were not solved by the other solvers.

II. Preliminaries

In this section, we present notations and background that is required for the description of LIAMC.

A. Linear Integer Arithmetic

We consider First Order Logic modulo the theory of quantifier free Linear Arithmetic either over Integers (QF_LIA) or over Integers modulo a constant 2^N (QF_LIA$_N$). In what follows we denote QF_LIA and QF_LIA$_N$ as LIA and LIA$_N$, respectively. The following grammar is used to define this theory:

$$\begin{aligned}\varphi &::= \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathit{term} \bowtie \mathit{term}\\ \mathit{term} &::= c \mid x \mid \mathit{term} + \mathit{term} \mid \mathit{term} - \mathit{term} \mid c \times \mathit{term} \mid\\ &\quad \mathit{ite}(\varphi, \mathit{term}, \mathit{term})\end{aligned}$$

where $\bowtie \in \{=, <, \leq, >, \geq\}$, c and x are a constant symbol and a variable either over $\mathbb{Z}/2^N\mathbb{Z}$ or $\mathbb{Z}$, respectively.

Let φ be a formula in LIA$_N$ (or LIA) over a set of variables $\mathcal{V}$. $\mathcal{V} := \mathcal{V}_I \cup \mathcal{V}_B$ where $\mathcal{V}_I$ and $\mathcal{V}_B$ are the sets of Integer and Boolean variables, respectively. Abusing notation, we write $c \in \varphi$ for a constant integer c appearing in φ. Let $N \in \mathbb{N}$ be a natural number such that $N \geq 2$. We refer to the interpretation of φ over the ring $\mathbb{Z}/2^N\mathbb{Z}$ as $\varphi|_N$. For consistency, we use either φ or $\varphi|_\infty$ as the interpretation of φ over $\mathbb{Z}$.

Note that the semantics of linear arithmetic over $\mathbb{Z}$ and over $\mathbb{Z}/2^N\mathbb{Z}$ is different. This is mainly due to "overflow". As an example, consider the following formula:

$$\varphi := (z = x + y) \wedge (x > 0) \wedge (y > 0) \wedge (z < 0)$$

While this formula is unsatisfiable over the ring $\mathbb{Z}$, it is satisfiable over $\mathbb{Z}/2^N\mathbb{Z}$. For example, for $\mathbb{Z}/4\mathbb{Z} = \{-2, -1, 0, 1\}$ $x = 1, y = 1$ and $z = -2$ is a satisfying assignment.

B. Integers as Bit-Vectors

Integers can be represented using bit-vectors. In this work, we use the 2's complement representation. Given an integer $c \in \mathbb{Z}$, there exists $N > 0$ s.t. for every $k \geq N$, there exists a bit-vector $b = \langle b_{k-1}, \ldots, b_0 \rangle$ of size k, and the following holds:

$$c = -b_{k-1} \cdot 2^{k-1} + \sum_{i=0}^{k-2} b_i \cdot 2^i$$

Note that a constant $c \in \mathbb{Z}/2^N\mathbb{Z}$ for some $N \geq 2$ can be represented by a bit-vector of size N. With abuse of notation, we define the function $\omega : \mathbb{Z} \to \mathbb{N}$ such that:

$$\omega(c) := \begin{cases} 2 & \text{if } c \in \mathbb{Z}/4\mathbb{Z} \\ N & \text{if } c \in \mathbb{Z}/2^N\mathbb{Z} \wedge c \notin \mathbb{Z}/2^{N-1}\mathbb{Z} \end{cases} \tag{1}$$

Note that $\omega(c) \geq 2$ for all $c \in \mathbb{Z}$.

C. Safety Verification

A transition system T is a tuple $(\mathcal{U}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathcal{U}$ is a set of Boolean variables, *Init* and *Bad* are formulas over $\mathcal{U}$ denoting the set of initial states and bad states, respectively, and *Tr* is a formula over $\mathcal{U} \cup \mathcal{U}'$ denoting the transition relation. A state $s \in 2^{\mathcal{U}}$ is said to be reachable in T if and only if there exists $k \geq 0$ and $s_0, s_1, \ldots, s_k$ s.t. $s_0 \in \mathit{Init}$, and $(s_i, s_{i+1}) \in \mathit{Tr}$ for $0 \leq i < k$, and $s = s_k$.

A transition system T is UNSAFE iff there exists a state $s \in \mathit{Bad}$ s.t. s is reachable. The path from $s_0 \in \mathit{Init}$ to $s \in \mathit{Bad}$ is called a *counterexample* (CEX).

A transition system T is SAFE iff all reachable states in T do not satisfy *Bad*. Equivalently, there exists a formula *Inv*, called a *safe inductive invariant*, that satisfies:

$$\mathit{Init}(\mathcal{U}) \Rightarrow \mathit{Inv}(\mathcal{U}) \tag{2}$$

$$\mathit{Inv}(\mathcal{U}) \wedge \mathit{Tr}(\mathcal{U}, \mathcal{U}') \Rightarrow \mathit{Inv}(\mathcal{U}') \tag{3}$$

$$\mathit{Inv}(\mathcal{U}) \Rightarrow \neg\mathit{Bad}(\mathcal{U}) \tag{4}$$

A *safety* verification problem is to decide whether a transition system T is SAFE or UNSAFE.

Note that a transition system can be modeled by a sequential circuit with a single output. In this case, the Boolean variables $\mathcal{U}$ represent registers and primary inputs, *Init* defines the initial values for the registers, and *Bad* defines the logic driving the output.

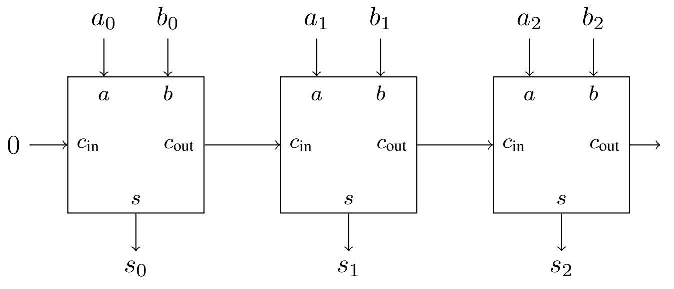


Fig. 1: 3-bit Adder

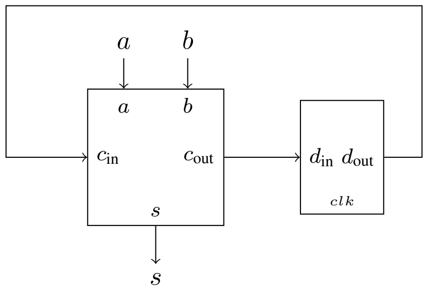


Fig. 2: Sequential Adder

III. Reducing LIA to Safety Verification

In this section we describe the transformation from constraints in LIA and LIA$_N$ to a safety verification problem.

First, we start with an intuitive example. Recall that linear arithmetic includes addition, subtraction and multiplication by a constant. Many arithmetic operations, and the above in particular, can be represented by either a combinational circuit or a sequential circuit. As an example, consider the case of an adder. A N-bit adder can be implemented by a combinational circuit by attaching N copies of a full-adder (Figure 1). Alternatively, it can be implemented by a sequential circuit (Figure 2) such that N bit addition takes N cycles. In the case of the combinational circuit, all bits of the operands must be available simultaneously. As a result, a combinational implementation requires a fixed-width bit-vector representation. In contrast, for the sequential adder, the bits "flow" in, one by one, where at the k-th cycle, only the k-th bit of a given operand is available. As a result the computation takes several cycles. Moreover, there is no restriction on the number of bits it can handle (wider bit-vectors mean more cycles are required to complete the computation). While the combinational implementation is considered more efficient, it may not be the best representation for formal reasoning.

A. LIA to Transition System

From this point on, unless stated otherwise, φ is a formula in either LIA$_N$ or LIA.

Given a formula φ, `LIAMC` reduces φ to a transition system T. The reduction is based on the representation of integers as bit-vectors. While an integer $c \in \mathbb{Z}/2^N\mathbb{Z}$, for some $N \geq 2$, can be represented by a bit-vector of size k for $k \geq N$, this is not the case when considering an arbitrary integer v over $\mathbb{Z}$. As a result, a formula in LIA cannot be represented using fixed-width bit-vectors. To overcome this issue, and considering our intuitive example, we represent input variables in φ (either fixed-width bit-vectors or integers) as inputs to a sequential circuit, and thus, as unbounded bit-vectors. Intuitively, an unbounded bit-vector b is modeled by an unbounded stream of bits, starting from the LSB. More precisely, the bits of b are read over time, such that the k-th bit b_k is available at the k-th time cycle. Representing a constant integer $c \in \mathbb{Z}/2^N\mathbb{Z}$ by a bit-vector of size k, where $k > N$, can be achieved by means of sign extension. Namely, by duplicating the N-th bit for every $k > N$.

Arithmetic constraints and relations in φ are modeled with sequential logic, and logical operators are treated using the corresponding logical gates.

The top level LiaToMc procedure appears in Algorithm 1. LiaToMc transforms a formula φ over variables $\mathcal{V}$, to a transition system T. T is represented by a sequential circuit C. We discuss three different parts of LiaToMc: initialization (lines 1-3), translation of constraints (lines 4-6), and the modeling of the property, i.e. Bad (line 9).

1) Initialization: The main part of initialization is to find the minimal width required to represent constants that appear in φ. Recall that in this work, we use the 2's complement representation. For example, if -3 (101 in binary) and 12 (01100 in binary) appear in φ, then the minimal width is 5. More formally, $k_{min} = \max_{c\in\varphi}\{\omega(c)\}$ (see Equation 1). Now, assume that for a constant $c \in \varphi$, there exists a wire $w_c \in C$ representing it. The value of w_c at a given cycle is determined by the 2's complement representation of c. For example, for -3, $w_c = 1$ at cycle 0 and at cycle 2, and $w_c = 0$ at cycle 1. To achieve this, we create a counter in C (line 3), which counts cycles up to k_{min}. We denote by w_{min} the wire in C that becomes $\top$ once the counter reaches $k_{min} - 1$ and is $\bot$ otherwise (i.e. from 0 to $k_{min} - 1$). For the example above, the counter counts from 0 to 4. Using this counter we can set w_c to the right value at the right cycle. After hitting the maximum value of the counter, w_c is sign-extended. Going back to our example, for every cycle $k > 4$, the value of w_c equals the value it was assigned to at the 4-th cycle.

2) Translating Linear Arithmetic Constraints: The function Translate operates on a Directed Acyclic Graph (DAG) mirroring the structure of φ. Leaf nodes in G represent either a variable (in $\mathcal{V}$) or a constant in φ, while internal nodes represent the different operators. Starting from the root, Translate recursively traverses G, and for each node in G, the proper logic is added to C. Translate appears in Algorithm 2.

Before describing the transformation in more detail, we highlight the handling of the sign bit. The input variables of φ are represented as streams of bits over time, namely, at each cycle, a new bit is added. As a result, at every cycle, the most recent bit is treated as the *sign* bit. Consequently, as the computation progresses, the sign bit is updated.

We now describe the transformation in more detail. W.l.o.g. every node $g \in G$ has at most two operands, a and b. In addition, for simplicity, we assume that $g := ite(c, a, b)$ is modeled by adding a new variable u s.t. $g := u$ and $(c \Rightarrow u = a) \vee (\neg c \Rightarrow u = b)$ is added as a conjunct

Algorithm 1: $\textsc{LiaToMc}(\varphi)$

```
Input: A LIA formula $\varphi$ over variables $\mathcal{V}$
Output: A safety verification problem $(Init, Tr, Bad)$
1  $C \leftarrow$ InitCircuit()
2  $k_{min} \leftarrow$ FindMinWidth($\varphi$)
3  $C$.CreateCounter($k_{min}$)
4  $G \leftarrow$ DAG($\varphi$)
5  $g_{root} \leftarrow G$.Root()
6  Translate($C, G, g_{root}$)
7  $Init \leftarrow C.Init()$
8  $Tr \leftarrow C.Tr()$
9  $Bad \leftarrow w_{min} \wedge C$.Output()
10 $T = (Init, Tr, Bad)$
11 return $T$
```

Algorithm 2: $\textsc{Translate}(C, G, g)$

```
Input: A circuit $C$, DAG $G$ and a node $g$
1  for $h \in g.Operands()$ do
2      if $h$ is undefined in $C$ then
3          Translate($C, G, h$)
4  $C$.CreateLogic($g$)
```

to the formula φ[1]. Once a node is translated, there exists a wire w_g in C that represents it. The logic of a full-adder is represented by $f(a, b, s, c_{in}, c_{out})$, where a and b are the input operands, s is the sum, c_{in} and c_{out} are the carry-in and carry-out, respectively.

Let us assume that for unary and binary operators the operands are a or a and b, respectively, where a and b can be of sort Integer, bit-vectors, or Boolean. The rules below describe the transformation.

- g is a leaf of sort integer/bit-vector: create an input terminal v_g in C. $w_g := v_g$.
- g is a leaf of a constant type (i.e. $c \in \mathbb{Z}$): use the counter to add logic that defines the right values for w_g over time.
- g is a leaf of sort Boolean: create an uninitialized latch v_g such that $v'_g := v_g$ and $w_g := v_g$.
- Boolean operations are implemented using their equivalent logical gates.
- $g := a + b$: add a sequential adder (see Figure 2). A latch v_+ and a full-adder $f(v_a, v_b, s, v_+, c_{out})$ are added. v_+ is defined as follows: $init(v_+) := \bot$ and $v'_+ := f.c_{out}$. $w_g := f.s$ (note that $f.c_{in} := v_+$).
- $g := a - b$: subtraction uses the identity: $x - y \equiv x + \bar{y} + 1$.
- $g := c \cdot a$: multiplication by constant uses the "Shift and Add" identity. Namely, $c \cdot a \equiv \sum_{i=0}^{k} c_i \cdot 2^i \cdot a$.
- The root node of G represents the output of the circuit C.

We describe equality and inequality in more detail.

a) Equality $g := a = b$: The equality operator amounts to bitwise comparison, namely, $a_i = b_i$ for every $i \geq 0$. The sequential implementation of it uses a latch $v_=$ s.t. $w_g := v_= \wedge (v_a = v_b)$, $init(v_=) := \top$ and $v'_= := w_g$. The latch "remembers" the comparison of earlier bits. Note that if at any point in time, the bits are unequal, the value of $v_=$ can never be $\top$ from that point on.

b) Inequality $g := a < b$: This case is more complex since the sign bit changes at each cycle. Therefore, the sequential circuit representing it is built of two parts. The first implements an unsigned comparison, and the second takes care of the sign bit. For the unsigned comparison a latch $v_<$ is added s.t. $init(v_<) := \bot$ and $v'_< := (\neg v_a \wedge v_b) \vee (\neg(v_a \wedge \neg v_b) \wedge v_<)$. The sign is handled by $w_g := \texttt{MUX}(v_<, v_a \vee \neg v_b, v_a \wedge \neg v_b)$

The other comparison operators $\leq, >$ and $\geq$ can naturally be adjusted based on the above reasoning. We therefore refrain from describing them in detail.

Related transformations can be found in [5], [12].

3) Modeling the Property (Bad): Recall that when modeling a transition system with a sequential circuit, the output represents Bad. The above reduction creates a sequential circuit C with an output o. A k-cycle execution of C represents the interpretation of φ over $\mathbb{Z}/2^k\mathbb{Z}$. Therefore, if o is evaluated to $\top$ in k cycles, then φ is satisfiable over $\mathbb{Z}/2^k\mathbb{Z}$.

Note that a k-cycle computation of the circuit is not necessarily well defined for all $k > 1$. The reason for this is the fact that φ includes constant values. Recall that k_{min} represents the minimum bit-vector width required to represent the constants in φ, and that w_{min} indicates when k_{min} cycles of C has been completed. We can therefore use w_{min} as a "guard" when defining Bad. The "guard" disables the output until k_{min}-th cycle.

To complete the reduction from LIA to a transition system, we create a safety verification problem $T = (Init, Tr, Bad)$ where $Init = C.Init()$, $Tr = C.Tr()$ and $Bad := w_{min} \wedge C.Output()$.

B. Naïve Decision Procedure

Before describing LIAMC, let us first provide an intuition. A well known SAT-based verification technique is Bounded Model Checking (BMC) [4]. Given a transition system T, BMC searches for an execution that starts from the initial states (i.e. $Init$) and reaches the bad states (i.e. Bad) s.t. it satisfies the transition relation. This path is called a *counterexample*. To find such a counterexample of length N, BMC generates the following N*-unrolling* formula:

$$\mu(T, N) := Init(U^0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(U^i, U^{i+1}) \right) \wedge Bad(U^N) \tag{5}$$

This formula is then passed to a SAT solver. If it is satisfiable, a counterexample of length N exists. When clear from the context, we omit T and write $\mu(N)$.

Consider again our example in Figure 2. The combinational adder that appears in Figure 1 is a result of unrolling the sequential circuit 3 times.

Recall that the reduction of LIA to a transition system treats integers as streams of bits. Let φ be a LIA formula and let $T = (Init, Tr, Bad)$ be the corresponding transition system.

Proposition 1 *For $N \geq k_{min}$, $\varphi|_N$ and $\mu(T, N)$ are equa-satisfiable.*

[1]Our implementation handles the ite operator directly.

Algorithm 3: `LIAMC` (φ, N)

```
Input: A LIA formula $\varphi$ over variables $V$, a constant $N \in \mathbb{N} \cup \{\infty\}$
Output: sat, unsat or unknown.
1  $T \leftarrow \textsc{LiaToMc}(\varphi)$
2  $\texttt{MC} \leftarrow InitMC(T, N)$
3  repeat
4      $(result, k) \leftarrow \texttt{MC}.Solve()$
5      if $result = SAFE$ then
6          return unsat
7      else if $result = UNSAFE$ then
8          if $k = N$ then
9              return sat
10         $\pi \leftarrow \texttt{MC}.GetCex()$
11         if $Extendable(\varphi, \pi, N)$ then
12             return sat
13         else
14             $\texttt{MC}.BlockCex(\pi)$
15 until $\infty$
16 return unknown
```

Given Proposition 1, BMC can be used to reason about linear arithmetic constraints over fixed-width bit-vectors, namely, over $\mathbb{Z}/2^N\mathbb{Z}$ for some N. It is important to note, in fact, that $\mu(N)$ is similar to a bit-blasted $\varphi|_N$ [5]. Consequently, this approach is, in general, not superior to solving the bit-blasted $\varphi|_N$ with a BV solver [5], since it requires a N-cycles long computation of the sequential circuit.

IV. Decision Procedure for LIA$_N$ and LIA

In this section we describe `LIAMC`, a decision procedure for LIA$_N$ and LIA. In the previous section we show how a formula φ can be reduced to a transition system, and how BMC can be used as a decision procedure. Yet, such a decision procedure is not more efficient than using BV solvers.

In order to achieve efficiency, `LIAMC` relies on the ability of state-of-the-art SATMC algorithms to generalize a bounded proof of correctness into a safe inductive invariant. This generalization proves the absence of a counterexample for any N. Note that this gives another intuitive justification for why the reduction from LIA$_N$ and LIA treats both integers and fixed-width bit-vectors as unbounded streams of bits. In case the SATMC algorithm finds an inductive invariant, there exists k such that $\varphi|_N$ is unsatisfiable for every $N \geq k$.

For the case a counterexample of length k exists in T, we have implemented a procedure that uses the "structure" for T such that it can, iteratively, and incrementally extend the counterexample to a target length N (for LIA$_N$) or extend the counterexample for the integers. The key insight here is to treat a counterexample of length k as a partial assignment.

`LIAMC` appears in Algorithm 3. It can operate in two modes, which are determined by the value of N. If $N = \infty$, then φ is interpreted over $\mathbb{Z}$ (LIA mode), otherwise it is interpreted over $\mathbb{Z}/2^N\mathbb{Z}$ (LIA$_N$ mode). The initialization (lines 1-2) of `LIAMC` starts by transforming φ to a transition system $T = (Init, Tr, Bad)$ and setting up an instance of a model checker MC. Note that `MC` receives N, the maximum time frame it needs to consider. The main loop (lines 3-15) uses a model checker to prove either T is SAFE or UNSAFE.

We assume that `MC`.$Solve()$ (line 4) returns a pair $(result, k)$, where $result$ is either SAFE or UNSAFE. In case $result =$ UNSAFE then k is the length of the counterexample. Otherwise, if $result =$ SAFE k is the depth at which an inductive invariant is found, or if no invariant is found $k = N$ (indicating no counterexample up to N).

We now describe `LIAMC` in more detail. We start by describing the case $\varphi|_N$ is satisfiable, and then the case it is unsatisfiable.

A. Satisfiability: T is UNSAFE (line 7)

Let us assume a counterexample of length k exists. In this case, $\varphi|_k$ is satisfiable. Since the satisfiability of $\varphi|_k$ does not entail the satisfiability of $\varphi|_N$, `LIAMC` checks if the returned counterexample can be "extended" into a counterexample of $\varphi|_N$. In what follows we detail this procedure for both LIA$_N$ and LIA.

1) Extending a Counterexample: As noted above, a counterexample of length k implies that $\varphi|_k$ is satisfiable. Due to the different semantics of LIA$_N$ and LIA$_k$ ($N > k$), the satisfying assignment for $\varphi|_k$ is not necessarily a satisfying assignment for $\varphi|_N$. The naive solution to the above problem, is to check whether the given assignment is also an assignment for $\varphi|_N$ (either when $N = \infty$ or $N < \infty$). This solution amounts to a sign-extension of the satisfying assignment. While it is simple, in most cases it does not work. As an example, consider again the following formula:

$$\varphi := (z = x + y) \wedge (x > 0) \wedge (y > 0) \wedge (z < 0)$$

As noted before, $x = 1, y = 1$ and $z = -2$ is a satisfying assignment for $\varphi|_2$, but it is not a satisfying assignment for $\varphi|_N$ for all $N > 2$.

Given a counterexample π of length k, let us assume we would like to extend it for $\varphi|_{k+1}$. Intuitively, π assigns values for k bits out of $k + 1$. Therefore, if there exists a satisfying assignment π^* to $\pi \wedge \mu(k+1)$ then π^* is an extension of π for $\varphi|_{k+1}$, since it satisfies $\mu(k+1)$.

Using the above intuition, we can iteratively, and *incrementally*, extend a counterexample up to a desired depth N such that $N > k$. This gives us an efficient procedure to determine satisfiability for LIA$_N$ ($N < \infty$).

We now need to handle the case of LIA ($N = \infty$). For LIA, we use the same intuition as above. Namely, a counterexample π of length k gives a valuation to the lower k bits. However, we need to adjust this intuition for integers in $\mathbb{Z}$. Let us assume an integer $v \in Int(\varphi)$ is evaluated to $c_v \in \mathbb{Z}/2^k\mathbb{Z}$ in π. In order to extend it, we can add the following constraint: $(v = v^* \cdot 2^k + |c_v|) \vee (v = -(v^* \cdot 2^k + |c_v|))$ where v^* is a fresh integer variable.

Given a counterexample π of length k, let us define:

$$\Delta(\pi) := \bigwedge_{v \in \mathcal{V}_I} \big((v = v^* \cdot 2^k + |c_v|) \vee (v = -(v^* \cdot 2^k + |c_v|))\big)$$

The function $\Delta(\pi)$ captures the value π assigns to the lower k bits of an integer in φ.

Lemma 1 *If $\varphi \wedge \Delta(\pi)$ is satisfiable, then φ is satisfiable.*

Algorithm 4: `Extendable`(φ, T, π, N)

```
Input: A LIA formula $\varphi$ and its corresponding safety
       verification problem $T = (Init, Tr, Bad)$, a
       counterexample $\pi$ of length $k$, and a constant
       $N > k$, s.t. $N \in \mathbb{N} \cup \{\infty\}$
Output: $(\mathtt{false}, \bot)$ or $(\mathtt{true}, \pi^*)$.
1  if $N = \infty$ then
2      $(result, \pi^*) \leftarrow \mathtt{LIA.IsSAT}(\varphi \wedge \Delta(\pi))$
3      if $result = sat$ then
4          return $(\mathtt{true}, \pi^*)$
5      return $(\mathtt{false}, \bot)$
6  else
7      $m \leftarrow k$
8      $i \leftarrow k + 1$
9      while $i \leq N$ do
10         $(result, \pi^*) \leftarrow \mathtt{IsSAT}(\mu(T, i) \wedge \pi)$
11         if $result = sat$ then
12             $m \leftarrow i$
13         $i \leftarrow i + 1$
14     if $m = N$ then return $(\mathtt{true}, \pi^*)$
15     return $(\mathtt{false}, \bot)$
```

In order to determine satisfiability of $\varphi \wedge \Delta(\pi)$, we use a LIA solver. In case $\varphi \wedge \Delta(\pi)$ is satisfiable, π can be extended to a satisfying assignment for φ. We would like to emphasize that using a LIA solver when $N = \infty$ (LIA mode) is intended to rule out counterexamples that may appear due to overflow. Note that it may be possible to model T s.t. overflow is not possible.

The procedure for extending counterexamples appears in Algorithm 4. In the LIA$_N$ mode, we try and iteratively extend a counterexample of length k to N (lines 9-13). As mentioned above, extending a counterexample in LIA mode requires a call to a LIA solver (line 2). In both cases, a counterexample that cannot be extended is blocked.

Theorem 1 *If `LIAMC` returns `sat`, then $\varphi|_N$ is satisfiable.*

Proof Sketch: Let us assume a counterexample π of length $k < N$ exists. If $N < \infty$, and π can be extended, then a counterexample of length N exists and thus $\varphi|N$ is satisfiable. For $N = \infty$, satisfiability of φ follows from Lemma 1.

■

B. Unsatisfiability: T is SAFE (line 5)

Due to the definition of Bad, the property being verified (i.e. $\neg Bad$) is of the form $w_{min} \Rightarrow o$. Moreover, since the reduction from φ to T uses unbounded bit-vectors, if a safe inductive invariant is found by the model checker, $\varphi|_N$ is unsatisfiable for all $N \geq k$.

Lemma 2 *Let φ be a formula in LIA. If there exists k s.t. for all $N > k$ $\varphi|_N$ is unsatisfiable then φ is unsatisfiable.*

The proof for Lemma 2 relies on the following: if φ is satisfiable, there exists $N \in \mathbb{N}$ s.t. $\varphi|_N$ is satisfiable.

The above lemma gives us a way to determine the unsatisfiability of φ: if T is SAFE and $(N = \infty)$, `LIAMC` only returns SAFE if an inductive invariant is found. In that case, using Lemma 2, `LIAMC` concludes φ is unsatisfiable.

In the case $N < \infty$ (LIA$_N$ mode), If T is SAFE up to bound N, `LIAMC` can terminate concluding $\varphi|_N$ is unsatisfiable even when an inductive invariant is not found. This is due to the fact that if no counterexample exists at depth N, $\varphi|_N$ is unsatisfiable

We do like to emphasize that while there is no requirement to find a safe inductive invariant in case $N < \infty$, if such an invariant is found at bound smaller than N, it implies the unsatisfiability of $\varphi|_N$.

Theorem 2 *If `LIAMC` returns `unsat`, then $\varphi|_N$ is unsatisfiable.*

Proof Sketch: We consider two cases. First, if no counterexample is found during the execution of `LIAMC`, and the model checker returns SAFE, then for $N < \infty$ the proof is immediate, and for $N = \infty$ we use Lemma 2.

The second case occurs when `LIAMC` blocks a counterexample π. It remains to be shown that π cannot be part of a real counterexample. Let us assume π is of length k (and $k < N$).

For the case of $N < \infty$, this is immediate - if π cannot be extended up to N it cannot be part of a real counterexample and therefore can safely be blocked.

For $N = \infty$, a similar logic applies. Let us assume that $\varphi \wedge \Delta(\pi)$ is unsatisfiable. Since every positive integer can be expressed by a sum of powers of 2, $\Delta(\pi)$ fixes only the first k elements of that sum. The rest of the elements in this summation are unrestricted, and therefore, if there exists an assignment for this valuation, the LIA solver finds it. However, if such an assignment does not exist, we can safely block this valuation for the first k elements of the sum.

■

V. Experiments

We implemented a prototype of `LIAMC`[2] using a generic SMT-LIB2 parser[3] and ABC [13]. We use the SMT-LIB2 parser to transform a LIA or LIA$_N$ formula to a transition system represented by an And-Inverter Graph (AIG) in ABC. For the SATMC procedure we use ABC's dprove command.

For evaluation, we use the QF_LIA[4] benchmarks from SMT-COMP'16[5]. Since `LIAMC` targets both LIA and LIA$_N$, we used two sets of experiments. First, we reinterpreted the LIA benchmark over $\mathbb{Z}/2^N\mathbb{Z}$ for $N \in \{32, 64, 128\}$ and evaluated `LIAMC` against Boolector [9] and the Bit-Vector solver in Z3 [10]. Second, we compared the performance of `LIAMC` against the LIA solver in CVC4 [11] and Z3.

We set a 900 seconds time limit for all benchmarks. All experiments were conducted on a machine running Ubuntu

[2]Available at http://www.cs.technion.ac.il/~yvizel/liamc.html

[3]Source code is available from https://es-static.fbk.eu/people/griggio/misc/smtlib2parser.html

[4]Only nec-smt subset was excluded due to its size, as even without this subset our benchmark includes more than 2700 test cases.

[5]Benchmarks are available from http://smtcomp.sourceforge.net/2016/benchmarks.shtml

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE I: Number of solved instances for LIA_N. *Total* stands for the total number of test cases in that benchmark. The difference is due to the fact that not all LIA test cases can be represented in LIA_N for certain values of N.

Benchmark	Total	Status	LIAMC	Boolector	Z3	Virtual Best
LIA_{32} (32bit)	2647	SAT	**1475**	1257	1373	1539
		UNSAT	784	**988**	881	995
LIA_{64} (64bit)	2784	SAT	**1630**	1340	1448	1781
		UNSAT	680	**1017**	889	1023
LIA_{128} (128bit)	2742	SAT	**1565**	1233	1347	1734
		UNSAT	637	**1013**	861	1020

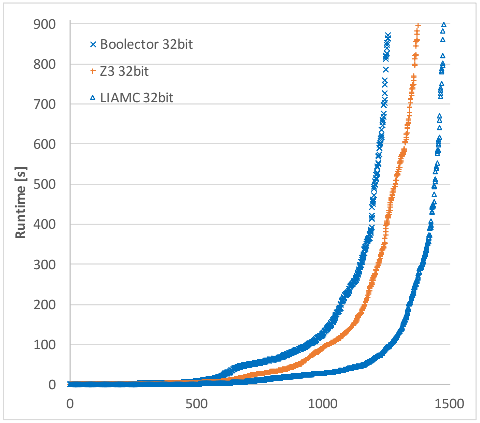


Fig. 3: $\mathbb{Z}/2^{32}\mathbb{Z}$: Trend for satisfiable instances (32 bit).

16.04.2 LTS, with Intel Xeon E3-1240V2 running at 3.4GHz and 32GB of RAM.

Table I shows the number of solved instances for the different experiments of LIA_N. As can be seen from the table, LIAMC has a big advantage specifically on satisfiable instances, for all values of N. LIAMC constructs a satisfying assignment, incrementally, starting from the LSB. We believe this is the main reason for the performance advantage of LIAMC over the other methods. Figures 3-5 further emphasize the performance advantage of LIAMC on satisfiable instances. Moreover, we can see the performance advantage of LIAMC grows as the width of bit-vectors grows.

It is important to note that the approaches are complementary as many test cases are solved by LIAMC and not by Boolector, and vice-versa. Overall, LIAMC solves 205 test cases not solved neither by Boolector nor Z3 for $N = 32$. For $N = 64$ and for $N = 128$, LIAMC solves 288 and 331 test cases that are not solvable by the other solvers. When compared to Boolector, for $N = 32$, LIAMC solves 370 test cases not solved by Boolector, and Boolector solves 331 test cases not solved by LIAMC. For $N = 64$ and $N = 128$, LIAMC solves 427 and 496 test cases not solved by Boolector, while Boolector solves 482 and 501 test cases not solved by LIAMC. In the case of Z3, for $N = 32, 64, 128$, LIAMC solves 324, 329 and 397 cases not solved by Z3, while Z3 solves 265, 337, 349 cases not solved by LIAMC.

When considering unsatisfiable instances, LIAMC finds an inductive invariant in 577 cases. It is important to note that for large values of N (e.g. $N \geq 512$) this fact translates to a clear advantage of LIAMC over the other solvers.

Table II presents solved instances in LIA mode. It compares

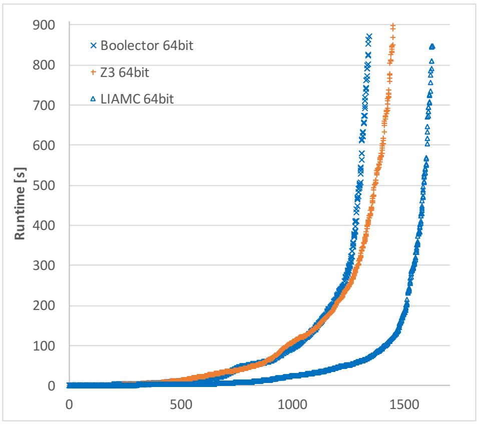


Fig. 4: $\mathbb{Z}/2^{64}\mathbb{Z}$: Trend for satisfiable instances (64 bit).

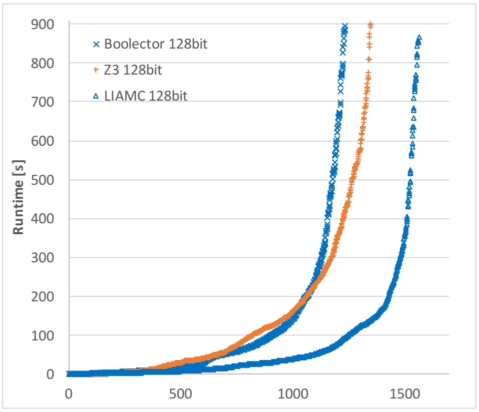


Fig. 5: $\mathbb{Z}/2^{128}\mathbb{Z}$: Trend for satisfiable instances (128 bit).

LIAMC to CVC4 and Z3. The table shows that both CVC4 and Z3 perform better than LIAMC. Analyzing the results shows that LIAMC can solve 88 instances not solved by CVC4, and 126 instances not solved by Z3.

We would like to note that the current results of LIAMC (in both modes) can be greatly improved. While the dprove command in ABC is capable, we are sure that LIAMC can benefit from a portfolio-based model checker, as well as from SATMC algorithms that target the kind of transition systems LIAMC generates. To evaluate this idea, we have chosen a random subset of unsolved instances and used the SATMC algorithm AVY [8] as part of LIAMC. Many of these unsolved instances (UNSAT) were solvable by AVY[6]. Moreover, an efficient BMC engine can probably solve many of the UNSAT cases LIAMC did not solve. This is due to the conceptual similarity between using BMC and an eager BV solver (as mentioned before). We intend to explore these avenues in our future work.

[6]We did not add these solved instances to the results presented in this paper since we had not run AVY on the entire benchmark set.

TABLE II: Number of solved instances for LIA.

	LIAMC	CVC4	Z3
SAT	1289	**1657**	1581
UNSAT	577	**1106**	1103
Uniquely Solved	18	18	0

VI. Related Work

A translation from linear arithmetic to finite automata had been proposed more that 20 years ago [14] and studied further in a number of works [15]–[17]. Our work belongs to that line of research as a finite automata can be thought of as a sequential circuit. The added value of our work is that we present an efficient decision procedure, achieved by leveraging a symbolic representation of the automata (i.e. the sequential circuit) and advancements from modern SATMC research, enhanced by novel generalization techniques. Our results challenge the pessimistic forecast in [16]: "there is little hope that these techniques will consistently outperform more traditional approaches when these can be applied".

Several related methods for synthesizing unbounded bit-vector arithmetic were proposed in [12], [18], but in these works the context is synthesis and no efficient decision procedure was detailed.

A closely related line of work is [5], [19], where a reduction from a fragment of BV (restricted to bitwise operators, addition, subtraction, shift by one, indexing and comparators) to propositional model checking has been introduced (as a by-product of studying the complexity of bit-vector logic). The proposed method has been implemented and shown to outperform traditional SMT solvers on crafted BV benchmarks, restricted to the aforementioned BV fragment. Unlike the transformation applied by LIAMC, the modeling suggested in [19] encodes the width of the bit-vectors into the model checking problem, making SATMC algorithms inefficient. As a result, BDD-based model checking algorithms were found to be the most efficient experimentally [19]. LIAMC shows how SATMC can be applied efficiently even for the subset supported by [19][7] by applying generalization techniques. Generalization is possible for LIAMC since the width of bit-vectors is not encoded in the transition system. In addition, our approach also handles multiplication by a constant, which makes it applicable to arbitrary formulas in LIA$_N$ and LIA.

LIAMC constructs a satisfying assignment incrementally by iteratively extending a satisfying assignment from a simple theory to a more complex one (i.e. from $\mathbb{Z}/2^k\mathbb{Z}$ to $\mathbb{Z}/2^N\mathbb{Z}$ where $N > k$). A somewhat similar concept is applied by [20] in the context of floating-point arithmetic (FPA). In [20], the formula is solved w.r.t. a simpler "proxy" theory. In case that a satisfying assignment is found, it is then tried to be adjusted to FPA semantics.

VII. Conclusion

In this paper we introduced LIAMC, a novel decision procedure for LIA$_N$ and LIA. LIAMC is based on a transformation of linear arithmetic constraints to a transition system. While this approach, in general, has been suggested and explored in the past in different contexts, to our knowledge LIAMC is the first efficient implementation. There are three key insights that make LIAMC efficient and different from previous approaches: 1) We treat both integers and fixed-width bit-vectors as unbounded streams of bits, which allows us to apply SATMC, and 2) We use *generalization* to efficiently reason about wide bit-vectors and integers.

[7]Note that throughout our experiments, the transition systems include more than thousands of state elements, making BDD-based MC intractable.

Our experiments show that LIAMC can solve many instances that cannot be solved by other top-tier SMT solvers, for both LIA$_N$ and LIA. Moreover, in the case of LIA$_N$, LIAMC is the best performer solving the most instances. We therefore believe that this approach has a promising future.

References

[1] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[2] D. Jovanovic and L. M. de Moura, "Cutting to the chase - solving linear integer arithmetic," *J. Autom. Reasoning*, vol. 51, no. 1, pp. 79–108, 2013.

[3] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[5] G. Kovásznai, A. Fröhlich, and A. Biere, "Complexity of fixed-size bit-vector logics," *Theory Comput. Syst.*, vol. 59, no. 2, pp. 323–376, 2016.

[6] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV*, 2003, pp. 1–13.

[7] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[8] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *CAV*, 2014, pp. 260–276.

[9] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009, pp. 174–177.

[10] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, 2008, pp. 337–340.

[11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV*, 2011, pp. 171–177.

[12] A. Spielmann and V. Kuncak, "Synthesis for unbounded bit-vector arithmetic," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012*, 2012, pp. 499–513.

[13] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.

[14] P. Wolper and B. Boigelot, "An automata-theoretic approach to presburger arithmetic constraints (extended abstract)," in *Static Analysis, Second International Symposium, SAS'95*, 1995, pp. 21–32.

[15] A. Boudet and H. Comon, "Diophantine equations, presburger arithmetic and finite automata," in *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, 1996*, 1996, pp. 30–43.

[16] B. Boigelot and P. Wolper, "Representing arithmetic constraints with finite automata: An overview," in *Logic Programming, 18th International Conference, ICLP 2002*, 2002, pp. 1–19.

[17] B. Boigelot, S. Jodogne, and P. Wolper, "An effective decision procedure for linear arithmetic over the integers and reals," *ACM Trans. Comput. Log.*, vol. 6, no. 3, pp. 614–633, 2005.

[18] J. Hamza, B. Jobstmann, and V. Kuncak, "Synthesis for regular specifications over unbounded domains," in *International Conference on Formal Methods in Computer-Aided Design (FMCAD'10)*, 2010, pp. 101–109.

[19] A. B. Andreas Fröhlich, Gergely Kovásznai, "Efficiently solving bit-vector problems using model checkers," in *11th International Workshop on Satisfiability Modulo Theories*, 2013.

[20] J. Ramachandran and T. Wahl, "Integrating proxy theories and numeric model lifting for floating-point arithmetic," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016*, 2016, pp. 153–160.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Z3str3: A String Solver with Theory-aware Heuristics

Murphy Berzish and Vijay Ganesh
University of Waterloo
Waterloo
Canada

Yunhui Zheng
IBM Research
Yorktown Heights
USA

***Abstract*—We present a new string SMT solver, Z3str3, that is faster than its competitors Z3str2, Norn, CVC4, S3, and S3P over a majority of three industrial-strength benchmarks, namely, Kaluza, PISA, and IBM AppScan. Z3str3 supports string equations, linear arithmetic over length function, and regular language membership predicate. The key algorithmic innovation behind the efficiency of Z3str3 is a technique we call theory-aware branching, wherein we modify Z3's branching heuristic to take into account the structure of theory literals to compute branching activities. In the traditional DPLL(T) architecture, the structure of theory literals is hidden from the DPLL(T) SAT solver because of the Boolean abstraction constructed over the input theory formula. By contrast, the theory-aware technique presented in this paper exposes the structure of theory literals to the DPLL(T) SAT solver's branching heuristic, thus enabling it to make much smarter decisions during its search than otherwise. As a consequence, Z3str3 has better performance than its competitors.**

I. Introduction

String SMT solvers are increasingly becoming important for security applications and in the context of analysis of string-intensive programs [6], [9], [11], [15], [16], [18], [22]. Many string SMT solvers, such as Z3str2 [23], [24] (and its predecessor Z3str [25]), CVC4 [12], Norn [2], S3 [20] (and its successor S3P [21]), and Stranger (and its successor ABC [4]) have been developed to address these challenges and applications. We have developed the Z3str3 string solver as a native first-class theory solver directly integrated into the Z3 SMT solver [7]. Z3str3 is the primary string solver in the official Z3 codebase. Our tool is competitive with respect to its predecessor Z3str2 and the CVC4 solver, and much faster than Norn, S3, and S3P. Having direct access to the core solver of Z3 has allowed us to develop and implement novel theory-aware DPLL(T) techniques, described below. We follow the latest string SMT language standard supported by all major string solvers, and published on the CVC4 website [12].

A. Contributions

1) **Theory-aware branching:** We leverage the integration between the Z3 SMT solver's DPLL(T) SAT layer (henceforth referred to as the "core solver") and the string solver to guide the search and prioritize certain branches of the search tree over others. In particular, we modify the activity computations of the branching heuristic of the Z3 core solver, making it aware of the structure of the theory literals underlying the Boolean abstraction of the input formula such that "simpler" theory literals are prioritized over more complex ones. The question of whether branching can be made theory-aware was first posed in a paper by Roberto Sebastiani [17]. However, to the best of our knowledge we are the first to propose a theory-aware branching technique which prioritizes certain branches over others in a DPLL(T) setting.
2) **Theory-aware case-split:** We add an optimization to Z3's core solver that enables efficient representation of mutually exclusive Boolean variables in the Boolean abstraction of the input theory formula.
3) **Experimental evaluation:** To validate the effectiveness of our techniques, we present a comprehensive and thorough evaluation of Z3str3, and compare against Z3str2, CVC4, S3, and Norn on several large industrial-strength benchmarks. We could not directly compare against S3P since its source is not available, but summarize the results from their CAV 2016 paper and compare against Z3str3. We also could not compare against Stranger/ABC because they do not produce models, do not support dis-equations over arbitrary string terms, and have correctness issues as noted in their paper [4].

II. Theory-Aware Branching

Several of the key enhancements we make in Z3str3 over Z3str2 involve changes to the Z3 core solver, which handles the Boolean structure of the formula and performs propagation and branching. The first of these enhancements is referred to as **theory-aware branching**. We modify the Z3 core solver to allow theory solvers to give certain literals increased or decreased priority during the search. Consider the case where the solver learns the equality $X \cdot Y = A \cdot B$ for non-constant terms X, Y, A, B. Z3str3, in line with Z3str2, handles this equality by considering a disjunction of three possible arrangements [23], [24]:

Arrangement 1: $X = A$ and $Y = B$
Arrangement 2: $X = A \cdot s_1$ and $s_1 \cdot Y = B$ for a fresh non-empty string variable s_1
Arrangement 3: $X \cdot s_2 = A$ and $Y = s_2 \cdot B$ for a fresh non-empty string variable s_2

Of the three possible arrangements, the first is the simplest to check because it does not introduce any new variables and

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

only asserts equalities between existing terms. Therefore, we would like Z3's core solver to prioritize checking this arrangement before the others. The advantage gained by theory-aware branching is the ability to give the core solver information regarding the relative importance of each branch, allowing the theory solver to exert additional control over the search. We always prioritize simpler branches over more complex ones.

We implement theory-aware branching as a modification of the branching heuristic in Z3. The default branching heuristic in Z3 is activity-based, similar to VSIDS [13]. The core solver will branch on the literal with the highest activity that has not yet been assigned. Activity is increased additively when a literal appears in a conflict clause, and decayed multiplicatively at regular intervals. Although we are not aware of any other work on theory-aware branching, there has been some work in taking domain-specific knowledge into account in the context of branching heuristics and custom decision strategies [10], [14], [8].

The theory-aware branching technique computes the activity of a literal A as the sum of two terms A_b and A_t, wherein the term A_b is the "base activity", which is the standard activity of the literal as computed and handled by Z3's core solver. The term A_t is the "theory-aware activity". The value of this term is provided for individual literals by theory solvers, and is taken to be 0 if no theory-aware branching information has been provided. This modification causes the core solver to branch on the literal with the highest activity A, taking into account both the standard activity value and the theory-aware activity. Therefore, assigning a (small) positive theory-aware activity to a literal will cause it to have higher activity than usual, making it more likely for the core solver to choose it to branch on. Conversely, assigning a (small) negative theory-aware activity will deter the core solver from choosing that literal. Theory-aware branching in Z3str3 modifies the activities of theory literals as follows:

1) Literals corresponding to arrangements that do not create new variables (as in Arrangement 1 above) are given a large (0.5) A_t. Other arrangements in the same case are given a small (0.1) A_t.
2) Arrangements that allow a variable to become equal to a constant string are given a small (0.2) A_t.
3) When searching for length of strings, literals corresponding to longer length values have small negative (-0.1) A_t.

The values of A_t were chosen to be similar in scale to the initial activity values assigned to literals by the default branching heuristic. Although this technique is currently used by the string solver component, theory-aware branching is also useful in many other contexts where new search paths may have unequal importance, such as non-linear arithmetic.

III. Theory-Aware Case-split

During the search, a theory solver can create terms which encode a disjunction of Boolean literals that are pairwise mutually exclusive, i.e., exactly one of the literals must be assigned true and the others must be assigned false. We refer to this as a **theory-aware case-split**. As an example, consider the case where the string solver learns that a concatenation of two string variables X and Y is equal to a string constant $c = c_1c_2 \dots c_n$ of length n, where each c_i is a character in c. There are $n+1$ possible ways in which we can split the constant c over X and Y resulting in different arrangements:

- $X = \epsilon, Y = c_1c_2 \dots c_n$
- $X = c_1, Y = c_2c_3 \dots c_n$
- $\dots$
- $X = c_1c_2 \dots c_n, Y = \epsilon$

Note that each of these arrangements represents a case that can be explored by the solver, and also that all of these cases are mutually exclusive (as clearly X cannot be equal to both ϵ and c_1 simultaneously, etc.). Thus, this represents a theory-aware case-split. However, the Boolean abstraction constructed over theory literals hides the fact that these are mutually exclusive cases. A naïve solution is to encode $O(n^2)$ extra mutual exclusion Boolean clauses over these variables. Unfortunately, this would result in very poor performance because of the quadratic blowup in formula size. Another option is to let the congruence closure solver in the Z3 core discover the mutual exclusivity of these Boolean variables. This can result in unnecessary backtracking, unnecessary calls to congruence closure, and, in the worst case, reduces to the same set of mutual exclusion clauses being learned in the form of conflict clauses.

The means of handling such cardinality constraints efficiently has been well-studied; previous work has investigated the possibility of alternate encodings, e.g. totalizers [5] and lazy cardinality [3]. Our implementation, by contrast, shows a way to handle these constraints in the inner loop of the SAT solver in a theory-aware manner. This means that theory solvers do not have to perform rewriting or assert extra clauses to enforce mutual exclusivity of choices. Instead, they can provide this information directly to the core solver, which can use these facts during the search. This saves on the propagation effort of the DPLL(T) framework. Our implementation of this technique is as follows:

1) The theory solver provides the core solver with a set S of mutually exclusive literals that correspond to a theory case-split. This set is maintained by the core solver in a list of all such sets.
2) During branching, the core solver checks if the current branching literal belongs to some such set S. If yes, the current branching literal is assigned true and all other theory case-split literals in S are assigned false. Otherwise, the default branching behaviour is used.
3) During propagation, the core solver may assign a truth value to a literal l in some set S of theory case-split literals. If so, the theory case-split check is invoked, i.e., the core solver checks whether two literals l_1, l_2 in the same set S have been assigned the value true. If this is the case, the core solver immediately generates the conflict clause $(\neg l_1 \vee \neg l_2)$.

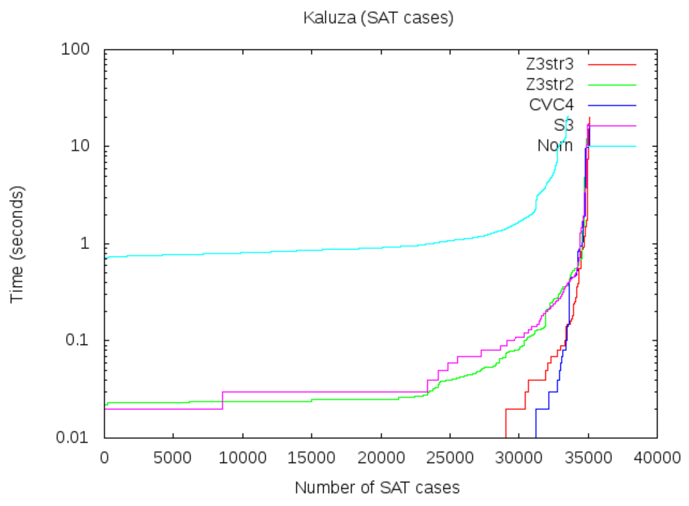


Fig. 1. Cactus plot of string solvers over the Kaluza benchmark (SAT cases).

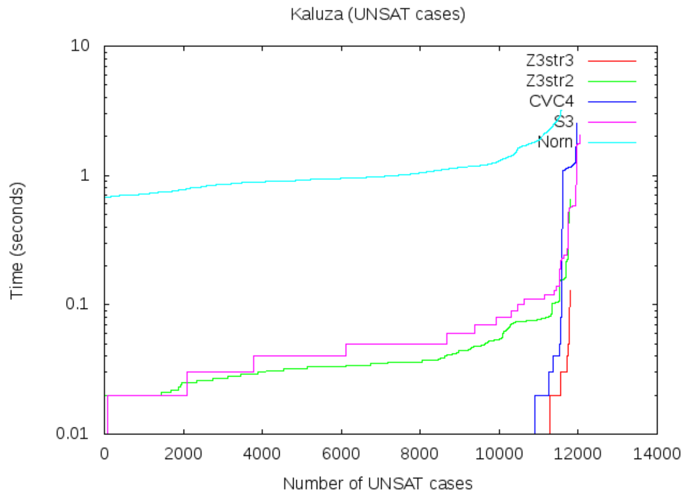


Fig. 2. Cactus plot of string solvers over the Kaluza benchmark (UNSAT cases).

IV. Experimental Results

In this section, we describe the experimental evaluation of the Z3str3 solver to validate the effectiveness of the techniques presented in this paper. We compare Z3str3 against four other state-of-the-art string solvers, namely, Z3str2 [24], [23], CVC4 [12], S3 [20], and Norn [2], across industrial benchmarks obtained from Kaluza [16], PISA [19], and AppScan Source [1]. Each of these benchmark suites draw from real-world applications with diverse characteristics. All experiments were performed on a workstation running Ubuntu 15.10 with an Intel i7-3770k CPU and 16GB of memory. Also, we cross-verified the models generated by Z3str3 against Z3str2 and CVC4, and vice-versa.

Table I shows the summary of results for the Kaluza benchmark. As can be seen from the cactus plots over SAT and UNSAT cases from the Kaluza benchmark, in Figures 1 and 2, Z3str3 (red series) outperforms competing solvers. On SAT cases, Z3str3 is competitive with CVC4 and significantly outperforms all other solvers. On UNSAT cases, Z3str3 is the fastest solver over all cases it can complete. As binaries for S3P are not publicly available, we report the aggregate results presented for this benchmark in the most recent S3P paper [21]. From Table I and Figures 1 and 2 it is clear that Z3str3 is highly competitive with respect to CVC4, and is much faster than other tools. Z3str3 solves more SAT instances than any other tool we benchmarked except S3P, and has the lowest total solving time on non-timeout cases. Notably, over all instances where both solvers finish, Z3str3 solves more cases in total than Z3str2 and completes 30% faster. The unknowns in Z3str3 are because it lacks the feature to handle string equations with overlapping variables, similar to Z3str2. However, Z3str3 has far fewer unknowns than Z3str2.

	Z3str3	Z3str2	CVC4	Norn	S3	S3P
sat	35147	34868	35128	33527	35016	35270
unsat	11799	11799	11957	11568	12049	12014
unknown	223	617	6	1913	0	0
timeout	115	0	0	276	219	0
error	0	0	193	0	0	0
Time (s)	4939.52	3997.63	4851.66	109280.76	10544.06	6972
Time w/o timeouts (s)	2971.02	3997.63	4851.66	97784.00	6164.06	6972

TABLE I
Kaluza benchmark results. Timeout=20 s. Total time includes all solved, timeout, unknown, and error instances.

Table II shows the results on the PISA benchmark, a set of industrial program analysis instances from IBM. Norn was not able to solve any of the cases as it crashed upon seeing unrecognized string operators (e.g. `indexof`). From Table II we make the following observations. The tools Z3str3, Z3str2, and CVC4 are in agreement on all cases they are able to solve, with CVC4 and Z3str2 timing out on one SAT case which Z3str3 can solve in 0.43 seconds. The results for S3 are significantly worse; it is unable to solve `pisa-009.smt2` while the other three solvers all answer SAT very quickly; and in addition S3 incorrectly answers UNSAT for `pisa-008.smt2`, `pisa-010.smt2`, and `pisa-011.smt2`, on which Z3str3 and (for two of these cases) Z3str2 and CVC4 all return SAT and produce a valid model. The performance of Z3str3 on this benchmark is highly competitive with other solvers, improving on the result from Z3str2.

Table III shows the results on the AppScan benchmark, a second set of industrial instances from IBM. Norn crashed on these cases as well upon seeing unrecognized string operators. From Table III we make the following observations. Z3str3, Z3str2, and CVC4 all agree on all cases they are able to solve. CVC4 performs slightly better than Z3str3 on 3 cases, equally well on 1, and worse on 4, timing out on one case that Z3str3 can solve in 0.73 seconds. In total, on non-timeout cases, CVC4 takes twice as long as Z3str3 (7.89 seconds vs. 4.15 seconds). Z3str2 performs better than Z3str3 on 1 case and worse on 7, taking almost ten times as long on all cases (33.17 seconds vs. 4.15 seconds). S3 returns UNKNOWN on two cases that are solved by the other three tools and produces invalid models which fail cross-validation for four other cases.

input	Z3str3		Z3str2		CVC4		S3	
	result	time (s)	result	time (s)	result	time (s)	result	time (s)
pisa-000.smt2	sat	0.03	sat	0.25	sat	0.08	sat	0.07
pisa-001.smt2	sat	0.05	sat	0.19	sat	0.00	sat	0.07
pisa-002.smt2	sat	0.03	sat	0.10	sat	0.00	sat	0.05
pisa-003.smt2	unsat	0.02	unsat	0.02	unsat	0.01	unsat	0.02
pisa-004.smt2	unsat	0.02	unsat	0.05	unsat	0.39	unsat	0.05
pisa-005.smt2	sat	0.02	sat	0.14	sat	0.02	sat	0.04
pisa-006.smt2	unsat	0.03	unsat	0.05	unsat	0.32	unsat	0.05
pisa-007.smt2	unsat	0.02	unsat	0.05	unsat	0.37	unsat	0.05
pisa-008.smt2	sat	0.43	timeout	20.00	timeout	20.00	unsat X	4.73
pisa-009.smt2	sat	0.60	sat	0.62	sat	0.00	timeout	20.00
pisa-010.smt2	sat	0.02	sat	0.09	sat	0.00	unsat X	0.02
pisa-011.smt2	sat	0.03	sat	0.06	sat	0.00	unsat X	0.02

TABLE II
PISA BENCHMARK RESULTS. TIMEOUT=20 S. X = INCORRECT RESPONSE.

input	Z3str3		Z3str2		CVC4		S3	
	result	time (s)	result	time (s)	result	time (s)	result	time (s)
t01.smt2	sat	0.18	sat	1.31	sat	0.01	sat	0.23
t02.smt2	sat	0.17	sat	0.38	sat	0.01	unknown	0.04
t03.smt2	sat	0.27	sat	9.54	sat	3.82	sat X	0.14
t04.smt2	sat	0.73	sat	4.45	timeout	20.00	sat X	0.10
t05.smt2	sat	0.57	sat	16.84	sat	3.87	sat X	0.55
t06.smt2	sat	0.02	sat	0.15	sat	0.01	sat	0.13
t07.smt2	sat	2.18	sat	0.25	sat	0.00	unknown	0.02
t08.smt2	sat	0.03	sat	0.25	sat	0.17	sat X	0.03

TABLE III
APPSCAN BENCHMARK RESULTS. TIMEOUT=20 S. X = INCORRECT RESPONSE.

Heuristic	Neither	Theory-aware branching	Theory-aware case split	Both
sat	35079	35147	35092	35147
unsat	11799	11799	11799	11799
unknown	221	230	223	223
timeout	185	108	170	115
Time (s)	6252.26	6055.04	5027.35	4939.52

TABLE IV
PERFORMANCE COMPARISON WITH THEORY-AWARE BRANCHING AND THEORY-AWARE CASE SPLIT ENABLED AND DISABLED IN ALL COMBINATIONS. TIMES TAKEN OVER KALUZA BENCHMARK WITH 20 S TIMEOUT. TOTAL TIME INCLUDES ALL SOLVED, TIMEOUT, AND UNKNOWN INSTANCES.

Table IV presents the results of a comparison in which each of the new heuristics in Z3str3, namely theory-aware branching and theory case split, was enabled and disabled in all combinations, in order to measure the change in behaviour of the solver when run over the same benchmark (Kaluza). The experiment clearly shows that both techniques improve the performance of the solver both in isolation and in combination. One intuition for the disparity in performance with respect to each heuristic is that the theory case-split heuristic applies in every instance, due to the frequency of generation of mutually-exclusive options during the search, while the theory-aware branching heuristic is only effective in cases with a large amount of backtracking and search activity, and as such the solver may not benefit from it if the solution is easy to find.

V. DISCUSSION ON EXPERIMENTAL RESULTS, AND CONCLUSIONS

The experimental results discussed here make clear the efficacy of theory-aware branching and case-split. The crucial insight behind these techniques is that biasing the search towards easier branches of the search tree (e.g., an arrangement that doesn't require splitting variables, as opposed to one with overlapping variables) is often very effective since most string constraints obtained from practical applications have the "small model" property. The slogan of theory-aware branching is "bias search towards easy cases first". We also note that Z3str3 and CVC4 do not give any incorrect results, and are more robust than Norn and S3 which sometimes give wrong answers or crash on the benchmarks we used.

REFERENCES

[1] IBM Security AppScan Tool and Source. URL: http://www-03.ibm.com/software/products/en/appscan-source.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.

[3] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and P. J. Stuckey. *To Encode or to Propagate? The Best Choice for Each Constraint in SAT*, pages 97–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[4] A. Aydin, L. Bang, and T. Bultan. *Automata-Based Model Counting for String Constraints*, pages 255–272. Springer International Publishing, Cham, 2015.

[5] O. Bailleux and Y. Boufkhad. *Efficient CNF Encoding of Boolean Cardinality Constraints*, pages 108–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 307–321, 2009.

[7] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, 2008.

[8] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[9] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[10] A. Erez and A. Nadel. Finding Bounded Path in Graph Using SMT for Automatic Clock Routing. In *Proceedings of the 27th International Conference on Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer International Publishing, 2015.

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.

[12] 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 *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. 2014.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[14] A. Nadel. Routing under constraints. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 125–132, Austin, TX, 2016. FMCAD Inc.

[15] G. Redelinghuys, W. Visser, and J. Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 139–148, 2012.

[16] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[17] R. Sebastiani. Lazy satisfiability modulo theories. In *Journal on Satisfiability, Boolean Modeling and Computation*, volume 3, 2007.

[18] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[19] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[20] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, 2014.

[21] M.-T. Trinh, D.-H. Chu, and J. Jaffar. *Progressive Reasoning over Recursively-Defined Strings*, pages 218–240. Springer International Publishing, Cham, 2016.

[22] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007.

[23] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, pages 1–40, 2016.

[24] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. *Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints*, pages 235–254. Springer International Publishing, Cham, 2015.

[25] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, 2013.

Verification of a lazy cache coherence protocol against a weak memory model

Christopher J. Banks[1], Marco Elver[1*], Ruth Hoffmann[2], Susmit Sarkar[2], Paul Jackson[1], Vijay Nagarajan[1]
[1]University of Edinburgh, [2]University of St Andrews

Abstract—**In this paper, we verify a modern lazy cache coherence protocol, TSO-CC, against the memory consistency model it was designed for, TSO. We achieve this by first showing a weak simulation relation between TSO-CC (with a fixed number of processors) and a novel finite-state operational model which exhibits the laziness of TSO-CC and satisfies TSO. We then extend this by an existing parameterisation technique, allowing verification for an unbounded number of processors. The approach is executed entirely within a model checker, no external tool is required and very little in-depth knowledge of formal verification methods is required of the verifier.**

I. Introduction

In parallel architectures with local caches, cached values can become stale. Therefore, it is imperative that the system guarantees shared memory correctness by ensuring that it correctly implements a *memory consistency model* (MCM)—the formal model that determines what value a read should return [1]. An integral component of enforcing an MCM is the *cache coherence protocol* (CCP), which is responsible for making writes visible to other caches in an order that is consistent with the MCM.

Traditionally, CCPs have been designed for the strictest of MCMs—*sequential consistency* (SC). Previously, this has been beneficial as a way to decouple the design of a CCP from the MCM; indeed, a CCP designed for the strongest of MCMs could bolt-on to other weaker MCMs. Unfortunately, this simplicity comes at a cost.

The strict program order requirements of SC mandates that writes are made globally visible before any subsequent memory operation from the same processor. To guarantee this, CCPs *eagerly* invalidate other non-local shared copies upon a write. In effect, such eager CCPs enforce the Single-Writer–Multiple-Reader (SWMR) invariant [2]—a cache line may only have either a single writer or multiple readers. To this end, eager CCPs must maintain a vector of processors sharing a cache line, but this vector scales linearly with the number of processors [3], [4]. Thus these protocols do not scale well to large-scale many-core processors.

Luckily, modern architectures tend to have more relaxed MCMs like Total Store Order (TSO)—used in prevalent architectures such as x86 and SPARC. Consequently, it is possible for CCP designers to take advantage of these relaxations. Indeed, there has been significant recent research on *lazy coherence protocols* [5], [3], [6], [4], that exploit the fact that relaxed models only require memory to be consistent at synchronisation boundaries. In these protocols, shared lines are *self-invalidated* on synchronisation boundaries and therefore no longer require a (poorly scaling) sharing vector.

This poses a problem for the verification of such protocols. Traditionally, formal verification approaches for CCPs [7], [8] have focused on model checking protocol-specific safety properties such as the SWMR invariant [2]. However, these new lazy CCPs that are designed to take advantage of weak MCMs violate SWMR by design and hence cannot be verified in the usual way. They need to be verified in a stronger manner: for adherence to the MCM. This is especially appropriate for the protocol we study, TSO-CC [6], because it was designed specifically with the TSO memory model in mind.

Challenges: If these new scalable lazy CCPs are to see the light of day, we believe they need to be formally verified against the MCM. A testing approach does not cover all corner cases and does not give the confidence that formal verification brings. Equally, the subtlety of behaviours exhibited by both lazy CCPs and weak MCMs warrants a rigorous approach. Only formal verification will suffice to allay skepticism surrounding the behaviour of lazy CCPs. Furthermore, the verification technique should be generally applicable, should not assume the verifier to have sophisticated knowledge beyond the protocol, and it should scale to many-core processors.

Our result: In this paper, for the first time, we formally and exhaustively verify a modern lazy CCP against the MCM which it is supposed to implement. Our protocol of interest is TSO-CC (Section II), a scalable lazy CCP which was designed to target TSO. We establish our result for fixed cache sizes, but for any number of processors. Our verification focuses on safety; we do not tackle liveness. This enables our verification approach to use a slightly abstract version of CCP where, for example, access counters are not modelled explicitly.

Our approach to verification proceeds as follows. First, we propose a novel finite-state operational model TSO-LB, based on load buffers, that abstracts our lazy CCP TSO-CC. Second, we use a model checker to establish that TSO-CC is a refinement of the TSO-LB operational model. Initially we show refinement for a fixed number of processors; subsequently we deploy the parameterised verification technique of Chou et al. [9] to extend our refinement result to an arbitrary number of processors. Finally, we show that the TSO-LB operational model is stricter than an axiomatic specification of TSO.

Contributions: Our approach is inspired by Chatterjee et al. [10], who showed how CCPs can be verified against their

*Now at Google.

†This work is supported by EPSRC grant EP/M027317/1

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

MCMs using a model checker. Beyond this work, we make a number of specific advances.

First, we support, for the first time, a lazy CCP through the use of a novel abstract operational model. A lazy CCP like TSO-CC *pulls* new values via self-invalidates upon a read, in contrast to conventionally eager CCPs which *push* invalidates upon a write. The nature and timing of invalidations in eager and lazy CCPs are different. Current operational models abstract the push-based invalidates, which makes it difficult to show that lazy CCPs refine them. We therefore needed to introduce this novel operational model we call TSO-LB which abstracts pull-based self-invalidates.

Second, we provide a proof that our TSO-LB model satisfies an axiomatic characterisation of TSO, however in Chatterjee et al. the task of showing the abstract operational models are consistent with axiomatic descriptions of the MCMs is not completed (and, as far as we can tell, was never subsequently completed). In our case, the proof is particularly important given how TSO-LB differs from conventional operational models for TSO.

Third, we employ the parameterisation technique of Chou et al. [9] to verify for an arbitrary number of processors (whereas Chatterjee et al. only verified for a fixed number of processors). In doing so, we demonstrate that the technique is not only useful when model checking CCP properties, but also is useful when using model checking to verify refinement and show a CCP satisfies the relevant MCM.

Other related work: Another alternative approach by Manerkar et al. [11] uses CCICheck, which explores ordering relations between CCP and MCM; however, protocols must be described in an axiomatic style—orthogonal to typical operational descriptions of protocols—and verification is with respect to specific litmus tests—which may not capture every MCM behaviour and hence not exhaustive. It is notable that these approaches only verify for a fixed number of processors; an approach to solving this problem is found in compositional model checking approaches pioneered by McMillan [12]. This method was further refined by Chou et al. [13], [9] and made practical; however, they, once again, only deal with protocol-specific properties. Likewise, Pong and Dubois [14], [15] verify compositionally, using Symbolic State Models, but again only against protocol-specific properties.

Abdulla et al. [16] recently propose the Dual-TSO operational model for TSO for program verification, in which they replace the store buffer in the traditional operational model with a load buffer. However, their notion of a load buffer has unbounded queues with potentially multiple values for an address, and thus does not help us with the infinite state-space problem. Our model also works very differently (but similar to CCP's like TSO-CC) by propagating multiple addresses to a load buffer atomically. So our model is not obviously a refinement of some finite restriction of the Dual-TSO model. It is also worth noting that we first defined our TSO-LB model [17] concurrently with Abdulla et al.

II. TSO-CC

TSO-CC [6] is a lazy CCP, designed to address the scalability issues surrounding CCPs for large numbers of cores. Lazy CCPs, like TSO-CC, take account of the fact that the relaxed memory models employed in modern multi-core processors only require memory to be consistent at synchronisation boundaries. Consequently, instead of eagerly enforcing coherence at every write, coherence is enforced lazily only at synchronisation boundaries. Thus, upon a write, data is merely written to a processor-local write-buffer, the contents of which are flushed to the shared cache upon a *release*. Upon an *acquire*, shared lines in the local caches are self-invalidated—thereby ensuring that reads to shared lines fetch the up-to-date data from the shared cache. In effect, the CCP may be much simpler and *does not require a sharing vector*.

However, the design of TSO-CC is specifically directed by the TSO memory model which has no explicit release or acquire instructions. It follows that, as reads have acquire semantics and writes have release semantics, a TSO compliant CCP would only need to consider each read/write an acquire/release; this, of course is not efficient because all reads and writes would need to be propagated, effectively negating the provision of local caches.

The approach in TSO-CC is that for each cache line in the shared cache, it keeps track of whether the line is exclusive, shared, or read-only. Shared lines do *not require tracking of sharers* (making TSO-CC more scalable than standard directory-based protocols). Additionally, for exclusive cache lines, it only maintains a pointer to the owner.

Since it does not track sharers, writes do not eagerly invalidate shared copies in other processors. On the contrary, writes are merely propagated to the shared cache in program order (thus ensuring write-write order). To save bandwidth, instead of writing the full data block to the shared cache, it merely propagates the coherence states. Intuitively, the *most recent* value of any data is maintained in the shared cache.

Reads to shared cache lines are allowed to read from the local cache, up to a predefined number of accesses (potentially causing a stale value to be read), but are forced to re-request the cache line from the shared cache after exceeding an access threshold (the implementation maintains an access counter per line). This ensures that any write (used as a release) will eventually be made visible to the matching acquire, ensuring *eventual write propagation*. When a read misses in the local cache, it is forced to obtain the most recent value from the shared cache. In order to ensure the read-read order, future reads will also need to read the most recent values. To guarantee this, whenever a read misses in the local cache, it self-invalidates all shared cache lines. Finer details of the protocol may be found in the original paper by Elver and Nagarajan [6]. It should be noted that our model implements the basic protocol, without timestamps.

Prior TSO-CC verification work: In order to check that the protocol implementation adheres to TSO, the original authors of TSO-CC used the diy [18] tool to generate litmus tests for TSO (according to the method detailed in Owens et al. [19]) and ran it in a full-system simulator. An independent approach to verification was made by CCICheck [11], using TSO-CC as a case study. CCICheck uses abstract axiomatic models of pipeline and memory system, and verifies that a set of litmus tests is not violated. However, whilst a litmus test

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

based approach provides some confidence that the protocol is correct, it is by no means an exhaustive means of verification and corner cases may be missed. In order to minimise the potential for missed corner cases in a detailed cycle-accurate full-system implementation, Elver and Nagarajan developed McVerSi [20], a test generation framework for fast memory consistency verification in simulation. This approach, whilst it further increased confidence and testing of corner cases, is still not exhaustive. The remainder of this paper solves this problem with an entirely exhaustive approach to verifying TSO-CC against the TSO memory model.

III. TSO-CC SATISFIES TSO

In this section, we show that the lazy cache coherence protocol TSO-CC does indeed satisfy the constraints of the TSO memory consistency model. This solves the problems associated with the previous verification approaches: corner cases which could be missed by insufficient testing would now be revealed by exhaustive exploration of the state space. For now, we only show that this is true for the simpler case of a fixed number of processors. We go on to show, in Section IV, that this is true for a parameterised model of TSO-CC with any number of processors.

We took a number of discrete steps in the process of verifying the protocol. The first step was to translate the protocol into a suitable model for verification. For this purpose we chose the Murφ language and model checker [21]. Murφ is a well-established model checker and extensively used in both previous academic studies [21], [10], [22] and in industry [12], [7], [9], [23], [3]. We then went on to show that this model satisfied some basic properties, such as freedom from deadlock, using the model checker. Our approach to this is detailed in Section III-A.

The next step in the process was to show that the TSO-CC model satisfied the constraints of TSO. One way to achieve this was to show there exists a *weak simulation relation* between TSO-CC and an operational model of TSO. A weak simulation relation exists if the observable actions (reads/writes to a memory location) in the CCP model can be matched by actions in the model of TSO. This concept is defined more formally in Section III-D, in which we also explain our approach to showing weak simulation using the model checker.

However, in order for our approach to work, we needed an operational model of TSO. Such models exist in the literature but tend to be *store buffer based* [19], [24]. These models, while abstracting push-based eager CCPs well, make it difficult to show that lazy CCPs (which pull new values via self-invalidates) refine them. Furthermore, whereas such models require unbounded store buffers, we needed a finitely enumerable model for use with a model checking approach. Hence, in Section III-B we define TSO-LB, a *load buffer based* operational model with bounded buffers that abstracts lazy CCPs. After establishing that TSO-LB exhibits only TSO behaviour, we were able to use the operational model as part of our verification strategy.

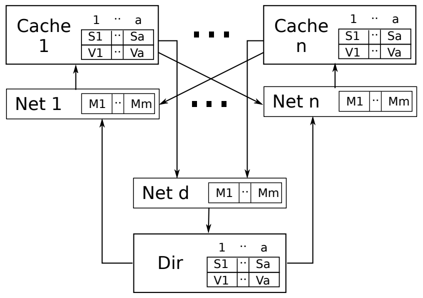


Fig. 1. Concrete model structure.

A. Model checking in Murφ

We began by defining a Murφ model of the TSO-CC protocol. The model implements the basic TSO-CC protocol as described in the original paper [6], with each rule in the protocol description relating to a rule in the Murφ model; it has parameters for the number of processors, number of addresses, and number of values; the model was checked using three address locations and two values. The model is a faithful implementation of the protocol with the only abstraction being the abstract interpretation of the access counter—as described below. The model is constructed as a set of caches and a directory, each having a state and a set of addresses or memory locations, each with a set of possible values. The interconnection network is represented as a set of sets of messages; each node (cache or directory) can write or read to or from the network (Figure 1).

A set of rules, each a *guard* $\implies$ *action* pair, then defines the behaviour of the model. As an example, the following is a pair of sample rules taken from the full ruleset[1]:

$$c[a].\texttt{state} = \texttt{I} \implies \texttt{SendGetS}(c, Dir, a); \quad c[a].\texttt{state} := WS \qquad \text{(Read I)}$$

$$c[a].\texttt{state} = \texttt{E} \implies c[a].\texttt{val} := v \quad c[a].\texttt{state} := M; \qquad \text{(Write E)}$$

where c is a cache, a is an address (memory location), v is a value, and $c[a].\texttt{state}$ (or $c[a].\texttt{val}$) is the state (or value) for the given cache and address. The first rule (Read `I`) is the Read rule for the `I`nvalid cache state and the second is the Write rule for the `E`xclusive cache state. When a cache is in state `I` and does a Read, it sends a `GetS` message to the directory and switches to state `WS`. When a cache is in state `E` it may do a Write, store the written value, and switch to state `M` (Modified). The function `SendGetS` handles the passing of a `GetS` message to the network.

We then define a rule that handles the receipt of messages from the network at each node (cache or directory). Within this rule are some functions which handle actions performed when a message is received; the following is an extract from the `DirectoryReceive` function for handling the messages in the previous example:

[1]The full ruleset can be found at https://github.com/icsa-caps/tso-cc

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

```
DirectoryReceive($msg, a$) =
        if $Dir[a]$.state = I $\wedge$ $msg$.type = GetS
        then SendDataS($msg$.src, $a, \ldots$);
             ReplaceOwner($msg$.src, $a$);
             $Dir[a]$.state := WE1
        else ...
```

There is a function `CacheReceive` which has similar conditions for receiving messages at a cache.

Another pair of rules which are of interest are the Read rules for the Shared cache state. Part of the lazy invalidation scheme for TSO-CC is that a cache must self-invalidate after a certain number of reads, specifically once an access counter reaches a predefined limit. In our model, we abstract the access counter by just having two rules corresponding to a Read in the Shared state: one where the access count is within its limit and another for when the limit has been reached. There is thus a non-deterministic choice between the two options:

```
$c[a]$.state = S $\Longrightarrow$ SendGetS($c, Dir, a$);
                      $c[a]$.state := $WS$
                                                  (Read S[MAX])

$c[a]$.state = S $\Longrightarrow$ //do nothing
                                                  (Read S[<MAX])
```

In the first rule, the access count has been reached, causing self-invalidation followed by re-requesting a fresh value from the directory; in the second rule, the access count has not been reached, and the cache is free to read hit on its own value with no further action. The model checker accounts for the non-deterministic choice between these rules.

Once the full ruleset in the model checker is defined, the rules are then exhaustively applied using an appropriate strategy (e.g. breadth first, depth first) until every possible state of the model has been enumerated; during this process of state enumeration the model checker checks that it can always proceed to another state (deadlock freedom) and that any defined invariants hold for each state. For efficiency, Murφ also reduces the set of states which need to be enumerated by using various techniques, such as symmetry reduction [25].

The next problem was to decide what property to check the model against. The derived properties which usually hold for CCPs, like SWMR do not hold for TSO-CC, by design, so a new strategy has to be applied. Our verification strategy is to establish that TSO-CC satisfies TSO by: (a) devising TSO-LB, a finite operational model for abstracting TSO-CC, (b) proving that TSO-LB shows only TSO behaviour, and (c) showing that TSO-CC is a refinement of TSO-LB within a model checker.

B. TSO-LB operational model

This section introduces the abstract TSO load-buffering model (TSO-LB). For our approach, existing operational models of TSO [19], [24] are not ideal for two reasons. The first being that they require unbounded buffers, making algorithmic verification difficult. Second, a refinement between a lazy CCP and an existing store-buffering model would be difficult, as a lazy CCP effectively follows a load-buffering rather than a store-buffering approach: loads, viz. reads, hitting on a locally "buffered" (potentially) stale value, until the current value is pulled in (i.e. propagates) from global memory via a self-invalidate. The load-buffering based operational model formalised below abstracts a lazy CCP better and hence simplifies verification.

Definition 1 (Labelled Transition System): A *labelled transition system* (LTS) is a tuple $(\mathcal{L}, \mathcal{Q}, \mathcal{I}, \mathcal{T})$ where, $\mathcal{L}$ is a set of *labels*, $\mathcal{Q}$ is a set of *states*, $\mathcal{I} \subseteq \mathcal{Q}$ is a set of initial states and $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{L} \times \mathcal{Q}$ is the *transition relation*.
If $(q, l, q') \in \mathcal{T}$ then we say there is a transition labelled $l \in \mathcal{L}$ from state $q \in \mathcal{Q}$ to state $q' \in \mathcal{Q}$ and we may abbreviate this as $q \xrightarrow{l} q'$.

Definition 2 (TSO-LB): We define an LTS for TSO-LB as follows. The transition relation is given by the rules:

$$\frac{\mathtt{local}_q(p)(a) = v}{q \xrightarrow{\mathsf{Read}(p,a,v)} q}\ \textsc{Read}$$

$$\frac{}{q \xrightarrow{\mathsf{Write}(p,a,v)} \langle \mathtt{local}_q[(p)(a) \mapsto v],\ \mathtt{global}_q[(a) \mapsto v] \rangle}\ \textsc{Write}$$

$$\frac{}{q \xrightarrow{\tau} \langle \mathtt{local}_q[(p) \mapsto \mathtt{global}_q],\ \mathtt{global}_q \rangle}\ \textsc{Propagate}$$

where P is a finite set of processors, with $p \in P$; A is a finite set of addresses (memory locations), with $a \in A$; V is a finite set of data values, with $v \in V$; $\mathtt{local}_q : P \to A \to V$ is a function where $\mathtt{local}_q(p)(a)$ is the value at address a in the local buffer of p in state q and $\mathtt{global}_q : A \to V$ is a function where $\mathtt{global}_q(a)$ is the value at address a in the global buffer in state q.

The set of states $\mathcal{Q}$ consists of all pairs $\langle \mathtt{local}_q, \mathtt{global}_q \rangle$ and the set of labels $\mathcal{L} = \{\tau, \mathsf{Read}(p, a, v), \mathsf{Write}(p, a, v)\}$ where $p \in P$, $a \in A$, $v \in V$ and τ is the silent action. We define the set of initial states $\mathcal{I}$ to be $\mathcal{I} \triangleq \{q\ :\ \forall p \in P.\ \forall a \in A.\ \mathtt{local}_q(p)(a) = \mathtt{global}_q(a)\}$.

C. TSO-LB satisfies TSO

In the following, we outline a proof sketch that the TSO-LB operational model defined in Definition 2 permits only TSO behaviour. Since TSO-LB is defined as a LTS, its behaviour is defined with respect to an arbitrary trace of this LTS. We show (Theorem 1), by means of an interpretation of logical and physical time over these traces, that the behaviour satisfies the `herd` axiomatic characterisation of TSO [26]. We also show via a counterexample that TSO-LB does not permit all allowable behaviours of TSO, i.e. TSO-LB is in fact stricter than TSO.

Theorem 1 (TSO-LB satisfies TSO): The read and write events of traces of the TSO-LB LTS satisfy the TSO axiomatic MCM (as formalised in Alglave et al. [26].

Our proof strategy starts with defining a trace P of TSO-LB (Definition 3). The trace order might be seen as the physical-time representation of events, which contains writes, reads, and propagates. We will then construct a *strict linear order* L from P which contains the same writes and reads (with the same values). We then show how to instantiate the required ordering relations from the herd framework of Alglave et al. [26] from L, and show all those orders are contained in L. This will then allow us to show that the herd axiomatic constraints of TSO hold over the write and read events. Note that we assume a simplified TSO model excluding fences, as TSO-LB does not model fences by definition.

Definition 3 (Trace): A *trace* of an LTS is a sequence (finite or infinite) of labels that results from a path of transitions starting at the initial state. Let us call this trace order P.

Definition 4 (Logical-time L): We define L to be an order on the read and write events in the trace P. All writes $\mathsf{Write}(p, a, v)$ appear in L in the same order as in the physical-time trace P. A read $\mathsf{Read}(p, a, v)$ is pulled backwards in the trace to just after the event in P which made the processor p get the value v for a. Such an event is either a write from the same processor p, or a propagate to the processor p (if $\mathsf{Read}(p, a, v)$ reads from a write on another processor).

Note that several reads from the same processor can be pulled back to the same point in this scheme, if the same address is read by multiple reads, or if the propagated values for different addresses for the same propagate event are read from by different reads. In such a case, we order these multiple reads in L (which have to be from the same processor) according to program order.

Definition 5 (co in TSO-LB): The order co is defined in TSO-LB as $\mathsf{Write}(p, a, v) \overset{\mathsf{co}}{\to} \mathsf{Write}(q, b, w)$ if and only if $\mathsf{Write}(p, a, v)$ occurs before $\mathsf{Write}(q, b, w)$ in the physical-time trace P, and the addresses a and b are the same.
Note that p and q may be the same or different processors, and v and w the same or different values.

Definition 6 (rf in TSO-LB): The order rf is defined as $\mathsf{Write}(p, a, v) \overset{\mathsf{rf}}{\to} \mathsf{Read}(q, a, v)$ where p and q may be the same or different processors, and the read gets its value from the write.

We can now show that co , rf , and all the derived relations of the herd TSO formalisation are sub-orders of L. Then all axioms state the acyclicity and irreflexivities of various order relations, which are satisfied by any sub-orders of a strict linear order L. For the complete proof we refer to the online appendix.[2]

D. Weak simulation by model checking

Our core goal here is to check that a value read from a memory location by a processor at any point in time adheres to the TSO-LB specification, if all memory accesses are governed by the TSO-CC protocol.

We model both the TSO-LB specification and the TSO-CC protocol as labelled transition systems. In both cases, the labels are either *observable actions* concerning reads and writes or they are *silent actions*. For convenience below, we use the single label τ for all silent actions, though in our implementation it is useful to consider each system having a number of silent actions.

Our formal notion of correctness is that every observable trace of the TSO-CC protocol LTS is also an observable trace of the TSO-LB specification LTS. An *observable trace* is a trace with all the silent actions removed. We establish this inclusion property of observable traces by exhibiting a weak simulation relation between the TSO-CC LTS and the TSO-LB LTS such that the pair of initial states of the two LTSs is included in the relation.

A *weak simulation relation* shows step-by-step that for every observable action in TSO-CC there is a corresponding observable action in TSO-LB; it makes no attempt to match the silent actions in the two LTSs. This notion of weak simulation may be defined more formally as follows (following Milner [27]).

Definition 7 (Weak transition): Let $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{I}, \mathcal{T})$ be an LTS. A *weak transition* $q \overset{l}{\Longrightarrow} q'$ is defined as $q \xrightarrow{\tau}{}^{*} x \xrightarrow{l} y \xrightarrow{\tau}{}^{*} q'$ for some x,y, where $\xrightarrow{\tau}{}^{*}$ is the reflexive transitive closure of $\xrightarrow{\tau}$ and $q, q', x, y \in \mathcal{Q}$, $l \in \mathcal{L}$ and $l \neq \tau$.

Later we use the notation $q \Longrightarrow q'$ for $q \xrightarrow{\tau}{}^{*} q'$ or, if we allow multiple silent-action labels, to say that q' can be reached from q by zero or more transitions labelled by silent actions.

Definition 8 (Weak simulation): Let $\mathcal{C} = (\mathcal{L}, \mathcal{Q}_C, \mathcal{I}_C, \mathcal{T}_C)$ and $\mathcal{A} = (\mathcal{L}, \mathcal{Q}_A, \mathcal{I}_A, \mathcal{T}_A)$ be two LTSs with the same label set. Let $l \in \mathcal{L}$ be an observable action. A *weak simulation* $\mathcal{W} \subseteq \mathcal{Q}_C \times \mathcal{Q}_A$ is a binary relation such that if $(p, q) \in \mathcal{W}$, written $p\mathcal{W}q$, then

1) if $p \xrightarrow{l} p'$ then there exists $q' \in \mathcal{Q}_A$ such that $q \overset{l}{\Longrightarrow} q'$ and $p'\mathcal{W}q'$, and
2) if $p \xrightarrow{\tau} p'$ then there exists $q' \in \mathcal{Q}_A$ such that $q \Longrightarrow q'$ and $p'\mathcal{W}q'$.

In our setting, $\mathcal{Q}_C$ are the states of the CCP and $\mathcal{Q}_A$ are the states of the MCM.

To prove that there exists a weak simulation relation using a model checker, we construct an unlabelled transition system $\mathcal{M} = (\mathcal{Q}, \mathcal{T})$ from the two LTSs with $\mathcal{Q} = \mathcal{Q}_C \times \mathcal{Q}_A$ and a specially crafted transition relation $\mathcal{T}$. If a certain property holds for every reachable state of $\mathcal{M}$, then the set of reachable states is a weak simulation relation between $\mathcal{C}$ and $\mathcal{A}$. As the initial state of $\mathcal{M}$ is a pair of the initial states of $\mathcal{C}$ and $\mathcal{A}$, we have that the initial state pair are related by the weak simulation, and hence every observable trace of $\mathcal{C}$ is also an observable trace of $\mathcal{A}$. We can describe the transition relation and checked property as follows. The transition relation $\langle p, q\rangle \longrightarrow \langle p', q'\rangle$ is defined as $\exists l \in L.\, p \xrightarrow{l} p' \wedge q' = \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ where $\mathrm{AbsWitness}(p, q, l)$ computes an alternating sequence of abstract states and labels $\langle q_0, l_0, q_1, l_1, \ldots, q_n\rangle$ for some $n \geq 0$, $q = q_0$ and the $\mathrm{last}()$ function picks out the last state q_n of such a sequence.

The checked property $\mathrm{Match}(\langle p, q\rangle)$ is defined as $\forall l \in L, p' \in \mathcal{Q}_C.\, p \xrightarrow{l} p' \Rightarrow \mathrm{AbsWitness}(p, q, l)$ is a witness for:

1) $q \overset{l}{\Longrightarrow} \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ if l is observable. and
2) $q \Longrightarrow \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ if $l = \tau$.

Here, an alternating sequence of abstract states and labels $\langle q_0, l_0, q_1, l_1, \ldots, q_n\rangle$ *is a witness for* $q_0 \Longrightarrow q_n$ if all the l_i are silent and $q_i \xrightarrow{l_i} q_{i+1}$ for all $i \in \{0, \ldots, n-1\}$, and *is a witness for* $q_0 \overset{l}{\Longrightarrow} q_n$ if there exists a unique $l_i = l$ in

[2]https://github.com/icsa-caps/tso-cc/blob/master/TSO-LB-proof.pdf

the sequence such that $\forall j \neq i\ l_j$ is silent and $q_i \xrightarrow{l_i} q_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. Witnesses for weak transition instances enable the straightforward checking of the truth of instances.

A conceptual sketch of the witness function $\mathrm{AbsWitness}$ we use is as follows:

- If TSO-CC does a write action, then TSO-LB is made to take a single corresponding write action step.
- For silent transitions of TSO-CC, the witness is a single state—i.e. TSO-LB takes no steps.
- If TSO-CC does a read action, then in TSO-LB we either do a read action, or a propagate action followed by a read. We settle for the single read step if it is allowed by TSO-LB. If not, we go for the 2 step witness. As propagate is the only silent action in TSO-LB and it is idempotent, there are no other options to consider.

In general the abstract LTS might permit several silent transitions and the $\mathrm{AbsWitness}$ function has to embody some strategy for testing possible silent actions; however, it is worthy of note that the trivial strategy, as described here, is generally applicable to checking *any* CCP against TSO-LB.

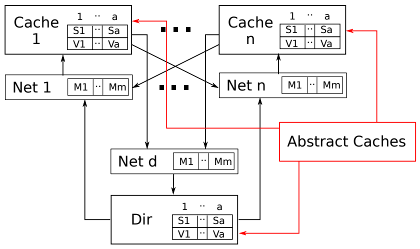


Fig. 2. Parameterised model structure with abstract caches.

E. *Weak simulation in Murφ*

To realise the above in Murφ we started with the TSO-CC Murφ model introduced in Section III-A and augmented the state with components for the TSO-LB specification. At the rules in the TSO-CC model where observable actions (reads/writes) are performed, we also step forward the TSO-LB model with the same actions, as explained above.

Coding the Match predicate is much simpler than the conceptual presentation above suggests. For the step forward of the TSO-LB system on write actions, the step is guaranteed by construction to satisfy the TSO-LB labelled transition relation, there is nothing to check. Only for the read action do we need to check that the value read in the TSO-LB specification actually matches that from the TSO-CC system. We simply use an invariant in Murφ to check the read value at each read step.

In the following, we detail how we implemented the transition system model and Match check in Murφ. The implementation of TSO-LB involves a pair of arrays to represent the global and local buffers for each cache and address, Murφ procedures `TSOStore` and `TSOUpdate`, and the Murφ function `TSOVerify`. These functions compute the next TSO-LB state for the Write, Propagate, and Read TSO-LB rules respectively. In addition `TSOVerify` returns a Boolean value indicating whether TSO-LB can indeed make one or two steps forward that result in a correct read. Calling `TSOVerify` returns true if the expected value is in the local buffer, or it tries a `TSOUpdate` and returns true if the expected value is now in the local buffer, else it returns false. A Murφ invariant ensures that `TSOVerify` always returns true.

The rules of TSO-CC incorporate these TSO-LB procedures and function. Taking our previous example rules, we amend them as follows:

$$
\begin{aligned}
c[a].\mathtt{state} = \mathtt{E} \implies\ & c[a].\mathtt{val} := v \\
& c[a].\mathtt{state} := M; \\
& \boxed{\mathtt{TSOStore}(c, a, v)}
\end{aligned}
$$

(Write E)

$$
\begin{aligned}
c[a].\mathtt{state} = \mathtt{S} \implies\ & \text{//do nothing;} \\
& \boxed{\mathtt{Assert}(\mathtt{TSOVerify}(c, a, c[a].\mathtt{val}))}
\end{aligned}
$$

(Read S[<MAX])

and likewise wherever a Read or Write action occurs in the CCP model. In this way our model shows that the values at the CCP level are consistent with the values at the MCM level.

Thus, for a fixed number of processors, we show that the simulation relation between TSO-CC and TSO-LB holds. The next problem was to show that the simulation relation holds for any number of processors. The next section shows how we solved this problem.

IV. TSO-CC WITH n PROCESSORS SATISFIES TSO

After showing that TSO-CC indeed satisfies TSO for a finite number of processors, we now show that this is also the case irrespective of the number of processors. In this section we present a parameterised model, parameterised in the number of processors, showing the same weak simulation relation between TSO-CC and TSO still applies with n processors.

In order to define a parameterised model we follow the method of Chou et al. [9], who in turn refined the ideas of McMillan [28]; the method is proven mathematically correct by Krstić [29]. The essence of the method is that one takes the original concrete model, but adds a new *abstract cache*. The abstract cache represents any number of caches connected to the concrete model (Figure 2). Initially the abstract cache can send any possible message to the concrete caches. This over-approximated set of messages is then reduced to only the set of legal messages by a process of counterexample guided abstraction refinement [30].

The initial over-approximated set of messages coming from the abstract cache will generate a counterexample when a spurious message is sent. One can then introduce a restriction to the abstract cache which disallows the spurious message. However it is then necessary to show that the restriction is valid and does not lead to an under-approximation of legal messages. In order to achieve this, one can write a *non-interference lemma* which shows the restricted message cannot occur in the concrete model. The key to the method is that the process is manual, but simply mechanical: the restriction is guided by the counterexample, the lemma is guided by the restriction, then the model checker checks both simultaneously and automatically. The apparent circular reasoning in proving the lemma on the amended system is dealt with by example by Chou et al. [9] and proved correct by Krstić [29].

A. Parameterised model in Murφ

The implementation of the parameterised model begins with the definition of a set of rules which generate all the possible messages which could be received at each node from the abstract caches. For example, one of the rules which handles the receipt of a `DataX` message at a cache is:

$$\begin{aligned} & c[a].\texttt{state} = \texttt{WX} \\ & \qquad \Longrightarrow \texttt{SendAck}(c, \mathit{Dir}, a); \\ & \qquad\quad c[a].\texttt{state} := \texttt{M}; \\ & \qquad\quad \texttt{InvalidateAllOtherLines}(c, a) \end{aligned}$$

(Cache Recv DataX Abs)

and similar rules are defined for each combination of node, state, and message received.

To extend the MCM to the parameterised model, we must consider what happens when our observable actions are performed by the abstract part of the parameterised model. As we do not track the state of abstract processors a Read action is not explicitly defined in the abstract part of the model, however a Write action by an abstract processor would have an effect (eventually) on the state of the concrete caches. We do not keep track of values at the abstract caches, so local buffers for abstract caches are not needed in the memory model. However, a Write at an abstract cache will go to the global buffer, because it may at some point be read by a concrete cache.

Thus, we implement a new function, `TSOStoreAbs`, which writes only to the global buffer. Now, in our abstract cache rules, we add a call to `TSOStoreAbs` wherever we see a Write action. For example, when the directory receives a Data message from an abstract cache a Write has occurred and we record this in the global buffer:

$$\begin{aligned} c[a].\texttt{state} = \texttt{WS} \Longrightarrow\ & \mathit{dir}[a].\texttt{val} := v; \\ & \texttt{TSOStoreAbs}(a, v) \\ & \mathit{dir}[a].\texttt{state} := \texttt{S} \end{aligned}$$

(Dir Recv Data Abs)

Once these rules are defined the model checker will generate all possible messages coming from the abstract cache. At this stage we have an *over-approximation* of the system. Of course, some of these messages will not be valid in the current state. When this occurs a counterexample will be generated by the model checker. The modeller must then inspect the counterexample and work out why the message was spurious. It is then possible to add a restriction to the rule that generated it such that the spurious message is eliminated.

For example, in the above rule (Cache Recv DataX Abs), we allow the cache to receive a `DataX` even when it is not the owner of the cache line. This produces a counterexample, because to receive a `DataX` from another cache (here an abstract cache) the other cache must have received a `FwdX` message first telling it to forward data to the new owner. Therefore the receiving cache must be the owner. To eradicate the counterexample we must add a restriction to check the receiving cache is the owner:

$$\begin{aligned} & c[a].\texttt{state} = \texttt{WX}\ \boxed{\wedge\ \ \texttt{IsOwner}(c, a)} \\ & \qquad \Longrightarrow \texttt{SendAck}(c, \mathit{Dir}, a); \\ & \qquad\quad c[a].\texttt{state} := \texttt{M}; \\ & \qquad\quad \texttt{InvalidateAllOtherLines}(c, a) \end{aligned}$$

(Cache Recv DataX Abs)

However, we must now show that the restriction is not too strict, i.e. we have not inadvertently caused the system to be *under-approximated* and, in essence, we are not changing the protocol. To do this, we introduce a *non-interference lemma*; this is a lemma which states the restriction as an invariant in the context of the concrete model, thus ensuring that the spurious messages eliminated are indeed not possible in the fully concrete model. For example, the lemma for the above restriction is:

$$\forall n \forall a \forall i.\ \mathit{net}[n][a][i].\texttt{msgType} = \texttt{DataX} \implies \texttt{IsOwner}(n, a)$$

where n is a node, a is an address, and i is a position in the message buffer. This is implemented in the model checker as an invariant[3] and if it does not fail then we know that we have not over-constrained the abstract cache.

Now, the model checker may catch a new counterexample. If this is the case then we repeat the process until all counterexamples are eliminated. Once all counterexamples are eliminated, we are done.

V. Results

In summary, the result of applying the method described in this paper to the TSO-CC protocol was that we showed that the protocol does, in fact, conform to the TSO memory model with any number of processors. Execution times for checking the full model are in the order of 14–15 hours on a single core of an Intel Xeon 1.8GHz machine with 64GB of RAM. The process of manually refining the model for parameterisation required 30 passes around the refinement loop, generating 30 non-interference lemmas. The time needed to define each lemma varied, depending on the complexity of the counterexample—at this stage, detailed knowledge of the protocol was a boon. Of note, however, is that for each pass around the refinement loop does not require 14 hours of model checking; generally, the model checker needed only to run

[3]Details of the restrictions and non-interference lemmas can be found in the model source at https://github.com/icsa-caps/tso-cc

for a few minutes to find the next counterexample—this time gradually increased as more counterexamples were eliminated.

It is also worth noting that one needs to consider vacuity in model checking and we must consider whether or not the specification holds trivially. We believe that our model is not vacuous, given that during the development of the model we observed and analysed a number of counter-example traces, both when we intentionally introduced bugs, during development, and when we iterated through the CEGAR loop, generating and refining the non-interference lemmas.

VI. Conclusion

We have shown that it is possible to verify a modern, lazy CCP against its counterpart TSO MCM. Our main contributions have been three fold:

1) the extension of a previous method [10] in order to formally verify a lazy CCP against the TSO weak MCM that it implements: the key novelty that enables this extension is the introduction of the new abstract operational model, TSO-LB;
2) a proof that our TSO-LB model satisfies a well-regarded axiomatic model of TSO;
3) extending the result of 1) to an arbitrary number of processors, using the parameterisation method of Chou et al. [9]: in establishing this result, we demonstrate that Chou et al.'s method for parameterised verification can be used to prove that a CCP refines an abstract operational model, not just for verifying protocol-specific properties such as SWMR.

We believe it would be straightforward to use our approach to verify other lazy CCPs that implement TSO. As such, one direction of future work is to improve the degree of automation in the method. Whilst the process of parameterisation of the model is simple, requiring more knowledge of the protocol than of formal methods, of note is the time and effort required to write restrictions, write lemmas, model check, and repeat. It is our belief that more of this process may be automated, as was the goal of both Chou et al. [9] and Krstić [29]. Some research on this topic already exists in the literature, for example O'Leary et al. [23] and Bingham et al. [31]. Another direction for future work is to check how we might use results similar to those of Henzinger et al. [32] to justify the verification for arbitrary numbers of addresses and data values.

References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[3] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," *PACT*, pp. 155–166, 2011.

[4] A. Ros, "Complexity-Effective Multicore Coherence," *ACM PACT*, pp. 241–251, 2012.

[5] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 472–483, 2011.

[6] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2014, pp. 165–176.

[7] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, 2003.

[8] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the Complexity of Hardware Cache Coherence and Some Implications," *ACM TACO*, vol. 11, no. 4, pp. 1–22, 2014.

[9] C. Chou, P. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," *FMCAD*, pp. 382–398, 2004.

[10] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking," *CAV*, pp. 121–138—-, 2002.

[11] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: using hb graphs to verify the coherence-consistency interface," in *MICRO'15*, 2015, pp. 26–37.

[12] K. L. McMillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 179–195, 2001.

[13] X. Chen, Y. Yang, G. Gopalakrishnan, and C. T. Chou, "Reducing verification complexity of a multicore coherence protocol using assume/guarantee," *FMCAD 2006*, vol. 3, pp. 81–88, 2006.

[14] F. Pong and M. Dubois, "Formal Verification Of Complex Coherence Protocols Using Symbolic State Models," *Journal of the ACM*, vol. 45, no. 4, pp. 557–587, 1998.

[15] ——, "Formal automatic verification of cache coherence in multiprocessors with relaxed memory models," *IEEE TPDS*, vol. 11, no. 9, pp. 989–1006, 2000.

[16] P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo, "The Benefits of Duality in Verifying Concurrent Programs under TSO," *27th International Conference on Concurrency Theory (CONCUR 2016)*, vol. 59, no. 5, pp. 1–5, 2016.

[17] M. Elver, "Memory Consistency Directed Cache Coherence Protocols for Scalable Multiprocessors," Ph.D. dissertation, University of Edinburgh, 2016.

[18] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," *Lecture Notes in Computer Science*, vol. 6605 LNCS, pp. 41–44, 2011.

[19] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-TSO," in *Lecture Notes in Computer Science*, vol. 5674 LNCS, 2009, pp. 391–407.

[20] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *HPCA*, vol. 2016-April, 2016, pp. 618–630.

[21] D. L. Dill, "The Murphi Verification System," in *CAV 96: Computer-Aided Verification*, vol. 1102, 1996, pp. 390–393.

[22] S. Burckhardt, R. Alur, and M. M. Martin, "Verifying safety of a token coherence implementation by parametric compositional refinement," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 130–145.

[23] J. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol verification using flows: An industrial experience," in *9th International Conference Formal Methods in Computer Aided Design, FMCAD 2009*, 2009, pp. 172–179.

[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO," *Communications of the ACM*, vol. 53, no. 7, p. 89, 2010.

[25] C. Norris IP and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 41–75, 1996.

[26] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats," *ACM TOPLAS*, vol. 36, no. 2, pp. 1–74, jul 2014.

[27] R. Milner, *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[28] K. L. McMillan, "Verification of infinite state systems by compositional model checking," *Lecture Notes in Computer Science*, vol. 1703, pp. 219–237, 1999.

[29] S. Krstić, "Parametrized System Verification with Guard Strengthening and Parameter Abstraction," *Electronic Notes in Theoretical Computer Science*, pp. 1–13, 2005.

[30] E. Clarke, "Counterexample-guided abstraction refinement," *Proceedings of the International Workshop on Temporal Representation and Reasoning*, pp. 7–8, 2003.

[31] J. Bingham, "Automatic non-interference lemmas for parameterized model checking," *FMCAD*, pp. 1–8, 2008.

[32] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Verifying sequential consistency on shared-memory multiprocessor systems," in *CAV*, vol. 1633, 1999, pp. 301–315.

Safety Verification of Phaser Programs

Zeinab Ganjei, Ahmed Rezine, Petru Eles, Zebo Peng
dept. of computer and information science
Linköping University, Sweden
firstname.surname@liu.se

***Abstract*—We address the problem of statically checking control state reachability (as in possibility of assertion violations, race conditions or runtime errors) and plain reachability (as in deadlock-freedom) of *phaser programs*. Phasers are a modern non-trivial synchronization construct that supports dynamic parallelism with runtime registration and deregistration of spawned tasks. They allow for collective and point-to-point synchronizations. For instance, phasers can enforce barriers or producer-consumer synchronization schemes among all or subsets of the running tasks. Implementations are found in modern languages such as Habanero Java. Phasers essentially associate phases to individual tasks and use their runtime values to restrict possible concurrent executions. Unbounded phases may result in infinite transition systems even in the case of programs only creating finite numbers of tasks and phasers. We introduce an exact gap-order based procedure that always terminates when checking control reachability for programs generating bounded numbers of coexisting tasks and phasers. We also show verifying plain reachability is undecidable even for programs generating few tasks and phasers. We then explain how to turn our procedure into a sound analysis for checking plain reachability (including deadlock freedom). We report on preliminary experiments with our open source tool.**

***Index Terms*—phasers, safety verification, dynamic synchronization, collective synchronization, Point-to-point synchronization, model checking**

I. Introduction

We focus on safety verification of programs using *phasers* for task synchronization [1]–[3]. This sophisticated construct dynamically unifies collective and point-to-point synchronizations. For instance, it allows for dynamic registration and deregistration of tasks allowing for a more balanced usage of the computing resources when compared to static producer-consumer or barrier constructs [4]. The construct can be added to any parallel programming language with a shared address space. For instance, it can be found in Habanero Java [3], an extension of the Java programming language. Phasers build on the clock construct from the X10 programming language [1]. They can be created dynamically and spawned tasks may get registered or deregistred at runtime.

Intuitively, each phaser associates two phases (hereafter *wait* and *signal* phases) to each registered task. Apart from creating phasers and registering each other to them, tasks can individually issue `wait` and `signal` commands to a phaser they are registered to. Intuitively, `signal` commands are used to inform other registered tasks the issuing task is done with its *signal* phase. The command is non-blocking. It increments the *signal* phase associated to the issuing task on the given phaser. The `wait` command is instead used to check whether all registered tasks are done with (i.e., have a *signal* phase that is strictly larger than) the issuing task's *wait* phase. This command may get blocked by a task that did not yet finish the corresponding phase. Unlike classical barriers, phasers need not force registered tasks to wait for each other at each single phase. Instead they allow them to proceed with the following phases (by issuing `signal` commands), or even to exit the construct by deregistering from the phaser. Such dynamic behavior allows for better load balancing and performance, but comes at the price of making it easy to introduce programming mistakes such as assertion violations, race conditions, runtime errors and, in the important situation where wait and signal commands are decoupled for maximum flexibility, deadlocks. We summarize our contributions in this work:

- We propose an operational model based on [2], [3], [5].
- We show undecidability of checking deadlock-freedom for programs with fixed numbers of tasks and phasers.
- We describe an exact gap-order based symbolic verification procedure for checking control state reachability (as in assertion violations, race conditions or runtime errors) and plain reachability (as in checking deadlock freedom).
- We show termination of the procedure for control state reachability when numbers of tasks and phasers are fixed.
- We describe how to turn the procedure into a sound over-approximation for plain reachability.
- We report on our preliminary experiments with our open source tool.

Related work. We are not aware of automatic formal verification works that focus on constructs allowing for such a degree of dynamic parallelism. Unlike [6], we focus on fully automatic verification and consider the richer and more challenging phaser construct. The work of [5] considers the dynamic verification of phaser programs and can therefore only reason about particular program inputs and runs. The work in [7] uses Java Path Finder [8] to explore several runs, but still for one concrete input at a time. The works in [9], [10] target gap-order systems. Although phaser programs share some of their properties (larger gaps can do more), the results in [9], [10] do not apply since gap-order systems crucially forbid exact increments.

Outline. We describe a phaser program and recall some preliminaries in Sections II and III. This is followed in Section IV by a formal description of phaser programs and

This work is partially supported by the CENIIT research organization.

of the properties we want to check. We also establish the undecidability of checking deadlock freedom. We introduce a gap-order based symbolic representation in Section V and describe in Section VI a simple verification procedure. We then show decidability of checking control state reachability and introduce a relaxation procedure for checking plain reachability. Finally, we report on our experiments and conclude the work. Descriptions of the proofs can be found in [11].

II. Motivating example

The program listed in Fig. (1) uses Boolean shared variables $\mathtt{B} = \{\mathtt{a}, \mathtt{b}, \mathtt{done}\}$. A *main* task creates two phasers (lines 5 and 6). When creating a phaser, the task gets automatically registered to it. The main task also creates three other task instances (lines 9, 10 and 11). Several tasks can be registered to several phasers. When a task t is registered to a phaser p, a pair of numbers $(wait_p^t, sig_p^t)$, each in $\mathbb{N} \cup \{+\infty\}$, is associated to the couple (t, p). The pair represents the individual *wait* and *signal* phases of task t on phaser p.

Registration of a task to a phaser can occur in one of three modes: SIG_WAIT, WAIT and SIG. In SIG_WAIT mode, a task may issue both `signal` and `wait` commands. In WAIT mode, a task may only issue `wait` commands on the phaser. Finally, when registered in SIG mode, a task may only issue `signal` commands. Issuing a `signal` command by a task on a phaser results in the task incrementing its signal phase associated to the phaser. This command is non-blocking. On the other-hand, issuing a `wait` command by a task on a phaser p will block until **all** tasks registered on p exhibit signal values on p that are strictly larger than the wait value of the issuing task on phaser p. In this case, the wait phase of the issuing task is incremented. Intuitively, a signal command allows the issuing task to state other tasks need not wait for it to complete its signal phase. In retrospect, a `wait` command allows a task to make sure all registered tasks have moved past its wait phase.

Upon creation of a phaser, wait and signal phases are initialized to 0 (except in WAIT mode where the signal phase is instead initialized to $+\infty$ in order to not block other waiters). The only other way a task may get registered to a phaser is if an already registred task does register it in the same mode (or in WAIT or SIG if the registrar is registered in SIG_WAIT). In this case, wait and signal phases of the newly registered task are initialized to those of the registrar. Tasks are therefore dynamically registered (e.g., lines 9-11). They can also dynamically deregister themselves (e.g., lines 25-26);

In this example, two producers and one consumer are synchronized using two phasers. The consumer requires the two producers to be ahead of it (wrt. the phaser main pointed to with `prod`) in order for it to consume their respective products. At the same time, the consumer needs to be ahead of both producers (wrt. the phaser main pointed to with `cons`) in order for these to produce their pair of products. It should be clear that phasers can be used as barriers for synchronizing dynamic subsets of concurrent tasks. Observe producers need not, in general, proceed in a lock step fashion. Producers may produce many items before consumers “catch up”.

We are interested in checking: (a) control state reachability as in assertions (e.g., line 44), race conditions (e.g., mutual exclusion of lines 20 and 49) or runtime errors (e.g., signaling a dropped phaser), and (b) plain reachability as in deadlocks (e.g., a producer at line 23 and a consumer at line 50 with equal phases). Intuitively, both problems concern themselves with the reachability of target sets of program configurations. The difference is that control state reachability defines the targets with the states of the tasks (their control locations and whether they are registered to some phasers). Plain reachability can, in addition, use values or relations between values of involved phases. Observe that control state reachability depends on the values of the actual phases, but these values are not used to define the target sets. For example, assertions are expressed as predicates over Boolean variables (e.g., line 44). Establishing such an assertion requires capturing the constraints imposed by the phasers on the program behaviors.

Our work proposes a sound and complete algorithm for checking control state reachability in case a bounded number of tasks and phasers are generated. The algorithm can handle arbitrarily large phases, e.g., generated using nested signaling loops. The algorithm starts from a symbolic representation of all bad configurations and successively computes sets of predecessor configurations. We show termination based on a well-quasi-ordering argument that imposes restrictions on what can be expressed with our symbolic representation. For instance putting upper bounds on differences between phases is forbidden. Deadlock configurations cannot be faithfully captured with such restricted representations. Intuitively, a deadlocked configuration will have a cycle where each involved task is waiting for the task to its right but where the wait phase of each task equals the signal phase of the task it is waiting for. We show the problem of checking deadlock freedom to be undecidable even for programs only generating a bounded number of tasks and phasers. We explain how to turn our verification algorithm into a sound but incomplete procedure for checking deadlock-freedom. Precision can then be augmented on demand to eliminate false positives.

III. Preliminaries

We use $\mathbb{N}$ and $\mathbb{Z}$ for natural and integer numbers respectively. We write $A \uplus B$ to mean the union of disjoint sets A and B. We let $\mathtt{Pfn}\,(A, B)$ be the set of partial functions from A to B and use $\varnothing_A$ for the empty function over A, i.e., $\varnothing_A(a)$ is undefined (written $\varnothing_A(a) \uparrow$) for all $a \in A$. Given function $g \in \mathtt{Pfn}\,(A, B)$ we write $g(a) \downarrow$ to mean that $g(a)$ is defined and write $g \setminus \{a\}$ to mean the restriction of g to the domain $A \setminus \{a\}$. We write $g[a \leftarrow b]$ for the function that coincides with g on A except for a that is sent to b. We abuse notation and let, for pairwise different $\{a_i \mid i \in I\}$, $g[\{a_i \leftarrow b_i \mid i \in I\}]$ mean the function that coincides with g on A except for each a_i that is sent to the corresponding b_i. We sometimes write a function g as a set $\{a \mapsto g(a) \mid a \in A\}$. It is then implicitly undefined outside of A.

```
1  bool a, b, done;
2  main()
3  {
4    done = false;
5    prod = newPhaser(SIG_WAIT);
6    cons = newPhaser(SIG_WAIT);
7    cons.signal();
8
9    asynch(aProducer, prod(SIG), cons(WAIT));
10   asynch(bProducer, prod(SIG), cons(WAIT));
11   asynch(abConsumer, prod(WAIT), cons(SIG));
12
13   prod.drop();
14   cons.drop();
15 }
16
17 aProducer(p(SIG), c(WAIT))
18 {
19   c.wait();
20   while(¬done){
21     a = true;
22     p.signal();
23     c.wait();
24   };
25   p.drop();
26   c.drop();
27 }
```

```
28  bProducer(p(SIG), c(WAIT))
29  {
30    c.wait();
31    while(¬done){
32      b = true;
33      p.signal();
34      c.wait();
35    };
36    p.drop();
37    c.drop();
38  }
39
40 abConsumer(p(WAIT), c(SIG))
41 {
42    while(¬done){
43     p.wait();
44     assert(a ∧ b);
45     a = false;
46     b = false;
47
48     if(ndet())
49       done = true;
50     c.signal();
51   };
52   c.drop();
53   p.drop();
54 }
```

Fig. 1. Two producers and one consumer are synchronized using two phasers. In this construction, the consumer requires both producers to be ahead of it (wrt. the `prod` phaser) in order for it to consume their respective products. At the same time, the consumer needs to be ahead of both producers (wrt. the `cons` phaser) in order for these to be able to produce their pair of products.

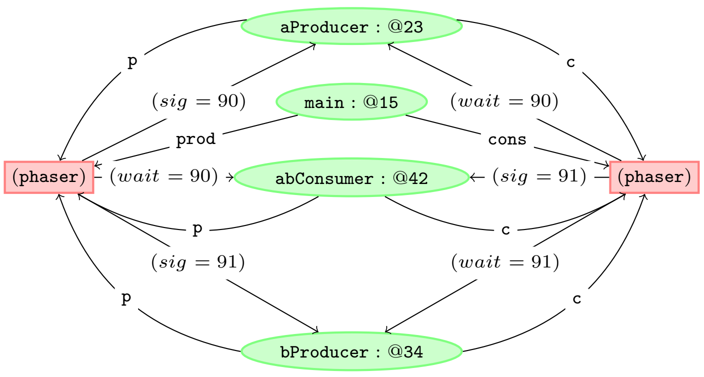


Fig. 2. Possible wait and signal phase values for Fig. (1). Observe that there is no a priori bound on the values of the different wait and signal phases. In this example, the difference between signal and wait phases is bounded. This is not always the case in general.

IV. Language

A program may use a set B of shared Boolean variables and a set V of local phaser variables:

```
prg   ::=  bool b_1, ..., b_|B|;
           task_1(v_1_1, ..., v_k_1) {stmt_1}
           ...
           task_n(v_1_n, ..., v_k_n) {stmt_n}

stmt  ::=    v = newPhaser() | asynch(task, v_1, ..., v_k)
           | v.drop() | v.signal() | v.wait() | exit
           | stmt; stmt | b = cond | assert(cond)
           | while(cond) {stmt} | if(cond) {stmt}

cond  ::=  ndet() | true | false | b | cond ∨ cond
           | cond ∧ cond | ¬cond
```

A program consists in a set of tasks T. A task is declared with $\mathtt{task}(\mathtt{v}_1, \ldots, \mathtt{v}_k)\,\{\mathtt{stmt}\}$ where $\mathtt{v}_1, \ldots \mathtt{v}_k$ are phaser variables that are local to the declared task. A task can also create a new phaser with $\mathtt{v} = \mathtt{newPhaser}()$ and store the identifier of the phaser in a local variable v. We let V be the union of all local phaser variables. When creating a phaser, a task gets registered to it. To simplify our description, we will assume all registrations to be in SIG_WAIT mode. Including the other modes is a matter of changing the initial phase values at registration and of statically ensuring the issued commands respect the registration mode. A task can deregister itself from a phaser referenced by a variable v with v.drop(). It can also issue signal or wait commands on a phaser on which it is registered and that is referenced by v. A task can spawn another task with $\mathtt{asynch}(\mathtt{task}, \mathtt{v}_1, \ldots, \mathtt{v}_n)$. The issuing task registers the spawned task to the phasers it points to with $\mathtt{v}_1, \ldots, \mathtt{v}_n$. The issuing task need not wait for the spawned task and may directly continue its execution.

Assume a phaser program $\mathtt{prg} = (\mathtt{B}, \mathtt{V}, \mathtt{T})$. We inductively define the finite set S of control sequences as follows. S is the smallest set containing: (i) suffixes of each "$\mathtt{stmt}_i$" appearing in some "$\mathtt{task}_i(\mathtt{v}_{1_i}, \ldots, \mathtt{v}_{k_i})\,\{\mathtt{stmt}_i\}$"; and (ii) suffixes of "$\mathtt{stmt}_i; \mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt_i}\}; \mathtt{stmt}_j$" (respectively "$\mathtt{stmt}_i; \mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt_i}\}$") for each "$\mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt_i}\}; \mathtt{stmt}_i$" (respectively "$\mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt_i}\}$") in S; and (iii) suffixes of "$\mathtt{stmt}_i; \mathtt{stmt}_j$" (respectively "$\mathtt{stmt}_i$") for each "$\mathtt{if}(\mathtt{cond})\,\{\mathtt{stmt_i}\}; \mathtt{stmt}_j$" (respectively "$\mathtt{if}(\mathtt{cond})\,\{\mathtt{stmt_i}\}$") appearing in S. We write s to mean some control sequence in S, and hd(s) and tl(s) to respectively mean the head and the tail of the sequence s.

A. Semantics.

A configuration c of $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ is a tuple $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ where:

- $\mathcal{T}$ is the current finite set of task identifiers. We let t, u range over the values in $\mathcal{T}$.
- $\mathcal{P}$ is the current finite set of phaser identifiers. We let p, q range over the values in $\mathcal{P}$.
- $\boldsymbol{bv} : \texttt{B} \to \{\texttt{true}, \texttt{false}\}$ is a total mapping that associates a value to each $\texttt{b} \in \texttt{B}$.
- $\boldsymbol{pc} : \mathcal{T} \to \texttt{S}$ is a total mapping that associates tasks to their remaining sequences (i.e., control location).
- $\boldsymbol{pv} : \mathcal{T} \to \texttt{Pfn}(\texttt{V}, \mathcal{P})$ is a total mapping that associates, to each task identifier in $\mathcal{T}$, a partial mapping from the local phaser variables $\texttt{V}$ to phaser identifiers $\mathcal{P}$. It captures the values of the phaser variables $\texttt{V}$ of each task.
- $\boldsymbol{\varphi} : \mathcal{P} \to \texttt{Pfn}(\mathcal{T}, \mathbb{N}^2)$ is a total mapping that associates to each phaser $p \in \mathcal{P}$ a partial mapping $\boldsymbol{\varphi}(p)$ that is defined exactly on the identifiers of the tasks registered to p. For such a task t, $\boldsymbol{\varphi}(p)(t)$ is the pair $(wait_p^t, sig_p^t)$ representing wait and signal values of t on p.

The set of tasks $\mathcal{T}$ is altered by $\texttt{asynch}(\texttt{task}, \texttt{v}_1, \ldots, \texttt{v}_n)$ and $\texttt{exit}$ statements (rules $(\texttt{asynch})$ and $(\texttt{exit})$ in Fig.(3)). The set of phasers $\mathcal{P}$ is updated upon creation of new phasers (rule $(\texttt{newPhaser})$ in Fig.(3)). The mapping $\boldsymbol{pv}$ associates values to program phaser variables. Accessing variables with undefined values, or phasers to which the task is not currently registered, leads to runtime errors (rule $(\texttt{runtime error})$). The total mapping $\boldsymbol{\varphi}$ captures states of phasers. It associates to each phaser identifier p in $\mathcal{P}$ a partial mapping $\boldsymbol{\varphi}(p)$. This partial mapping is defined for a task identifier $t \in \mathcal{T}$ (i.e., $\boldsymbol{\varphi}(p)(t) \downarrow$) iff the task t is registered to the phaser p. In this case, $\boldsymbol{\varphi}(p)$ gives the waiting phase $wait_p^t$ and the signaling phase sig_p^t of the task t on the phaser p. Initially, a unique "main" task $\texttt{t}_0$ starts executing its $\texttt{stmt}_{main}$ with no phasers. $\boldsymbol{\varphi}$ is the empty function with an empty domain $\varnothing_\emptyset$. After a task t executes a $\texttt{v := newPhaser()}$ statement (rule $(\texttt{newPhaser})$ in Fig.(3)), a new phaser p is associated to the variable $\texttt{v}$ using $\boldsymbol{pv}$ and $\boldsymbol{\varphi}(p)$ becomes the partial function $\{t \mapsto (0,0)\}$. The initial configuration is $c_{init} = (\{\texttt{t}_0\}, \{\}, \boldsymbol{bv}_{\texttt{false}}, \{\texttt{t}_0 \mapsto \texttt{stmt}\}, \varnothing, \varnothing)$, where a "main" task with identifier $\texttt{t}_0$ and code $\texttt{stmt}$ is the unique initial task. No phasers are present in the initial configuration, and all Boolean variables are mapped to $\texttt{false}$.

Given two configurations c and c' with $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$, we write $c \xrightarrow{t} c'$ if there is a task $t \in \mathcal{T}$ such that one of the rules in Fig.(3) holds. We use $\xrightarrow{*}$ for the reflexive transitive closure of $\to$ and write $c \xrightarrow{*} c'$ to mean that c' is reachable from c. A configuration is said reachable if it is reachable from the initial configuration c_{init}.

1) Control-state reachability: Checking the possibility of assertion violations, of runtime errors and of race conditions amounts to checking reachability of configurations respectively in $\texttt{badConfs}^{(n,p)}_{\texttt{assert}}$, $\texttt{badConfs}^{(n,p)}_{\texttt{runtime}}$ and in $\texttt{badConfs}^{(n,p)}_{\texttt{race}}$ for some number of tasks n and number of phasers p. We introduce in Section V a complete procedure for checking reachability of such sets of configurations and show it to be sound for programs with fixed upper bounds on numbers of generated phasers and tasks.

2) Deadlocks as in plain reachability: We are also interested in checking the possibility of deadlocks. For this we need to define the notion of a blocked task. Assume in the following a configuration $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$.

Definition 1 (Blocked)**.** *A task $t \in \mathcal{T}$ is blocked at phaser $p \in \mathcal{P}$ by task $u \in \mathcal{T}$ if $\texttt{hd}(\boldsymbol{pc}(\texttt{t})) = \texttt{v.wait}()$ with $\boldsymbol{pv}(t)(\texttt{v}) = p$ and $\boldsymbol{\varphi}(p)(t) = (wait_p^t, \_)$ when $\boldsymbol{\varphi}(p)(u) = (\_, sig_p^u)$ and $sig_p^u \le wait_p^t$.*

Intuitively, a task t is blocked by a task u if it cannot finish its $\texttt{wait}$ command on some phaser because it is waiting for task u that did not issue enough $\texttt{signal}$ commands on the same phaser.

Definition 2 (Deadlock)**.** *$(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ is a deadlock configuration if each task of a non empty subset $\mathcal{U} \subseteq \mathcal{T}$ is blocked by some task in $\mathcal{U}$.*

Theorem 1 (Deadlock-Freedom)**.** *It is undecidable in general, even for programs with only three phasers and four tasks, to check for deadlock-freedom.*

The idea of the proof is to encode the reachability problem of any given 3-counters reset-VAS (vector addition system with reset arcs) as the reachability problem of a configuration with a cycle involving three phasers and three tasks (in addition to the main task). Indeed, reachability of configuration $(s_F, 0, 0, 0)$ (three counters with zero values at some control location s_F) is undecidable for reset-VASs. The idea then is to spawn three tasks and as many phasers. The value of each counter is captured with the difference between the signal and the wait of a pair of tasks on one phaser. Resets are encoded by asking a task to drop a phaser and exit and spawning a new task. The encoding ensures that a deadlock is reached exactly when the vector addition system reaches configuration $(s_F, 0, 0, 0)$. (See [11] for more details.)

V. Symbolic verification of phaser programs

We briefly introduce gap-order constraints and use them to define a symbolic representation (hereafter constraints) that we use in Section VI for checking reachability.

A. Gap-order constraints and graphs [9], [10], [12], [13].

Gap-order constraints can be regarded as a particular case of the octagons or the *unit two variables per inequality* (utvpi) constraints. Assume in this section that x and y are integer variables and that k is an integer constant. We use X and Y to mean finite sets of integer variables. A valuation val is a total function $X \to \mathbb{Z}$. Valuations are implicitly extended to preserve constants (i.e. $val(k) = k$ for any $k \in \mathbb{Z}$). A *gap-order clause* δ over X is an inequality of the form $a - b \ge k$ where $a, b \in X \cup \{0\}$. A *gap-order constraint* Δ over X is a finite conjunction of gap-order clauses over the same set X. Observe that $(x = y + 2 \ \wedge \ y \le 5)$ is essentially a gap-order constraint because it can be equivalently rewritten as

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{v} := \mathtt{newPhaser}() \;\wedge\; p \notin \mathcal{P} \;\wedge \\ \mathcal{P}' = \mathcal{P} \cup \{p\} \;\wedge\; \boldsymbol{pv}' = \boldsymbol{pv}[t \leftarrow \boldsymbol{pv}(t)[\mathtt{v} \leftarrow p]] \\ \wedge\, \boldsymbol{\varphi}' = \boldsymbol{\varphi}[p \leftarrow \{t \mapsto (0,0)\}]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T},\mathcal{P}',\boldsymbol{bv},\boldsymbol{pc}[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{pv}',\boldsymbol{\varphi}'\right)}\ (\mathtt{newPhaser})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{v.signal}() \;\wedge \\ \boldsymbol{pv}(t)(\mathtt{v}) = p \;\wedge\; \boldsymbol{\varphi}(p)(t) = (wait_p^t, sig_p^t) \;\wedge \\ \boldsymbol{\varphi}' = \boldsymbol{\varphi}\Big[p \leftarrow \boldsymbol{\varphi}(p)\Big[t \leftarrow (wait_p^t, 1 + sig_p^t)\Big]\Big]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{\varphi}'\right)}\ (\mathtt{signal})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{assert}(\mathtt{cond}) \;\wedge \\ \boldsymbol{bv}(\mathtt{cond}) = \mathtt{true}\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} (\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{pv},\boldsymbol{\varphi})}\ \begin{pmatrix}\mathtt{assert.} \\ \mathtt{ok}\end{pmatrix}$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{v.drop}() \;\wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \;\wedge \\ \boldsymbol{\varphi}(p)(t)\downarrow \;\wedge\; \boldsymbol{\varphi}' = \boldsymbol{\varphi}[p \leftarrow \boldsymbol{\varphi}(p)[t \leftarrow \uparrow]]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{pv},\boldsymbol{\varphi}'\right)}\ (\mathtt{drop})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{asynch}(\mathtt{task},\mathtt{v}_1,\ldots\mathtt{v}_\mathtt{k})\{\mathtt{s}_1\} \;\wedge\; \mathtt{paramOf}(\mathtt{task}) = (\mathtt{w}_1,\ldots\mathtt{w}_k) \;\wedge \\ \text{for each } i : 1 \leq i \leq k.\ \boldsymbol{pv}(t)(\mathtt{v}_i) = p_i \;\wedge\; \boldsymbol{\varphi}(p_i)\downarrow \;\wedge \\ u \notin \mathcal{T} \;\wedge\; \boldsymbol{pv}' = \boldsymbol{pv}[u \leftarrow \{\mathtt{w}_i \mapsto \boldsymbol{pv}(t)(\mathtt{v}_i) \mid 1 \leq i \leq k\}] \;\wedge\; \boldsymbol{pc}' = \boldsymbol{pc}[u \leftarrow \mathtt{s}_1] \;\wedge \\ \boldsymbol{\varphi}' = \boldsymbol{\varphi}[\{p_i \leftarrow \boldsymbol{\varphi}(p_i)[u \leftarrow \boldsymbol{\varphi}(p_i)(t)] \mid \boldsymbol{pv}(t)(\mathtt{v}_i) = p_i \text{ for } p_i \in \mathcal{P} \text{ and } 1 \leq i \leq k\}]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}'[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{pv},\boldsymbol{\varphi}'\right)}\ (\mathtt{asynch})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{v.wait}() \;\wedge\; \boldsymbol{pv}(t)(\mathtt{v}) = p \;\wedge\; \boldsymbol{\varphi}(p)(t) = (wait_p^t, sig_p^t) \;\wedge \\ \forall u \in \mathcal{T}.\ \Big(\boldsymbol{\varphi}(p)(u) = (wait_p^u, sig_p^u) \Rightarrow wait_p^t < sig_p^u\Big) \;\wedge \\ \boldsymbol{\varphi}' = \boldsymbol{\varphi}\Big[p \leftarrow \boldsymbol{\varphi}(p)\Big[t \leftarrow (1 + wait_p^t, sig_p^t)\Big]\Big]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T},\mathcal{P},\boldsymbol{pv},\boldsymbol{pc}[t \leftarrow \mathtt{tl}(\boldsymbol{pc}(t))],\boldsymbol{\varphi}'\right)}\ (\mathtt{wait})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{exit} \;\wedge\; \boldsymbol{pv}' = \boldsymbol{pv} \setminus \{t\} \;\wedge\; \boldsymbol{pc}' = \boldsymbol{pc} \setminus \{t\} \;\wedge \\ \boldsymbol{\varphi}' = \boldsymbol{\varphi}[\{p \leftarrow (\boldsymbol{\varphi}(p) \setminus \{t\}) \mid p \in \mathcal{P}\}]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \xrightarrow{t} \left(\mathcal{T} \setminus \{t\},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv}',\boldsymbol{\varphi}'\right)}\ (\mathtt{exit})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{assert}(\mathtt{cond}) \;\wedge \\ \boldsymbol{bv}(\mathtt{cond}) = \mathtt{false}\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \in \mathtt{badConfs}_{\mathtt{assert}}^{(|\mathcal{T}|,|\mathcal{P}|)}}\ \begin{pmatrix}\mathtt{assert.} \\ \mathtt{fault}\end{pmatrix}$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{s} \;\wedge (\mathtt{s} = \mathtt{v.drop}() \;\vee\; \mathtt{s} = \mathtt{v.signal}() \\ \vee\; \mathtt{s} = \mathtt{v.wait}() \;\vee\; \mathtt{s} = \mathtt{asynch}(\mathtt{task},\ldots,\mathtt{v},\ldots)\ ) \\ \wedge\; (\boldsymbol{pv}(t)(\mathtt{v})\uparrow \;\vee\; \boldsymbol{\varphi}(\boldsymbol{pv}(t)(\mathtt{v}))(t)\uparrow)\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \in \mathtt{badConfs}_{\mathtt{runtime}}^{(|\mathcal{T}|,|\mathcal{P}|)}}\ \begin{pmatrix}\mathtt{runtime} \\ \mathtt{error}\end{pmatrix}$$

$$\frac{\begin{array}{c}\{t_0,\ldots t_n\} \subseteq \mathcal{T} \wedge \{p_0,\ldots p_n\} \subseteq \mathcal{P} \;\wedge \\ \forall\ i : 0 \leq i \leq n.\ \mathtt{hd}(\boldsymbol{pc}(t_i)) = \mathtt{v}_i\mathtt{.wait}() \;\wedge \\ \boldsymbol{pv}(t_i)(\mathtt{w}_i) = p_{(i+1)\%n} \wedge \boldsymbol{pv}(t_i)(\mathtt{v}_i) = p_i \;\wedge \\ wait_{p_{(i+1)\%n}}^{t_i} \geq sig_{p_{(i+1)\%n}}^{t_{(i+1)\%n}}\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \in \mathtt{badConfs}_{\mathtt{deadlock}}^{(|\mathcal{T}|,|\mathcal{P}|)}}\ (\mathtt{deadlock})$$

$$\frac{\begin{array}{c}\mathtt{hd}(\boldsymbol{pc}(t)) = \mathtt{b} := \mathtt{cond} \;\wedge \mathtt{hd}(\boldsymbol{pc}(u)) = \mathtt{s}' \wedge\; t \neq u \;\wedge \\ \left(\begin{array}{c}\mathtt{s}' = \mathtt{b} := \mathtt{cond}' \vee \mathtt{b} \text{ appears in } \mathtt{cond}' \text{ and} \\ \left(\begin{array}{c}(\mathtt{s}' = \mathtt{if}(\mathtt{cond}')\,\{\mathtt{stmt}\}\vee \\ \mathtt{s}' = \mathtt{while}(\mathtt{cond}')\,\{\mathtt{stmt}'\}\vee \\ \mathtt{s}' = \mathtt{assert}(\mathtt{cond}') \vee \mathtt{s}' = \mathtt{b}' = \mathtt{cond}')\end{array}\right)\end{array}\right)\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\varphi}) \in \mathtt{badConfs}_{\mathtt{race}}^{(|\mathcal{T}|,|\mathcal{P}|)}}\ (\mathtt{race})$$

Fig. 3. Operational semantics of phaser statements.

the conjunction $(x - y \geq 2 \;\wedge\; y - x \geq -2 \;\wedge\; 0 - y \geq -5)$. Given a gap-order constraint Δ over X and a valuation $val : X \to \mathbb{Z}$, we write $val \models \Delta$ to mean that $val(a) - val(b) \geq k$ holds for each gap-order clause $\delta : a - b \geq k$ appearing in Δ. We let $Sat(\Delta)$ be the set $\{val : X \to \mathbb{Z} \mid val \models \Delta\}$.

A gap-order graph (or graph for short) $\wp$ over X is a graph (V, E) with vertices $V = X \cup \{0\}$ where edges in E are of the form $a \xrightarrow{k} b$ with $a, b \in V$ and weight k in $\mathbb{Z} \cup \{-\infty, +\infty\}$. We let $\mathtt{varsOf}(\wp) = X$. Given a gap-constraint Δ over X, we can build the graph $\mathtt{graphOf}\,(\Delta)$ with vertices $X \cup \{0\}$ and where E only contains a representative $a \xrightarrow{k} b$ edge for each clause $a - b \geq k$ appearing in Δ. A valuation $val : X \to \mathbb{Z}$ satisfies a graph $\wp = (V, E)$ (written $val \models \wp$) iff $val(a) - val(b) \geq k$ for each $a \xrightarrow{k} b \in E$. We let $Sat(\wp)$ be the set $\{val : X \to \mathbb{Z} \mid val \models \wp\}$. Clearly, $Sat(\mathtt{graphOf}\,(\Delta)) = Sat(\Delta)$. The closure $\mathtt{clo}\,(\wp)$ of a graph $\wp = (V, E)$ is the unique complete graph with the same vertices V and where $a \xrightarrow{k'} b$ is an edge of $\mathtt{clo}\,(\wp)$ iff $k' \in \mathbb{Z} \cup \{-\infty, +\infty\}$ is the least upper bound of all weight-sums for any path in $\wp$ from a to b. Closure allows us to deduce $(0 - x \geq -7)$ from $(y - x \geq -2 \wedge 0 - y \geq -5)$. The result of the closure procedure is a special graph $\wp_{\mathtt{false}}$ denoting the graph without any satisfying valuation each time a weight k=$+\infty$ is generated. The closure of a graph can be computed in polynomial time and we get $Sat(\mathtt{clo}\,(\wp)) = Sat(\wp)$. We define the *degree* of a graph $\wp$ (written $\mathtt{degreeOf}(\wp)$) to be 0 if no edge in $\mathtt{clo}\,(\wp)$ has a negative weight apart from $-\infty$. Otherwise, $\mathtt{degreeOf}(\wp)$ is the largest natural $k \in \mathbb{N}$ such that there is an edge in $\mathtt{clo}\,(\wp)$ with weight $-k$. For instance, the degree of the graph resulting from $(x - y \geq 2 \wedge y - x \geq -4)$ is 4. We systematically close all manipulated graphs and write $\mathcal{G}(X)$ for the set of closed graphs over X. Given a graph $\wp$, we write $\wp[x/y]$ to mean the graph obtained by replacing the vertex x by the vertex y. We abuse notation and write $\wp[\{x_i/y_i \mid i \in I\}]$, for pairwise different x_i elements to mean the simultaneous application of the individual substitutions. For a set of variables Y, we write $\wp \ominus Y$ to mean the graph obtained by removing the variables in Y from the vertices of $\wp$. Given two closed graphs $\wp$ and $\wp'$ over the same X, we write $\wp \sqsubseteq_{\mathcal{G}} \wp'$ to mean that each directed edge in $\wp$ is labeled with a larger weight in $\wp'$. As a result, $Sat(\wp') \subseteq Sat(\wp)$. Finally, we write $\wp \owedge \wp'$ to mean the closure of the graph obtained with merging the two sets of vertices and edges. As a result, $Sat(\wp \owedge \wp') = Sat(\wp) \cap Sat(\wp')$.

B. Constraints as a symbolic representation.

A constraint ϕ is a tuple $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ where the only difference with the definition of a configuration $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ is the adoption of a gap-order constraint $\boldsymbol{\gamma}$ instead of $\boldsymbol{\varphi}$. More specifically, $\boldsymbol{\gamma} : \mathcal{P} \to \cup_{\mathcal{U} \subseteq \mathcal{T}} \mathcal{G}(\cup_{t \in \mathcal{U}} \{\omega_p^t, \sigma_p^t\})$ is a total mapping that associates a gap-order graph to each phaser $p \in \mathcal{P}$. Intuitively, we use variables ω_p^t and σ_p^t to constrain in graph $\boldsymbol{\gamma}(p)$ possible values of both wait ($wait_p^t$) and signal (sig_p^t) phases of each task t registered to phaser p. As a result, we can check if task t is registered to phaser p according to graph $\wp = \boldsymbol{\gamma}(p)$ by checking if $\{\omega_p^t, \sigma_p^t\} \subseteq \texttt{varsOf}(\wp)$. We will write $\texttt{Reg}(p, \wp)$ to mean the set of tasks $\{t \mid \{\omega_p^t, \sigma_p^t\} \subseteq \texttt{varsOf}(\wp)\}$. We also write $\texttt{isReg}(t, p, \wp)$ for the predicate $t \in \texttt{Reg}(p, \wp)$. Observe that the language semantics impose that, for each phaser p and for any pair t, u of tasks in $\texttt{Reg}(p, \wp)$, the predicate $0 \leq wait_p^t \leq sig_p^u$ is an invariant. For this reason, we always safely strengthen, in any obtained $\boldsymbol{\gamma}(p) = \wp$, weights k in $\sigma_p^t \xrightarrow{k} \omega_p^u$, $\sigma_p^t \xrightarrow{k} 0$ and $\omega_p^t \xrightarrow{k} 0$ with $max(k, 0)$. The following definition helps us characterize configurations for which our procedure terminates.

Definition 3 (degree and freeness of constraints)**.** *A constraint $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ has as degree the largest degree among all its graphs $\boldsymbol{\gamma}(p)$ for $p \in \mathcal{P}$ if $\mathcal{P}$ is not empty and 0 otherwise. Furthermore, a constraint is said to be "free" if, for any $p \in \mathcal{P}$, the only edges in $\boldsymbol{\gamma}(p)$ with weights different from $-\infty$ are edges of the forms (i) $\sigma_p^t \xrightarrow{k_{(\sigma_p^t, \omega_p^u)}} \omega_p^u$, (ii) $\sigma_p^t \xrightarrow{k_{(\sigma_p^t)}} 0$, or (iii) $\omega_p^t \xrightarrow{k_{(\omega_p^t)}} 0$ for some $t, u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))$ and $k_{(\sigma_p^t, \omega_p^u)}, k_{(\sigma_p^t)}, k_{(\omega_p^t)} \in \mathbb{N}$*

Free constraints are only allowed to impose, for the same phaser, non-negative lower bounds on differences between signals and waits, between signals and 0, and between waits and 0. Like degree-0-constraints, free constraints are not allowed to put a positive upper bound on how much a signal is larger than a wait. Unlike degree-0-constraints, they are not allowed to put bounds on the differences among signal values, or among wait values. For instance a free constraint cannot impose $\sigma_p^t - \sigma_p^u = 0$ while a degree-0-constraint can. Intuitively, freeness does not oblige our verification procedure to maintain exact differences when firing "signal" or "wait" instructions, jeopardizing termination. This will be stated in Section VI.

C. Denotations of constraints.

Given a configuration $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ and a constraint $\phi = (\mathcal{T}', \mathcal{P}', \boldsymbol{bv}', \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}')$, we say that c satisfies ϕ, and write $c \models \phi$, if c satisfies (up to a renaming of the tasks and the phasers) conditions imposed by ϕ. More concretely, $c \models \phi$ if $\boldsymbol{bv} = \boldsymbol{bv}'$ and there are bijections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ such that: (i) $\boldsymbol{pc}(t) = \boldsymbol{pc}'(\tau(t))$ for each $t \in \mathcal{T}$; and (ii) $\pi(\boldsymbol{pv}(t)(\mathtt{v})) = \boldsymbol{pv}'(\tau(t))(\mathtt{v})$ for each $t \in \mathcal{T}$ and $\mathtt{v} \in \mathtt{V}$; and (iii) the renaming of tasks and phasers in $\boldsymbol{\varphi}$ wrt. τ and π satisfies $\boldsymbol{\gamma}$, i.e., (iii.a) for each $t \in \mathcal{T}$ and each $p \in \mathcal{P}$, $\boldsymbol{\varphi}(p)(t) \downarrow$ iff $\texttt{isReg}(\tau(t), \pi(p), \boldsymbol{\gamma}(\pi(p)))$, and (iii.b) for each $p' \in \mathcal{P}'$, $\wp(\bigwedge_{t' \in \texttt{Reg}(p', \boldsymbol{\gamma}(p'))}((\omega_{p'}^{t'}, \sigma_{p'}^{t'}) = \boldsymbol{\varphi}(\pi^{-1}(p'))(\tau^{-1}(t')))) \models \boldsymbol{\gamma}(p')$. We let $[\![\phi]\!]$ denote $\{c \mid c \models \phi\}$. Intuitively, $[\![(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})]\!]$ contains all configurations c with the same number of tasks and phasers and such that there are renamings of tasks and phasers that preserve in c the correspondence between $\boldsymbol{pc}$, $\boldsymbol{pv}$ and $\boldsymbol{\gamma}$. We write $[\![\Phi]\!]$, for a set Φ of constraints, to mean the union $\cup_{\phi \in \Phi} [\![\phi]\!]$. Given a program $(\mathtt{B}, \mathtt{V}, \mathtt{T})$, we can exactly characterize with a finite set of constraints all configurations involving n tasks and p phasers and satisfying the premises of rules ($\texttt{runtime error}$), ($\texttt{assert. fault}$), ($\texttt{race}$) and ($\texttt{deadlock}$) from Fig.(3).

Lemma 1 (Characterizing badness)**.** *Given a program* $(\mathtt{B}, \mathtt{V}, \mathtt{T})$ *and natural numbers (n, p), we can exhibit finite sets of constraints* $\texttt{badCstrs}_{\texttt{race}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}$ *and* $\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}$ *such that:*

$$\begin{aligned}
\texttt{badConfs}_{\texttt{race}}^{(n,p)} &= [\![\texttt{badCstrs}_{\texttt{race}}^{(n,p)}]\!] \\
\texttt{badConfs}_{\texttt{assert}}^{(n,p)} &= [\![\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}]\!] \\
\texttt{badConfs}_{\texttt{runtime}}^{(n,p)} &= [\![\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}]\!] \\
\texttt{badConfs}_{\texttt{deadlock}}^{(n,p)} &= [\![\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}]\!]
\end{aligned}$$

In addition, we can choose the constraints in $\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}$ *to be of degree* 0 *while those in* $\texttt{badCstrs}_{\texttt{race}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}$ *or in* $\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}$ *to be free.*

D. Entailment.

We say that a constraint $\phi = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ is weaker than a constraint $\phi' = (\mathcal{T}', \mathcal{P}', \boldsymbol{bv}', \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}')$, written $\phi \sqsubseteq \phi'$, to mean the following. First, the two constraints have the same number of phasers and tasks, agree on the values of the Boolean variables and, up to renamings, on the values of the phaser variables and on which tasks are registered to which phasers. Second, the constraints on the wait and signal values are stronger in ϕ' than in ϕ. More formally, $\phi \sqsubseteq \phi'$ if $\boldsymbol{bv} = \boldsymbol{bv}'$ and there are bijections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ s.t. for each $t \in \mathcal{T}$ and $p \in \mathcal{P}$ the following four conditions hold: (i) $\boldsymbol{pc}(t) = \boldsymbol{pc}'(\tau(t))$; and (ii) $\pi(\boldsymbol{pv}(t)(\mathtt{v})) = \boldsymbol{pv}'(\tau(t))(\mathtt{v})$; and (iii) $\pi(\texttt{Reg}(p, \boldsymbol{\gamma}(p))) = \texttt{Reg}(\pi(p), \boldsymbol{\gamma}'(\pi(p)))$; and (iv) $\boldsymbol{\gamma}(p) \sqsubseteq_{\mathcal{G}} \boldsymbol{\gamma}'(\pi(p)) \left[\left\{ \omega_{\pi(p)}^{\tau(t)} / \omega_p^t, \sigma_{\pi(p)}^{\tau(t)} / \sigma_p^t \mid t \in \texttt{Reg}(p, \boldsymbol{\gamma}(p)) \right\} \right]$. Clearly, $\phi \sqsubseteq \phi'$ implies $[\![\phi']\!] \subseteq [\![\phi]\!]$. We say that $\sqsubseteq$ is sound.

We can show that $\sqsubseteq$ is a well-quasi-order[1] over constraints of bounded degrees and involving fixed numbers of tasks and phasers since $\sqsubseteq_{\mathcal{G}}$ is itself a well-quasi-ordering over graphs of bounded degrees over a finite set of variables ([9], [12]).

Lemma 2 (WQO)**.** *Given $k, n, p \in \mathbb{N}$, the entailment relation $\sqsubseteq$ over the set of constraints of degree k involving at most n tasks and p phasers is a well-quasi-order.*

[1]A reflexive and transitive binary relation $\preceq$ is a well-quasi-order over a set A if there is no infinite sequence $a_0, a_1, \ldots$ of A elements s.t. $a_i \not\preceq a_j$ for all $i < j$.

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{v := newPhaser()}; \boldsymbol{pc}(t)] \ \wedge \\ \boldsymbol{pv}(t)(\texttt{v}) = p \ \wedge \ \boldsymbol{pv}' = \boldsymbol{pv}[t \leftarrow \boldsymbol{pv}(t)[\texttt{v} \leftarrow \uparrow]] \ \wedge \\ \left\{\omega_p^t \mapsto 0, \sigma_p^t \mapsto 0\right\} \models \boldsymbol{\gamma}(p) \ \wedge \ \boldsymbol{\gamma}' = \boldsymbol{\gamma} \setminus \{p\} \ \wedge \\ (\texttt{isReg}(u, p, \boldsymbol{\gamma}(p)) \implies u = t)\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} \left(\mathcal{T}, \mathcal{P} \setminus \{p\}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}'\right)} \ (\texttt{newPhaser I})$$

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{v := newPhaser()}; \boldsymbol{pc}(t)] \ \wedge \\ \boldsymbol{pv}(t)(\texttt{v}) = p \ \wedge \ \boldsymbol{pv}' = \boldsymbol{pv}[t \leftarrow \boldsymbol{pv}(t)[\texttt{v} \leftarrow q]] \ \wedge \\ \left\{\omega_p^t \mapsto 0, \sigma_p^t \mapsto 0\right\} \models \boldsymbol{\gamma}(p) \ \wedge \ \boldsymbol{\gamma}' = \boldsymbol{\gamma} \setminus \{p\} \ \wedge \\ (\texttt{isReg}(u, p, \boldsymbol{\gamma}(p)) \implies u = t)\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} \left(\mathcal{T}, \mathcal{P} \setminus \{p\}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}'\right)} \ (\texttt{newPhaser II})$$

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{v.signal()}; \boldsymbol{pc}(t)] \ \wedge \ \boldsymbol{pv}(t)(\texttt{v}) = p \ \wedge \ \texttt{isReg}(t, p, \boldsymbol{\gamma}(p)) \ \wedge \ \wp = \left(\boldsymbol{\gamma}(p) \oslash \texttt{graphOf}\left(\wedge_{u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))} (\sigma_p^t > \omega_p^u \geq 0)\right)\right) \\ \wedge \ \texttt{isSat}(\wp) \ \wedge \ \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\left(\wp\left[\sigma_p^t / \sigma\right] \oslash \texttt{graphOf}\left(\sigma_p^t = \sigma - 1\right)\right) \ominus \{\sigma\}\right)\right]\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}, \boldsymbol{\gamma}')} \ (\texttt{signal})$$

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{v.wait()}; \boldsymbol{pc}(t)] \ \wedge \ \boldsymbol{pv}(t)(\texttt{v}) = p \ \wedge \ \texttt{isReg}(t, p, \boldsymbol{\gamma}(p)) \ \wedge \ \wp = \left(\boldsymbol{\gamma}(p) \oslash \texttt{graphOf}\left(\wedge_{\{u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))\}} (\sigma_p^u \geq \omega_p^t > 0)\right)\right) \\ \wedge \ \texttt{isSat}(\wp) \ \wedge \ \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\left(\wp\left[\omega_p^t / \omega\right] \oslash \texttt{graphOf}\left(\omega_p^t = \omega - 1\right)\right) \ominus \{\omega\}\right)\right]\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}, \boldsymbol{\gamma}')} \ (\texttt{wait})$$

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{v.drop()}; \boldsymbol{pc}(t)] \ \wedge \ \boldsymbol{pv}(t)(\texttt{v}) = p \ \wedge \ \neg\texttt{isReg}(t, p, \boldsymbol{\gamma}(p)) \ \wedge \\ \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\boldsymbol{\gamma}(p) \oslash \texttt{graphOf}\left((\sigma_p^t \geq \omega_p^t \geq 0) \wedge_{u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))} (\sigma_p^u \geq \omega_p^t \geq 0) \wedge (\sigma_p^t \geq \omega_p^u \geq 0)\right)\right)\right]\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}, \boldsymbol{\gamma}')} \ (\texttt{drop})$$

$$\dfrac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{asynch}(\texttt{task}, \texttt{v}_1, \dots \texttt{v}_\texttt{k})\{\texttt{s}_1\}; \boldsymbol{pc}(t)] \ \wedge \ \texttt{paramOf}(\texttt{task}) = (\texttt{w}_1, \dots \texttt{w}_k) \ \wedge \ u \in \mathcal{T} \setminus \{t\} \ \wedge \\ \boldsymbol{pv}' = \boldsymbol{pv} \setminus \{u\} \ \wedge \ \boldsymbol{pc}(u) = \texttt{s}_1 \ \wedge \ \forall i : 1 \leq i \leq k.\ \boldsymbol{pv}(t)(\texttt{v}_i) = \boldsymbol{pv}(u)(\texttt{w}_i) = p_i \ \wedge \ (\texttt{isReg}(t, p_i, \boldsymbol{\gamma}(p_i)) \Leftrightarrow \texttt{isReg}(u, p_i, \boldsymbol{\gamma}(p_i))) \ \wedge \\ \wp_i = \left(\boldsymbol{\gamma}(p_i) \oslash \texttt{graphOf}\left(\omega_{p_i}^t = \omega_{p_i}^u \wedge \sigma_{p_i}^t = \sigma_{p_i}^u\right)\right) \ \wedge \ \texttt{isSat}(\wp_i) \ \wedge \ \boldsymbol{\gamma}_0 = \boldsymbol{\gamma} \ \wedge \ \boldsymbol{\gamma}_i = \boldsymbol{\gamma}_{i-1}\left[p_i \leftarrow \left(\wp_i \ominus \left\{\omega_{p_i}^u, \sigma_{p_i}^u\right\}\right)\right]\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} \left(\mathcal{T} \setminus \{u\}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}' \setminus \{u\}, \boldsymbol{pv}', \boldsymbol{\gamma}_n\right)} \ (\texttt{asynch})$$

$$\dfrac{\begin{array}{c}t \notin \mathcal{T} \ \wedge \ \boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \texttt{exit}] \ \wedge \ f \in \texttt{Pfn}\,(\texttt{V}, \mathcal{P}) \ \wedge \boldsymbol{pv}' = \boldsymbol{pv} \uplus \{t \mapsto f\} \ \wedge \\ \mathcal{Q} \subseteq \mathcal{P} \ \wedge \ \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[\left\{p \leftarrow \boldsymbol{\varphi}(p) \oslash \texttt{graphOf}\left((\sigma_p^t \geq \omega_p^t \geq 0) \wedge_{u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))} (\sigma_p^u \geq \omega_p^t \geq 0) \wedge (\sigma_p^t \geq \omega_p^u \geq 0)\right) \ \mid \ p \in \mathcal{Q}\right\}\right]\end{array}}{(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} \left(\mathcal{T} \cup \{t\}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}'\right)} \ (\texttt{exit})$$

Fig. 4. Derivation rules for computing $\texttt{pre}(t, \phi)$ for phaser statements as union of all $\{\phi' \mid \phi \overset{t}{\rightsquigarrow} \phi' \text{ with } \phi = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) \text{ and } t \in \mathcal{T}\}$. Derivations for other program statements are straightforward.

VI. Verification Procedure

```
Input: A program $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$, a set $\Phi_{bad}$ of pairwise
       $\sqsubseteq$-incomparable constraints, maximum upper bounds $t^\bullet$ and $p^\bullet$
       (in $\mathbb{N} \cup \{+\infty\}$) on coexisting tasks and phasers.
Output: A symbolic run to $\Phi_{bad}$ or the value unreachable
1  Initialize both Working and Visited to $\{(\phi, \phi) \mid \phi \in \Phi_{bad}\}$;
2  while there exists $(\phi, \tau) \in$ Working do
3      remove $(\phi, \tau)$ from Working;
4      let $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}) = \phi$;
5      if $|\mathcal{T}| > t^\bullet$ or $|\mathcal{P}| > p^\bullet$ then continue;
6      if $c_{init} \models \phi$ then return $\tau$;
7      foreach $t \in \mathcal{T}$ do
8          foreach $\phi' \in \texttt{pre}(t, \phi)$ do
9              if $\psi \not\sqsubseteq \phi'$ for all $(\psi, \_) \in$ Visited then
10                 Remove from Working and Visited each $(\psi, \_)$
                   for which $\phi' \sqsubseteq \psi$;
11                 Add $(\phi', \phi' \cdot t \cdot \tau)$ to both Working and Visited;
12 return unreachable ;
```

Procedure check($\texttt{prg}$,Φ_{bad},$t^\bullet$, $p^\bullet$), a simple working list procedure for checking constraints reachability.

We discuss in the following the procedure *check* depicted above and assume a program $\texttt{prg}$ and a set Φ_{bad} of constraints the reachability of which we want to check. Φ_{bad} can for example be any subset of $\texttt{badCstrs}^{(n,p)}_{\texttt{deadlock}}$ (degree 0) or of $\texttt{badCstrs}^{(n,p)}_{\texttt{assert}}$ (free) in case we want to check the possibility of a deadlock or of an assertion violation.

It is not difficult to show that $[\![\texttt{pre}(t, \phi)]\!]$ (obtained as described in Fig.(4)) coincides with $\{c' \mid c' \xrightarrow{t} c \text{ and } c \in [\![\phi]\!]\}$. Using the soundness of $\sqsubseteq$, we can show by induction the partial correctness of the procedure check($\texttt{prg}$,Φ_{bad},$+\infty$,$+\infty$).

Lemma 3 (Partial correctness)**.** *If* check($\texttt{prg}$,Φ_{bad},$+\infty$,$+\infty$) *returns* $\texttt{unreachable}$*, then* $c_{init} \overset{*}{\not\to} [\![\Phi_{bad}]\!]$*. If it returns a trace* $\phi_n \cdot t_n \cdots t_1 \cdot \phi_1$ *then there are* $c_n, \dots c_1$ *with* $c_n = c_{init}$*,* $c_1 \in [\![\Phi_{bad}]\!]$ *and* $c_i \xrightarrow{t_i} c_{i-1}$ *for* $i : 1 < i \leq n$*.*

Theorem 2 (Free termination)**.** check($\texttt{prg}$,Φ_{bad},$t^\bullet$,$p^\bullet$) *terminates for* $t^\bullet, p^\bullet \in \mathbb{N}$ *and free* Φ_{bad}*.*

Proof. Sketch. Freeness is preserved by the $\texttt{pre}$ computation (Fig.(4)). Suppose the procedure does not terminate. The infinite sequence of constraints passing the test at line 9 of the procedure violates well-quasi-orderness of $\sqsubseteq$ over free constraints with fixed numbers of tasks and phasers. $\square$

In order to check reachability of arbitrary constraints, we may need to force termination. We do this by soundly bounding the degree of generated constraints using a relaxation ρ_k. The relaxation $\rho_k((\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma}))$ replaces, in each graph $\boldsymbol{\gamma}(p)$, each weight k'' s.t. $k'' < -k$ with $-\infty$.

```
1 foreach $\phi'' \in \texttt{pre}(t, \phi)$ do
2     Let $\phi' = \rho_k(\phi'')$;
```

Fig. 5. Systematic relaxation

Theorem 3 (Forced termination). *Procedure* check($\mathtt{prg}$,Φ_{bad},$t^\bullet$,$p^\bullet$) *for* $t^\bullet, p^\bullet \in \mathbb{N}$*, with line 8 replaced by the lines of Fig. (5), is sound and guaranteed to terminate.*

Proof. Soundness is due to the validity of $\rho_k(\phi) \sqsubseteq \phi$ while the termination argument relies, similarly to Theorem (2), on well-quasi orederness of $\sqsubseteq$ on the set of constraints with bounded degree and fixed numbers of tasks and phasers. □

VII. Experimental Results

We report on experiments with our open source prototype *hjVerify* (https://gitlab.ida.liu.se/apv/hjVerify) for the verification of phaser programs. We conducted experiments on 12 different programs (some of which are from [5]). We considered both deadlock and assertion reachability problems. For each property, we considered correct and buggy versions. This gave 48 different instances with 2 to 3 phasers and 2 to 4 tasks (except for the parameterized case). Our tool uses global phaser and task variables as in [5]. We have experimented with adapting the view abstraction technique [14] to verify phaser programs generating arbitrary many tasks, i.e., parameterized verification where the number of phasers is fixed. (see [11] for more details.) We report on two parameterized examples. Experiments were conducted on a 2.9GHz processor with 8GB of memory.

program	property	safe / buggy	times
01.Loopless	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
02.Iterative averaging	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
03.Ordered phasers	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	13s / 1s
04.Conditional	deadlock:	ok / trace	2s / 1s
	assertion:	ok / trace	4s / 7s
05.Loop Synch.	deadlock:	ok / trace	178s / 145s
	assertion:	ok / trace	7s / 13s
06.Nested forks	deadlock:	ok / trace	2s / 1s
	assertion:	ok / trace	1s / 1s
07.Conditional membership	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	12s / 3s
08.Producer-consumer	deadlock:	ok / trace	37s / 222s
	assertion:	ok / trace	79s / 34s
09.Parameterized loopless	deadlock:	ok / trace	20s / 1s
	assertion:	ok / trace	67s / 1s
10.Parameterized iterative-averaging	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
11.Running-2	deadlock:	ok / trace	5s / 1s
	assertion:	ok / trace	26s / 4s
12.Running-3	deadlock:	ok / trace	4318s / 128s
	assertion:	ok / trace	18631s / 54s

Our implemented procedure does not eagerly concretize all task states as described in the predecessor computation of Section V. Instead we collect conditions on the phases of the tasks that did not take any action yet and lazily concretize them. Reported times for checking deadlocks are the sums of the times required to check reachability for each cycle. The prototype is only a proof of concept. For instance, the example (12.Running-3) is a variant of (11-Running-2) where a task instance is spawned twice leading to two symmetrical tasks (out of four). This required up to three orders of magnitude more time to check. We believe partial order reduction techniques would help here. Other relevant heuristics would be to make use of priority queues and to organize the minimal sets. All examples are available on the tool homepage.

VIII. Conclusion

We have proposed a gap-order based reachability analysis for phaser programs. We have showed our analysis to be exact and guaranteed to terminate when checking runtime, race and assertion errors. We have established the undecidability of deadlock verification and explained how to turn our analysis into a sound over-approximation. To the best of our knowledge, this is beyond the capabilities of current verification techniques which currently only target concrete inputs to phaser programs. We are currently working on tackling the parameterized case and have obtained preliminary encouraging results. Apart from improving the scalability of the tool and from using it in combination with predicate abstraction and abstract interpretation in order to analyze actual source code, we are investigating the applicability of the presented techniques for the verification of similar synchronization constructs.

References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[2] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, “Phasers: a unified deadlock-free construct for collective and point-to-point synchronization,” in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 277–288.

[3] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-java: the new adventures of old x10,” in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, 2011, pp. 51–61.

[4] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, “Phaser accumulators: A new reduction construct for dynamic parallelism,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[5] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida, “Dynamic deadlock verification for general barrier synchronisation,” in *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 150–160.

[6] D.-K. Le, W.-N. Chin, and Y.-M. Teo, “Verification of static and dynamic barrier synchronization using bounded permissions,” in *Int. Conf. on Formal Engineering Methods*. Springer, 2013, pp. 231–248.

[7] P. Anderson, B. Chase, and E. Mercer, “Jpf verification of habanero java programs,” *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–7, 2014.

[8] K. Havelund and T. Pressburger, “Model checking java programs using java pathfinder,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, 2000.

[9] R. Mayr and P. Totzke, “Branching-time model checking gap-order constraint systems,” *Fundamenta Informaticae*, vol. 143, no. 3-4, pp. 339–353, 2016.

[10] L. Bozzelli and S. Pinchinat, “Verification of gap-order constraint abstractions of counter systems,” *Theoretical Computer Science*, vol. 523, pp. 1 – 36, 2014.

[11] G. Zeinab, R. Ahmed, E. Petru, and P. Zebo, “Safety verification of phaser programs,” *CoRR*, vol. abs/1708.02801, 2017. [Online]. Available: https://arxiv.org/abs/1708.02801

[12] P. Z. Revesz, “A closed-form evaluation for datalog queries with integer (gap)-order constraints,” *Theoretical Computer Science*, vol. 116, no. 1, pp. 117–149, 1993.

[13] S. Lahiri and M. Musuvathi, “An efficient decision procedure for utvpi constraints,” in *Frontiers of Combining Systems (FroCos ’05)*. Springer Verlag, May 2005.

[14] P. A. Abdulla, F. Haziza, and L. Holík, “All for the price of few,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 476–495.

Learning to Prove Safety over Parameterised Concurrent Systems

Yu-Fang Chen
Academia Sinica

Chih-Duo Hong
Oxford University

Anthony W. Lin
Oxford University

Philipp Rümmer
Uppsala University

***Abstract*—We revisit the classic problem of proving safety over parameterised concurrent systems, i.e., an infinite family of finite-state concurrent systems that are represented by some finite (symbolic) means. An example of such an infinite family is a dining philosopher protocol with any number n of processes (n being the parameter that defines the infinite family). Regular model checking is a well-known generic framework for modelling parameterised concurrent systems, where an infinite set of configurations (resp. transitions) is represented by a regular set (resp. regular transducer). Although verifying safety properties in the regular model checking framework is undecidable in general, many sophisticated semi-algorithms have been developed in the past fifteen years that can successfully prove safety in many practical instances. In this paper, we propose a simple solution to synthesise regular inductive invariants that makes use of Angluin's classic L^* algorithm (and its variants). We provide a termination guarantee when the set of configurations reachable from a given set of initial configurations is regular. We have tested L^* algorithm on standard (as well as new) examples in regular model checking including the dining philosopher protocol, the dining cryptographer protocol, and several mutual exclusion protocols (e.g. Bakery, Burns, Szymanski, and German). Our experiments show that, despite the simplicity of our solution, it can perform at least as well as existing semi-algorithms.**

I. Introduction

Parameterised concurrent systems are infinite families of finite-state concurrent systems, parameterised by the number n of processes. There are numerous examples of parameterised concurrent systems, including models of distributed algorithms which are typically designed to handle an arbitrary number n of processes [32], [53]. Verification of such systems, then, amounts to proving that a desired property holds for *all* permitted values of n. For example, proving that the safety property holds for a dining philosopher protocol entails proving that the protocol with any given number n of philosophers ($n \geq 3$) can never reach a state when two neighbouring philosophers eat simultaneously. For each given value of n, verifying safety/liveness is decidable, albeit the exponential state-space explosion in the parameter n. However, when the property has to hold for each value of n, the number of system configurations a verification algorithm has to explore is potentially infinite. Indeed, even safety checking is already undecidable for parameterised concurrent systems [9], [12], [30]; see [13] for a comprehensive survey on the decidability aspect of the parameterised verification problem.

Various sophisticated semi-algorithms for verifying parameterised concurrent systems are available. These semi-algorithms typically rely on a symbolic framework for representing infinite sets of system configurations and transitions. *Regular model checking* [42], [7], [14], [15], [6], [1], [22], [45], [65] is one well-known symbolic framework for modelling and verifying parameterised concurrent systems. In regular model checking, configurations are modelled using words over a finite alphabet, sets of configurations are represented as regular languages, and the transition relation is defined by a regular transducer. From the research programme of regular model checking, not only are regular languages/transducers known to be highly expressive symbolic representations for modelling parameterised concurrent systems, they are also amenable to an automata-theoretic approach (due to many nice closure properties of regular languages/transducers), which have often proven effective in verification.

In this paper, we revisit the classic problem of verifying safety in the regular model checking framework. Many sophisticated semi-algorithms for dealing with this problem have been developed in the literature using methods such as abstraction [4], [5], [21], [20], widening [15], [23], acceleration [57], [42], [11], and learning [54], [55], [38], [63], [62]. One standard technique for proving safety for an infinite-state systems is by exhibiting an *inductive invariant* Inv (i.e. a set of configurations that is closed under an application of the transition relation) such that **(i)** Inv subsumes the set Init of all initial configurations, but **(ii)** Inv does not intersect with the set Bad of unsafe configurations. In regular model checking, the sets Init and Bad are given as regular sets. For this reason, a natural method for proving safety in regular model checking is to exhibit a *regular* inductive invariant satisfying (i) and (ii). The regular set Inv can be constructed as a "regular proof" for safety since checking that a candidate regular set Inv is a proof for safety is decidable. A few semi-algorithms inspired by automata learning — some based on the passive learning algorithms [38], [55], [2] and some others based on active learning algorithms [55], [62]— have been proposed to synthesise a regular inductive invariant in regular model checking. Despite these semi-algorithms, not much attention has been paid to applications of automata learning in regular model checking.

In this paper, we are interested in one basic research question in regular model checking: *can we effectively apply the classic Angluin's L^* automata learning [8] (or variants [58], [44]) to learn a regular inductive invariant?* Hitherto this question, perhaps surprisingly, has no satisfactory answer in the literature. A more careful consideration reveals at least

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

two problems. Firstly, membership queries (i.e. is a word w reachable from $Init$?) may be asked by the L^* algorithm, which amounts to checking reachability in an infinite-state system, which is undecidable in general. This problem was already noted in [54], [55], [62], [63]. Secondly, a regular inductive invariant satisfying (i) and (ii) might not be unique, and so strictly speaking we are not dealing with a well-defined learning problem. More precisely, consider the question of *what the teacher should answer when the learner asks whether v is in the desired invariant, but v turns out not to be reachable from $Init$?* Discarding v might not be a good idea, since this could force the learning algorithm to look for a *minimal* (in the sense of set inclusion) inductive invariant, which might not be regular. Similarly, let us consider what the teacher should answer in the case when we found a pair (v, w) of configurations such that (1) v is in the candidate Inv, (2) $w \notin Inv$, and (3) there is a transition from v to w. In the ICE-learning framework [35], [34], [54], the pair (v, w) is called an *implication counterexample*. To satisfy the inductive invariant constraint, the teacher may respond that w should be added to Inv, or that v should be removed from Inv. Some works in the literature have proposed using a three-valued logic/automaton (with "don't know" as an answer) because of the teacher's incomplete information [37], [26].

a) Contribution: In this paper, we propose a simple and practical solution to the problem of applying the classic L^* automata learning algorithm and its variants to synthesise a regular inductive invariant in regular model checking. To deal with the first problem mentioned in the previous paragraph, we propose to restrict to *length-preserving* regular transducers. In theory, length-preservation is not a restriction for safety analysis, since it just implies that each instance of the considered parameterised system is operating on bounded memory of size n (but the parameter n is unbounded). Experience shows that many practical examples in parameterised concurrent systems can be captured naturally in terms of length-preserving systems, e.g., see [52], [7], [6], [42], [22], [57], [1]. The benefit of the restriction is that the problem of membership queries is now decidable, since the set of configurations that may reach (be reachable from) any given configuration w is finite and can be solved by a standard finite-state model checker. For the second problem mentioned in the previous paragraph, we propose that a *strict teacher* be employed in L^* learning for regular inductive invariants in regular model checking. A strict teacher attempts to teach the learner the minimal inductive invariant (be it regular or not), but is satisfied when the candidate answer posed by the learner is an inductive invariant satisfying (i) and (ii) without being minimal. [In this sense, perhaps a more appropriate term is a *strict but generous teacher*, who tries to let a student pass a final exam whenever possible.] For this reason, when the learner asks whether w is in the desired inductive invariant, the teacher will reply NO if w is not reachable from $Init$. The same goes with an implication counterexample (v, w) such that the teacher will say that an unreachable v is not in the desired inductive invariant.

We have implemented the learning-based approach in a prototype tool with an interface to the libalf library, which includes the L^* algorithm and its variants. Despite the simplicity of our solution, it (perhaps surprisingly) works extremely well in practice, as our experiments suggest. We have taken numerous standard examples from regular model checking, including cache coherence protocols (German's Protocol), self-stabilising protocols (Israeli-Jalfon's Protocol and Herman's Protocol), synchronisation protocols (Lehmann-Rabin's Dining Philosopher Protocol), secure multi-party computation protocols (Dining Cryptographers Protocol [25]), and mutual exclusion protocols (Szymanski's Protocol, Burn's Protocol, Dijkstra's Protocol, Lamport's bakery algorithm, and Resource-Allocator Protocol). We show that L^* algorithm can perform at least as well as (and, in fact, often outperform) existing semi-algorithms. We compared the performance of our algorithm with well-known and established techniques such as SAT-based learning [55], [54], [51], [52], abstract regular model checking (ARMC), which is based on abstraction-refinement using predicate abstractions and finite-length abstractions [20], [21], and T(O)RMC, which is based on extrapolation (a widening technique) [16]. Our experiments show that, despite the simplicity of our solution, it can perform at least as well as existing semi-algorithms.

b) Related Work: The work of Vardhan *et al.* [63], [62] applies L^* learning to infinite-state systems and, amongst other, regular model checking. The learning algorithm attempts to learn an inductive invariant enriched with "distance" information, which is one way to make membership queries (i.e. reachability for general infinite-state systems) decidable. This often makes the resulting set not regular, even if the set of reachable configurations is regular, in which case our algorithm is guaranteed to terminate (recall our algorithm is only learning a regular invariant without distance information). Conversely, when an inductive invariant enriched with distance information is regular, so is the projection that omits the distance information. Unfortunately, neither their tool Lever [63], nor the models used in their experiments are available, so that we cannot make a direct comparison to our approach. A learning algorithm allowing incomplete information [37] has been applied in [55] for inferring inductive invariants of regular model checking. Although the learning algorithm in [37] uses the same data structure as the standard L^* algorithm, it is essentially a SAT-based learning algorithm (its termination is not guaranteed by the Myhill-Nerode theorem).

Despite our results that SAT-based learning seems to be less efficient than L^* learning for synthesising regular inductive invariants in regular model checking, SAT-based learning is more general and more easily applicable when verifying other properties, e.g., liveness [52], fair termination [48], and safety games [56]. View abstraction [5] is a novel technique for parameterised verification. Comparing to parameterised verification based on view abstraction, our framework (i.e. general regular model checking framework with transducers) provides a more expressive modelling language that is required in specifying protocols with near-neighbour communication (e.g. Dining Cryptographers and Dining Philosophers).

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

When preparing the final version, we found that a very similar algorithm had already appeared in Vardhan's thesis [61, Section 6] from 2006; in particular, including the trick to make a membership query (i.e. point-to-point reachability) decidable by bounding the space of the transducers. The research presented here was conducted independently, and considers several aspects that were not yet present in [61], including experimental results on systems that are not counter systems (parameterised concurrent systems with topologies), and heuristics like the use of shortest counterexamples and caching. We cannot compare our implementation in detail with the one from [61], since the latter tool is not publicly available.

c) Organisation: The notations are defined in Section II. A brief introduction to regular model checking and automata learning is given in Section III and Section IV, respectively. The learning-based algorithm is provided in Section V. The result of the experiments is in Section VI.

II. Preliminaries

d) General Notations: Let Σ be a finite set of symbols called *alphabet*. A word over Σ is a finite sequence of symbols of Σ. We use λ to represent an empty word. For a set $I \subseteq \Sigma^*$ and a relation $T \subseteq \Sigma^* \times \Sigma^*$, we define $T(I)$ to be the post-image of I under T, i.e., $T(I) = \{y \mid \exists x.\ x \in I \wedge (x,y) \in T\}$. Let $id = \{(x,x) \mid x \in \Sigma^*\}$ be the *identity relation*. We define T^n for all $n \in \mathbb{N}$ in the standard way by induction: $T^0 = id$, and $T^k = T \circ T^{k-1}$, where $\circ$ denotes the *composition* of relations. Let T^* denote the transitive closure of T, i.e., $T^* = \bigcup_{i=1}^{\infty} T^i$. For any two sets A and B, we use $A \ominus B$ to denote their *symmetric difference*, i.e., the set $A \setminus B \cup B \setminus A$.

e) Finite Automata and Transducer: In this paper, automata/transducers are denoted in calligraphic fonts $\mathcal{A}, \mathcal{B}, \mathcal{I}, \mathcal{T}$ to represent automata/transducers, while the corresponding languages/relations are denoted in roman fonts A, B, I, T.

A *finite automaton* (FA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A *run* of $\mathcal{A}$ on a word $w = a_1 a_2 a_3 \cdots a_n$ is a sequence of states $q_0, q_1, \cdots, q_n$ such that $(q_i, a_{i+1}, q_{i+1}) \in \delta$. A run is *accepting* if the last state $q_n \in F$. A word is *accepted* by $\mathcal{A}$ if it has an accepting run. The *language* of $\mathcal{A}$, denoted by A, is the set of word accepted by $\mathcal{A}$. A language is *regular* if it can be recognised by a finite automaton. $\mathcal{A}$ is a *deterministic finite automaton* (DFA) if $|\{q' \mid (q, a, q') \in \delta\}| \leq 1$ for each $q \in Q$ and $a \in \Sigma$.

Let $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. A *(finite) transducer* is a tuple $\mathcal{T} = (Q, \Sigma_\lambda, \delta, q_0, F)$ where Q is a finite set of states, $\delta \subseteq Q \times \Sigma_\lambda \times \Sigma_\lambda \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We say that $\mathcal{T}$ is *length-preserving* if $\delta \subseteq Q \times \Sigma \times \Sigma \times Q$. We define relation $\delta^* \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ as the smallest relation satisfying (1) $(q, \lambda, \lambda, q) \in \delta^*$ for any $q \in Q$ and (2) $(q_1, x, y, q_2) \in \delta^* \wedge (q_2, a, b, q_3) \in \delta \implies (q_1, xa, yb, q_3) \in \delta^*$. The relation represented by $\mathcal{T}$ is the set $\{(x,y) \mid (q_0, x, y, q) \in \delta^* \wedge q \in F\}$. A relation is *regular and length-preserving* if it can be represented by a length-preserving transducer.

III. Regular model checking

Regular model checking (RMC) is a uniform framework for modelling and automatically analysing parameterised concurrent systems. In the paper, we focus on the regular model checking framework for safety properties. Under the framework, each *system configuration* is represented as a word in Σ^*. The sets of *initial configurations* and of *bad configurations* are captured by regular languages over Σ. The *transition relation* is captured by a regular and length-preserving relation on Σ^*. We use a triple $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ to denote a *regular model checking problem*, where $\mathcal{I}$ is an FA recognizing the set of initial configurations, $\mathcal{T}$ is a transducer representing the transition relation, and $\mathcal{B}$ is an FA recognizing the set of bad configurations. Then the regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ asks if $T^*(I) \cap B = \emptyset$. A standard way to prove $T^*(I) \cap B = \emptyset$ is to find a proof based on a set V satisfying the following three conditions: (1)$I \subseteq V$ (i.e. all initial configurations are contained in V), (2) $V \cap B = \emptyset$ (i.e. V does not contain bad configurations), (3) $T(V) \subseteq V$ (i.e. V is *inductive*: applying T to any configuration in V does not take it outside V). We call the set V an *inductive invariant* for the regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. In the framework of regular model checking, a standard method for proving safety (e.g. see [55], [7]) is to find a *regular proof*, i.e., an inductive invariant that can be captured by finite automaton. Because regular languages are effectively closed under Boolean operations and taking pre-/post-images w.r.t. finite transducers, an algorithm for verifying whether a given regular language is an inductive invariant can be obtained by using language inclusion algorithms for FA [3], [19].

Example 1 (Herman's Protocol)**.** *Herman's Protocol is a* self-stabilising *protocol for n processes (say with ids $0, \ldots, n-1$) organised as a ring structure. A* configuration *in the Herman's Protocol is* correct *iff only one process has a token. The protocol ensures that any system configuration where the processes collectively holding any odd number of tokens will almost surely be recovered to a correct configuration. More concretely, the protocol works iteratively. In each iteration, the scheduler randomly chooses a process. If the process with the number i is chosen by the scheduler, it will toss a coin to decide whether to keep the token or pass the token to the next process, i.e. the one with the number $(i+1)\%n$. If a process holds two tokens in the same iteration, it will discard* both *tokens. One safety property the protocol guarantees is that every system configuration has at least one token.*

The protocol and the corresponding safety property can be modelled as a regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. Each process has two states; the symbol T *denotes the state that the process has a token and* N *denotes the state that the process does not have a token. The word* NNTTNN *denotes a system configuration with six processes, where only the processes with numbers 2 and 3 are in the state with tokens. The set of initial configurations is $I = \mathsf{N}^* \mathsf{T} (\mathsf{N}^* \mathsf{T} \mathsf{N}^* \mathsf{T} \mathsf{N}^*)^*$, i.e., an odd number of processes has tokens. The set of bad configuration is $B = \mathsf{N}^*$, i.e., all tokens have disappeared. We use the regular* *language $E = ((\mathsf{T},\mathsf{T}) + (\mathsf{N},\mathsf{N}))$ to denote the relation that a process is idle. The transition relation T can be specified as a union of the following regular expressions: (1) E^* [Idle], (2) $E^*(\mathsf{T},\mathsf{N})(\mathsf{T},\mathsf{N})E^* + (\mathsf{T},\mathsf{N})E^*(\mathsf{T},\mathsf{N})$ [Discard both tokens], and (3) $E^*(\mathsf{T},\mathsf{N})(\mathsf{N},\mathsf{T})E^* + (\mathsf{N},\mathsf{T})E^*(\mathsf{T},\mathsf{N})$ [Pass the token].*

IV. Automata Learning

Suppose R is a regular *target* language whose definition is not directly accessible. *Automata learning* algorithms [8], [58], [44], [17] automatically infer a FA $\mathcal{A}$ recognising R. The setting of an online learning algorithm assumes a *teacher* who has access to R and can answer the following two queries: (1) Membership query $Mem(w)$: is the word w a member of R, i.e., $w \in R$? (2) Equivalence query $Equ(\mathcal{A})$: is the language of FA $\mathcal{A}$ equal to R, i.e., $A = R$? If not, it returns a counterexample $w \in A \ominus R$. The learning algorithm will then construct a FA $\mathcal{A}$ such that $A = R$ by interacting with the teacher. Such an algorithm works iteratively: In each iteration, it performs membership queries to get from the teacher information about R. Using the results of the queries, it proceeds by constructing a candidate automaton $\mathcal{A}_h$ and makes an equivalence query $Equ(\mathcal{A}_h)$. If $A_h = R$, the algorithm terminates with $\mathcal{A}_h$ as the resulting FA. Otherwise, the teacher returns a word w distinguishing A_h from R. The learning algorithm uses w to refine the candidate automaton of the next iteration. In the last decade, automata learning algorithms have been frequently applied to solve formal verification and synthesis problems, c.f., [27], [24], [38], [37], [26], [31].

More concretely, below we explain the details of the automata learning algorithm proposed by Rivest and Schapire [58] (RS), which is an improved version of the classic L^* learning algorithm by Angluin [8]. The foundation of the learning algorithm is the Myhill-Nerode theorem, from which one can infer that the states of the minimal DFA recognizing R are isomorphic to the set of equivalence classes defined by the following relations: $x \equiv_R y$ iff $\forall z \in \Sigma^* : xz \in R \leftrightarrow yz \in R$. Informally, two strings x and y belong to the same state of the minimal DFA recognising R iff they cannot be distinguished by any suffix z. In other words, if one can find a suffix z' such that $xz' \in R$ and $yz' \notin R$ or vice versa, then x and y belong to different states of the minimal DFA.

The algorithm uses a data structure called *observation table* (S, E, T) to find the equivalence classes correspond to $\equiv_R$, where S is a set of strings denoting the set of identified states, E is the set of suffixes to distinguish if two strings belong to the same state of the minimal DFA, and T is a mapping from $(S \cup (S \cdot \Sigma)) \cdot E$ to $\{\top, \bot\}$. The value of $T(w) = \top$ iff $w \in R$. We use $row_E(x) = row_E(y)$ as a shorthand for $\forall z \in E : T(xz) = T(yz)$. That is, the strings x and y cannot be identified as two different states using only strings in the set E as the suffixes. Observe that $x \equiv_R y$ implies $row_E(x) = row_E(y)$ for all $E \subseteq \Sigma^*$. We say that an observation table is *closed* iff $\forall x \in S, a \in \Sigma : \exists y \in S : row_E(xa) = row_E(y)$. Informally, with a closed table, every state can find its successors wrt. all symbols in Σ. Initially, $S = E = \{\lambda\}$, and $T(w) = Mem(w)$ for all $w \in \{\lambda\} \cup \Sigma$.

Algorithm 1: The improved L^* algorithm by Rivest and Schapire

```
Input: A teacher answers $Mem(w)$ and $Equ(\mathcal{A})$ about a
       target regular language $R$ and the initial
       observation table $(S, E, T)$.
1 repeat
2   while $(S, E, T)$ is not closed do
3     Find a pair $(x, a) \in S \times \Sigma$ such that
      $\forall y \in S : row_E(xa) \neq row_E(y)$. Extend $S$ to
      $S \cup \{xa\}$ and update $T$ using membership
      queries accordingly;
4   Build a candidate DFA $\mathcal{A}_h = (S, \Sigma, \delta, \lambda, F)$, where
    $\delta = \{(s, a, s') \mid s, s' \in S \wedge row_E(sa) = row_E(s)\}$,
    the empty string $\lambda$ is the initial state, and
    $F = \{s \mid T(s) = \top \wedge s \in S\}$;
5   if $Equ(\mathcal{A}_h) = (\mathsf{false}, w)$, where $w \in A \ominus R$ then
      Analyse $w$ and add a suffix of $w$ to $E$;
6 until $Equ(\mathcal{A}_h) = \mathsf{true}$;
7 return $\mathcal{A}_h$ is the minimal DFA for $R$;
```

The details of of the improved L^* algorithm by Rivest and Schapire can be found in Algorithm 1. Observe that, in the algorithm, two strings x, y with $x \equiv_R y$ will never be simultaneously contained in the set S. When the equivalence query $Equ(\mathcal{A})$ returns false together with a counterexample $w \in A \ominus R$, the algorithm will perform a binary search over w using membership queries to find a suffix e of w and extend E to $E \cup \{e\}$. The suffix e has the property that $\exists x, y \in S, a \in \Sigma : row_E(xa) = row_E(y) \wedge row_{E\cup\{e\}}(xa) \neq row_{E\cup\{e\}}(y)$, that is, add e to E will identify at least one more state. The existence of such a suffix is guaranteed. We refer the readers to [58] for the proof.

Proposition 1. *[58] Algorithm 1 will find the minimal DFA $\mathcal{R}$ for R using at most n equivalence queries and $n(n + n|\Sigma|) + n \log m$ membership queries, where n is the number of state of $\mathcal{R}$ and m is the length of the longest counterexample returned from the teacher.*

Because each equivalence query with a false answer will increase the size (number of states) of the candidate DFA by at least one and the size of the candidate DFA is bounded by n according to the Myhill-Nerode theorem, the learning algorithm uses at most n equivalence queries. The number of membership queries required to fill in the entire observation table is bounded by $n(n + n|\Sigma|)$. Since a binary search is used to analyse the counterexample and the number of counterexample from the teacher is bounded by n, the number of membership queries required is bounded by $n \log m$.

We would like to introduce the other two important variants of the L^* learning algorithm. The algorithm proposed by Kearns and Vazirani [44] (KV) uses a *classification tree* data structure to replace the observation table data structure of the classic L^* algorithm. The algorithm of Kearns and Vazirani has a similar query complexity to the one of Rivest and Schapire [58]; it uses at most n equivalence queries and $n^2(n|\Sigma| + m)$ membership queries. However, the worst case

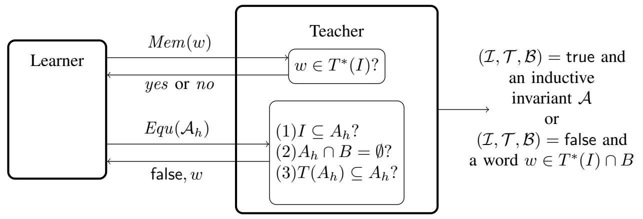

Teacher
Learner
$Mem(w)$
$w \in T^*(I)$?
yes or no
$Equ(\mathcal{A}_h)$
(1)$I \subseteq A_h$?
(2)$A_h \cap B = \emptyset$?
(3)$T(A_h) \subseteq A_h$?
false, w
$(\mathcal{I}, \mathcal{T}, \mathcal{B})$ = true and an inductive invariant $\mathcal{A}$
or
$(\mathcal{I}, \mathcal{T}, \mathcal{B})$ = false and a word $w \in T^*(I) \cap B$

Fig. 1. Overview: using automata learning to solve the regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. Recall that we use calligraphy font for automata/transducers and roman font for the corresponding languages/relations.

bound of the number of membership queries is very loose. It assumes the structure of the classification tree is linear, i.e., each node has at most one child, which happens very rarely in practice. In our experience, the algorithm of Kearns and Vazirani usually requires a few more equivalence queries, with a significant lower number of membership queries comparing to Rivest and Schapire when applied to verification problems.

The NL^* algorithm [17] learns a non-deterministic finite automaton instead of a deterministic one. More concretely, it makes use of a canonical form of nondeterministic finite automaton, named *residual finite-state automaton (RFSA)* to express the target regular language. In some examples, RFSA can be exponentially more succinct than DFA recognising the same languages. In the worst case, the NL^* algorithm uses $O(n^2)$ equivalence queries and $O(m|\Sigma|n^3)$ membership queries to infer a canonical RFSA of the target language.

V. Algorithm

We apply automata learning algorithms, including Angluin's L^* and its variants, to solve the regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. Those learning algorithms require a teacher answering both equivalence and membership queries. Our strategy is to design a "strict teacher" targeting the minimal inductive invariant $T^*(I)$. For a membership query on a word w, the teacher checks if $w \in T^*(I)$, which is decidable under the assumption that $\mathcal{T}$ is length-preserving. For an equivalence query on a candidate FA $\mathcal{A}_h$, the teacher analyses if $\mathcal{A}_h$ can be used as an inductive invariant in a proof of the problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. It performs one of the following actions depending on the result of the analysis (Fig. 1):

- Determine that $\mathcal{A}_h$ does not represent an inductive invariant, and return false together with an explanation $w \in \Sigma^*$ to the learner.
- Conclude that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{true}$, and terminate the learning process with an inductive invariant $\mathcal{A}_h$ as the proof.
- Conclude that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{false}$, and terminate the learning with a word $w \in T^*(I) \cap B$ as an evidence.

Similar to the typical regular model checking approach, our learning-based technique tries to find a "regular proof", which amounts to finding an inductive invariant in the form of a regular language. Our approach is incomplete in general since it could happen that there only non-regular inductive invariants exist. Pathological cases where only non-regular inductive invariant exist do not, however, seem to occur frequently in practice, c.f., [21], [38], [20], [22], [60], [57], [50].

Answering a membership query on a word w, i.e., checking whether $w \in T^*(I)$, is the easy part: since $\mathcal{T}$ is length-preserving, we can construct an FA recognising $Post^{|w|} = \{w' \mid |w'| = |w| \wedge w' \in T^*(I)\}$ and then check if $w \in Post^{|w|}$. In practice, $Post^{|w|}$ can be efficiently computed and represented using BDDs and symbolic model checking.

For an equivalence query on a candidate FA $\mathcal{A}_h$, we need to check if $\mathcal{A}_h$ can be used as an inductive invariant for the regular model checking problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. More concretely, we check the three conditions (1) $I \subseteq A_h$, (2) $A_h \cap B = \emptyset$, and (3) $T(A_h) \subseteq A_h$ using Algorithm 2.

Algorithm 2: Answer equivalence query on candidate FA

```
Input: An FA $\mathcal{A}_h$ and an RMC problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$
1  if $I \not\subseteq A_h$ then
2      Find a word $w \in \mathcal{I} \setminus A_h$;
3      return (false, $w$) to the learner;
4  else if $A_h \cap B \neq \emptyset$ then
5      Find a word $w \in A_h \cap B$;
6      if $w \in T^*(I)$ then Output $\{cex = w,$
       $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{false}\}$ and halt;
7      else return (false, $w$) to the learner;
8  else if $T(A_h) \not\subseteq A_h$ then
9      Find a pair of words $(w, w') \in T$ such that $w \in A_h$
       but $w' \notin A_h$;
10     if $w \in T^*(I)$ then return (false, $w'$) to the learner;
11     else return (false, $w$) to the learner;
12 else Output $\{inv = \mathcal{A}_h, (\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{true}\}$ and halt;
```

If the condition (1) is violated, i.e., $I \not\subseteq A_h$, there is a word $w \in I \setminus A_h$. Since $I \subseteq T^*(I)$, the teacher can infer that $w \in T^*(I) \setminus A_h$ and return w as a *positive* counterexample to the learner. A counterexample is positive if it represents a word in the target language that was missing in the candidate language. The definition negative counterexamples is symmetric.

If the condition (2) is violated, i.e., $A_h \cap B \neq \emptyset$, there is a word $w \in A_h \cap B$. The teacher checks if $w \in T^*(I)$ by constructing $Post^{|w|}$ and checking if $w \in Post^{|w|}$. If $w \notin T^*(I)$, the teacher obtains that $w \in A_h \setminus T^*(I)$ and returns false together with w as a negative counterexample to the learner. Otherwise, the teacher infers that $w \in T^*(I) \cap B$ and outputs $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{false}$ with the word w as an evidence.

The case that the condition (3) is violated, i.e., $T(A_h) \not\subseteq A_h$, is more involved. There exists a pair of words $(w, w') \in T$ such that $w \in A_h \wedge w' \notin A_h$. The teacher will check if $w \in T^*(I)$. If it is, then the teacher knows that $w' \in T^*(I) \wedge w' \notin A_h$ and hence returns false together with w' as a positive counterexample to the learner. If $w \notin T^*(I)$, then the teacher knows that $w \notin T^*(I) \wedge w \in A_h$ and hence returns false together with w as a negative counterexample to the learner.

If all conditions hold, the "strict teacher" shows its generosity (A_h might not equal to $T^*(I)$, but it will still pass) and concludes that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{true}$ with a proof using $\mathcal{A}_h$ as the inductive invariant.

Theorem 1 (Correctness)**.** *If the algorithm from Fig. 1 terminates, it gives correct answer to the RMC problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$.*

To see this, observe that the algorithm provides an inductive invariant when it concludes $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{true}$ and a word in $T^*(I) \cap B$ when it concludes $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{false}$. In addition, if one of the L^* learning algorithms[1] from Section IV is used, we can obtain an additional result about termination:

Theorem 2 (Termination)**.** *When $T^*(I)$ is regular, the algorithm from Fig. 1 is guaranteed to terminate in at most k iterations, where k is the size of the minimal DFA of $T^*(I)$.*

Proof. Observe that in the algorithm, the counterexample obtained by the learner in each iteration locates in the symmetric difference of the candidate language and $T^*(I)$. Hence, when $T^*(I)$ can be recognized by a DFA of k states, the algorithm will not execute more than k iterations by Proposition 1. □

Two remarks are in order. Firstly, the set $T^*(I)$ tends to be regular in practice, e.g., see [21], [38], [20], [22], [60], [57], [10], [11], [50], [49]. In fact, it is known that $T^*(I)$ is regular for many subclasses of infinite-state systems that can be modelled in regular model checking [60], [50], [40], [11], [49] including pushdown systems, reversal-bounded counter systems, two-dimensional VASS (Vector Addition Systems with States), and other subclasses of counter systems. Secondly, even in the case when $T^*(I)$ is not regular, termination may still happen due to the "generosity" of the teacher, which will accept any inductive invariant as an answer.

Considerations on Implementation: The implementation of the learning-based algorithm is very simple. Since it is based on standard automata learning algorithms and uses only basic automata/transducer operations, one can find existing libraries for them. The implementation only need to take care of how to answer queries. The core of our implementation has only around 150 lines of code (excluding the parser of the input models). We provide a few suggestions to make the implementation more efficient. First, each time when an FA recognising $Post^k$ is produced, we store the pair $(k, Post^k)$ in a cache. It can be reused when a query on any word of length k is posed. We can also check if $Post^k \cap B = \emptyset$. The algorithm can immediately terminate and return $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = \mathsf{false}$ if $Post^k \cap B \neq \emptyset$. Second, for each language inclusion test, if the inclusion does not hold, we suggest to return the shortest counterexample. This heuristic helped to shorten the average length of strings sent for membership queries and hence reduced the cost of answering them. Recall that the algorithm needs to build the FA of $Post^k$ to answer membership queries. The shorter the average length of query strings is, the fewer instances of $Post^k$ have to be built.

VI. Evaluation

To evaluate our techniques, we have developed a prototype[2] in Java and used the libalf library [18] as the default inference engine. We used our prototype to check safety properties for a range of parameterised systems, including cache coherence protocols (German's Protocol), self-stabilising protocols (Israeli-Jalfon's Protocol and Herman's Protocol), synchronisation protocols (Lehmann-Rabin's Dining Philosopher Protocol), secure multi-party computation protocol (David Chaums' Dining Cryptographers Protocol), and mutual exclusion protocols (Szymanski's Protocol, Burn's Protocol, Dijkstra's Protocol, Lamport's Bakery Algorithm, and Resource-Allocator Protocol). Most of the examples we consider are standard benchmarks in the literature of regular model checking (c.f. [4], [5], [20], [22], [57]). Among them, German's Protocol and Kanban are more difficult than the other examples for fully automatic verification (c.f. [4], [5], [43]).

Based on these examples, we compare our learning method with existing techniques such as SAT-based learning [54], [55], [51], [52], extrapolating [16], [46], and abstract regular model checking (ARMC) [20], [21]. The SAT-based learning approach encodes automata as Boolean formulae and exploits a SAT-solver to search for candidate automata representing inductive invariants. It uses automata-based algorithms to either verify the correctness of the candidate or obtain a counterexample that can be further encoded as a Boolean constraint. T(O)RMC [16], [46] extrapolates the limit of the reachable configurations represented by an infinite sequence of automata. The extrapolation is computed by first identifying the increment between successive automata, and then over-approximating the repetition of the increment by adding loops to the automata. ARMC is an efficient technique that integrates abstraction refinement into the fixed-point computation. It begins with an existential abstraction obtained by merging states in the automata/transducers. Each time a spurious counterexample is found, the abstraction can be refined by splitting some of the merged states. ARMC is among the most efficient algorithms for regular model checking [38].

The comparison of those algorithms are reported in Table I, running on a MinGW64 system with 3GHz Intel i7 processor, 2GB memory limit, and 60-second timeout. The experiments show that the learning method is quite efficient: the results of our prototype are comparable with those of the ARMC algorithm[3] on all examples but Kanban, for which the minimal inductive invariant, if it is regular, has at least 400 states. On the other hand, our algorithm is significantly faster than ARMC in two cases, namely German's Protocol and Dining Cryptographers. ARMC comes with a bundle of options and heuristics, but not all of them work for our benchmarks. We have tested all the heuristics available from the tool and adopted the ones[4] that had the best performance in our experiments. The performance of SAT-based learning is comparable to the previous two approaches whenever inductive invariants representable by automata with few states exist. However, as its runtime grows exponentially with the sizes of candidate automata, the SAT-based algorithm fails to solve four examples that do not have small regular inductive invariants. T(O)RMC seems to suffer from similar problems as it timeouts on all

[1]If NL^* is used, the bound in Theorem 2 will increase to $O(k^2)$.

[2]Available at https://github.com/ericpony/safety-prover.

[3]Available at http://www.fit.vutbr.cz/research/groups/verifit/tools/hades.

[4]The heuristics are structure preserving, backward computation, and backward collapsing with all states being predicates. See [21] for explanations.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

The RMC problems								RS			SAT			T(O)RMC			ARMC
Name	#label	S_{init}	T_{init}	S_{trans}	T_{trans}	S_{bad}	T_{bad}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time
Bakery [32]	3	3	3	5	19	3	9	0.0s	6	18	0.5s	2	5	0.0s	6	11	0.0s
Burns [4]	12	3	3	10	125	3	36	0.2s	8	96	1.1s	2	10	0.1s	7	38	0.0s
Szymanski [59]	11	9	9	118	412	13	40	0.3s	43	473	1.6s	2	21	2.0s	51	102	0.1s
German [36]	581	3	3	17	9.5k	4	2112	4.8s	14	8134	t.o.	–	–	t.o.	–	–	10s
Dijkstra [4]	42	1	1	13	827	3	126	0.1s	9	378	1.7s	2	24	6.1s	8	83	0.3s
Dijkstra, ring [28], [33]	12	3	3	13	199	3	36	1.4s	22	264	0.9s	2	14	t.o.	–	–	0.1s
Dining Crypto. [25]	14	10	30	17	70	12	70	0.1s	32	448	t.o.	–	–	t.o.	–	–	7.2s
Coffee Can [52]	6	8	18	13	34	5	8	0.0s	3	18	0.2s	2	7	0.1s	6	13	0.0s
Herman, linear [39]	2	2	4	4	10	1	1	0.0s	2	4	0.2s	2	4	0.0s	2	4	0.0s
Herman, ring [39]	2	2	4	9	22	1	1	0.0s	2	4	0.4s	2	4	0.0s	2	4	0.0s
Israeli-Jalfon [41]	2	3	6	24	62	1	1	0.0s	4	8	0.1s	2	4	0.0s	4	8	0.0s
Lehmann-Rabin [47]	6	4	4	14	96	3	13	0.1s	8	48	0.5s	2	11	0.8s	19	105	0.0s
LR Dining Philo. [52]	4	4	4	3	10	3	4	0.0s	4	16	0.2s	2	6	0.1s	7	18	0.0s
Mux Array [33]	6	3	3	4	31	3	18	0.0s	5	30	0.4s	2	7	0.2s	4	14	0.0s
Res. Allocator [29]	3	3	3	7	25	4	9	0.0s	5	15	0.0s	1	3	0.0s	4	9	0.0s
Kanban [5], [43]	3	25	48	98	250	37	68	t.o.	–	–	t.o.	–	–	t.o.	–	–	3.5s
Water Jugs [64]	11	5	6	23	132	5	12	0.1s	24	264	t.o.	–	–	t.o.	–	–	0.0s

TABLE I

COMPARING THE PERFORMANCE OF DIFFERENT RMC TECHNIQUES. #LABEL STANDS FOR THE SIZE OF ALPHABET; S_X AND T_X STAND FOR THE NUMBERS OF STATES AND TRANSITIONS, RESPECTIVELY, IN THE AUTOMATA/TRANSDUCERS. **RS** IS THE RESULT OF OUR PROTOTYPE USING RIVEST AND SCHAPIRE'S VERSION OF L^*; **SAT**, **T(O)RMC**, AND **ARMC** ARE THE RESULTS OF THE OTHER THREE TECHNIQUES.

examples that cannot be proved by the SAT-based approach.

Table II reports the results of the learning-based algorithm geared with different automata learning algorithms implemented in libalf. As the table shows, these algorithms have similar performance on small examples; however, the algorithm of Rivest and Schapire [58] and the algorithm of Kearns and Varzirani [44] are significantly more efficient than the other algorithms on some large examples such as Szymanski and German. Table II shows that Kearns and Varzirani's algorithm can often find smaller inductive invariants (fewer states) than the other L^* variants, which explains the performance difference. For NL^*, our implementation pays an additional cost to determinise the learned FA in order to answer the equivalence queries; this cost is significant when a large invariant is needed.

Recall that our approach uses a "strict but generous teacher". Namely, the target language of the teacher is $T^*(I)$ for an RMC problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. We have tried the version where a "flexible and generous teacher" is used, that is, the target language of the teacher is the complement of $(T^{-1})^*(B)$. The performance, however, is worse than that of our current version. This result may reflect the fact that the set $T^*(I)$ is "more regular" (i.e., can be expressed by a DFA with fewer states) than the set $(T^{-1})^*(B)$ in practical cases.

VII. CONCLUSION

The encouraging experimental results suggest that the performance of the L^* algorithm for synthesising regular inductive invariants is comparable to the most sophisticated algorithm for regular model checking for proving safety. From a theoretical viewpoint, learning-based approaches (including ours and [54], [55], [38]) have a termination guarantee when the set $T^*(I)$ is regular, which is not guaranteed by approaches based on a fixed-point computation (e.g., the ARMC [21]). An interesting research question is whether L^* algorithm can be effectively used for verifying other properties, e.g., liveness.

Acknowledgements: We thank anonymous referees for their useful comments. Rümmer was supported by the Swedish Research Council under grant 2014-5484.

REFERENCES

[1] P. A. Abdulla. Regular model checking. *STTT*, 14(2):109–118, 2012.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV'14*, pages 150–166.

[3] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *TACAS'10*, pages 158–174.

[4] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS'07*, pages 721–736.

[5] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI'13*, pages 476–495.

[6] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for LTL(MSO). *STTT*, 14(2):223–241, 2012.

[7] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *CONCUR'04*, pages 35–48.

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[9] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 22(6):307–309, 1986.

[10] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[11] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA'05*, pages 474–488.

[12] N. Bertrand and P. Fournier. Parameterized verification of many identical probabilistic timed processes. In *FSTTCS'13*, volume 24 of *LIPIcs*, pages 501–513. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[13] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[14] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1999.

[15] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV'03*, pages 223–235.

[16] B. Boigelot, A. Legay, and P. Wolper. Omega-regular model checking. In *TACAS'04*, pages 561–575.

[17] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI'09*, pages 1004–1009.

[18] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *CAV'10*, pages 360–364.

[19] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL'13*, pages 457–468.

[20] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *ENTCS*, 149(1):37–48, 2006.

[21] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV'04*, pages 372–386.

[22] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV'00*, pages 403–418.

	RS			L*			L*c			KV			NL*		
	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}
Bakery	0.0s	6	18	0.0s	6	18	0.1s	6	18	0.0s	6	18	0.1s	6	18
Burns	0.2s	8	96	0.5s	8	96	0.2s	8	96	0.2s	8	96	0.4s	6	72
Szymanski	0.3s	43	473	2.4s	51	561	1.2s	41	451	0.3s	41	451	1.4s	59	649
German	4.8s	14	8134	13s	15	8715	26s	15	8715	4.2s	14	8134	40s	15	8715
Dijkstra	0.1s	9	378	0.4s	9	378	0.1s	9	378	0.2s	9	378	0.2s	10	420
Dijkstra, ring	1.4s	22	264	2.7s	20	240	8.9s	22	264	1.5s	14	168	1.8s	20	240
Dining Crypto.	0.1s	32	448	0.2s	34	476	0.2s	38	532	0.1s	19	266	0.3s	36	504
Coffee Can	0.0s	3	18	0.0s	3	18	0.0s	4	24	0.0s	3	18	0.0s	4	24
Herman, linear	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4
Herman, ring	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4
Israeli-Jalfon	0.0s	4	8	0.0s	4	8	0.0s	4	8	0.0s	4	8	0.0s	4	8
Lehmann-Rabin	0.1s	8	48	0.2s	8	48	0.1s	8	48	0.1s	8	48	0.2s	8	48
LR D. Philo.	0.0s	4	16	0.2s	4	16	0.0s	5	20	0.0s	4	16	0.0s	8	32
Mux Array	0.0s	5	30	0.0s	5	30	0.0s	5	30	0.0s	5	30	0.0s	5	30
Res. Allocator	0.0s	5	15	0.0s	4	12	0.0s	5	15	0.0s	5	15	0.0s	5	15
Kanban	>60s	–	–	>60s	–	–	>60s	–	–	>60s	–	–	>60s	–	–
Water Jugs	0.1s	24	264	0.5s	25	275	0.5s	25	275	0.1s	24	264	0.5s	25	275

TABLE II

COMPARING THE PERFORMANCE BASED ON DIFFERENT AUTOMATA LEARNING ALGORITHMS. THE COLUMNS $\mathbf{L^*}$, $\mathbf{L^*c}$, $\mathbf{RS}$, $\mathbf{KV}$, AND $\mathbf{NL^*}$ ARE THE RESULTS OF THE ORIGINAL L^* ALGORITHM BY ANGLUIN [8], A VARIANT OF L^* THAT ADDS ALL SUFFIXES OF THE COUNTEREXAMPLE TO COLUMNS, THE VERSION BY RIVEST AND SHAPIRE [58], THE VERSION BY KEARNS AND VAZIRANI [44], AND THE NL^* ALGORITHM [17], RESPECTIVELY.

[23] A. Bouajjani and T. Touili. Widening techniques for regular tree model checking. *STTT*, 14(2):145–165, 2012.

[24] M. Chapman, H. Chockler, P. Kesseli, D. Kroening, O. Strichman, and M. Tautschnig. Learning the language of error. In *ATVA'15*, pages 114–130.

[25] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[26] Y. Chen, A. Farzan, E. M. Clarke, Y. Tsay, and B. Wang. Learning minimal separating DFA's for compositional verification. In *TACAS'09*, pages 31–45.

[27] Y. Chen, C. Hsieh, O. Lengál, T. Lii, M. Tsai, B. Wang, and F. Wang. PAC learning-based verification and model synthesis. In *ICSE'16*, pages 714–724.

[28] E. W. Dijkstra, R. Bird, M. Rogers, and O.-J. Dahl. Invariance and non-determinacy [and discussion]. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 312(1522):491–499, 1984.

[29] A. F. Donaldson. *Automatic techniques for detecting and exploiting symmetry in model checking*. PhD thesis, University of Glasgow, 2007.

[30] J. Esparza. Parameterized verification of crowds of anonymous processes. *Dependable Software Systems Engineering*, 45:59–71, 2016.

[31] 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 *TACAS'08*, pages 2–17.

[32] W. Fokkink. *Distributed Algorithms*. MIT Press, 2013.

[33] L. Fribourg and H. Olsén. Reachability sets of parameterized rings as regular languages. *ENTCS*, 9:40, 1997.

[34] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV'14*, pages 69–87.

[35] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV'13*, pages 813–829.

[36] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *JACM*, 39(3):675–735, 1992.

[37] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR'06*, pages 483–497.

[38] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *ENTCS*, 138(3):21–36, 2005.

[39] T. Herman. Probabilistic self-stabilization. *IPL*, 35(2):63–67, 1990.

[40] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

[41] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *PODC'90*, pages 119–131.

[42] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *TACAS'00*, pages 220–234.

[43] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV'10*, pages 645–659.

[44] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT press, 1994.

[45] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *TCS*, 256(1-2):93–112, 2001.

[46] A. Legay. T(O)RMC: A tool for (ω)-regular model checking. In *CAV'08*, pages 548–551.

[47] D. Lehmann and M. O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *POPL'81*, pages 133–138.

[48] O. Lengál, A. W. Lin, R. Majumdar, and P. Rümmer. Fair termination for parameterized probabilistic concurrent systems. In *TACAS'17*.

[49] J. Leroux and G. Sutre. Flat counter automata almost everywhere! In *ATVA'05*, pages 489–503.

[50] A. W. Lin. Accelerating tree-automatic relations. In *FSTTCS'12*, pages 313–324.

[51] A. W. Lin, T. K. Nguyen, P. Rümmer, and J. Sun. Regular symmetry patterns. In *VMCAI'16*, pages 455–475.

[52] A. W. Lin and P. Rümmer. Liveness of randomised parameterised systems under arbitrary schedulers. In *CAV'16*, pages 112–133.

[53] N. A. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *PODC'94*, pages 314–323.

[54] D. Neider. *Applications of Automata Learning in Verification and Synthesis*. PhD thesis, RWTH Aachen, 2014.

[55] D. Neider and N. Jansen. Regular model checking using solver technologies and automata learning. In *NFM*, pages 16–31, 2013.

[56] D. Neider and U. Topcu. An automaton learning approach to solving safety games over infinite graphs. In *TACAS'16*, pages 204–221.

[57] M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala Univ., 2005.

[58] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[59] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS'88*, pages 621–626.

[60] A. W. To and L. Libkin. Algorithmic metatheorems for decidable LTL model checking over infinite systems. In *FoSSaCS'10*, pages 221–236.

[61] A. Vardhan. *Learning To Verify Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[62] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *ICFME'04*, pages 274–289.

[63] A. Vardhan and M. Viswanathan. LEVER: A tool for learning based verification. In *CAV'06*, pages 471–474.

[64] Wikipedia. Liquid water pouring puzzles. https://en.wikipedia.org/w/index.php?title=Liquid_water_pouring_puzzles&oldid=764748113, 2017. [Accessed: 24-February-2017].

[65] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV'98*, pages 88–97.

Lasso Detection using Partial-State Caching

Rashmi Mudduluru*, Pantazis Deligiannis*, Ankush Desai†, Akash Lal*, Shaz Qadeer*
*Microsoft Research, {t-rasmud, pdeligia, akashl, qadeer}@microsoft.com
†UC Berkeley, ankushdesai@gmail.com

***Abstract*—We study the problem of finding liveness violations in real-world asynchronous and distributed systems. Unlike a safety property, which asserts that certain bad states should never occur during execution, a liveness property states that a program should not remain in a bad state for an infinitely long period of time. Checking for liveness violations is essential to ensure that a system will always make progress in production.**

The violation of a liveness property can be demonstrated by a finite execution where the same system state repeats twice (known as lasso). However, this requires the ability to capture the state precisely, which is arguably impossible in real-world systems. For this reason, previous approaches have instead relied on demonstrating a long execution where the system remains in a bad state. However, this hampers debugging because the produced trace can be very long, making it hard to understand.

Our work aims to find liveness violations in real-world systems while still producing lassos as a bug witness. Our technique relies only on partially caching the system state, which is feasible to achieve efficiently in practice. To make up for imprecision in caching, we use retries: a potential lasso, where the same partial state repeats twice, is replayed multiple times to gain certainty that the execution is indeed stuck in a bad state.

We have implemented our technique in the P# programming language and evaluated it on real production systems and several challenging academic benchmarks.

***Index Terms*—Liveness checking, Distributed systems, Lasso detection, Testing**

I. Introduction

Concurrent programming is essential in modern software development, especially as the data and computing requirements grow beyond what is possible on a single processor core. Concurrency can be found either in the form of multi-threaded programs for multi-core processors, or as asynchronous distributed programs for multiple interconnected machines. In either case, writing *correct* concurrent programs is challenging. Subtle interactions between concurrently-running computations, such as thread interleavings or message reorderings, can lead to unexpected behaviors. Further, the non-deterministic nature of concurrency, often not controlled by the programmer, makes testing for such erroneous behaviors very difficult.

The expectations of correct behavior for a concurrent program (or any program for that matter) comes in two flavors: *safety* and *liveness* properties. Safety properties assert that a program never enters a *bad* or undesired state. The most natural form of a safety property is an *assertion*, a construct that is provided by most programming languages. Liveness properties, which are the focus of this paper, assert that the program does not *stay* in a bad state for an indefinite amount of time. (We are going to use the term *hot state* instead of *bad state* when discussing liveness properties.) Liveness properties are used to ensure that the program always makes progress. While safety specifications can be asserted and tested, there is no natural way to express liveness properties in programs, making it important to develop tools that help catch liveness violations.

As an example, consider the design of the Azure Storage vNext system [5]. As a typical cloud-storage system, vNext is a distributed program that stores user data reliably even under machine or disk failures. At a high level, it comprises of two main components: an extent node, which stores data on the local disk, and an extent manager that manages a set of extent nodes making sure that each piece of data lives on at least three extent nodes. Because machines or disks can fail at any time, it is possible that an extent node goes down. In such a case, the extent manager must detect the failure and use one (or both) of the other replicas to recreate another extent node with that data. Thus, while it is possible that the system enters a state where user data is not present on three nodes (a hot state), it must always eventually recover provided there are no more failures. A liveness violation for vNext is that the system remains in a hot state for an indefinite amount of time, even when there are no additional failures. Such kinds of bugs are very hard to find using traditional methods of testing [5]. The goal of our work is to build tools that help find liveness violations.

The model checking community has long studied this problem. The work concentrates on finding a *lasso*: a program execution that visits the same program state (say, s) twice. A lasso indicates the presence of an infinite execution because the execution from s to s can be repeated infinitely often. If, further, the lasso is hot, i.e., it is continuously in a hot state as it goes from s to s (including the state s itself) then it indicates a violation of the liveness property. A hot lasso naturally provides the user exact information on the execution segment that fails to make progress, according to the definition of what constitutes a hot state. For example, for vNext, the cycle in a potential violation would indicate the steps that the system is taking when some replica has gone down, but they fail to create a new replica.

There are several algorithms in this space that look for a hot lasso.[1] These algorithms are either exhaustive [3] or randomized [11], however they require access to the complete state of the program to know that the cycle in a lasso can be repeated indefinitely. For a distributed program, for instance,

[1]More generally, the algorithms look for violations of properties written in a temporal logic like LTL.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

the state would include the individual states of all concurrent processes as well as all the messages on the network. It may even have to include the state of the operating system if the program uses system resources (e.g., disk). This is an unrealistic task in a real-world setting, especially because it has to be done at each step of the program's execution. State-of-the-art tools like SPIN [12] and ZING [1] thus work on *models* of actual systems. The creation of the model is the user's responsibility. However, programmers are often reluctant to write models or maintain them as the software evolves given the time pressures of a fast-moving software industry.

A different approach of checking liveness is to directly execute the program (with some modifications to make executions deterministic) instead of relying on a model of the program. Without capturing the program state, these approaches are unable to find a lasso. Instead, they attempt to find a sufficiently long execution with a hot suffix (i.e., all states in the suffix are hot). Instances of this approach are implemented in the MACEMC tool [13] for distributed programs and the CHESS tool [15] for multi-threaded programs. We refer to this approach as the *temperature method*. (The system is additionally tagged with a *temperature* property. The temperature goes up by a unit when the system is in a hot state and it goes to zero when it transitions to a non-hot state. When the temperature exceeds a threshold, a violation is reported.) The temperature method requires the user to set the temperature threshold. Setting this threshold too low can result in false positives and setting it too high will produce long and hard-to-understand traces because there is no lasso or cycle that tells the user where the program failed to make progress.

Our goal is to provide the user with a lasso *without* relying on lengthy executions or the ability to cache the entire program state. There are two key ingredients to our approach. First, we use a *partial-state caching* mechanism that only captures a small part of the program state. By default, we capture partial details of each concurrently executing process and the *types* of the messages currently in flight between the processes. (We do provide convenient APIs so that users can *optionally* capture additional state.) Second, we execute the program (not a model) while taking over the program scheduling and message delivery. In each execution, we use the partial-state cache to find a repeating state. Because the caching is partial, we cannot say for sure if we have found a lasso or not. We overcome this limitation by taking the potential cycle and repeatedly re-executing it. If we are able to successfully re-execute the cycle (while continuing to stay in a hot state) for a large number of iterations then we flag this as a liveness violation and show the lasso (without subsequent re-executions) to the user.

We have implemented our approach and integrated it with the P# suite [4], [17]. P# is an extension of the C# language meant for developing asynchronous systems. P# comes with tools for thorough systematic testing of programs written in the language. P# is currently in use inside Microsoft for developing production systems. Using a collection of challenging academic benchmarks as well as production systems, we report on several interesting aspects of our approach and its comparison against the temperature method:

- We present a case study (§V) to show the advantage of inspecting a lasso, as compared to looking at a long trace.
- We evaluate and compare the algorithms on a set of benchmarks (§VI). We find that our lasso detection is more robust in terms of finding liveness violations; it has higher number of true positives and fewer false positives. The partial-state caching mechanism incurs an overhead of 2X in the running time on average compared to the temperature method which does not require any caching.

The rest of the paper is organized as follows. Section II sets up the notation and Section III formally describes our liveness checking algorithm. Section IV discusses our implementation based on P#. Section V presents a case study on the utility of having a lasso. Section VI presents our experimental results. Section VII discusses related work.

II. NOTATION AND DEFINITIONS

We consider a program as consisting of concurrently executing processes that communicate with each other via message passing. We formally model a program as a transition system. In particular, an asynchronous program P is a tuple $(S, \mathit{Pid}, T, \mathit{Hot}, s_0)$ where:

- S is the set of states of P.
- Pid is the set of process identifiers in the system.
- T: $\mathit{Pid} \times S \rightarrow S$ is the transition function of the program. T is a partial function. If $T(m, s) = s'$, then the process m can execute a step to take the program from state s to s'. We also say in this case that (s, s') is a transition of P and m is the *scheduled* process of that transition.
- Hot: $S \rightarrow \mathit{bool}$ is a function that maps a state to a Boolean indicating whether the state is *hot*.
- s_0 is the initial program state.

For the sake of convenience (and without loss of generality) we assume that for all states s_1 and s_2, if $T(m_1, s_1) = s_2$ and $T(m_2, s_1) = s_2$ then $m_1 = m_2$. Thus, a transition is uniquely identified by the source and target states. Let *Scheduled* be a partial function that maps a pair of states (s_1, s_2) to the unique process identifier that takes state s_1 to s_2. Formally, if $\mathit{Scheduled}(s_1, s_2)$ is defined then $T(\mathit{Scheduled}(s_1, s_2), s_1) = s_2$. Let *Enabled* be a function that maps a state s to the set of all processes *enabled* in that state. Formally, $\mathit{Enabled}(s) = \{m \mid \exists s'.T(m, s) = s'\}$. An example depicting the transition system of a program is shown in Figure 1. For instance, $\mathit{Enabled}(s_5) = \{p_2, p_3\}$ and $\mathit{Scheduled}(s_1, s_3) = p_2$.

An *execution trace* of P is a sequence of states $s_0, s_1, \cdots, s_n$ such that $\forall i \in \{0, 1, \cdots, n-1\}, \exists m \in \mathit{Pid} : T(m, s_i) = s_{i+1}$. A *lasso* L is an execution trace $s_0, \cdots, s_n$ such that for some i with $0 \leq i < n$, $s_n = s_i$. In this case, the execution trace $s_0, \cdots, s_{i-1}$ is called the *stem* of the lasso and the sequence $s_i, \cdots, s_n$ is the *cycle* of the lasso. The presence of a lasso indicates an infinite execution because the cycle can be repeated infinitely often. L is additionally a *hot* lasso if all states in its cycles are hot. Formally, $\mathit{HotLasso}(L)$ holds if $\forall k : i \leq k < n \Rightarrow \mathit{Hot}(s_k)$. Similarly, L is called a *fair* lasso

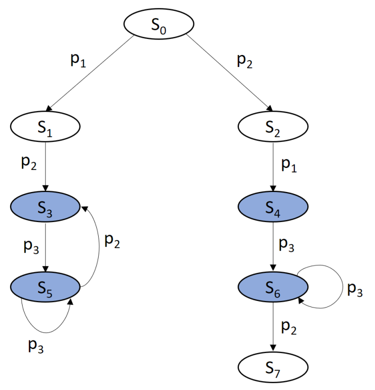


Fig. 1. A transition system. Nodes are program states and edges are transitions labelled with scheduled process name. The shaded nodes depict *hot* states.

if all processes that are enabled at some point in the cycle are scheduled in the cycle as well.[2] Formally, *FairLasso*(L) holds if $\bigcup_{k=i}^{n-1} \mathit{Enabled}(s_k) \subseteq \{\mathit{Scheduled}(s_k, s_{k+1}) | i \leq k \leq n-1\}$. A *liveness violation* of a program is a lasso that is both hot and fair.

Fairness is an important criteria while considering liveness properties. For unbounded runs, it is unlikely that a real system will starve an enabled process from executing. Programmers expect fairness and design their progress guarantees under this assumption. Thus, a hot lasso that is unfair will be a false positive from the user's perspective.

The example of Figure 1 has multiple lassos, for example $L_1 = s_0, s_1, s_3, s_5, s_3$, $L_2 = s_0, s_1, s_3, s_5, s_5$, and $L_3 = s_0, s_2, s_4, s_6, s_6$. All of these are hot lassos but only L_1 is fair. Lassos L_2 and L_3 are not fair because their cycles have p_2 enabled but it is never scheduled. For the purpose of this paper, only L_1 constitutes a liveness violation.

III. Liveness Checking Algorithms

This section outlines the two algorithms used for exploring the state space of a program in an attempt to find a liveness violation. We present the algorithm for the temperature method, inspired by previous work [13], [15]. Though this algorithm is not one of our contributions, we have implemented and presented it here mainly for the purpose of comparison. We then present our algorithm based on partial-state caching. A key hurdle in these algorithms is that, unlike a typical model-checking scenario, we cannot capture or store the entire state of a program. Instead, we only have the ability to inspect the current state, identify enabled processes and schedule an enabled process to make the program take a step. We cannot checkpoint the state, thus, we cannot identify a lasso by detecting a state seen previously in the execution.

Our algorithms will be parameterized by a *scheduler*, represented as a method GetNext that takes a state s as input and returns a process $m \in \mathit{Enabled}(s)$ that should be scheduled next. This method may have its own internal logic to decide which process to schedule next. One can use different implementations of GetNext that will, for instance, do a depth-first or a breadth-first exploration of the program by keeping track of previous decisions made. A GetNext implementation can also be randomized. It can, for example, pick and return a random element of *Enabled*(s) to do a pure random exploration.

[2]This property is usually termed as *strong fairness*.

Algorithm 1 ExecProgram

```
Input: Initial State $s_0$
Input: Scheduler method GetNext
Input: Method $M \in \{\mathit{TempMethod}, \mathit{PartialCaching}\}$,
Input: Maximum steps $B$: Int,
Input: Temperature threshold $TT$: Int
Input: Replay threshold $RT$: Int
 1: $s \leftarrow s_0$
 2: $n \leftarrow 0$
 3: $\mathit{Temp} \leftarrow 0$
 4: $\mathit{Trace} \leftarrow []$
 5: while $\mathit{Enabled}(s) \neq \emptyset \wedge n < B$ do
 6:     $m \leftarrow$ GetNext$(s)$
 7:     $s' \leftarrow T(m, s)$
 8:     $\mathit{Trace} \leftarrow \mathit{Trace} + (m, \mathit{Enabled}(s), \mathit{Hot}(s), \mathit{Hash}(s))$
 9:     $s \leftarrow s'$
10:     $n \leftarrow n + 1$
11:     if $M == \mathit{TempMethod}$ then
12:         $\mathit{Temp} \leftarrow$ CheckTemp$(s, \mathit{Trace}, \mathit{Temp}, TT)$
13:     else if $M == \mathit{PartialCaching}$ then
14:         $s, \mathit{Trace} \leftarrow$ CheckLasso$(s, \mathit{Trace}, RT)$
15:     end if
16: end while
```

Each of the two liveness detection methods repeatedly run ExecProgram (Algorithm 1) up to a user-specified bound on the maximum number of executions to explore. Each run of ExecProgram generates a program execution, according to the GetNext scheduler, and the execution is monitored for possible liveness violation. The two methods differ in how they perform the detection.

ExecProgram takes as input the initial state of the program s_0, a scheduler GetNext, the method to use for detecting violations, the maximum length of an execution B (after which exploration is truncated), and two threshold values TT and RT that we explain later.

The main loop of ExecProgram (line 5) runs as long as there are processes available to be scheduled or until the maximum length of the execution is reached. Each iteration of the loop executes the program for one step according to the scheduler (line 7) and then calls the selected detection method (lines 12 and 14).

Algorithm 2 describes the CheckTemp method. It simply increases the temperature value (line 2) if the current state is hot and checks if the threshold has been reached. If the current state is not hot, then the temperature is reset (line 7). It is easy to see that this method reports a liveness violation on a trace if the last TT steps of the trace were in a hot state.

Algorithm 2 CheckTemp$(s, \mathit{Trace}, \mathit{Temp}, TT)$

```
Input: Current state $s$
Input: Current trace $\mathit{Trace}$
Input: Current temperature $\mathit{Temp}$: Int
Input: Threshold $TT$: Int
Output: Updated temperature value
1: if $Hot(s)$ then
2:     $\mathit{Temp} \leftarrow \mathit{Temp} + 1$
3:     if $\mathit{Temp} = TT$ then
4:         REPORT-LIVENESS-BUG($\mathit{Trace}$)
5:     end if
6: else
7:     $\mathit{Temp} \leftarrow 0$
8: end if
9: return $\mathit{Temp}$
```

Algorithm 3 CheckLasso(s, Trace, RT)

```
Input: Current state $s$
Input: Current trace $\mathit{Trace}$
Input: Threshold value $RT$: Int
Output: New current state
Output: Updated trace
1: for all i: $\mathit{Hash}$(s) = $\mathit{hash}$($\mathit{Trace}$[i]) do
2:     $C \leftarrow \mathit{Trace}[i..\mathit{length}(\mathit{Trace})]$
3:     if $\mathit{Hot}(C) \wedge \mathit{Fair}(C)$ then
4:         return ReplayCycle$(s, C, \mathit{Trace}, RT)$
5:     end if
6: end for
```

Algorithm 4 ReplayCycle$(s, C, \mathit{Trace}, RT)$

```
Input: Current state $s$
Input: Potential cycle $C$
Input: Current trace $\mathit{Trace}$
Input: Threshold $RT$: Int
Output: New current state
Output: Updated trace
1: $\mathit{Trace}' \leftarrow \mathit{Trace}$
2: for $j = 0$ to $RT \times \mathit{length}(C) - 1$ do
3:     $i \leftarrow j$ mod $\mathit{length}(C)$
4:     $m \leftarrow \mathit{scheduled}(C[i])$
5:     if $m \notin \mathit{Enabled}(s)$ then
6:         return $(s, \mathit{Trace})$
7:     end if
8:     $s' \leftarrow T(m, s)$
9:     $\mathit{Trace} \leftarrow \mathit{Trace} + (m, \mathit{Enabled}(s), \mathit{Hot}(s), \mathit{Hash}(s))$
10:    $s \leftarrow s'$
11:    if $\mathit{Enabled}(s) \neq \mathit{enabled}(C[i + 1$ mod $\mathit{length}(C)]) \vee$
       $\neg \mathit{Hot}(s)$ then
12:        return $(s, \mathit{Trace})$
13:    end if
14: end for
15: REPORT-LIVENESS-BUG($\mathit{Trace}'$)
```

The *Trace* variable captures a summary of the current execution. It is a list of *trace events*. For a program transition $T(m, s_1) = s_2$, we record the trace event $(m, \mathit{Enabled}(s_1), \mathit{Hot}(s_1), \mathit{Hash}(s_1))$ capturing the process m that was scheduled and information about the source state s_1: the set of enabled machines in the state, if the state is hot or not, and a *hash* of the state. The function *Hash* computes a fingerprint of a state by hashing partial information gleaned from the program state. The next section details the information that we hash by default in our implementation, but users also have access to convenient APIs for hashing additional program state that is relevant to their own program. For the purpose of our algorithm, we only assume that *Hash* is indeed a function, i.e., identical states must be mapped to the same value. But the more information that is hashed about a state, the less likely it becomes that two different states are mapped to the same value.

The temperature method uses *Trace* for reporting a violation. A user can use the list of scheduled processes to replay the execution. Our implementation of the temperature method optimizes *Trace* by only keeping the process names in the trace events. The *PartialCaching* method, however, makes full use of trace events.

The PartialCaching Algorithm. For a trace event e, let *scheduled*(e) be its first element, *enabled*(e) be its second element, *hot*(e) be its third element and *hash*(e) be its last element. For a trace t (a list of trace events), let $t[i]$ be its i^{th} trace event. Let *length*(t) be the length of the trace. Let $t[i..j]$ be a sub-trace consisting of trace events $t[i], \cdots, t[j-1]$. We say that a trace t is *hot* (*Hot*(t)) if for each trace event $e \in t$, *hot*(e) is *true*. We say that a trace t is *fair* (*Fair*(t)) if $\bigcup_{e \in t} \mathit{enabled}(e) \subseteq \{\mathit{scheduled}(e) \mid e \in t\}$.

The CheckLasso method (algorithm 3) works as follows. For each new state s in the execution, it checks if *Hash*(s) has been seen earlier in the trace (line 1). A hit in the trace corresponds to a potential cycle, however, we cannot be sure because the hashing was only partial. It first checks if the potential cycle is hot and fair (line 3). If not, then it considers some other cycle. If it finds a hot and fair (potential) cycle, then to make sure, the method tries to *replay* the cycle.

The method ReplayCycle$(s, C, \mathit{Trace}, RT)$ (algorithm 4) takes over the scheduling of the execution and instead of calling GetNext, it uses C to make scheduling decisions. It attempts to replay C for RT number of times. If successful, the input *Trace* is reported as a liveness violation, with C marked as the hot and fair cycle of the lasso. The method proceeds as follows. Line 2 is the replay loop (for RT number of times). At line 4, the process to schedule is chosen from C. If m is not currently enabled (line 5), then the replay fails. Otherwise a step is executed by scheduling m (line 8). Next, line 11 checks if the new state matches the corresponding step ($i + 1$ **mod** *length*(C)) of C. If not then the replay fails, otherwise it keeps going.

Remarks. First, ReplayCycle does not check that the state hash matches with C during replay. We are only interested in replaying the scheduling decisions in C while making sure that the same set of processes are enabled (to ensure fairness). *Second*, when replay fails, we simply continue the program execution (in algorithm 1) from where the replay failed. This is because we cannot checkpoint state to rollback the execution from where the replay had started. We can potentially re-execute the program from the beginning to simulate rollback, but it adds extra cost to the algorithm. *Third*, in CHECKLASSO there may be many potential cycles on line 1. In our implementation, we go through these in a random order. Only the first hot and fair (potential) cycle is replayed and not the rest (line 4 executes a **return**) because the trace gets extended by the time replay fails.

A comparison of the two methods. Note that the temperature method does not have a fairness check. This is not possible because it does not produce a cycle that can be checked. To avoid false positives, previous work has relied instead on the scheduler to generate executions without starvation. For instance, MACEMC uses a randomized scheduler that picks a process randomly from the set of enabled processes; this makes it probabilistically unlikely that an enabled process will be starved in a long execution. For example, in the transition system of Figure 1, it is unlikely that a random scheduler will generate the execution $s_0, s_1, s_3, s_5, s_5, s_5, \cdots$. The randomness will ensure that p_2 is scheduled in state s_5 at some point; and likely the execution will have some alternation between states s_3 and s_5. Thus, in our experiments we limit the temperature method to use the random scheduler, whereas our partial-state caching method can use any scheduler. Random scheduling helps guard against unfairness, but it can also reduce bug-finding capabilities as illustrated by the following example.

A Dining Philosophers example. Consider a program with multiple processes, playing the role of a philosopher or a fork, arranged in a ring with alternating philosophers and forks. Each philosopher tries to acquire the fork on her left followed by the fork on her right. If she succeeds in getting both forks, she (eats and) releases both forks and quits. If she does not succeed in getting both the forks, she releases any fork with her and tries over again. This program has an infinite fair execution where each philosopher first gets the fork on their left, then they release them all realizing that the fork on the right is unavailable and so on.

The temperature method, with a randomized scheduler, is unable to detect the liveness violation: the ability to generate a particular trace decays roughly exponentially with the length of the trace, thus the method is unable to generate long traces. However, our partial caching method is able to find the violation, even while using a randomized scheduler. The reason is that it only needs to find the first iteration of a cycle after which replay will take over the scheduling. Even chances of hitting the first iteration decays exponentially with the number of philosophers. For example, for 2 to 5 philosophers, our partial-state caching method reports a (correct) liveness violation in $17.3\%, 4\%, 0.4\%, 0.03\%$ of the executions, respectively. The temperature method is not able to find a violation even for two philosophers (we used $TT = 50$).

IV. IMPLEMENTATION

We have implemented our techniques in the P# language [4], [17]. P# is designed for writing asynchronous programs. A P# program is a collection of *state machines* that run concurrently and communicate with each other by passing messages. A P# state machine (or machine for short) has an *input queue* that stores received messages and it can have an arbitrary number of fields of any C# type, just like a regular C# object. A machine can have multiple *states* in the sense of a finite-state-machine. (To avoid ambiguity with the multiple uses of the word "state", we will refer to this as a *MachineState*.) The messages are handled in a FIFO order. The user defines, separately for each MachineState, how the machine will handle a message of a particular type. It can execute a handler or transition to another MachineState. A handler can execute arbitrary (but sequential) C# code that may create more machines, send messages to other machines, block until it receives a specific message, or update the internal fields of the machine.

A P# program can run inside a single process (using a thread pool) or be deployed on a cluster of interconnected machines. P# is being used internally inside Microsoft to develop production services for Azure.

A liveness property in a P# program is specified with the help of a *monitor*. A *monitor* is a state machine that can receive but not send messages and whose *MachineStates* are optionally annotated as *hot*.[3] A monitor essentially observes the execution of the program. A liveness violation occurs if the program has a monitor in a hot state for an indefinite amount of time (for fair executions).

The P# runtime has a *bug-finding mode* that serializes the program execution on a single thread and systematically explores different interleavings of the program. P# has several *scheduling strategies* that can be used for exploration [8]. P# recommends a *portfolio* mode where testing is done in parallel, with each parallel instance using a different scheduler.

Our formalism in Section II assumed that the transition system of the program is deterministic except for the choice of which process to schedule next, i.e., given a program state s and an enabled process m, the state resulting from the execution of m was fixed ($T(m, s)$). A P# program, however, can have other sources of non-determinism, such as generating non-deterministic values. Our implementation is able to handle this non-determinism in data as well by generating these values randomly and capturing the generated value in the trace to allow for replay.

By default, we compute the fingerprint of a program state by hashing together the fingerprints of each machine. For a machine, we only look at the information that is directly visible to the P# runtime: this includes the name of MachineState that the machine is in currently and the sequence of message types in its inbox. We do not take into account the internal fields of the machine or payloads of the messages, each of

[3]P# also has the notion of *cold* and *warm* states but we do not discuss this feature in this paper.

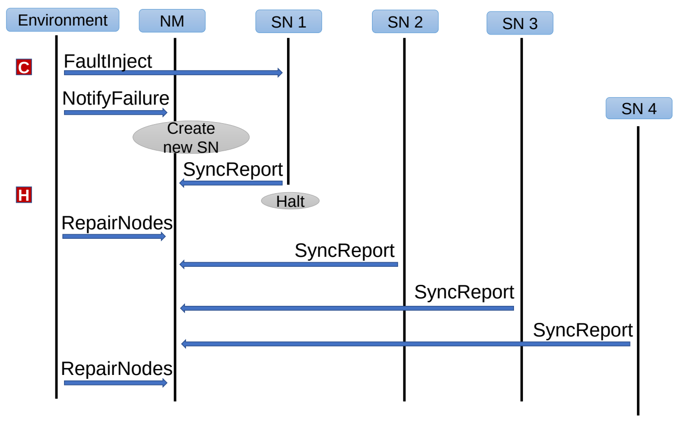


Fig. 2. ReplicatingStorage liveness bug

which can be of an arbitrary C# type, thus, hard to capture efficiently and automatically. P# offers an API by which a user can provide a more precise hash of a machine or of a message.

V. Case Study

This section compares the violations reported by the temperature method and our partial-state caching method on one benchmark. We find that a lasso expresses the liveness violation very naturally, and offers information that would otherwise be hard to deduce from a long trace.

It is important to note that even the MaceMc work [13] acknowledged that a user must be given more information than just a trace for identifying a liveness bug. MaceMc finds and displays a *critical transition* of the trace: a step of the program execution after which the program is doomed to violate the liveness property. In practice, this transition is one after which several random explorations fail to reach a non-hot state. A critical transition need not always exist (an example is the dining philosophers program), but it was present in the benchmark that follows.

The ReplicatingStorage benchmark is a simplified version of Azure Storage vNext (described in §I) that manifests a known real bug of the system. The P# program consists of a Node Manager (NM) that is responsible for handling the failure of Storage Nodes (SN) by creating new replicas. The SNs periodically send a **SyncReport** to the NM reporting a summary of the data that they currently store. We model the environment as a P# machine that randomly induces a failure to help us test the system. The program has a bug that is triggered when a SN sends a **SyncReport** to NM and then fails; the NM detects the failure, but just before it starts the repair, it gets the **SyncReport**. This causes the repair to not happen and the system continues without enough replicas. The fix is for NM to ignore **SyncReport** messages from nodes that it believes have failed.

Figure 2 shows the sequence of events that trigger the liveness bug. The Environment machine induces a node failure by sending a **FaultInject** message to SN-1. At this point, SN-1 simply enqueues this message. The Environment also sends a **NotifyFailure** to NM, which results in the creation of SN-4. Next, SN-1 sends a **SyncReport** to NM just before it handles the pending **FaultInject** message. Handling **FaultInject** causes the SN-1 machine to halt. When NM receives the **SyncReport** from SN-1, it updates its internal structures and (incorrectly) assumes that all replicas have the latest data. Subsequently, upon receipt of a periodic **RepairNodes** message (simulating a periodic callback), NM does not start a repair action and does not send the latest data to SN-4. It also ignores all the **SyncReport** messages that it receives from SN-4. As a result the system is stuck in a hot state from which it cannot recover.

In the scenario described above, the liveness monitor enters a *hot* state when SN-1 halts and the system never recovers after this. Therefore, the termination of SN-1 is the critical transition of this bug. However, this transition does not convey any useful information by itself: *any* liveness violation of this spec must start with a node failure.

In contrast, the states and messages in the cycle detected by our approach reflect the following information: NM repeatedly receives a **RepairNodes** message, which it handles, but NM does not send the latest data to any node and the SNs keep generating and sending **SyncReport** messages to NM. This information is more relevant to a user who knows that the newly created SN-4 needs to receive the latest data from NM, but it never does.

VI. Experiments

We experimented with a number of challenging academic benchmarks (which were authored by us), as well as production systems (for which we were not involved in their development). All benchmarks are written in P# [4] and are summarized in Table I, which shows lines of code (LoC), total number of machine types, total number of MachineStates and total number of message types[5].

The production systems include `PoolServer` and `Azure Storage vNext`. For the former, we picked two versions where the developers found interesting liveness violations. `Proposers` is a simplified version of the Paxos protocol obtained from previous work [9]. `Chord` [18] is a protocol implementing a distributed key-value store. `ReplicatingStorage` was described earlier (§V). `FailureDetector` is a failure detection protocol. `Process Scheduler`, `Leader Election`, and `Sliding Window` are P# versions of Spin benchmarks [12].

All benchmarks have a liveness bug, except for `Leader Election` and `Sliding Window`. A description of the bugs can be found in our technical report [14]. The production systems are proprietary; their liveness bugs were found using P# for the first time. We performed all our experiments on a 64-bit Windows Server machine with 64 GB RAM and 16 logical cores.

Table II reports results from our experiments. All benchmarks were executed with maximum steps set to 500 (variable

[4]https://github.com/p-org/PSharp
[5]https://github.com/p-org/PSharpLab/tree/master/FMCAD17

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE I
BENCHMARK CHARACTERISTICS

Benchmark	LoC	#Machines	#States	#Messages
Proposers	176	3	3	5
Chord	762	3	7	22
ReplicatingStorage	757	7	20	41
FailureDetector	436	4	10	19
Process Scheduler	641	7	7	27
Leader Election	340	3	4	9
Sliding Window	344	3	4	8
PoolServer - v1	12160	10	54	49
PoolServer - v2	27908	18	139	94
Azure Storage vNext	22967	6	15	27

B in Algorithm 1). The temperature threshold (TT) was set to 250. This value was chosen after initial experimentation which revealed that smaller values led to many false positives. The threshold on cycle replay (RT) was set to 10. It is interesting to note that our algorithm was robust with respect to this value though it was set arbitrarily: there were no false positives among the traces that we manually inspected; for the remaining traces, we confirmed that once a cycle was replayed for 10 iterations, it could be also replayed for at least 10K iterations. If replay failed, it would almost always fail in the first iteration of the cycle. Each benchmark was tested for 10K executions, with 1K executions performed in parallel with 10 parallel instances. As mentioned in Section III, we use a random scheduler for the temperature method, whereas we ran a portfolio of schedulers (suggested as default to P# users) for the partial-state caching method.

Table II shows the total time taken by the two approaches (in seconds); the percentage of executions that reported bugs (along with false positive ratio, when present); and the average length of reported traces. For PartialCaching, we report the average length of the trace (L_T) along with the average length of the cycle (L_C). We also show the number of times replay failed for potential cycles that were fair and hot (D). For `PoolServer-v2`, we were surprised to not find any bugs; when we checked with the developers, we found that they were using a maximum step bound of 5000. The row `Poolserver-v2-5k` uses this setting.

The results show that the extra information tracked by the PartialCaching method incurs an overhead over the temperature method (3.5X maximum, 2X on average). The overhead is mostly due to cycle detection in the trace, but also because of the partial hash computation and failed replay attempts. However, PartialCaching method has no false positives and has consistently better bug-finding capabilities, except for `ReplicatingStorage`.

In the case of `Azure Storage vNext`, the temperature method reports nearly 88% of the executions to be buggy, whereas our partial-state caching approach reports just 0.02%. To investigate the stark difference in the number of bugs reported by both approaches, we placed checks to see if the known buggy code was reached in the executions. It turns out that in nearly 82% of the executions, the bug was never triggered. These were all false positives. The temperature threshold was reached prematurely and the execution did not get sufficient time to do the repair. We also note that this is a lower bound on the number of false positives because triggering the buggy code is a necessary but not a sufficient condition for the liveness violation. The actual number of false positives may be higher. Setting the temperature threshold to 450 still reports over 80% false positives.

PartialCaching is able to find a bug in `Proposers` and `Poolserver-v2-5k` that is not found otherwise. The former is similar to the dining philosophers example (see §III); it requires the *proposer* machines to continuously out-bid each other in alternation. Thus, generating long traces is probabilistically unlikely. For `Poolserver-v2-5k`, its because one of the portfolio schedulers (based on priority-based scheduling [2]) exposed the corner case, which the random scheduler is unable to find. Additional experiments, which use just the random scheduler or improved state hashing, can be found in our technical report [14].

Discussion and Summary. The temperature method is easy to implement. It has been shown to work well in past work [13], and our experiments confirm this to some extent. However, initial experience with the developers using P# indicated two shortcomings. First, it required an understanding of the temperature threshold; one must give the system enough time to recover from a hot state. Like in the case of `vNext`, a low value can result in false positives. Second, when a trace was reported, developers had to spend time identifying a "loop" in their logic to see why the system failed to make progress. Our work on the partial-state caching algorithm was directly inspired by these shortcomings, under the constraint that full-state caching would not be possible in a real setting.

Our method finds a short cycle in most cases, usually much shorter than the trace itself and points directly to why the execution failed to make progress. Further, the technique is more robust, with fewer false positives and higher true positives. It is able to find bugs (e.g., `Poolserver-v2-5k`) that would be missed otherwise, which is invaluable to the user. We believe these advantages justify the relative modest runtime overhead of the approach. It also has the added advantage of supporting multiple schedulers, which we knew from past reported experience with safety properties, that it will be useful in exposing interesting behaviors [8].

VII. Related Work

Formal methods for checking liveness properties on programs is a widely studied area. The properties themselves are expressed in a temporal logic, most commonly in Linear Temporal Logic (LTL). These are compiled to a Buchi automaton, which is complemented and then intersected (via a cross-product construction) with the program. In the resulting system, the problem is then to find a lasso where the cycle contains an *accepting* state. The problem of limiting attention to fair traces is then just a matter of encoding fairness in LTL. The classical algorithm for finding such a lasso is the *Nested Depth First Search* (NDFS) algorithm [3]. State-of-the-art implementations of this algorithm, with various improvements

TABLE II
MAX STEPS: 500, ITERS:10000 (L_T : TRACE LENGTH; L_C: CYCLE LENGTH; D: DISCARDED CYCLES)

Benchmark	Time taken		% of Buggy Schedules		Trace length		
	Temperature	PartialCaching	Temperature	PartialCaching	Temperature	PartialCaching (L_T, L_C)	D
Proposers	5.87	10.23	0	**1.16**	-	21.9, 13	237
Chord	5.95	6.20	6.02	**6.03**	280.2	36.4, 3.2	0
ReplicatingStorage	45.80	160.76	**13.78**	11.96	367.7	295.4, 72.3	238
FailureDetector	48.17	74.34	0.03	**0.6**	254	78.1, 8	10076
Process Scheduler	36.77	117.3	0.33	**11.7**	411.5	236.1, 12.4	94671
Leader Election	6.83	7.38	0	0	-	-	60921
Sliding Window	35.95	125	0	0	-	-	0
PoolServer-v1	11.6	24.53	1.5	**2.57**	250.4	287.6, 26.6	14442
PoolServer-v2	46.63	68.85	0	0	-	-	0
PoolServer-v2-5k	28.75	64.11	0	**0.03**	-	624.2, 10.6	2
Azure Storage vNext	80.5	139.8	88.95 | 82.21 FP	0.02	309.6	235, 24	50

[7], [10], can be found in tools such as SPIN [12] and ZING [1]. However, this methodology requires the ability to cache the entire state (or a fingerprint of it). Consequently, both SPIN and ZING support their own input languages for writing models of actual systems. This is not readily possible in our setting.

The P programming language [6], [16] was co-designed with P#. It carries the same state-machine and message-passing structure as P#. However, unlike P# which is an extension of the C# language, P is its own programming language with its own data types and type system, designed in a manner that a P program can be compiled directly to ZING's input language. (A P program can interface with external C procedures for deployment in production, but the programmer is required to provide P models of any external procedure.) Thus, a P program can be analyzed using ZING. We coded some of our simpler benchmarks in P, where it was possible to capture the entire program state. However, ZING was often unable to find the bug in the program. This was because of two main reasons. First, the encoding of fairness in the translation to ZING introduced a lot of non-determinism in the model. Second, NDFS insists on a DFS order to explore the state space. Our benchmarks have infinite state spaces (or very large, even when the execution depth is constrained [7]). We found NDFS often getting lost in exploring sub-regions of the state space that did not have bugs and not being able to exhaustively cover the sub-region before it timed out.

Previous work on *stateless* techniques, which do not capture the program state, is usually restricted to safety properties. They advocate encoding liveness properties as safety assertions that check for progress explicitly [19]. Our approach instead automatically reports a lasso as a proof of "no progress". MACEMC [13] and CHESS [15] are stateless approaches that directly look for liveness violations. They have already been covered in the paper.

REFERENCES

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Conference on Concurrency Theory (CONCUR)*, pages 1–15, 2004.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[3] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer Aided Verification (CAV)*, pages 233–242, 1990.

[4] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous programming, analysis and testing with state machines. In *Programming Language Design and Implementation (PLDI)*, pages 154–164, 2015.

[5] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *File and Storage Technologies (FAST)*, pages 249–262, 2016.

[6] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *Programming Language Design and Implementation (PLDI)*, pages 321–332, 2013.

[7] A. Desai, S. Qadeer, S. Rajamani, and S. Seshia. Iterative cycle detection via delaying explorers. Technical report, March 2015.

[8] A. Desai, S. Qadeer, and S. A. Seshia. Systematic testing of asynchronous reactive systems. In *Foundations of Software Engineering (FSE)*, pages 73–83, 2015.

[9] M. Emmi and A. Lal. Finding non-terminating executions in distributed asynchronous programs. In *Static Analysis Symposium (SAS)*, pages 439–455, 2012.

[10] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Protocol Specification, Testing and Verification (PSTV)*, pages 109–124, 1993.

[11] R. Grosu and S. A. Smolka. Monte carlo model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 271–286, 2005.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Networked Systems Design and Implementation*, 2007.

[14] R. Mudduluru, P. Deligiannis, A. Desai, A. Lal, and S. Qadeer. Lasso detection using partial-state caching. Technical Report MSR-TR-2017-37, Microsoft Research, July 2017.

[15] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Programming Language Design and Implementation (PLDI)*, pages 362–371, 2008.

[16] P: Safe asynchronous event-driven programming. https://github.com/p-org/P.

[17] P#: Safe asynchronous event-driven .NET programming. https://github.com/p-org/PSharp.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001.

[19] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Networked Systems Design and Implementation (NSDI)*, pages 213–228, 2009.

Exact Quantitative Probabilistic Model Checking Through Rational Search

Matthew S. Bauer*, Umang Mathur*, Rohit Chadha†, A. Prasad Sistla‡ and Mahesh Viswanathan*
*University of Illinois at Urbana-Champaign, †University of Missouri,‡University of Illinois at Chicago

***Abstract*—Model checking of systems formalized using probabilistic models such as discrete time Markov chains (DTMCs) and Markov decision processes (MDPs) can be reduced to computing constrained reachability properties. Linear programming methods to compute reachability probabilities for DTMCs and MDPs do not scale to large models. Thus, model checking tools often employ iterative methods to approximate reachability probabilities. These approximations can be far from the actual probabilities, leading to inaccurate model checking results. In this article, we present a new algorithm and its implementation that improves approximate results obtained by scalable techniques like value iteration to compute exact reachability probabilities.**

I. Introduction

Probabilistic models such as discrete time Markov chains (DTMCs) and Markov decision processes (MDPs) are often used to describe systems in many application areas such as distributed systems [16], [35], hardware communication protocols [17], reliability engineering in circuits [11], [22], [32], [33], dynamic power management [10], [34], networking [28], [29] and security [13]. Probabilistic transitions in these models are used to capture random faults, uncertainty of environment and explicit randomization used in algorithms. The key verification tasks for such systems are often accomplished through Probabilistic Computation Tree Logic (PCTL) model checking [36]. The logic PCTL extends the temporal logic CTL with operators that provide the ability to reason quantitatively. For example, given $\bowtie \in \{\leq, <, \geq, >\}$, the formula $\mathcal{P}_{\bowtie p}[\psi]$ expresses the property that the measure of computation paths satisfying ψ is $\bowtie p$. PCTL model checking proceeds by recursively computing the set of states that satisfy subformulas of a given formula. Each recursive step, in turn, reduces to *constrained quantitative reachability*, wherein, given a DTMC/MDP, a set of good states G and a target set of states T, the goal is to compute the measure of the paths that reach T while remaining in G. If the model is decorated with *costs* or *rewards*, one may also be interested in computing the expected cost/reward of reaching T. It is well known that the constrained quantitative reachability problem for DTMCs and MDPs can be solved in polynomial time by reducing to linear programming [8], [36].

Despite its low asymptotic complexity, linear programming, unfortunately, doesn't scale well to large models and is rarely used in practice to solve the quantitative reachability problem. Instead, probabilistic model checkers [14], [15], [21], [24], [25], [30], typically compute *approximations* to the exact reachability probabilities through an iterative process. The results computed by these tools can be incorrect, primarily due to two sources of imprecision. The first is the use of finite precision arithmetic and floating point numbers to carry out calculations. The second is the use of approximate techniques like *value iteration*, where the exact reachability probabilities may only be approached in the limit. It is common practice for model checking tools to terminate value iteration in a finite number of steps, based on several different criteria, such as, when the change in the computed reachability probability between successive iterations is "small". This approximation step may lead to unsound results, for example, in systems where high magnitude changes in value iteration are preceded by periods of stability that cause iteration to terminate prematurely. Inaccuracies in model checking can get further compounded by the presence of nested probability operators in PCTL formulas when the sets of good states G and target states T are not correctly computed in the recursive step (see Example 3 on in Section III).

Contributions

In this article, we present a new algorithm and its implementation that *sharpens* approximate solutions computed by value iteration, to obtain the *exact* constrained reachability probability, allowing one to obtain accurate and reliable model checking results. The starting point of our approach is the observation that when transition probabilities in the model are rational numbers, the exact solution is also a rational number of polynomially many bits. The second ingredient in our technique is an algorithm due to Kwek and Mehlhorn [26], which, given a "close enough" approximation to a rational number, finds the rational number efficiently. Our algorithm works roughly as follows. We use value iteration to compute an approximate solution and then apply the Kwek-Mehlhorn algorithm to find a close candidate rational solution. Since the approximate solution we start with is of unknown quality, the candidate rational solution obtained may not be the exact answer. Therefore, we check if the candidate is the unique solution to the linear program that describes the system. This allows one to confirm the correctness of the candidate rational solution. If it is not correct, the process is repeated, starting with an approximate solution of improved precision. Precise details of the algorithm are given in Section IV.

We gratefully acknowledge the support of the following grants— Matthew S. Bauer was partially supported by NSF CNS-1314485; Umang Mathur was partially supported by NSF CCF-1422798; Rohit Chadha was partially supported by NSF CNS-1314338 and NSF CNS-1553548; A. Prasad Sistla was partially supported by CNS-1314485, CCF-1319754 and CCF-1564296; and Mahesh Viswanathan was partially supported by NSF CNS-1329991.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

We have implemented this approach as an extension of the PRISM model checker, called RATIONALSEARCH. Our tool computes exact constrained reachability probabilities and exact expected rewards for model checking DTMCs and MDPs against PCTL specifications. Evaluation of our implementation against a large set of examples from the PRISM benchmark suite [6] and case studies [7] shows that our technique can be applied to a wide array of examples. In many cases, our tool is orders of magnitude faster than the exact model checking engines implemented in state-of-the-art tools like PRISM [30] and STORM [15].

Related Work

The work closest in spirit to ours is [19], which presents an approach to obtain exact solutions for reachability properties for MDPs and discounted MDPs. The basic idea there is to interpret the scheduler obtained for an approximate solution, as a *basis* for the linear program corresponding to the verification question. By examining the optimality of the solution associated with this basis, the exact solution can be obtained by improving the scheduler using the Simplex algorithm. This is significantly different from our approach. In particular, for DTMCs (where there is no scheduler), the approach of [19] reduces to solving a linear program, which is known to be not scalable. Since the implementation from [19] is not available, we could not experimentally compare with this approach.

To overcome the convergence problems with value iteration, techniques like *interval iteration* [9], [20], [38], which utilize two simultaneous value iteration procedures converging to the exact probabilities values from above and below, have been proposed. This allows one to bound the error produced by approximation techniques. Additionally, several tools [15], [30] implement exact quantitative model checking as an extension of *parametric model checking*, which synthesizes symbolic algebraic expressions over parameters of the model, representing quantitative properties of a system. These expressions can be evaluated under a concrete instantiation of the parameters to produce exact solutions.

II. BACKGROUND

A common technique in the analysis of systems is to model them as *state transitions systems* where states describe information about the system at a point in time and transitions describe how the system evolves from one state to another. When this evolution is governed by random phenomena, such state transition systems can then be enriched to capture probabilistic behavior. The resulting model is known as a DTMC, in which every state is mapped to a distribution over the successor states. MDPs generalize DTMCs, in that, the distribution over the successor states is non-deterministically chosen. We next formalize DTMCs and MDPs.

Discrete time Markov chains (DTMCs)

A DTMC is a tuple $\mathcal{M} = (Z, \Delta, \mathbf{C}, L)$ where Z is a set of states, $\Delta : Z \rightarrow \mathsf{Dist}(Z)$ is the *probabilistic transition function* that maps every state to a probability distribution over Z, $\mathbf{C} : Z \times Z \mapsto \mathbb{Q}^{\geq 0}$ is a cost (or reward) structure and $L : Z \rightarrow 2^{\mathsf{AP}}$ is a labeling function that maps states to subsets of AP, the set of atomic propositions. We will restrict our attention to DTMCs with a finite number of states. For each $z \in Z$, $\Delta(z)$ defines a discrete probability distribution over Z, that is, $\Delta(z)(z') \geq 0$ for all $z' \in Z$, and $\sum_{z' \in Z} \Delta(z)(z') = 1$. We will henceforth denote $\Delta(z)(z')$ by $\Delta(z, z')$. A path ρ of $\mathcal{M}$ is a sequence of states $z_0 \rightarrow z_1 \rightarrow \cdots$ such that $\Delta(z_i, z_{i+1}) > 0$. We write $\rho(i)$ to denote the i^{th} state z_i in ρ. We denote the set of all infinite paths of $\mathcal{M}$ by $\mathsf{Paths}(\mathcal{M})$ and the set of all infinite paths of $\mathcal{M}$ starting from state z by $\mathsf{Paths}_z(\mathcal{M})$. For a finite path $\rho_{\mathsf{fin}} = z_0 \rightarrow \cdots \rightarrow z_m$ we associate a probability measure $\mathsf{prob}(\rho_{\mathsf{fin}}) = \prod_{i=0}^{m-1} \Delta(z_i, z_{i+1})$. The cylinder set of ρ_{fin} is $\mathsf{Cyl}(\rho_{\mathsf{fin}}) = \{\rho \in \mathsf{Paths}(\mathcal{M}) \mid \rho_{\mathsf{fin}} \text{ is a prefix of } \rho\}$ and its associated probability measure is $\mathsf{prob}(\mathsf{Cyl}(\rho_{\mathsf{fin}})) = \mathsf{prob}(\rho_{\mathsf{fin}})$, which can be extended to a unique probability measure over the smallest σ-algebra containing all cylinder sets. The cost associated with ρ_{fin} is $\mathsf{cost}(\rho_{\mathsf{fin}}) = \sum_{i=0}^{m-1} \mathbf{C}(z_i, z_{i+1})$. Let $F \subseteq Z$. For a path $\rho \in \mathsf{Paths}(\mathcal{M})$, the cost of reaching F, denoted $\mathsf{cost}(F)(\rho)$, is the cost of the shortest prefix of ρ that reaches F, and is ∞ if no such prefix exists. The expected cost incurred for reaching F starting from z is given by $\mathsf{E}[\mathsf{cost}_z(F)] = \sum\limits_{\rho \in \mathsf{Paths}_z(\mathcal{M})} \mathsf{prob}(\rho) \cdot \mathsf{cost}(F)(\rho)$.

Example 1: Consider an embedded control system [27] comprised of an input processor, a main processor, an output processor and a bus. In each cycle of the system, the input processor collects data from a set of n sensors $S_1, S_2, \ldots, S_n$. The main processor polls the input processor and passes instructions to the output processor controlling a set of m actuators $A_1, A_2, \ldots A_m$. Communication between processors occurs over the bus. The system is designed to tolerate failures in a limited number of components. If the input processor reports that the number of sensor failures exceeds some threshold MAX_FAILURES, then the main processor shuts the system down. Otherwise, it activates the actuators, which again, are prone to failure. When the probabilities, with which each of these components fail, are known, one can model the system's reliability using a DTMC. In Figure 1, we give a DTMC that models a single cycle of such a system with $n = 2$ sensors and $m = 1$ actuator. For simplicity, we assume that each sensor fails with probability E_s and each actuator fails with probability E_a. States of the model are labeled with $e_1^s, ..., e_n^s \in \{0, 1\}$ and $e_1^a, ..., e_m^a \in \{0, 1\}$, where $e_i^s = 1$ denotes the failure of sensor S_i and $e_i^a = 1$ denotes the failure of actuator A_i. In Figure 1, we omit labels if they are not relevant in a particular state.

Markov decision processes (MDPs)

An MDP is a tuple $\mathcal{M} = (Z, \mathsf{Act}, \Delta, \mathbf{C}, L)$ where Z is a finite set of states, Act is a set of actions, $\Delta : Z \times \mathsf{Act} \hookrightarrow \mathsf{Dist}(Z)$ is the *probabilistic transition function* that maps pairs of states and actions to probability distributions over Z, $\mathbf{C} : Z \times \mathsf{Act} \times Z \rightarrow \mathbb{Q}^{\geq 0}$ is a cost (or reward) structure and $L : Z \rightarrow 2^{\mathsf{AP}}$ is a labeling function. The set $\mathsf{enabled}(z) =$

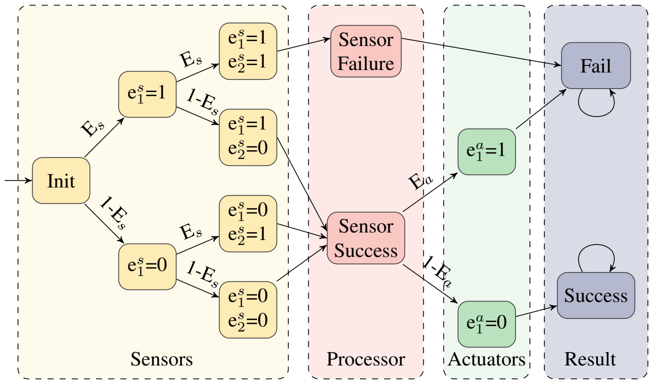


Fig. 1. Markov chain for a simple embedded control system with two sensors and one actuator tolerating a single sensor fault.

$\{\alpha \in \mathsf{Act} \mid \Delta(z, \alpha)$ is defined$\}$ of actions enabled from each state z is assumed to be non-empty for every $z \in Z$. An MDP, therefore, differs from a DTMC, in that, at each state z, there is a choice among several possible distributions. The choice of which distribution to *trigger* is resolved by a *scheduler* (or an attacker). Informally, an MDP $\mathcal{M}$ evolves as follows. It starts from some state $z_0 \in Z$. After i execution steps, if $\mathcal{M}$ is in state z, the scheduler chooses an action $\alpha \in \mathsf{enabled}(z)$, which then defines a unique probability distribution μ given by $\Delta(z, \alpha)$. The process then moves to state z' in step $(i+1)$ with probability $\Delta(z, \alpha)(z')$. We will write $\Delta(z, \alpha, z')$ to denote $\Delta(z, \alpha)(z')$ when $\alpha \in \mathsf{enabled}(z)$. A path ρ of an MDP $\mathcal{M}$ is a sequence $z_0 \xrightarrow{\alpha_1} z_1 \xrightarrow{\alpha_2} \cdots$ such that for each $i \geq 0$, $\alpha_{i+1} \in \mathsf{enabled}(z_i)$ and $\Delta(z_i, \alpha_{i+1}, z_{i+1}) > 0$.

Formally, a scheduler is a function $\mathfrak{S} : Z^+ \to \mathsf{Act}$ such that for every finite sequence $\tau = z_0 z_1 \ldots z_k \in Z^+$, we have $\mathfrak{S}(\tau) \in \mathsf{enabled}(z_k)$. A path $z_0 \xrightarrow{\alpha_1} z_1 \xrightarrow{\alpha_2} \cdots$ is a $\mathfrak{S}$-path if $\mathfrak{S}(z_0 z_1 \ldots z_i) = \alpha_{i+1}$ for all $i \geq 0$. We will write $\mathsf{Paths}(\mathcal{M})$ for the set of infinite paths, $\mathsf{Paths}_z(\mathcal{M})$ for the set of infinite paths starting from z, $\mathsf{Paths}^{\mathfrak{S}}(\mathcal{M})$ for the set of infinite $\mathfrak{S}$-paths, and $\mathsf{Paths}^{\mathfrak{S}}_z(\mathcal{M})$ for the set of infinite $\mathfrak{S}$-paths starting from z. The set of all schedulers will be denoted by $\mathcal{S}$. A scheduler $\mathfrak{S} \in \mathcal{S}$ for MDP $\mathcal{M}$ induces a (potentially infinite) DTMC $\mathcal{M}^{\mathfrak{S}}$ such that $\mathsf{Paths}(\mathcal{M}^{\mathfrak{S}}) = \mathsf{Paths}^{\mathfrak{S}}(\mathcal{M})$. The definitions for the measure and cost associated with paths can then be naturally lifted from DTMCs. Interested readers should refer to standard texts such as [8], [37] for more details.

Probabilistic computation tree logic (PCTL)

Properties of DTMCs and MDPs can be expressed in the logic PCTL, which extends the temporal logic CTL with the ability to reason quantitatively. Let $a \in \mathsf{AP}$ be an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$, $c \in \mathbb{Q}^{\geq 0}$ and $k \in \mathbb{N}$. Below, we begin by defining PCTL for DTMCs and then give the extension to MDPs.

Definition 1: The syntax of PCTL is

$$\phi ::= \mathsf{true} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{E}_{\bowtie c}[\phi]$$

where

$$\psi ::= \mathcal{X}\phi \mid \phi \mathcal{U} \phi$$

In Definition 1, ϕ is a state formula used to describe properties of states and ψ are path formulas used to model properties of paths. We now formalize the semantics of PCTL.

Definition 2: Let $\mathcal{M} = (Z, \Delta, \mathbf{C}, L)$ be a DTMC, ϕ, ϕ_1, ϕ_2 be state formulas and ψ be a path formula. The satisfaction relation $\models$ for PCTL state formulae is defined inductively as

$$\begin{array}{ll} z \models \mathsf{true} & \text{for all } z \in Z \\ z \models a & \Leftrightarrow a \in L(z) \\ z \models \neg\phi & \Leftrightarrow z \not\models \phi \\ z \models \phi_1 \wedge \phi_2 & \Leftrightarrow z \models \phi_1 \text{ and } z \models \phi_2 \\ z \models \mathcal{P}_{\bowtie p}[\psi] & \Leftrightarrow p_z(\psi) \bowtie p \\ z \models \mathcal{E}_{\bowtie c}[\phi] & \Leftrightarrow e_z(\phi) \bowtie c \end{array}$$

where $p_z(\psi) = \mathsf{prob}(\{\rho \in \mathsf{Paths}_z(\mathcal{M}) \mid \rho \models \psi\})$, $e_z(\phi) = \mathsf{E}[\mathsf{cost}_z(Z_\phi)]$ with $Z_\phi = \{z' \in Z | z' \models \phi\}$, and the satisfaction relation for paths and path formulae is defined inductively as

$$\begin{array}{ll} \rho \models \mathcal{X}\phi & \Leftrightarrow \rho(1) \models \phi \\ \rho \models \phi_1 \mathcal{U} \phi_2 & \Leftrightarrow \exists i \geq 0 : (\rho(i) \models \phi_2 \ \&\ \forall j < i : \rho(j) \models \phi_1) \end{array}$$

When the underlying model $\mathcal{M}$ is an MDP, the semantics of PCTL formulae stay the same, except for the semantics of $\mathcal{P}_{\bowtie p}[\psi]$ and $\mathcal{E}_{\bowtie c}[\phi]$, which now require a quantification over all schedulers. Let $p^{\mathfrak{S}}_z(\psi) = \mathsf{prob}(\{\rho \in \mathsf{Paths}^{\mathfrak{S}}_z(\mathcal{M}) | \rho \models \psi\})$. One can analogously define $e^{\mathfrak{S}}_z(\phi)$ for a scheduler $\mathfrak{S}$.

Definition 3: Let $\mathcal{M}$ be an MDP, ϕ be a state formula and ψ be a path formula. The satisfaction relation $\models$ for PCTL state formulae is defined identically to Definition 2, with the exception of the following cases.

$$\begin{array}{ll} z \models \mathcal{P}_{\bowtie p}[\psi] & \Leftrightarrow \forall \mathfrak{S} \in \mathcal{S},\ p^{\mathfrak{S}}_z(\psi) \bowtie p \\ z \models \mathcal{E}_{\bowtie c}[\phi] & \Leftrightarrow \forall \mathfrak{S} \in \mathcal{S},\ e^{\mathfrak{S}}_z(\phi) \bowtie c \end{array}$$

For a path formula ψ (resp. state formula ϕ), we write $\mathcal{P}_{=?}[\psi]$ (resp. $\mathcal{E}_{=?}[\phi]$) to represent the solution vector V, given by $\mathsf{V}(z) = p_z(\psi)$ (resp. $e_z(\phi)$) for all $z \in Z$. Strictly speaking, $\mathcal{P}_{=?}[\cdot]$ and $\mathcal{E}_{=?}[\cdot]$ are not part of PCTL syntax. However, we henceforth extend the PCTL syntax to allow $\mathcal{P}_{=?}[\cdot]$ and $\mathcal{E}_{=?}[\cdot]$ as the outermost operator.

Example 2: Consider the DTMC modeling an embedded control system from Example 1. One can describe many important properties of this model using PCTL:

1) The probability of success.
   $\mathcal{P}_{=?}$ [$\mathsf{true}\, \mathcal{U}$ "Sucess"]

2) The probability that there are no sensor failures.
   $\mathcal{P}_{=?}$ [$\mathsf{true}\, \mathcal{U}\ (e^s_1 + ... + e^s_n = 0)$]

3) The probability that actuator A_1 does not fail given that sensor S_1 fails with probability $\bowtie 1/2$.
   $\mathcal{P}_{=?}$ [$\mathcal{P}_{\bowtie \frac{1}{2}}[\mathsf{true}\, \mathcal{U}\ (e^s_1{=}1)]\ \mathcal{U}\ \mathcal{P}_{\leq 0}[\mathsf{true}\, \mathcal{U}\ (e^a_1{=}1)]$]

PCTL model checking

Similar to the model checking algorithm for CTL, the PCTL model checking algorithm recursively computes the set of states satisfying a state sub-formula. We will begin by restricting our attention to DTMCs.

Let ϕ, ϕ' be state formulas. To compute $\mathcal{P}_{=?}[\phi\ \mathcal{U}\ \phi']$, one recursively computes the set of states Z_ϕ and $Z_{\phi'}$ satisfying

ϕ and ϕ' respectively. These can be used to derive, for every $z{\in}Z$, the quantity $p_z(\phi\,\mathcal{U}\,\phi')$ representing the probability of reaching the set $Z_{\phi'}$ while remaining in the set Z_ϕ, starting from the state z.

Now, $p_z(\phi\ \mathcal{U}\ \phi')$ is the unique solution to the following linear program:

$$p_z(\phi\,\mathcal{U}\,\phi'){=}\begin{cases} 0 & \text{if } z \in \mathsf{Prob0} \\ 1 & \text{if } z \in \mathsf{Prob1} \\ \sum\limits_{z' \in Z} \Delta(z,z') \cdot p_{z'}(\phi\,\mathcal{U}\,\phi') & \text{otherwise} \end{cases} \tag{1}$$

where $\mathsf{Prob0}, \mathsf{Prob1}$ can be determined via a pre-computation step that analyzes the underlying graph of the DTMC. For computing $\mathcal{P}_{\bowtie p}[\phi\ \mathcal{U}\ \phi']$, one computes $\mathcal{P}_{=?}[\phi\ \mathcal{U}\ \phi']$ and compares $p_z(z\,\mathcal{U}\,z') \bowtie p$ for every $z \in Z$. The computation for $\neg\phi$, $\phi \wedge \phi'$, $\mathcal{E}_{=?}[\phi]$ and $\mathcal{P}_{\bowtie c}[\mathcal{X}\phi]$ is similar.

When the underlying model is an MDP, the computation for $\mathcal{P}_{\bowtie p}[\phi\ \mathcal{U}\ \phi']$ reduces to solving the following linear optimization problem when $\bowtie \in \{<, \leq\}$

$$\begin{array}{ll} \min \sum\limits_{z \in Z} p_z(\phi\,\mathcal{U}\,\phi') \quad \text{subject to} & \\ p_z(\phi\,\mathcal{U}\,\phi') = 0 & \text{if } z \in \mathsf{Prob0} \\ p_z(\phi\,\mathcal{U}\,\phi') = 1 & \text{if } z \in \mathsf{Prob1} \\ p_z(\phi\,\mathcal{U}\,\phi') \geq \sum\limits_{z' \in Z} \Delta(z,\alpha,z') \cdot p_{z'}(\phi\,\mathcal{U}\,\phi') & \\ \qquad \text{for each } \alpha \in \mathsf{enabled}(z) & \text{otherwise} \end{array} \tag{2}$$

When $\bowtie \in \{>, \geq\}$, the objective changes to maximization and the direction the last inequality is reversed.

Value iteration

One can equivalently express the system of equations described in (1) and (2) as (3) and (4) for DTMCs and MDPs respectively (for some appropriate matrix A and vector b),

$$\bar{x} = A\bar{x} + b \tag{3}$$

$$\bar{x}(z) = \max\{\Delta(z,\alpha,z') \cdot \bar{x} \mid \alpha{\in}\mathsf{enabled}(z)\} \tag{4}$$

An alternate approach to solving the above set of equations is *value iteration*, which iteratively computes the solution vector as the limit of the sequence $\{\bar{x}_i\}_{i\geq 0}$ given by $\bar{x}_{i+1} = A\bar{x}_i + \bar{b}$ starting with $\bar{x}_0(z) = 1$ if $z \in \mathsf{Prob1}$ and $\bar{x}_0(z) = 0$ otherwise, for the case of DTMCs. The iterative formulation for MDPs is also similar. Value iteration techniques remain the popular choice for industrial tools that analyze PCTL properties, because, when equipped with a suitable stopping criterion, value iteration beats state-of-the-art linear programming techniques, despite their theoretically better asymptotic complexity. State-of-the-art quantitative model checkers further enhance the performance of value iteration by performing arithmetic operations using *Multi-terminal binary decision diagrams* (MTBDDs) [18], [23]. MTBDDs generalize BDDs [12] by allowing terminal values to be different from 0 or 1. Similar to the role of BDDs in symbolic model checking [31], MTBDD based model checkers leverage the performance benefit due to the succinct representations of the data structures involved.

III. Approximate model checking

As discussed above, solving quantitative properties of DTMCs and MDPs by a reduction to linear programming does not scale well enough to make it a viable solution technique in practice. As a result, techniques to approximate solutions using floating point arithmetic, such as value iteration, have been widely adopted. In addition to errors introduced by overflows in floating point numbers, several other sources of imprecision can arise in quantitative model checkers that employ approximate solution techniques.

a) Value iteration and convergence: Value iteration, discussed above, computes a sequence of vectors $\{v_i\}_{i\geq 0}$ that converge to the solution vector V for a PCTL formula. In many cases, the sequence does not converge in a finite number of steps, and therefore model checkers terminate the sequence when successive vectors v_k and v_{k+1} become "close enough". The choice of stopping criterion is based largely on heuristics. The PRISM model checker, for example, implements two criteria (i) *absolute criterion*, and (ii) *relative criterion*. Under the absolute criterion, value iteration terminates if $\|v_{k+1} - v_k\| < \epsilon$ for some $\epsilon > 0$. Under the relative criterion, termination occurs when $\frac{\|v_{k+1}-v_k\|}{\|v_k\|} < \epsilon$. Both of these convergence criteria can result in solutions that are very far from the actual answers. In [20], the authors give a DTMC and a PCTL property whose solution is $\frac{1}{2}$, yet PRISM reports 9.77×10^{-4} for the absolute criterion and 0.198 for the relative criterion.

b) Nested reachability and PCTL: State-of-the-art quantitative model checking tools employing floating point arithmetic often fail to produce accurate solutions to properties of the form $\mathcal{P}_{\bowtie p}(\psi)$ when the probability of satisfying ψ is very close to p. First observed in [39], we shall also demonstrate this phenomenon using the DTMC from Example 1. For the sake of illustration, let $\mathrm{E}_s = \frac{1}{2}$. Clearly, from the initial state, the probability of reaching a state where sensor 1 fails is exactly $\frac{1}{2}$ and hence the formula $\mathcal{P}_{<\frac{1}{2}}$ [$\mathsf{true}\ \mathcal{U}\ (e_1^s{=}1)$] evaluates to false for the initial state. However, PRISM returns true. Errors such as these can be compounded in PCTL formulas containing nested operators, wherein the recursive step of the model checking algorithm returns an incorrect set of states. This can lead to substantial logical errors in model analysis where the reported probabilities are very far from the actual ones.

Example 3: Let us instantiate the DTMC from Example 1 with $n = 14$ sensors, $m = 1$ actuator, MAX_FAILURES=1 and with E_s = E_a = $\frac{1}{2}$. Recall the third PCTL property of the embedded control system given in Example 2:

$$\mathcal{P}_{=?}\ [\ \mathcal{P}_{\bowtie \frac{1}{2}}[\mathsf{true}\,\mathcal{U}\ (e_1^s{=}1)]\ \mathcal{U}\ \mathcal{P}_{\leq 0}[\mathsf{true}\,\mathcal{U}\ (e_1^a{=}1)]\ ].$$

When $\bowtie$ is $\leq$, the PRISM model checker returns the value "0.7096993582589287" for the initial state. Using our tool RationalSearch, one can verify that the probability is actually 212895/229376, or "0.9281485421316964". Further, when $\bowtie$ is $<$, PRISM again returns "0.7096993582589287". This time, the actual solution, is 0, and our tool concurs with this. This is because, PRISM incorrectly computes the set of states satisfying $\mathcal{P}_{\bowtie\frac{1}{2}}[\mathsf{true}\ \mathcal{U}\ (e^s_1{=}1)]$. This error in the recursive step results in an incorrect formulation of the constraints in the outermost constrained reachability problem.

IV. Exact model checking

As demonstrated in the previous section, approximate solution techniques can lead to unreliable results and to incorrect analysis of systems. To rectify this serious limitation, several approaches that attempt to give exact (or high-precision) solutions to the PCTL model checking problem have been proposed. One such technique, *interval iteration* [9], [20] carries out two simultaneous value iterations converging to the solution from above and below, allowing one to bound the error in the approximate solution. This approach is again vulnerable to floating point errors and it doesn't directly give exact answers. Another approach for computing exact answers is *parametric model checking*. In this approach, one synthesizes symbolic algebraic expressions representing the quantitative properties of the model, by treating the parameters in the underlying model (constants, transition probabilities and rewards) symbolically. These symbolic expressions evaluate to the exact arithmetic values when instantiated with the concrete values in the model. However, generating these expressions, in general, is an expensive task. See our experiential evaluation in Section V for a comparison with this technique.

Example 4: Again consider the DTMC modeling an embedded control system with the parameters given in Example 3, where it was demonstrated that approximate model checking techniques can lead to incorrect logical analysis of the system. To guarantee the correctness of one's analysis, exact solution techniques must be employed. Unfortunately, the exact model checking engines of PRISM and STORM do not scale well enough to analyze this example, which contains about 4.8 million states and about 44 million transitions. Under our test setup (see Section V), both tools reached a 30 minute timeout when trying to analyze the properties from Example 3. On the other hand, our tool RationalSearch found the exact answer to both the formulae in under 1 minute.

The Kwek-Mehlhorn algorithm

Given an ordered set of integers of bounded size, the classical binary search algorithm can be used to locate the smallest element larger than a given value. Kwek and Mehlhorn [26] extend this methodology to efficiently locate the rational number with the smallest size in a given interval. Here we present a novel application of this technique where approximate answers to quantitative model checking problems can be used to efficiently generate exact solutions.

Consider an interval $I = [\frac{\alpha}{\beta}, \frac{\gamma}{\delta}]$ with rational end-points. It was established [26] that for any interval $I = [\frac{\alpha}{\beta}, \frac{\gamma}{\delta}]$, there exists a unique rational $a_{\min}(I)/b_{\min}(I)$ such that for all rational numbers $\frac{a}{b} \in I$, $a_{\min}(I) \leq a$ and $b_{\min}(I) \leq b$. Further, this minimal fraction $a_{\min}(I)/b_{\min}(I)$ can be found using Algorithm 1 from [26].

Let $Q_M = \{p/q \mid p, q \in \{1, ..., M\}\} \cap [0, 1]$. For $\mu \in \mathbb{N}$, if $\frac{a}{b} \in Q_M$ is contained in the interval $[\frac{\mu}{2M^2}, \frac{\mu+1}{2M^2}]$ of length $\frac{1}{2M^2}$

Algorithm 1 Compute the minimal rational in $[\frac{\alpha}{\beta}, \frac{\gamma}{\delta}]$

```
function FINDFRACTION($\alpha$, $\beta$, $\gamma$, $\delta$):
    if $\left\lfloor\frac{\alpha}{\beta}\right\rfloor = \lfloor\frac{\gamma}{\delta}\rfloor$ and $\frac{\alpha}{\beta} \notin \mathbb{N}$ then
        $b, a \leftarrow$ FINDFRACTION($\delta$, $\gamma$ mod $\delta$, $\beta$, $\alpha$ mod $\beta$)
        return $\left\lfloor\frac{\alpha}{\beta}\right\rfloor b + a,\ b$
    else
        return $\left\lceil\frac{\alpha}{\beta}\right\rceil, 1$
    end if
end function
```

then $\frac{a}{b}$ is the unique element of Q_M in $[\frac{\mu}{2M^2}, \frac{\mu+1}{2M^2}]$. It turns out that $\frac{a}{b}$ must also be the minimal element of $[\frac{\mu}{2M^2}, \frac{\mu+1}{2M^2}]$, meaning it can be found using the algorithm from Algorithm 1 in time $O(\log M)$.

Rational search

In this section, we explain our approach for exact quantitative model checking of PCTL formulas. The key insight we exploit is that value iteration typically converges very fast and produces a precise enough answer. Using this precise approximation, we can then effectively construct a small interval for which the Kwek-Mehlhorn algorithm can find the exact answer. In the following, we formalize this procedure.

We begin the presentation of our exact model checking algorithm by first describing how a given approximate solution vector corresponding to a set of equations, like those in (1) and (2), can be refined to get the exact vector. This process is formalized in Algorithm 2, which takes as input the model $\mathcal{M}$, a maximum precision P and a state-indexed vector $\mathsf{V}^\dagger$ that approximates V (defined after Definition 2).

Algorithm 2 Sharpen

```
function SHARPEN($\mathcal{M}$, $P$, $\mathsf{V}^\dagger$):
    for all $p \in \{1, ..., P\}$ do
        for all $z \in Z$ do
            $\alpha, \beta, \gamma, \delta \leftarrow$ BOUNDS($p$, $\mathsf{V}^\dagger(z)$)
            $\mathsf{V}^\star(z) \leftarrow \lfloor\mathsf{V}^\dagger(z)\rfloor +$ FINDFRACTION($\alpha, \beta, \gamma, \delta$)
        end for
        if FIXPOINT($\mathcal{M}$, $\mathsf{V}^\star$) then
            return $\mathsf{V}^\star$
        end if
    end for
    return null
end function
```

For a given precision p and state z, BOUNDS$(p, \mathsf{V}^\dagger(z))$ returns $\alpha, \beta, \gamma, \delta$ such that α is the first p decimal digits of the fractional part of $\mathsf{V}^\dagger(z)$, $\beta = 10^p$, $\gamma = \alpha + 1$ and $\delta = \beta$. Observe that α/β is the rational representation of the first p digits of the fractional part of $\mathsf{V}^\dagger(z)$. From this approximation, we identify a *sharpened* solution vector $\mathsf{V}^\star$ using the FINDFRACTION procedure from Algorithm 1. The procedure FIXPOINT then tests if $\mathsf{V}^\star$ is the correct solution by checking if it satisfies (3) or (4), whichever is appropriate. The uniqueness of the solutions to the these equation systems (which follows from those of (1) and (2)) ensures that the fixpoint check is only satisfied by the desired solution vector.

If the input vector $\mathsf{V}^\dagger$ is not precise enough, then SHAPREN returns "null".

The guarantees of Algorithm 2 are formalized as follows. Let V^b satisfying $\mathsf{V}(z) - \mathsf{V}^b(z) \leq 10^{-b}$ for all $z \in Z$ be an approximate solution vector of precision b[1]. Then, Lemma 1 establishes that starting from a close enough approximation, Algorithm 2 finds the actual solution vector.

***Lemma** 1:* Let $\mathcal{M}$ be an MDP, ψ be a PCTL path formula and V be the solution vector for $\mathcal{P}_{=?}[\psi]$. Let $b, P \in \mathbb{N}$ be such that $P \geq b$ and V^b is an approximate solution vector of precision b. If $\mathsf{V}(z) \in Q_{\lfloor\sqrt{10^b/2}\rfloor}$ for every $z \in Z$, then SHARPEN$(\mathcal{M}, P, \mathsf{V}^b) = \mathsf{V}$.

Proof (Sketch): Fix a state z and assume $\mathsf{V}(z) \in Q_M$ for $M{=}\lfloor\sqrt{10^b/2}\rfloor$. If $P{\geq}b$ then SHARPEN$(\mathcal{M}, P, \mathsf{V}^b)$ searches for $\mathsf{V}(z)$ in $I = [\alpha/\beta, \gamma/\delta]$ for $\alpha, \beta, \gamma, \delta = \text{BOUNDS}(b, \mathsf{V}^b(z))$. Now, $\mathsf{V}(z) \in I$ since $\mathsf{V}^b(z)$ satisfies $\mathsf{V}(z) - \mathsf{V}^b(z) \leq 10^{-b}$. Further, $|I| = 10^{-b} \leq \frac{1}{2M^2}$. Due to Kwek et. al. [26], we have that an interval of size $\frac{1}{2M^2}$ contains at most 1 element of Q_M. Clearly, FINDFRACTION$(\alpha, \beta, \gamma, \delta)$ returns $\mathsf{V}(z)$ which is the unique "minimal" element in $I \cap Q_M$. □

Building on the SHARPEN procedure, Algorithm 3 computes the values in $\mathcal{P}_{=?}[\phi_1 \mathcal{U} \phi_2]$ for state formulas of the form $\mathcal{P}_{\bowtie p}[\phi_1 \mathcal{U} \phi_2]$. It augments the value iteration phase from the standard PCTL model checking algorithm.

Algorithm 3 Rational Search

```
function RATIONALSEARCH($\mathcal{M}$, $\phi$, $\epsilon_0$):
    $\mathsf{V}^{\text{init}} \leftarrow$ INIT($\mathcal{M}, \phi$)
    $\epsilon \leftarrow \epsilon_0$
    while true do
        $\mathsf{V}^\dagger \leftarrow$ VALUEITERATION($\mathcal{M}$, $\phi$, $V^{\text{init}}$, $\epsilon$)
        $\mathsf{V}^\star \leftarrow$ SHARPEN($\mathcal{M}$, $\lceil\log(\frac{1}{\epsilon})\rceil$, $\mathsf{V}^\dagger$)
        if $\mathsf{V}^\star \neq$ null then
            return $\mathsf{V}^\star$
        end if
        $\mathsf{V}^{\text{init}} \leftarrow \mathsf{V}^\dagger$
        $\epsilon \leftarrow \epsilon/10$
    end while
end function
```

Algorithm 3 begins by running value iteration up to a given precision ϵ (where ϵ is used in the convergence criterion — absolute or relative — described in Section III) to determine an approximate solution vector $\mathsf{V}^\dagger$. Alternatively, value iteration could be replaced by an interval iteration algorithm and then ϵ would then represent a bound on the maximum error in the approximate solution vector. Once $\mathsf{V}^\dagger$ is computed, Algorithm 2 attempts to sharpen the approximate answer to an exact one. If it succeeds, the whole process terminates. Otherwise, $\mathsf{V}^\dagger$ is further refined by re-invoking value iteration with an increased ϵ precision and the sharpening process is repeated. When successive approximations in value iteration are computed using arbitrary precision arithmetic, Theorem 1 establishes the correctness of Algorithm 3.

[1]Notice that we require $\mathsf{V}^b(z)$ to be a lower bound for $\mathsf{V}(z)$, instead of $|\mathsf{V}(z) - \mathsf{V}^b(z)| \leq 10^{-b}$. Such an approximation can indeed be obtained from, say, value iteration.

***Theorem** 1:* Let $\mathcal{M}$ be a MDP, ψ be a PCTL path formula and V be the solution vector for $\mathcal{P}_{=?}[\psi]$ and $\epsilon_0 \in \mathbb{Q}^{>0}$. Then, RATIONALSEARCH$(\mathcal{M}, \mathcal{P}_{\bowtie p}[\psi], \epsilon_0)$ terminates and returns the exact solution vector V.

Proof (Sketch): It is easy to see that there is a $b > 0$ such that, for every state z, $\mathsf{V}(z) \in Q_N$ for $N = \lfloor\sqrt{10^b/2}\rfloor$. Now, since value iteration converges in the limit, we have that the first b digits of $\mathsf{V}^\dagger(z)$ match that of $\mathsf{V}(z)$ for each state $z \in Z$, eventually. Also, in every iteration of the loop in Algorithm 3, SHARPEN is invoked with an incremented value of P and eventually $P \geq b$. □

While both Lemma 1 and Theorem 1 are stated for formulae of the kind $\mathcal{P}_{=?}[\psi]$, they can be easily re-factored to reason about formulas of the form $\mathcal{E}_{=?}[\phi]$.

Example 5: Our experiments show that Algorithm 3 can make non-trivial improvements to solution quality. Consider the standard example of tossing N biased coins independently, where each coin yields heads with probability 1/3 and tails with probability 2/3. Analyzing the DTMC model to compute the probability of the event that 11 coins land heads, PRISM's floating-point model checker returned the decimal "0.000005645029269476758". Our tool was able to correctly determine the exact probability to be 1/177,147 by starting with the first 12 digits of this approximate answer. This is remarkable given that the period of this fraction (and hence its most succinct decimal representation) is almost 20,000 digits long. Moreover, the algorithm is able to simultaneously infer the reachability probabilities for *all* of the roughly 200,000 states of the model during a single fixpoint check. This illustrates another advantage of our technique; the algorithm is agnostic of the number of initial states in the system. The exact model checking engine of PRISM, on the other hand, currently only supports systems with a single initial state.

V. RESULTS

We have implemented Algorithm 3 in our tool RATIONALSEARCH as an extension of the PRISM model checker (version 4.3.1). RATIONALSEARCH is available for download at [5]. PRISM is comprised of four solution engines, three of which (MTBDD, HYBRID, SPARSE) are based on symbolic methods using compact data structures like MTBDDs. The fourth engine (EXPLICIT) manipulates sparse matrices, vectors and bit-sets directly. RATIONALSEARCH implements Algorithm 3 on top of all four engines. It intercepts PRISM's routine for solving constrained reachability probabilities and rewards, sharpening the probabilities every time it is invoked.

The EXPLICIT engine of PRISM is implemented in Java. To support this engine, our tool uses the libraries JScience [4] and Apfloat [1] to construct the transition matrix using rational entries, perform matrix-vector multiplications for the fixpoint

1	2	3	4	5	6	7	8	9	10	11
Model			RATIONALSEARCH						PRISM	STORM
			EXPLICIT		MTBDD		HYBRID			
Name	Parameter	States	Time (s)	Overhead (%)	Time (s)	Overhead (%)	Time (s)	Overhead (%)	Time (s)	Time (s)
Biased Coins	11	177,147	23.1	336	0.125	179	0.178	225	1449.7	3.2
Dice	6	4,826,809	OOM	N/A	1.8	2.1	6.5	12	TO	63
Din. Cryptographers	8	187,457	18.9	197	0.278	70	0.364	105	356.2	2.4
Din. Philosophers	3	956	0.41	165	1.9	4.8	0.133	98	3.128	0.65
ECS	14	4,815,782	OOM	N/A	2.4	23	11.6	79	TO	TO
Fair Exchange	400	321,600	14.6	423	2.0	44	2.2	51	TO	1.1
Firewire	11,000	428,364	122.2	225	15.1	0.2	19.5	21	232.3	29.5
Leader Election	4	12,302	1.8	226	5.0	30	20.4	25	80	0.042
Virus Infection	3	809	0.5	165	2.8	52	0.17	93	0.98	0.032

Fig. 2. **Experimental Results:**. Columns 1-3 describe the benchmark examples. Columns 4-11 report the performance metrics for the various exact solution engines. Running times are reported in seconds, averaging over 5 measurements. For the RATIONALSEARCH engines [EXPLICIT, MTBDD, HYBRID], we additionally report overhead percentages which are calculated by comparing the running times of PRISM's approximate engines with the corresponding extensions in RATIONALSEARCH. We use absolute convergence criterion ($\epsilon = 10^{-12}$) for the three engines in RATIONALSEARCH and the corresponding approx. engines in PRISM. A TO in columns 8, 10 and 11 represents a timeout (set to be 30 minutes). OOM indicates an out of memory exception.

check in Algorithm 3, and implement the Kwek-Mehlhorn algorithm (Algorithm 1). PRISM implements the remaining three engines using an extension of the CUDD library [2]. The off-the-shelf version of CUDD only supports floating point numbers at the terminals. RATIONALSEARCH enhances CUDD by allowing terminals to hold either floating points or arbitrary precision rational numbers provided by the GNU MP library [3]. Our extension allows the data type at the node to be easily interchanged and the full suite of MTBDD operations can be performed regardless of the data type. RATIONALSEARCH constructs the transition matrix as a rational MTBDD and uses Algorithm 2 to generate candidate solution vectors over rationals starting from approximate solution vectors represented as floating point MTBDDs given by PRISM's value iteration procedure.

Evaluation: We evaluated our tool against all of the examples involving quantitative reachability and rewards from the PRISM benchmark suite and case studies [6], [7] and compared the results with the exact parametric engines implemented in PRISM and STORM. Our tests were carried out on an Intel core i7 dual core processor @2.2GHz with 4Gb RAM running macOS 10.12.4. A summary of the performance on quantitative PCTL properties is given in Figure 2. The model checking times reported in the table include the time required to build the transition matrix using rational numbers. The reported times are an average of five runs for each engine/tool. We observed that, among the engines based on symbolic techniques, the SPARSE engine of PRISM was being consistently outperformed by the MTBDD and HYBRID engines. We, therefore, do not report its performance statistics.

Analysis of Results: The objective of our experimental evaluation is two-fold. First, we would like to compare our implementation against state-of-the-art tools for exact quantitative model checking (see Columns 4,6,8,10,11). The second objective is to analyze the performance overhead that our technique adds to approximate model checking techniques (see Columns 5,7,9). The overhead measures the additional time incurred by RATIONALSEARCH when compared to PRISM's engines that perform only value iteration using inexact floating point arithmetic.

Each implementation EXPLICIT, MTBDD and HYBRID of RATIONALSEARCH significantly outperforms PRISM's exact engine; in many cases, by several orders of magnitude. We also found at least one class of examples (Biased Coins) where PRISM's exact engine gave incorrect probabilities.

The comparison with STORM is more competitive. On most examples with a large number of states (ECS, Biased Coins, Dice), the running times achieve by RATIONALSEARCH are much lower than those from STORM. On smaller examples, the times were more comparable, with RATIONALSEARCH running slightly faster on the majority of examples.

On several examples with large state spaces, the EXPLICIT engine fails due to an out of memory exception. This can be attributed to the fact that the implementation stores two copies of the transition matrix in memory. On all the examples where EXPLICIT fails, the symbolic engines (MTBDD and HYBRID) find the solution quickly.

For the symbolic engines MTBDD and HYBRID, we found that RATIONALSEARCH can infer exact solutions from approximate ones, while typically not adding more than double overhead to approximate engines that are known to run extremely fast. The EXPLICIT engine incurs a much higher overhead. This difference is due to the fact that MTBDD's perform symmetry reductions, storing a single copy of each possible terminal value. This allows our implementation to run the Kwek-Mehlhorn algorithm a single time for all states sharing the same approximate value. This luxury is not afforded by the EXPLICIT engine, with carries out the Kwek-Mehlhorn procedure for every state in the model. For nested PCTL properties (such as ECS), the SHARPEN procedure must compute multiple fixpoints, adding to the overhead time. We would note, however, that for this example RATIONALSEARCH is the only tool that found a solution without hitting a timeout.

VI. CONCLUSION

Techniques for exact model checking allow one to avoid logical errors in system analysis that can arise due to approximation techniques. We presented an algorithm and tool, RATIONALSEARCH, that computes the exact probabilities described by PCTL formulas for DTMCs and MDPs. Our tool works by sharpening approximate results obtained through value iteration, allowing it to benefit from the performance enhancements gained through approximation techniques. Our experimental evaluation concurs with this hypothesis, and shows that our approach often performs significantly better than existing exact quantitative model checking tools while also scaling to large model sizes. For future work, we plan to combine our algorithm with more precise approximation techniques such as interval iteration. We believe there are also performance enhancements that can be achieved by a tighter integration with the Kwek-Mehlhorn algorithm, wherein computations from previous iterations can be reused.

REFERENCES

[1] Apfloat. http://www.apfloat.org/.

[2] CUDD. http://vlsi.colorado.edu/~fabio/CUDD/html/.

[3] GNU Multiple Precision Arithmetic Library. https://gmplib.org/.

[4] JScience. http://jscience.org/.

[5] RationalSearch. https://publish.illinois.edu/rationalmodelchecker/.

[6] (2017) PRISM Benchmark Suite. http://www.prismmodelchecker.org/benchmarks/. [Online; accessed 1-January-2017].

[7] (2017) PRISM Case Studies. http://www.prismmodelchecker.org/casestudies/. [Online; accessed 1-January-2017].

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] C. Baier, J. Klein, L. Leuschner, D. Parker, and S. Wunderlich, "Ensuring the reliability of your model checker: Interval iteration for markov decision processes," in *Computer Aided Verification*, 2017.

[10] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1999.

[11] D. Bhaduri, S. K. Shukla, P. S. Graham, and M. B. Gokhale, "Reliability analysis of large circuits using scalable techniques and tools," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, 2007.

[12] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, 1986.

[13] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, 1988.

[14] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Abraham, "Prophesy: A probabilistic parameter synthesis tool," in *International Conference on Computer Aided Verification, CAV*, 2015.

[15] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "A storm is coming: A modern probabilistic model checker," in *Computer Aided Verification: 29th International Conference, CAV 2017*.

[16] E. W. Dijkstra, "Self-stabilization in spite of distributed control," in *Selected writings on computing: a personal perspective*. Springer, 1982.

[17] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker, "A formal analysis of bluetooth device discovery," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 6, pp. 621–632, 2006.

[18] M. Fujita, P. C. McGeer, and J.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal methods in system design*, vol. 10, no. 2-3, pp. 149–169, 1997.

[19] S. Giro, "Efficient computation of exact solutions for quantitative model checking," in *Proc. 10th Workshop on Quantitative Aspects of Programming Languages (QAPL'12)*, 2012.

[20] S. Haddad and B. Monmege, "Reachability in mdps: Refining convergence of value iteration," in *International Workshop on Reachability Problems*. Springer, 2014, pp. 125–137.

[21] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang, "PARAM: A model checker for parametric Markov models," in *International Conference on Computer Aided Verification (CAV'10)*, 2010.

[22] J. Han, H. Chen, E. Boykin, and J. Fortes, "Reliability evaluation of logic circuits using probabilistic gate models," *Microelectronics Reliability*, 2011.

[23] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "Spudd: Stochastic planning using decision diagrams," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, 1999.

[24] B. Jeannet, P. D'Argenio, and K. Larsen, "Rapture: A tool for verifying Markov decision processes," in *Proc. Tools Day, affiliated to 13th Int. Conf. Concurrency Theory (CONCUR'02)*, 2002.

[25] J.-P. Katoen, M. Khattri, and I. Zapreevt, "A markov reward model checker," in *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*. IEEE, 2005.

[26] S. Kwek and K. Mehlhorn, "Optimal search for rationals," *Information Processing Letters*, vol. 86, no. 1, pp. 23–26, 2003.

[27] M. Kwiatkowska, G. Norman, and D. Parker, "Controller dependability analysis by probabilistic model checking," in *11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'04)*, 2004.

[28] M. Kwiatkowska, G. Norman, and J. Sproston, "Probabilistic model checking of the IEEE 802.11 wireless local area network protocol," in *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, 2002.

[29] ——, "Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol," *Formal Aspects of Computing*, vol. 14, no. 3, pp. 295–318, 2003.

[30] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 585–591.

[31] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[32] N. Mohyuddin, E. Pakbaznia, and M. Pedram, "Probabilistic error propagation in a logic circuit using the boolean difference calculus," in *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. Springer, 2011, pp. 359–381.

[33] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla, "Evaluating the reliability of nand multiplexing with prism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[34] Q. Qiu, Q. Qu, and M. Pedram, "Stochastic modeling of a power-managed system-construction and optimization," *IEEE Transactions on computer-aided design of integrated circuits and systems*, 2001.

[35] M. Rabin, "Randomized Byzantine generals," in *Proc. Symposium on Foundations of Computer Science*, 1983, pp. 403–409.

[36] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker, *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), ser. CRM Monograph Series. American Mathematical Society, 2004, vol. 23.

[37] J. J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker, *Mathematical techniques for analyzing concurrent and probabilistic systems*. American Mathematical Soc., 2004.

[38] R. St-Aubin, J. Hoey, and C. Boutilier, "Apricodd: Approximate policy construction using decision diagrams," in *Advances in Neural Information Processing Systems*, 2001, pp. 1089–1095.

[39] R. Wimmer, A. Kortus, M. Herbstritt, and B. Becker, "Probabilistic model checking and reliability of results," in *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*. IEEE, 2008, pp. 1–6.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Sampling Invariants from Frequency Distributions

Grigory Fedyukovich Samuel J. Kaufman Rastislav Bodík
University of Washington Paul G. Allen School of Computer Science & Engineering
{grigory, kaufmans, bodik}@cs.washington.edu

***Abstract*—We present a new SMT-based, probabilistic, syntax-guided method to discover numerical inductive invariants. The core idea is to initialize frequency distributions from the program's source code, then repeatedly sample lemmas from those distributions, and terminate when the conjunction of learned lemmas becomes a safe invariant. The sampling process gets further optimized by priority distributions fine-tuned after each positive and negative sample. The stochastic nature of this approach admits simple, asynchronous parallelization. We implemented and evaluated this approach in a tool called FreqHorn which shows competitive performance on well-known linear and some non-linear programs.**

I. Introduction

Automated formal verification of programs handling unbounded loops is reduced to finding safe inductive invariants that over-approximate the sets of reachable states, but precise enough to prove unreachability of the error state. Successful solutions to this problem include Counterexample-Guided Abstraction Refinement [1] and Property Directed Reachability (PDR) [2], [3], [4], but they are not guaranteed to deliver appropriate invariants.

We aim at learning inductive invariants in an "*enumerate-and-check*" manner [5], [6]. While this approach in general meets a lot of skepticism, there are particular synthesis tasks which can be efficiently solved using tailored heuristics. Our intuition behind applying this synthesis paradigm to discover invariants is that an invariant can often be caught on the surface, i.e., it to some degree imitates the syntactical constructions which appear in the source code.

Source code can give hints for guessing a candidate to be checked for invariance. Any information of occurrences of variables, constants, arithmetic and comparison operators, and their applications can potentially guide the search of invariants. The research question we address in this paper is whether a probability distribution constructed by processing the source code could help sampling successful candidates. This reduces the number of invariance checks and decreases the total verification time.

We contribute a framework for learning invariants using sampling from probability distributions obtained after collecting multiple facts about the given source code. Before sampling, we fix a number of features which could belong to each invariant. We then split the code in clauses, normalize each clause, and check how many of the predetermined features belong to it. The statistics collected from all normalized clauses define a number of *frequency distributions*.

The main workhorse in our framework is a repetitive process that samples different pieces of a candidate invariant from the frequency distributions and then assembles them together. Each assembled candidate has a certain feature with a probability specified in the corresponding distribution. So it is likely that our sampled candidates are in some sense *representative*. Finally, the candidate is checked using an off-the-shelf SMT solver. If it is proven to be an actual invariant, our algorithm stores it and proceeds to discovering other invariants. The search continues until all invariants that are needed to verify safety are discovered, or until the search space is exhausted.

Our second contribution is an algorithm that combines sampling from frequency distributions and sampling from *priority distributions* created on the fly and adjusted after each positive and negative sample. That is, once a candidate is checked, some "*likely unrelated*" candidates get higher priorities for being sampled in the coming iterations. We show how this strategy can be made aggressive, i.e., the completeness of the search space exploration is traded off for widening of candidate diversity.

The approach has been implemented in a tool called FreqHorn which naturally admits parallelization. FreqHorn uses an SMT solver to check each sample for invariance. The learning strategy with priority distributions is shown to be extremely effective in practice, despite for some pathological situations it affects the convergence. As expected, our tool is competitive to the closely related machine-learning-based tools for invariants learning, and in some cases it is more effective than PDR-based tools.

The rest of the paper is structured as follows. Sect. II briefly discusses the fundamentals of our verification problem. Then, in Sect. III, we introduce the framework to learn numerical invariants, describe its optimizations and drawbacks, and in Sect. IV, we describe our parallel implementation, evaluation, and comparison with other tools. Sect. V has an overview of the related work, and Sect. VI concludes the paper.

II. Background

A. Programs and their inductive invariants

We use vector notation to denote sequences (e.g., of variables or constants). We assume the first-order formulas $\varphi(\vec{x}) \in Expr$ in the paper. For simplicity, we write φ when the arguments are clear from the context. For an

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

implication between $\varphi, \psi \in Expr$, we write $\varphi \implies \psi$; φ is said to be stronger than ψ, and ψ – weaker than φ. If φ is satisfiable, we write $\varphi \not\Longrightarrow \bot$ (and $\varphi \implies \bot$ otherwise). Formula ψ is called a tautology if $\neg\psi \implies \bot$.

Definition 1. *A* program *P is a tuple* $\langle Var, Init, Tr\rangle$, *where* $Var \overset{\text{def}}{=} V \cup V'$ *is a set of* input *and* output variables*;* $Init \in Expr$ *encodes the* initial states *over* V*; and* $Tr \in Expr$ *encodes the* transition relation *over* Var.

A state is a valuation to all variables in V. For every input variable $x \in V$, there is a corresponding output variable $x' \in V'$ (i.e., the value of x in the next state).

Definition 2. *Let* $P = \langle V \cup V', Init, Tr\rangle$*; a formula Inv over V is an* inductive invariant *if the following conditions (respectively called initiation and consecution) hold:*

$$Init(V) \implies Inv(V) \tag{1}$$

$$Inv(V) \wedge Tr(V, V') \implies Inv(V') \tag{2}$$

Example 1. *Consider Fig. 1 showing program* `Bradley` *named after its appearance in [2]. It has counter* `x` *that gets repeatedly added to variable* `y`*. Examples of inductive invariants include* $x \geqslant 0$, $x \geqslant 0 \wedge y \geqslant 0$, $x \geqslant 0 \wedge x + y \geqslant 0$, *and* $x \geqslant 0 \wedge y - x \geqslant 0$. □

Lemma 1. *Given program P, if* Inv_1 *and* Inv_2 *are inductive invariants for P, then* $Inv_1 \wedge Inv_2$ *is also an inductive invariant.*

Lemma 1 does not work in the reverse direction: if a conjunction of formulas is an inductive invariant then each conjunct in isolation could not be an inductive invariant. For example, $x \geqslant 0 \wedge y \geqslant 0$ is an inductive invariant for `Bradley`, but $y \geqslant 0$ is not an inductive invariant.

Lemma 2. *Given program* $P = \langle Var, Init, Tr\rangle$*, let* Inv_1 *be an inductive invariant for P, program* P_1 *be* $\langle Var, Init, Tr \wedge Inv_1\rangle$*, and* Inv_2 *be an inductive invariant for* P_1*; then* $Inv_1 \wedge Inv_2$ *is an inductive invariant for P.*

Lemma 2 enables incremental invariant discovery. For example, for `Bradley`, one could find an inductive invariant $x \geqslant 0$ first and then conjoin it to the transition relation. It remains to find an inductive invariant satisfying the strengthened transition relation, e.g., $y \geqslant 0$. Finally, conjunction $x \geqslant 0 \wedge y \geqslant 0$ is an invariant for `Bradley`.

Definition 3. *A* verification task *is a pair* $\langle P, Bad\rangle$*, where* $P = \langle V \cup V', Init, Tr\rangle$ *is a program, and Bad is a formula encoding the* error states *over V.*

A verification task has a solution if the set of error states is not reachable. We call the program *safe* in this case. Safety is decided by discovering a safe inductive invariant, a formula that covers the initial state, is closed under the transition relation, and does not cover the error state.

```
int x = y = 0;
while (*) {
  x = x + 1;
  y = y + x;
}
assert(y >= 0);
```

Figure 1: Possibly infinite loop over algebraic integers (cf. [2]).

Definition 4. *Let* $P = \langle V \cup V', Init, Tr\rangle$ *and* $\langle P, Bad\rangle$ *be a verification task; an inductive invariant Inv for P is called* safe *if:*

$$Inv(V) \wedge Bad(V) \implies \bot \tag{3}$$

Examples of safe inductive invariants for `Bradley` include $x \geqslant 0 \wedge y \geqslant 0$ and $x \geqslant 0 \wedge y - x \geqslant 0$.

B. Sampling from probability distributions

Definition 5. *A probability distribution on a set A is a function* $p : A \to \mathbb{R}$*, such that* $\forall a \in A \,.\, 0 \leqslant p(a) \leqslant 1$ *and* $\sum\limits_{a \in A} p(a) = 1$.

In this paper, we consider a process which, given a set of formulas and a probability distribution, chooses at each iteration an element from the set with a probability determined by the distribution.

Example 2. *Given four formulas over x and y, a probability distribution* $p^{\texttt{Bradley}}$ *could be defined as follows:*

$$\begin{aligned} x \geqslant 0 &\mapsto {}^{4}\!/_{10} \\ y \geqslant 0 &\mapsto {}^{3}\!/_{10} \\ x + y \geqslant 0 &\mapsto {}^{2}\!/_{10} \\ y - x \geqslant 0 &\mapsto {}^{1}\!/_{10} \end{aligned}$$

In order to prove program `Bradley` *safe, it could be sufficient to sample from distribution* $p^{\texttt{Bradley}}$ *two times and to check invariance incrementally for each sample. Assuming that formula* $x \geqslant 0$ *was sampled at the first round (with probability* 0.4*), it is enough to sample either* $y \geqslant 0$ *or* $y - x \geqslant 0$ *(with probability* $0.3 + 0.1$*). Thus, the probability of discovering a safe inductive invariant in two steps equals* $0.4 \cdot (0.3 + 0.1) = 0.16$. □

Consider now a scenario that rejects all samples that were already checked for invariance (e.g., by nudging the distribution accordingly). It is easy to see that the probability of discovering a safe inductive invariant (in two steps) increases.

In the next section, we discuss a practical way of creating both, the sets of samples, and their probability distributions.

III. Learning Numerical Invariants

A. Grammar and probabilistic production rules

Fig. 2 shows a grammar for generating the candidate inductive invariants (also referred to as the *sampling*

$$\begin{aligned}
c &::= c_1 \mid c_2 \mid \ldots \mid c_\ell \\
k &::= k_1 \mid k_2 \mid \ldots \mid k_m \\
x &::= x_1 \mid x_2 \mid \ldots \mid x_n \\
\mathit{lincom} &::= k \cdot x + k \cdot x + \ldots + k \cdot x \\
\mathit{ineq} &::= \mathit{lincom} > c \mid \mathit{lincom} \geqslant c \\
\mathit{cand} &::= \mathit{ineq} \vee \mathit{ineq} \vee \ldots \vee \mathit{ineq}
\end{aligned}$$

Figure 2: Sampling grammar.

grammar). The formulas are generated using *probabilistic production rules*. In contrast to standard non-deterministic production rules, each choice is in line with a particular *probability distribution*, making the samples more predictable.

The sampling works in a top-to-bottom manner. Given a probability distribution $p_\vee$ for the arities of the *or*-operator, we sample a value n from $p_\vee$ and reserve n slots for operands of $\vee$ (linear inequalities). Then, for each $1 \leqslant i \leqslant n$, we sample a non-empty subset $\vec{x} \subseteq V$ of variables from a given probability distribution p_+. Then, given a sequence of probability distributions $\{p_{k_j}\}$ for scalar coefficients for each variable $x_j \in V$, we sample a value $k_j \in K \subseteq \mathbb{R}\backslash\{0\}$. Summing products of each k_j with x_j, we get a linear combination (denoted $\{x_j, k_j\}$). Lastly, for each inequality, we sample a binary comparison operator (either $>$ or $\geqslant$) and a constant $c \in C \subseteq \mathbb{R}$ from given probability distributions p_{op} and p_c, respectively.

Each conjunction-free sample is individually checked for invariance. Following Lemma 2, each successful sample is conjoined to the transition relation, and thus it will be used while checking invariance of samples in the future. This lets us discover conjunctive invariants without having the conjunction operator in the sampling grammar.

Note that the sampling grammar does not contain the comparison operators other than $>$ and $\geqslant$. The expressiveness of formulas is achieved by providing large enough sets of K and C for numerical coefficients and constants: if an element is in a set, its additive inverse is also in the set. Thus, instead of generating formula $k_1 \cdot x_1 + \ldots + k_n \cdot x_n < c$, we generate an equivalent formula $(-k_1) \cdot x_1 + \ldots + (-k_n) \cdot x_n > -c$ (see more details in Sect. III-B).

In practice, there could be dependencies among ingredients of a sample. For instance, the number of disjuncts in a sample could affect the variables, coefficients, constants, and the comparison operators appearing in each disjunct. Because our sampling is hierarchical, the dependencies propagate top-to-bottom. To formally address this, we allow the production rules operate over conditional probability distributions.

B. Value ranges and frequency distributions

The sampling grammar imposes the fixed structure on the candidate invariants. The key to success while assembling each candidate is to fix the sets of numerical constants K and C. Our contribution is the technique that 1) automatically constructs these sets and 2) supplies each production rule in the grammar with the probability distribution. We achieve both targets via exploring the *Init*, *Tr*, and *Bad* formulas, included in the verification task, and calculating the frequencies of appearances of particular constants.

The algorithm of frequency calculations is informally described below. It starts with converting the *Init*, *Tr*, and *Bad* formulas to the Conjunctive Normal Form, splitting them into *clauses* (i.e., disjunctions of linear inequalities) and inserting the clauses to two sets, denoted respectively A_V and $A_{V \cup V'}$. Set A_V contains elements which have appearances of input variables V only (i.e., all elements obtained from *Init* or *Bad* and possibly some elements from *Tr*). Set $A_{V \cup V'}$ contains elements which have appearances of input and output variables V and V' at the same time (e.g., $x' = x + 2$ which can be originated from *Tr*).

Then, for each clause $a \in A_V$, an application of $\neq$, $=$, $<$, or $\leqslant$ is replaced by application(s) of $>$ or $\geqslant$:

$$\frac{A < B}{-A > -B} \qquad \frac{A \leqslant B}{-A \geqslant -B}$$

$$\frac{A = B}{A \geqslant B \wedge -A \geqslant -B} \qquad \frac{A \neq B}{A > B \vee -A > -B}$$

Note that in case $a = (A = B)$ the resulting formula is conjunction $a_+ \wedge a_-$, and thus a is replaced by a_+ and a_-, i.e., $A_V \leftarrow A_V \backslash \{a\} \cup \{a_+, a_-\}$. After this rewriting, we assume that each clause $a \in A_V$ matches the sampling grammar. Thus, it is straightforward to determine the arity of the $\vee$-operator (and include them to set N), numerical coefficients (and include them to set K), and constants (and include them to set C).

Additionally, we collect constants which appear in clauses $A_{V \cup V'}$ and include them to K. The last trick is to include products $c \cdot k$ to K, for any $c \in C$ and $k \in K$; and products $c_1 \cdot c_2$ to C, for any $c_1, c_2 \in C$.

Definition 6. *The set of formulas specified by the grammar in Fig. 2, in which the sets of arities of the $\vee$-operator N, numerical coefficients K, and constants C are obtained from A_V and $A_{V \cup V'}$ is called an* appearance-guided search space*.*

Finally, we are ready to calculate various statistics, in particular:

- how often $a \in A_V$ has arity $i \in N$,
- how often each combination of variables $\vec{x} \subseteq V$ appears among the inequalities,
- how often a variable $x \in V$ has a coefficient $k \in K$,
- how often a constant $c \in C$ appears among the inequalities,

Algorithm 1: Sampling inductive invariants.

```
Input: $P = \langle V \cup V', \mathit{Init}, \mathit{Tr} \rangle$: program;
       $\langle P, \mathit{Bad} \rangle$: verification task
Output: $\mathit{learnedLemmas}$: set of $\mathit{Expr}$
$A_V, A_{V \cup V'} \leftarrow$ NORMALIZE$(P)$;
$C, K, N \leftarrow$ GETRANGES$(A_V, A_{V \cup V'})$;
$\{p_*\} \leftarrow$ GETFREQUENCIES$(A_V, A_{V \cup V'})$;
$\mathit{learnedLemmas} \leftarrow \varnothing$;
while $(\mathit{Bad}(V) \wedge \bigwedge_{\ell \in \mathit{learnedLemmas}} \ell(V) \not\Longrightarrow \bot)$ do
    $\mathit{Cand} \leftarrow \bot$;
    $n \leftarrow$ SAMPLE$(p_\vee)$;
    for $(i \in [1, n])$ do
        $\vec{x}_i \leftarrow$ SAMPLE$(p_+ \mid n)$;
        $\vec{k}_i \leftarrow$ SAMPLE$(p_k \mid n, \vec{x})$;
    for $(i \in [1, n])$ do
        $c_i \leftarrow$ SAMPLE$(p_c \mid i, n, \{\vec{x}_i, \vec{k}_i\})$;
        $op_i \leftarrow$ SAMPLE$(p_{op} \mid i, n, c_i, \{\vec{x}_i, \vec{k}_i\})$;
        $\mathit{Cand} \leftarrow \mathit{Cand} \vee$ ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c_i, op_i)$;
    if $(\neg \mathit{Cand}(V) \Longrightarrow \bot)$ then continue;
    if $(\mathit{Init}(V) \wedge \neg \mathit{Cand}(V) \not\Longrightarrow \bot)$ then continue;
    if $(\mathit{Cand}(V) \wedge \mathit{Tr}(V, V') \wedge \neg \mathit{Cand}(V') \wedge$
            $\bigwedge_{\ell \in \mathit{learnedLemmas}} \ell(V) \not\Longrightarrow \bot)$ then continue;
    $\mathit{learnedLemmas} \leftarrow \mathit{learnedLemmas} \cup \mathit{Cand}$;
```

- how often an operator $op \in \{>, \geqslant\}$ appears among the inequalities.

These statistics are used to construct frequency distributions, respectively: $p_\vee$, p_+, p_{k_0}, $\ldots$, p_{k_n}, p_c, and p_{op}, and to guide the sampling process. To enlarge the search space, an artificial ϵ-frequency representing appearances that never happened in the actual code (e.g., by connecting a variable and a constant that never appear together) could be introduced. The value of ϵ-frequency could be chosen heuristically based on values of other frequencies, as long as it stays sufficiently small and positive.

C. Core algorithm

Alg. 1 shows the routines of the *sampler*, the invariance *checker*, and their interaction. As a preprocessing step (lines 1-3), the algorithm normalizes formulas, collects sets N, K, and C, and calculates frequencies as described in Sect. III-B.

The sampler generates a formula (line 14) from the appearance-guided search space using the frequency distributions. In a naive scenario, the sampler deals with distributions $p_\vee$, p_+, p_{k_0},$\ldots$, p_{k_n}, p_c, and p_{op} directly. However, in order to make the sampling more predictable, the algorithm creates conditional distributions and samples from them (lines 9-13), in particular:

- how often each combination of variables $\vec{x} \subseteq V$ appears among inequalities which are contained in a clause of the given arity n,
- how often a constant $c \in C$ appears in *ineq*, given *ineq* is the i-th inequality among n inequalities $\{\mathit{ineq}_j\}$ and each ineq_j is over particular $\{\vec{x}_j, \vec{k}_j\}$,
- etc ...

As mentioned in Sect. III-A, the choice of conditional distributions is justified by the order of sampling of each ingredient of a candidate *Cand*. Thus, any change in this order may affect the conditional distributions used for sampling. At the same time, any conditions for the distributions could be made optional (depending on the problem in hand). Evidently, this does not affect soundness of the entire approach, but affects the speed of convergence.

If the sampled *Cand* is a tautology (performed by an SMT solver, line 15), then it is known to be an inductive invariant, but it does not make any progress towards completing the verification; thus *Cand* should be withdrawn. It would also make sense to withdraw all unsatisfiable candidates, but by construction, the number of products $k_i \cdot x_i$ in each disjunct is always positive, which makes *Cand* always satisfiable.

The checker decides a number of local SMT queries per each *Cand*. A negative result – i.e., whenever *Cand* is a tautology or the initiation or the consecution check fails (line 16 or 17, respectively) – is called an *invariance failure.* Otherwise, the result is positive and is called a *learned lemma.* Each new learned lemma is book-kept (line 18) for the safety check (line 5), and also for the consecution checks (line 17) of candidates coming in the next iterations (recall Lemma 2). The sampler and the checker alternate until a safe inductive invariant is found.

Theorem 1. *If a safe inductive invariant can be expressed by a conjunction of formulas within the appearance-guided search space; then the probability that Alg. 1 eventually discovers it tends to 1.*

D. Prioritizing the search space

The success of techniques based on syntax-guided synthesis (SyGuS) depends on how effectively the search space is pruned after a positive or negative sample is examined. To avoid repeatedly appearing learned lemmas and invariance failures in the future, we introduce a pool of all samples and refer to it whenever a new candidate is sampled. Furthermore, we employ a lightweight analysis to identify and block some closely related candidates from being sampled and prioritize some likely unrelated candidates to being sampled and checked for invariance.

Definition 7. *The* $\mathit{priorMap}_{\langle \vec{x}, \vec{k} \rangle}$ *function maps linear combinations* $\{\vec{x}_i, \vec{k}_i\}$ *to joint probability distributions for* op_i *and* c_i *(called* priority distributions*).*

One natural goal of $\mathit{priorMap}_{\langle \vec{x}, \vec{k} \rangle}$ is to set to zero probabilities of sampling the candidates which are 1) already checked, 2) stronger than failures, and 3) weaker than learned lemmas. Consequently, the probabilities of other candidates should be increased, and we achieve it by exploiting the ordering of constants and comparison operators in the sampling grammar. Alg. 2 shows a version of Alg. 1, augmented with prioritizing capabilities (the pseudocode inherited from Alg. 1 is decoloured).

Algorithm 2: Sampling inductive invariants from priority distributions.

```
    Input: $P = \langle V \cup V', Init, Tr\rangle$: program;
           $\langle P, Bad\rangle$: verification task
    Output: $learnedLemmas$: set of $Expr$
 1  $A_V, A_{V \cup V'} \leftarrow$ NORMALIZE($P$);
 2  $C, K, N \leftarrow$ GETRANGES($A_V, A_{V \cup V'}$);
 3  $\{p_*\} \leftarrow$ GETFREQUENCIES($A_V, A_{V \cup V'}$);
 4  $learnedLemmas \leftarrow \varnothing$;
 5  while ($Bad(V) \wedge \bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot$) do
 6      $Cand \leftarrow \bot$;
 7      $n \leftarrow$ SAMPLE($p_\vee$);
 8      for ($i \in [1, n]$) do
 9          $\vec{x}_i \leftarrow$ SAMPLE($p_+ \mid n$);
10          $\vec{k}_i \leftarrow$ SAMPLE($p_k \mid n, \vec{x}$);
11      if ($priorMap_{\langle \vec{x}, \vec{k}\rangle} = \varnothing$) then
12          $priorMap_{\langle \vec{x}, \vec{k}\rangle} \leftarrow uniform_{\langle \vec{x}, \vec{k}\rangle}$;
13          for ($i \in [1, n]$) do
14              $c_i \leftarrow$ SAMPLE($p_c \mid i, n, \{\vec{x}_i, \vec{k}_i\}$);
15              $op_i \leftarrow$ SAMPLE($p_{op} \mid i, n, c_i, \{\vec{x}_i, \vec{k}_i\}$);
16              $Cand \leftarrow Cand \vee$ ASSEMBLEINEQ($\vec{x}_i, \vec{k}_i, c_i, op_i$);
17      else if ($priorMap_{\langle \vec{x}, \vec{k}\rangle} =$ undefined) then continue;
18      else
19          for ($i \in [1, n]$) do
20              $c_i, op_i \leftarrow$ SAMPLE($priorMap_{\langle \vec{x}, \vec{k}\rangle}(\{\vec{x}_i, \vec{k}_i\})$);
21              $Cand \leftarrow Cand \vee$ ASSEMBLEINEQ($\vec{x}_i, \vec{k}_i, c_i, op_i$);
22      if ($\neg Cand(V) \Longrightarrow \bot$) then
23          UPDATE($priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE${}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$);
24          continue;
25      if ($Init(V) \wedge \neg Cand(V) \not\Longrightarrow \bot$) then
26          UPDATE($priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE${}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle)$);
27          continue;
28      if ($Cand(V) \wedge Tr(V, V') \wedge \neg Cand(V') \wedge \bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot$) then
29          UPDATE($priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE${}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle)$);
30          continue;
31      $learnedLemmas \leftarrow learnedLemmas \cup Cand$;
32      UPDATE($priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE${}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$);
```

For each sequence of linear combinations $\langle \vec{x}, \vec{k}\rangle \overset{\text{def}}{=} \{\{\vec{x}_i, \vec{k}_i\}\}$, that has been sampled for the first time: 1) $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ gets assigned a sequence of *uniform* joint distributions for each $\vec{x}_i$ and $\vec{k}_i$ (line 12), and 2) the remaining ingredients for assembling *Cand* are sampled from the frequency distributions (lines 13-16, as in Alg. 1). Before proceeding to the next iteration, each positive or negative result nudges distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ (line 23, 26 29, or 32).

For each $\langle \vec{x}, \vec{k}\rangle$, that has been sampled not for the first time, there exists a sequence of *non-uniform* distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$. Further, the sampler produces candidates from that distribution instead of the frequency distributions (lines 19-21), and again, distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ get nudged after each positive and negative result.

Once for some $\langle \vec{x}, \vec{k}\rangle$ the search is exhausted then the distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ are said to be undefined (line 17), and the sampler proceeds to exploring another sequence of linear combinations. In the rest of the subsection, we clarify how distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ get nudged and how it might make them undefined.

Definition 8. *Given $\langle \vec{x}, \vec{k}\rangle$ and $ineq_1 \vee \ldots \vee ineq_n$, such that each $ineq_i$ is over a linear combination $\{\vec{x}_i, \vec{k}_i\}$, operator op_i, and constant c_i, we write:*

- PRIORITIZE${}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$ *– to produce a joint probability distribution for each $1 \leqslant i \leqslant n$ to sample op'_i and c'_i and to produce $ineq'_i =$* ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c'_i, op'_i)$*, such that if $ineq_i \Longrightarrow ineq'_i$ then the probability of sampling op'_i and c'_i is set to zero, and otherwise it* increases *with the growth of c'_i;*
- PRIORITIZE${}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle)$ *– to produce a joint probability distribution for each $1 \leqslant i \leqslant n$ to sample op'_i and c'_i and to produce $ineq'_i =$* ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c'_i, op'_i)$*, such that if $ineq'_i \Longrightarrow ineq_i$ then the probability of sampling op'_i and c'_i is set to zero, and otherwise it* decreases *with the growth of c'_i (symmetrically to* PRIORITIZE${}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$*).*

Example 3. *Let $C = \{-5, 0, 5\}$, and $ineq_1 \vee ineq_2 = x > -5 \vee x + y \geqslant 5$ be a learned lemma for some program P. Then, any disjunction $ineq'_1 \vee ineq'_2$ where $ineq'_1 \in \{x \geqslant -5, x > -5\}$, and $ineq'_2 \in \{x + y \geqslant -5, x + y > -5, x + y \geqslant 0, x + y > 0, x + y \geqslant 5\}$ is weaker or equal to $ineq_1 \vee ineq_2$, and checking its invariance would not affect our verification process. The* PRIORITIZE${}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$ *function outputs two probability distributions p_x and p_{x+y} to sample a new comparison operator and a new constant for x and $x + y$, respectively:*

$x > 5 \mapsto {}^4\!/_{10}$	$x + y > 5 \mapsto 1$
$x \geqslant 5 \mapsto {}^3\!/_{10}$	$x + y \geqslant 5 \mapsto 0$
$x > 0 \mapsto {}^2\!/_{10}$	$x + y > 0 \mapsto 0$
$x \geqslant 0 \mapsto {}^1\!/_{10}$	$x + y \geqslant 0 \mapsto 0$
$x > -5 \mapsto 0$	$x + y > -5 \mapsto 0$
$x \geqslant -5 \mapsto 0$	$x + y \geqslant -5 \mapsto 0$

□

Distributions $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ block the sampler from producing more formulas than needed, i.e., not only the ones which are strictly weaker (or stronger) that the learned lemmas (or invariance failures). Thus there is a risk to miss some invariants and to meet divergence. However, the intention of our synthesis procedure is to encourage exploring a wide range of unrelated samples. While having the risk to miss a "*next door*" invariant, we *aggressively* increase probabilities of "*far away*" candidates to being sampled. Our experiments confirm that such aggressive-pruning strategy in many cases accelerates the invariant discovery (see Sect. IV-B).

Note that in Alg. 2, we prioritize the search space after a tautology in a similar fashion to how we proceed after a learned lemma (but as in Alg. 1, we do not add it to *learnedLemmas*). This trick helps blocking some more tautologies from being sampled.

Example 4. *Let $C = \{-5, 0, 5\}$, and $ineq_3 \vee ineq_4 = x \geqslant 5 \vee x + y > 5$ be an invariance failure for P. The* $\text{PRIORITIZE}^{\overset{\vec{op},\vec{c}}{\downarrow}}(\langle\vec{x},\vec{k}\rangle)$ *function outputs two probability distributions p'_x and p'_{x+y}:*

$x > 5 \mapsto 0$	$x + y > 5 \mapsto 0$
$x \geqslant 5 \mapsto 0$	$x + y \geqslant 5 \mapsto {}^1/_{15}$
$x > 0 \mapsto {}^1/_{10}$	$x + y > 0 \mapsto {}^2/_{15}$
$x \geqslant 0 \mapsto {}^2/_{10}$	$x + y \geqslant 0 \mapsto {}^3/_{15}$
$x > -5 \mapsto {}^3/_{10}$	$x + y > -5 \mapsto {}^4/_{15}$
$x \geqslant -5 \mapsto {}^4/_{10}$	$x + y \geqslant -5 \mapsto {}^5/_{15}$

$\square$

Definition 9. *Given two probability distributions p, p' on set A, a probability distribution $p_m(p, p')$ on A is defined as follows. Let $s \overset{\text{def}}{=} \sum_{a \in A} min(p(a), p'(a))$, then $\forall a \in A \,.\, p_m(p, p')(a) \overset{\text{def}}{=} \frac{min(p(a),p'(a))}{s}$.*

We write $\text{UPDATE}(priorMap_{\langle\vec{x},\vec{k}\rangle}, \{p_i\})$ to produce a distribution $p_m(priorMap_{\langle\vec{x},\vec{k}\rangle}(\{\vec{x}_i, \vec{k}_i\}), p_i)$ for each i and to store all of them in $priorMap_{\langle\vec{x},\vec{k}\rangle}$.

Example 5. *Given p_x and p_{x+y}, p'_x and p'_{x+y} obtained in Examples 3-4 respectively, two distributions $p_m(p_x, p'_x)$ and $p_m(p_{x+y}, p'_{x+y})$ are as follows. Note that the latter is* undefined *since all formulas are mapped to 0, and the condition of Def. 5 is violated.*

$x > 5 \mapsto 0$	$x + y > 5 \mapsto$ undefined
$x \geqslant 5 \mapsto 0$	$x + y \geqslant 5 \mapsto$ undefined
$x > 0 \mapsto {}^1/_2$	$x + y > 0 \mapsto$ undefined
$x \geqslant 0 \mapsto {}^1/_2$	$x + y \geqslant 0 \mapsto$ undefined
$x > -5 \mapsto 0$	$x + y > -5 \mapsto$ undefined
$x \geqslant -5 \mapsto 0$	$x + y \geqslant -5 \mapsto$ undefined

This way, since one of the two distributions at $priorMap_{\langle\vec{x},\vec{k}\rangle}$ is undefined, *the entire $\langle\vec{x},\vec{k}\rangle$ is withdrawn by the algorithm and is not going to be considered in the next iterations.* $\square$

E. Invariants over non-linear arithmetic

Our approach has a limited support for learning numerical invariants for programs with non-linear arithmetic computations. It naturally extends the idea of collecting and exploiting features of the program source code. In addition to populating sets N, K, and C from *Init*, *Tr*, and *Bad* (described in Sect. III-B), our algorithm populates set W by applications of either 1) the modulo operator, 2) the division operator, or 3) the multiplication operator, the list of arguments of which contains more than one variable.

The grammar in Fig. 3 enhances the sampling grammar with elements of W, which are treated as fresh variables. That is, the sampler may end up with candidates having elements of W, possibly multiplied by numeric constants and appeared in linear combinations.

$$\begin{aligned}
&\ldots\\
x ::= & x_1 \mid \ldots \mid x_n \mid x_i \texttt{ div } k_i \mid x_i \texttt{ mod } k_j\\
& \mid x_i \cdot x_j \mid x_i \texttt{ div } x_j \mid x_i \texttt{ mod } x_j\\
&\ldots
\end{aligned}$$

Figure 3: Non-linear sampling grammar (see Fig. 2 for the omissions).

```
int x = y = 0;
int z = *;
while (*) {
  x = x + z;
  y = y + 1;
}
assert(x == y * z);
```

Figure 4: Program with non-linear computations.

Example 6. *For program shown in Fig. 4, $Bad = (x = y \cdot z)$, thus $W = \{y \cdot z\}$. This lets our sampler generate candidates such as $-1 \cdot x + y \cdot z \geqslant 0$ and $x + -1 \cdot y \cdot z \geqslant 0$ which would pass the invariance check by a theory solver over non-linear arithmetic.* $\square$

F. Further extensions

Extending frequencies with redundant clauses. Before populating the set of clauses A_V from *Init*, *Tr*, and *Bad*, these formulas could be enhanced by conjoining with some redundant clauses. A straightforward approach would consider pairs of conjuncts of *Init* (respectively, *Tr*, and *Bad*), infer a new clause and conjoin it back to *Init*. For instance, if $Init = (x = 0) \wedge (y = 0)$ then it implies $x = y$; and thus we can rewrite *Init* to be $(x = 0) \wedge (y = 0) \wedge (x = y)$. After the frequency distributions are created and used for sampling, the probability of getting formulas $-1 \cdot x + y \geqslant 0$ and $x + -1 \cdot y \geqslant 0$ increases. In practice, there could be many possible ways of inferring redundant clauses, and we leave the investigation of which way is the best for the future work.

More aggressive pruning. Besides of aggressive priority distributions, other tricks could be applied to shrink the search space. From a set of learned lemmas $\{ineq_1 \vee \ldots \vee ineq_n, \neg ineq_1, \ldots, \neg ineq_{n-1}\}$, it follows that formula $\neg ineq_n$ is not an inductive invariant. Thus, it could be withdrawn by the algorithm and not considered for sampling. Furthermore, the sets C and K might allow equivalent formulas (e.g., $x > 1$ and $2 \cdot x > 2$). The priority distributions could be nudged to block those as well.

Compensating aggressive pruning. One could introduce a "*reincarnating*" function, which turns distributions at $priorMap_{\langle\vec{x},\vec{k}\rangle}$ back to $uniform_{\langle\vec{x},\vec{k}\rangle}$ once they become undefined, or once each next lemma is learned.

IV. Implementation and Evaluation

We implemented the proposed approach in a tool FreqHorn[1]. It takes as input a verification task in a form of linear constrained Horn clauses. Despite we described the algorithm in a setting of single-loop programs, FreqHorn also supports multiple (possibly, nested) loops, but the risk of divergence due to the search space pruning in this case is higher.

A. Parallel architecture

FreqHorn is designed to benefit from asynchronous parallelism. The *master process* takes care of the preprocessing, sampling, learning, and prioritizing steps. It is equipped with an incremental SMT solver, called safeSolver which holds the conjunction of all learned lemmas till the end of the entire verification.

The most expensive computation at the FreqHorn workflow happens to be the invariance checking. Each sampled *Cand* requires a number of isolated SMT checks, performed by *worker processes*. Thus each worker is equipped with its own SMT solver, called invSolver, which examines if *Cand* is an inductive invariant. In particular, invSolver gets reset before each tautology check, initiation check, and consecution check. After all checks are done, a worker communicates its positive or negative result back to the master, and the worker becomes available for another candidate.

When the verification starts, n workers are available. The master samples n candidates in a row and sends one sample per worker. Since each sample requires unpredictable worker's time, the communication between the master and workers is asynchronous. After a learned lemma is received, the master re-checks safety. If the safety check failed (or an invariance failure is received), the master creates / nudges the priority distributions, samples a new candidate and sends it to the available worker. If the safety check succeeded, the verification is done.

B. Evaluation

Benchmarks. We evaluated FreqHorn on a set of 76 loopy programs, taken from various sources including SVCOMP[2], literature (e.g., [2], [7]) and crafted programs. The set contains 16 benchmarks over non-linear arithmetic (i.e., with $\cdot$, `div`, and `mod` operators).

Experimental setup. We used m4.xlarge instances on Amazon Web Services, which have Intel Xeon E5-2676 v3 processors ("base clock rate of 2.4 GHz and can go as high as 3.0 GHz with Intel Turbo Boost") and 16GiB of RAM. All solvers were instantiated with Z3 [8]. Due to the stochastic nature of our learning, the FreqHorn timings are the means of 10 independent runs. We used a timeout of 10 mins for all runs.

[1]The source code and benchmarks are available at https://github.com/grigoryfedyukovich/aeval/tree/rnd-parallel-master-slave.

[2]Software Verification Competition, http://sv-comp.sosy-lab.org/

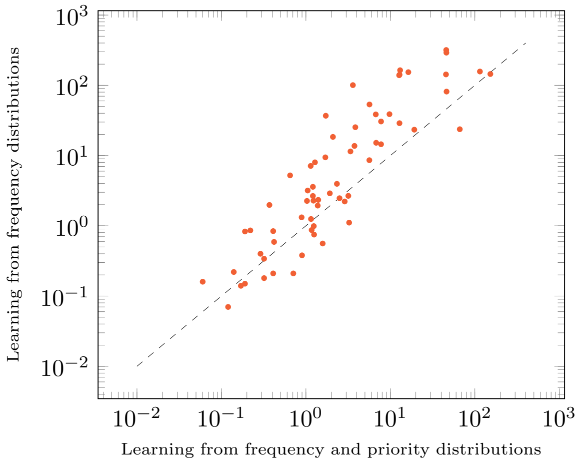


Figure 5: Effect of the search space pruning / prioritizing.

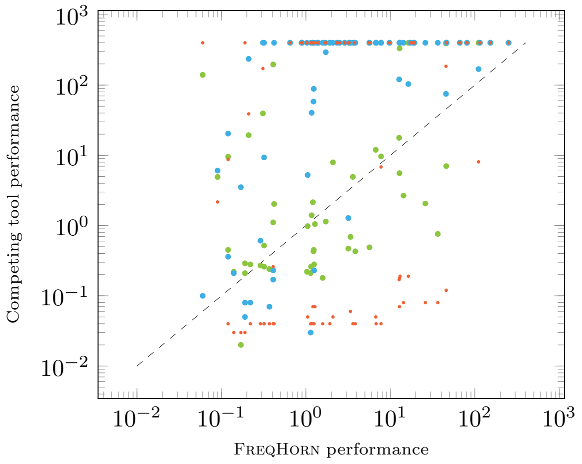


Figure 6: Comparison with μZ (red); ICE-DT (green); MCMC (blue).

Comparing FreqHorn's learning strategies. Fig. 5 shows a scatter plot comparing the average running times of FreqHorn with / without prioritizing the search space. Each point in the plot represents a pair of learning runs for the same benchmark: Alg. 2 (x-axis) and Alg. 1 (y-axis). Priority distributions accelerated the synthesis in 48 cases, and we witnessed a speedup up to 28%. As expected, there were six benchmarks, for which priority distributions helped finding an invariant (i.e., Alg. 1 diverged), but there were two another benchmarks, for which the search space was pruned too aggressively (i.e., Alg. 2 diverged).

Comparing with other tools. Fig. 6 shows a scatter plot comparing the timings of Alg. 2 and μZ v.4.5.0 [4], ICE-DT [7], and MCMC [9] invariant synthesizers. With four workers, FreqHorn outperformed μZ for 37 benchmarks (including 32 for which μZ crashed or reached timeout), ICE-DT for 53 (respectively, 30), and MCMC for 67 (respectively, 49).

Compared to μZ, FreqHorn solved all non-linear tasks and the tasks requiring large disjunctive invariants. Interestingly, FreqHorn was able to deliver a compact conjunctive representation of them. Compared to its closest competitor, MCMC (see Sect. V for more details), FreqHorn exhibited more consistent performance. We compared benchmarks for which both tools succeeded by their coefficients of variation (i.e., the ratio of the standard deviation to mean of the benchmark time): FreqHorn gets 0.60, and MCMC gets 0.92. Finally, FreqHorn for most benchmarks outperformed ICE-DT. This could be possibly explained by the reliance of the latter on the actual program executions, which are hard to get for non-deterministic programs.

V. Related work

Our enumerative approach can be considered as *data-driven* since it treats frequencies of various features in the source code as *data*. Among other enumerative-learning techniques, MCMC [9] employs Metropolis Hastings MCMC sampler to produce candidates for the whole invariant. Similarly to our approach, it obtains some statistics from the code (namely, constants), but as can be seen from Sect. IV-B, it is not enough to guarantee consistent results. In [10], [11], [7], the *data* is obtained by executing programs. Then, the learning of invariants proceeds by analyzing the program traces and does not take into account the source code.

There is a large body of inductive and non-enumerative SMT-based techniques for invariant synthesis, and due to lack of space we list only a few here. IC3 / PDR [12], [2], [3], [4] and abductive inference [13] depend crucially on the background theory of verification conditions which should admit interpolation and / or quantifier elimination. Those approaches were shown effective for propositional logic, linear arithmetic, and arrays. Our tool, in contrast, can discover non-linear invariants since it reduces the synthesis task to only quantifier-free queries and does not require interpolation.

Some prior work considered non-enumerative invariant inference from the source code. In Formula Slicing [14], a variant of Houdini [15], candidate invariants are chosen from the *Init* formulas (not from *Tr* or *Bad*, as in our case). In Niagara [16], [17] candidate invariants are obtained from the previous versions of the program. Despite those techniques proceed in the "*guess-and-check*" manner, for each new guess they just weaken the formula from the previous guess. In contrast, we permit much wider search space.

Syntax-guided approaches to synthesis [6] in general proceed by exploring the pre-determined (and adjusted for a particular problem) grammar. Recently, the "*Divide-and-Conquer*" [18] methodology, in which the problem is being solved in small pieces, has been successfully applied to SyGuS. Our approach has a similar underlying idea – to learn each lemma individually and conjoin it to the invariant – but the implementation via frequency and priority distributions is entirely new.

VI. Conclusion

We addressed the challenge of inductive invariant synthesis. Motivated by the observation that invariants can often be learned from the "*easy-to-get*" ingredients, we proposed to guide the learning process by frequency distributions, collected after a lightweight syntactic analysis of the source code, and to further prune / prioritize the search space. We built FreqHorn, the first tool that constructs the grammars for candidate invariants and distributions completely automatically, enables parallel verification of well-known programs over linear arithmetic, and to a limited extent supports non-linear arithmetic. In the future, we plan to investigate how deeper statistics about the code can help discovering more complicated inductive invariants.

References

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, vol. 1855 of *LNCS*, pp. 154–169, Springer, 2000.

[2] A. R. Bradley, "Understanding IC3," in *SAT*, vol. 7317 of *LNCS*, pp. 1–14, Springer, 2012.

[3] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, pp. 125–134, IEEE, 2011.

[4] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, vol. 7317, pp. 157–171, Springer, 2012.

[5] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, pp. 404–415, ACM, 2006.

[6] 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.

[7] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *POPL*, pp. 499–512, ACM, 2016.

[8] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, vol. 4963 of *LNCS*, pp. 337–340, Springer, 2008.

[9] 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.

[10] S. Gulwani and N. Jojic, "Program verification as probabilistic inference," in *POPL*, pp. 277–289, ACM, 2007.

[11] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV*, vol. 8559 of *LNCS*, pp. 69–87, Springer, 2014.

[12] A. R. Bradley and Z. Manna, "Property-directed incremental invariant generation," *Formal Asp. Comput.*, vol. 20, no. 4-5, pp. 379–405, 2008.

[13] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, pp. 443–456, ACM, 2013.

[14] E. G. Karpenkov and D. Monniaux, "Formula slicing: Inductive invariants from preconditions," in *HVC*, vol. 10028 of *LNCS*, pp. 169–185, Springer, 2016.

[15] 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.

[16] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Incremental verification of compiler optimizations," in *NFM*, vol. 8430 of *LNCS*, pp. 300–306, Springer, 2014.

[17] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Property directed equivalence via abstract simulation," in *CAV*, vol. 9780, Part II, pp. 433–453, Springer, 2016.

[18] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling Enumerative Program Synthesis via Divide and Conquer," in *TACAS, Part I*, vol. 10205 of *LNCS*, pp. 319–336, 2017.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Tagged BDDs: Combining Reduction Rules from Different Decision Diagram Types

Tom van Dijk
Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria
Email: tom.vandijk@jku.at

Robert Wille
Institute for Integrated Circuits
Johannes Kepler University Linz, Austria
Email: robert.wille@jku.at

Robert Meolic
Faculty of EE and CS
University of Maribor, Slovenia
Email: robert.meolic@um.si

***Abstract*—Binary decision diagrams are fundamental data structures in discrete mathematics, electrical engineering and computer science. Many different variations of binary decision diagrams exist, in particular variations that employ different reduction rules. For some applications, such as on-the-fly state space exploration, multiple reduction rules are beneficial to minimize the size of the involved graphs. We propose tagged binary decision diagrams, an edge-based approach that allows to use two reduction rules simultaneously. Experimental evaluations demonstrate that on-the-fly state space exploration is an order of magnitude faster using tagged binary decision diagrams compared to traditional binary decision diagrams.**

I. Introduction

Binary decision diagrams are fundamental data structures which emerged in the ’80s as a means to efficiently represent Boolean functions. They now find applications in many areas including logic synthesis, model checking, verification, automated reasoning, reachability analysis, and combinatorics. Section 7.1.4 in Knuth’s encyclopedia [17] and a recent survey paper by Minato [24] provide an accessible history of the research activities into binary decision diagrams.

Many variations of binary decision diagrams have been proposed. They differ, e.g., in the type of leaves, decomposition rules, and reduction rules. The existence of these variants motivates dedicated application areas. Original binary decision diagrams (BDDs) are, e.g., applied in model checking and logic synthesis [9], [8], [21], multi-terminal binary decision diagrams (MTBDDs) are, e.g., used to compute properties of probabilistic models [2], [14], [3] and zero-suppressed binary decision diagrams (ZBDDs) are exploited to compactly represent subsets and sparse matrices [23].

For some applications, it could be beneficial to combine characteristics of different binary decision diagram types. In this paper, we consider on-the-fly state space exploration such as conducted in [5], [16]. The efficiency of on-the-fly state space exploration relies on a compact representation of the involved Boolean functions, which initially include a large number of Boolean variables set to 0 (suitably represented by ZBDDs) but, over time, are extended by partial assignments to variables which become redundant (suitably represented by BDDs). Thus far, no solution combines the reduction rules from both BDDs *and* ZBDDs at the same time. Existing methods use either original BDDs or ZBDDs only and thus do not exploit the full potential of these reduction rules.

In the literature, different decomposition rules have been combined using a node-based approach, where nodes store the information about the applied decomposition rule [13]. However, combining BDDs and ZBDDs using this method would not be efficient, as it limits node sharing. Hence, we propose *tagged binary decision diagrams* (TBDDs), an edge-based approach that allows the simultaneous use of two reduction rules. To distinguish which rule is used to remove nodes, we introduce tags on every edge in the graph. We adapt several algorithms to handle both reduction rules and this leads to a more compact representation. We evaluate the proposed TBDDs for on-the-fly state space exploration and observe that they improve upon BDDs by an order of magnitude.

The rest of this paper is structured as follows. We cover the preliminaries in Section II. Section III motivates the new binary decision diagram type by reviewing on-the-fly symbolic state space exploration – an application where reduction rules from both BDDs and ZBDDs are beneficial. We describe the concepts of TBDDs in Section IV. In Section V, we present several algorithms that construct and manipulate TBDDs and discuss how they are used in the considered application. The benefits of TBDDs are illustrated and evaluated in Section VI. Finally, we reflect upon the obtained results in Section VII.

II. Preliminaries

This paper proposes a type of binary decision diagrams that exploits characteristics of both traditional binary decision diagrams and zero-suppressed binary decision diagrams. We briefly review both types in the following.

A. Binary Decision Diagrams

A *binary decision diagram* [7] is a rooted directed acyclic graph. There are only two distinct leaves labeled with 0 and 1. Each internal node v is labeled with $\mathrm{var}(v)$ and has two outgoing edges to successors denoted by $\mathrm{low}(v)$ (the *low* successor) and $\mathrm{high}(v)$ (the *high* successor). The edge leading to the root is called a *top edge*. A binary decision diagram is called *ordered* if each variable is encountered at most once on each path from the root to a leaf and the variables are encountered in the same order on all such paths. The *size* of a binary decision diagram is defined by the number of its nodes.

If binary decision diagrams are used to represent Boolean functions, then labels in nodes are Boolean variables and the

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

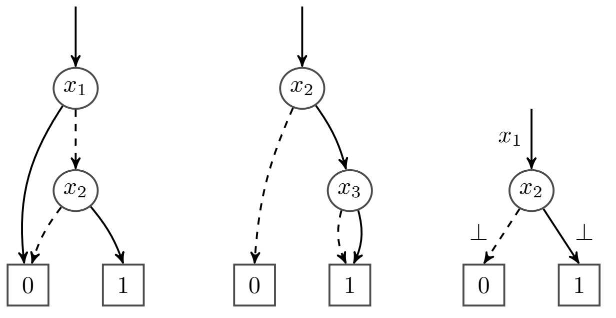
leaves 0 and 1 represent the Boolean constants. The Boolean function represented by the graph with root v is recursively determined using the *decomposition rule* which is based on Shannon's expansion theorem [27], i.e.,

$$f(v) = \overline{\mathrm{var}(v)} \cdot f(\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\mathrm{high}(v)).$$

Consequently, Boolean functions $f(\mathrm{low}(v))$ and $f(\mathrm{high}(v))$ coincide with $f(v)|_{\mathrm{var}(v)=0}$ and $f(v)|_{\mathrm{var}(v)=1}$, respectively.

The efficiency of binary decision diagrams is achieved by minimizing the structure. This is done by employing several *reduction rules*. The primary rule is that a binary decision diagram may not contain isomorphic subgraphs, i.e., *equivalent nodes* with the same $\mathrm{var}(v)$, $\mathrm{low}(v)$ and $\mathrm{high}(v)$. Traditional BDDs obey an additional rule, which removes so-called *redundant nodes*, i.e., nodes where $\mathrm{low}(v) = \mathrm{high}(v)$ [7]. With ZBDDs, this rule is substituted by one where a node v is called redundant iff $\mathrm{high}(v) = 0$. In both approaches, a decision diagram is *reduced* if it contains neither equivalent nodes nor redundant nodes. Both reduced BDDs and reduced ZBDDs are a canonical representation of Boolean functions and combinatorial sets.

Example 1. *Fig. 1 shows three binary decision diagrams representing the Boolean function $f(x_1, x_2, x_3) = \overline{x_1}x_2$. We use dashed lines for low edges and solid lines for high edges. TBDDs are proposed in Section IV.*



(a) BDD (b) ZBDD (c) TBDD

Fig. 1: Different binary decision diagrams representing the same Boolean function.

B. Complemented Edges

Modern binary decision diagram packages often use complemented edges [6], [25] as an additional mechanism to minimize the graph size. A complemented edge modifies the interpretation of the subgraph to which it points. In this section, we write $\neg v$ to denote that edge pointing to v is complemented, i.e., marked with a complement bit.

Complemented edges on BDDs are interpreted as follows. A complemented edge to 0 is interpreted as 1 (or vice-versa), and a complemented edge to an internal node is interpreted by toggling the complement bit on both successors, i.e.,

$$f(\neg v) := \overline{\mathrm{var}(v)} \cdot f(\neg\,\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\neg\,\mathrm{high}(v)).$$

It follows that $f(\neg v)$ (complemented edge) is equal to $\overline{f(v)}$ (negation) for BDDs and therefore that computing the negation of a formula is free with BDDs.

With ZBDDs, complemented edges cannot be used such that they coincide with negation. Nevertheless, complemented edges can be used to increase node sharing. Since forwarding the complement bit to the high successor counteracts the "zero-suppressing" property of ZBDDs, the complement bit is only forwarded to the low successor:

$$f(\neg v) := \overline{\mathrm{var}(v)} \cdot f(\neg\,\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\mathrm{high}(v))$$

Complemented edges allow multiple representations of the same function by toggling the complement bit on the incoming edge and both successors (for BDDs), or on the incoming edge and the low successor (for ZBDDs). A well-known method to ensure a canonical representation is to forbid the complement bit on the low successor, which works well for both BDDs and ZBDDs [6], [23].

C. Generalizing Reduction Rules

The reduction rules of BDDs and ZBDDs target two *patterns* of nodes that are removed. For BDDs these are nodes with equivalent successors, whereas for ZBDDs these are nodes whose high successors are 0. In general, given a graph in which no variable is skipped, the reduction rule removes all nodes that match a given pattern. Given the successors of a node v as a pair $(\mathrm{low}(v), \mathrm{high}(v))$, we can identify the following *simple* reduction patterns involving only one node:

(1)	(2)	(3)	(4)	(5)
$(k,k)\Rightarrow k$	$(k,0)\Rightarrow k$	$(k,1)\Rightarrow k$	$(0,k)\Rightarrow k$	$(1,k)\Rightarrow k$

Pattern 1 is the rule for BDDs, while pattern 2 is the rule for ZBDDs. If we also allow patterns involving negation, we get 12 patterns [22]. This can be generalized further for patterns that involve multiple nodes, such as representations of $x = x'$. However, this is beyond the scope of this paper.

III. Motivation

BDDs and ZBDDs offer compact representations for Boolean functions. As they use different reduction rules, their size significantly depends on the kind of Boolean functions they represent. BDDs are particularly suited for functions where adjacent input assignments have the same outcome, whereas ZBDDs are more compact for functions that often evaluate to 1 when many variables are set to 0. This motivates dedicated application areas for either type. BDDs are heavily applied, e.g., in symbolic model checking, while ZBDDs find particular application, e.g., to represent sparse matrices and subsets.

However, applications exist where characteristics from BDDs *and* ZBDDs could both be beneficial. This section reviews such an application. Afterwards, we illustrate how using characteristics from both types could advance the state-of-the-art. This provides the main motivation for the novel binary decision diagram type proposed in Section IV which combines reduction rules of BDDs and ZBDDs.

A. On-the-fly State Space Exploration

Binary decision diagrams are an important data structure in symbolic model checking, which creates models of complex systems to verify that they function according to certain properties or a given specification. Systems are modeled as a set of states S and a transition relation $T \subseteq S \times S$. We encode these sets using their characteristic functions $\mathbf{S}$ and $\mathbf{T}$ (such that $S = \{\, s \mid \mathbf{S}(s) \,\}$ and $T = \{\, (s, s') \mid \mathbf{T}(s, s') \,\}$), which we represent by binary decision diagrams. Since algorithms on binary decision diagrams effectively operate on sets of states rather than individual states, they have successfully been applied to verify systems with a large number of states [9], [8], [28] (infeasible for explicit-state model checking).

A central role in model checking algorithms, such as verifying properties expressed in the modal μ-calculus [18], [4], LTL [10] or CTL [8], [19] is state space exploration. Here, all reachable states in (a part of) the state space are determined starting from a given set of initial states. This is typically conducted by computing the successor states according to the transition relation until a fixed point has been reached.

The model checking toolset LTSMIN [5], [16], [20] offers a framework where transition relations are updated on-the-fly as the model is explored. Initially, LTSMIN does not have knowledge of the transitions in the system. The transition system is explored by learning new transitions via a language-independent interface called PINS,which connects various input languages to model checking algorithms [16]. In PINS, the states of a system are represented by vectors of integers.

B. Utilizing Characteristics from BDDs and ZBDDs

One of the challenges in on-the-fly state space exploration is that the number of bits required to encode the state space is not known in advance. As LTSMIN uses integers as the fundamental data type, these integers are encoded with, e.g., 16 or 32 bits per integer. However, state variables often hold only few values in the reachable state space, so most bits are always set to 0. Hence, ZBDDs can more effectively represent the state space, as nodes for variables set to 0 are then eliminated [15]. On the other hand, the set of reachable states often includes adjacent states, where some variable x can be either True or False. Such sets are effectively represented with traditional BDDs. Hence, both the reduction rules of ZBDDs and traditional BDDs would significantly reduce the size of the binary decision diagrams.

Example 2. *In Fig. 2 we consider one integer variable of a possibly much larger system, encoded using 8 bits. In this example, we have discovered that the variable may hold values* $\{0, 2, 4, 6\}$*. Using ZBDDs would allow for a compact representation for variables* $x_0, x_1, x_2, x_3, x_4, x_7$ *(since these variables would have their high edge to* 0*). In contrast, using BDDs would allow for a compact representation for variables* x_5, x_6 *(since these variables would have identical successors). Hence, both the reduction rules of ZBDDs and ordinary BDDs would significantly reduce the size of the representation.*

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	0	1	1	0
		A			B		C

Fig. 2: The BDD representing the four states would have nodes with the high edge to 0 for the variables in regions “A” and “C” and with identical successors in region “B”.

IV. TAGGED BINARY DECISION DIAGRAMS

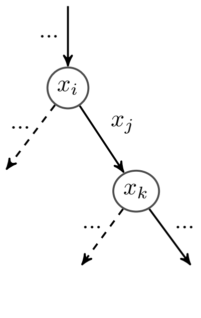
This section introduces a novel binary decision diagram type which addresses the drawback discussed in the previous section. The general idea is that nodes are removed according to two different reduction rules. To distinguish which rule has been applied between two adjacent nodes in the graph, a variable label is used as a *tag* on every edge (note that edge tags have also been used for other purposes, e.g., in [25]). Missing variables *before* the tag are removed according to the first rule, while variables missing *starting from* the tag are removed according to the second rule. Both reduction rules are maximally applied. Overall, this combines the benefits from both decision diagram types and thus leads to a more compact representation. In the following, the concepts are described first. Afterwards, the prototype implementation is described in Section V, while their efficiency is experimentally evaluated in Section VI.

A. Definition

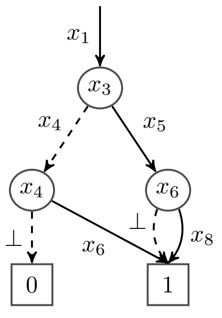
A *tagged binary decision diagram* (TBDD) is a rooted directed acyclic graph. There are two leaves that are labeled with 0 and 1. Each internal node v is labeled with $\mathrm{var}(v)$ and has two outgoing edges to successors denoted by $\mathrm{low}(v)$ (the *low* successor) and $\mathrm{high}(v)$ (the *high* successor). Like ordinary binary decision diagrams, variables are encountered along each directed path according to a fixed variable ordering and equivalent nodes are forbidden. In addition, edges are labeled with a *tag*, which can be a variable label or $\perp$. Edges to internal nodes are always labeled with a variable label, while edges to leaves may also be labeled with $\perp$. The variable used as a variable label must always be *after* the variable of the source node and *before or equal to* the variable of the target node, if the target node is not a leaf.

TBDDs admit two reduction rules that forbid certain nodes. Examples of these rules are given in Section II-C. In the remainder of this paper, we use the reduction rule of BDDs as the first rule and the reduction rule of ZBDDs as the second rule. Both rules are maximally applied. The tag determines how missing nodes are treated. Informally, the tag x_{tag} means that all skipped variables *before* x_{tag} were removed due to the first rule and that all skipped variables *starting from* x_{tag} were removed due to the second rule. The tag $\perp$ on an edge to a leaf means that all remaining variables were removed due to the first reduction rule.

Example 3. *Fig. 3a shows a fragment of a TBDD with the variable ordering $x_i < x_j \leq x_k$. Nodes with variable x_m such that $x_i < x_m < x_j$ were removed by the first reduction rule, while nodes with variable x_n such that $x_j \leq x_n < x_k$ were removed by the second reduction rule. Fig. 3b shows a concrete TBDD where the first reduction rule is those of BDDs and the second reduction rule is those of ZBDDs. For the variables $\{x_0, \dots, x_8\}$, this TBDD represents $\overline{x_1 x_2 x_3} x_4 \overline{x_6} x_7 \overline{x_8} \vee \overline{x_1 x_2} x_3 \overline{x_5 x_6} \vee \overline{x_1 x_2} x_3 \overline{x_5} x_6 \overline{x_8}$. This expression is obtained by looking at each path from the root to leaf 1 and assigning False to all variables that are skipped and equal to or greater than the tag.*



(a)



(b)

Fig. 3: Examples of tagged binary decision diagrams.

To interpret a BDD as a Boolean function, the variable domain is not explicitly needed, while for ZBDDs and for TBDDs the represented Boolean function depends on the variable domain. In the following, for a given edge and variable domain *Dom* the notations Dom_1 and Dom_2 are used, where Dom_1 denotes the variables in *Dom* before the tag and Dom_2 denotes the variables in *Dom* starting from the tag and before the variable of the target node if it is not a leaf. If the tag is $\perp$, then $Dom_1 = Dom$ and $Dom_2 = \emptyset$. For the two reduction rules, we can deduce interpretation functions $Interp_1$ and $Interp_2$. These functions map a set of variables and a Boolean formula to the Boolean formula of the interpretation. An edge to a node or leaf v is interpreted as $Interp_1(Dom_1, Interp_2(Dom_2, f(v)))$.

The interpretation function of the reduction rule of BDDs is simply $Interp(Dom, f) := f$, while the interpretation function of the reduction rule of ZBDDs is $Interp(Dom, f) := \bigwedge_{x \in Dom} \overline{x} \wedge f$. An edge to a node or leaf v in a TBDD using these two rules is interpreted as $\bigwedge_{x \in Dom_2} \overline{x} \wedge f(v)$. Furthermore, edges to the leaf 0 can only have tag $\perp$. Finally, we introduce complemented edges similarly as for ZBDDs. This means that we cannot use complemented edges to compute negation, but at least we can sometimes reuse nodes.

B. Reduction Rules During Construction

The two reduction rules are maximally applied. However, a special case occurs when nodes that are eliminated by the two rules alternate. In this case, either every last node matching the second rule or every first node matching the first rule must be kept. In the following, we maximally apply the second rule (rule of ZBDDs) and thus some nodes of the first rule (rule of BDDs) are kept to denote these alternating sequences.

Example 4. *The binary decision diagram in Fig. 4 has two nodes that have the high edge to 0 and a node with identical successors. Applying the reduction rules removes all nodes with the high edge to 0. The remaining node cannot be eliminated, because it would no longer be possible to distinguish which rule was used to eliminate which node.*

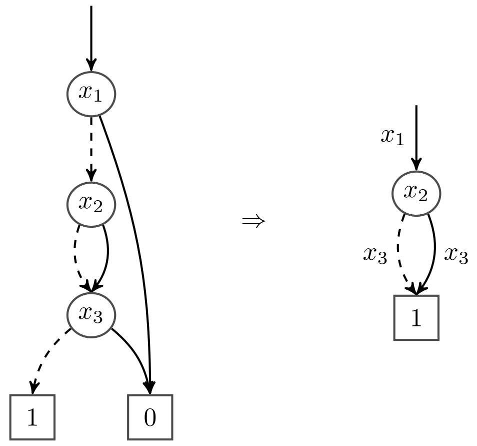


Fig. 4: A TBDD where not all nodes can be eliminated.

Binary decision diagrams are reduced from bottom to top during their construction. Whenever a node is constructed that matches one of the reduction rules, the rule is applied, in the case of the rule of BDDs by simply returning the original low (or high) successor, and in case of the rule of ZBDDs by returning the low successor with an updated tag. However, if we deduce that the low successor already has skipped variables according to the rule of BDDs, then we keep an extra node.

Fig. 5 illustrates the respective reduction rules for TBDDs. The bottom-up reduction is started with a graph in which no variable is skipped. All edges are initially labeled with the same variable as the node that they point to, or $\perp$ if they point to a leaf node. Rules 1 and 2a are applied in the most cases. For the special case where nodes that match both reduction rules alternate, rule 2b ensures that an extra node is created. As we maximally apply the rule of ZBDDs, rule 2b creates a node that matches the rule of BDDs; if we would maximally apply the rule of BDDs, then rule 2b would be modified to accomplish this.

C. Canonical Representation

TBDDs are, like BDDs and ZBDDs, a canonical representation of Boolean functions, under the condition that the reduction rules are maximally applied. In general, this removes all nodes of the two chosen patterns, except exactly those nodes that are required when rule applications alternate (as discussed above). In the considered case (combining BDDs and ZBDDs), the canonicity can be simply derived by observing the bijection between BDDs, ZBDDs, and TBDDs, i.e., every reduced TBDD matches with a unique reduced BDD (ZBDD)

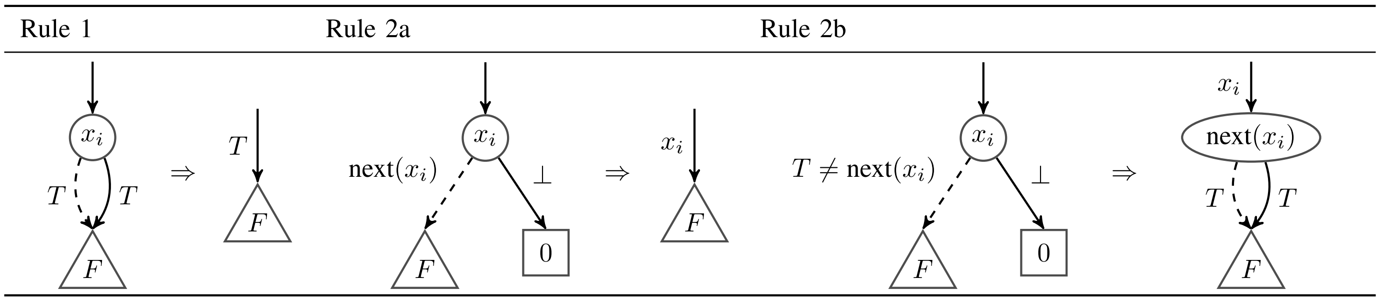


Fig. 5: The reduction rules of a TBDD for bottom-up reduction. Here T stands for any tag and F for any reduced TBDD; next(x) is the next variable in the variable domain, or $\perp$ if x is the last variable.

with a deterministic conversion procedure, and vice versa. The conversion procedure can be defined as a two step procedure such that the first step is a creation of a graph in which no variable is skipped. Therefore, since BDDs (and ZBDDs) are a canonical representation, so are TBDDs.

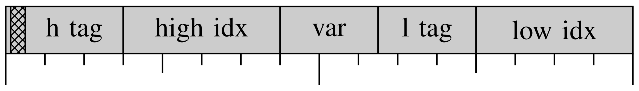
V. Implementation

Algorithms on TBDDs are not trivial, since they have to deal with various special cases introduced by the two reduction rules. This section first describes the representation of nodes and edges of TBDDs followed by the algorithm `FoaNode` that constructs TBDDs for a given function by finding or adding nodes. Afterwards, the algorithms `ite` and `exists` are described which provide basic operations on TBDDs. Finally, we briefly discuss a more specialized algorithm that is used for the on-the-fly state space exploration.

A. Representation of Nodes and Edges

We implemented TBDDs in the parallel decision diagram package Sylvan [11], [12]. As Sylvan allocates 16 bytes per node (for performance), we design the internal structure of a TBDD node to fit these constraints. In binary decision diagram implementations, nodes are stored in a *unique table*. This ensures that no equivalent nodes are created. We use 32 bits to store the index of a node in the unique table. This is sufficient to store up to 2^{32} nodes, i.e., 96 gigabytes of nodes, excluding overhead costs. For the variable labels and the tags on the edges, we allocate 20 bits. This allows up to 1,048,576 variables; we reserve the highest value to encode $\perp$.

In our implementation, we use a 64-bit integer to encode a tagged edge to a TBDD node. The lowest 32 bits represent the location of the node in the table, and the highest bit stores the complement bit [6]. The TBDD 0 is reserved for the leaf 0 (False), with the complemented edge to 0 for True. The 31 remaining bits provide space to store the 20-bit tag. A TBDD node in memory stores the variable label (20 bits), the low edge index (32 bits), the high edge index (32 bits), the low edge tag (20 bits), the high edge tag (20 bits) and the complement bit of the high edge (1 bit, the first bit below) as follows:



B. Constructing TBDD Nodes

The core function to obtain nodes is `FoaNode` (find or add a node), which applies the reduction rules presented in Fig. 5 and creates a new node if necessary. This function is given the variable x_i and tagged edges L (low) and H (high), as well as the next variable x'_i (which can be $\perp$) which is necessary to apply the reduction rules of Fig. 5. The function's result is a tagged edge to a TBDD node. The `FoaNode` algorithm is shown as Alg. 1. It first applies Rule 1 (line 2). If there is a complement on the low edge, we apply the rule that forbids complemented edges on the low edge (line 3). We then apply Rules 2a and 2b (lines 4–6) by comparing the tag on the low edge to the next domain variable. If they are the same, Rule 2a is applied by returning an edge to the given node with the tag according to Rule 2a (line 5). Otherwise, an extra node is created (line 6) and returned with the appropriate tag (line 8). If no reduction rule can be applied, a TBDD node is created via the unique table (line 7) and returned with the appropriate tag (line 8). Note that `find-or-insert` is provided by the unique table and it does not create a new TBDD node if the requested node already exists.

```
1 def FoaNode($x_i$, $L$, $H$, $x'_i$):
2     if $L = H$ : return $L$
3     if comp($L$) : return $\neg$FoaNode($x_i$, $\neg L$, $H$, $x'_i$)
4     if $H = 0$ :
5         if tag($L$) $= x'_i$ : return settag($L$, $x_i$)
6         else: node $\leftarrow$ find-or-insert($\{x'_i, L, L\}$)
7     else:  node $\leftarrow$ find-or-insert($\{x_i, L, H\}$)
8     return settag(node, $x_i$)
```

Algorithm 1: The `FoaNode` method that constructs a TBDD node and applies the reduction rules.

C. Basic Operations

This section describes several basic TBDD operations. All operations use an operation cache to store subresults, like virtually all binary decision diagram operations. We assume that the reader is familiar with this technique and omit it here. We also assume that the involved TBDDs are interpreted in the same variable domain as the result. This variable domain

```
1  def cofactors($F$, $x_i$, $x_i'$):
2      if $F = 0$ : return $0, 0$
3      elif $x_i <$ tag($F$) : return $F, F$
4      elif $F = 1$ : return $F, 0$
5      elif $x_i <$ var($F$) :
6          $F_0 \leftarrow$ low($F$)
7          if $F_0 \notin \{0,1\} \wedge$ var($F_0$)$= x_i' \wedge$ low($F_0$)$=$high($F_0$) :
8              $F_0 \leftarrow$ low($F_0$)
9          return $F_0$, $0$
10     else: return low($F$), high($F$)
```

Algorithm 2: Fragment used by TBDD algorithms to compute the cofactors F_0 and F_1, given a domain variable $x_i \leq \text{var}(F)$ and the next domain variable x_i' (or $\perp$ if x_i is the last).

```
1  def ite($F$, $G$, $H$):
2      if $F = 1$ : return $G$
3      if $F = 0$ : return $H$
4      if $G = H$ : return $G$
5      $t \leftarrow$ min(tag(F), tag(G), tag(H))
6      if tag(F) = tag(G) = tag(H) :
7          $v \leftarrow$ min(var(F), var(G), var(H))
8      else: $v \leftarrow t$
9      $F_0, F_1 \leftarrow$ cofactors ($F$, $v$, next($v$))
10     $G_0, G_1 \leftarrow$ cofactors ($G$, $v$, next($v$))
11     $H_0, H_1 \leftarrow$ cofactors ($H$, $v$, next($v$))
12     $r \leftarrow$ FoaNode($v$, ite($F_0$, $G_0$, $H_0$), ite($F_1$, $G_1$,
           $H_1$), next($v$))
13     if $t \neq v$ : $r \leftarrow$ FoaNode($t$, $r$, 0, $v$)
14     return $r$
```

Algorithm 3: The implementation of the algorithm `ite`.

is given as an additional parameter in the operations, but we omit it here in the interest of clarity and brevity.

Many binary decision diagram operations first find a *pivot variable*, typically the topmost variable of the root nodes of the parameters, then recursively compute the subresults of the operation on the cofactors of the parameters obtained by setting the pivot variable to False or True, and finally compute the result by creating a node with `FoaNode`.

Computing the cofactors is straightforward and summarized in Alg. 2. Note that this is *not* a "generic cofactor" operation; just a helper method for the recursion step. This method returns $F|_{x_i=0}$ and $F|_{x_i=1}$ for a given TBDD F with top variable x_i and also compensates for the application of Rule 2b. We use `low` and `high` to obtain the low and high successor of F, where `low` also applies any complement on F to the low edge. After checking whether F is an edge to 0 (line 2), we check whether Rule 1 was applied (line 3). If not, we check if Rule 2 was applied to leaf 1 (line 4). If not, then we check if Rule 2 was applied to the node (line 5) and if so, we check if the node is a redundant node inserted by Rule 2b (line 7) and return the appropriate result (lines 6–9). Finally, at line 10 it is established that no reduction rule was used and we return the successors of the TBDD.

```
1  def exists($F$, $\vec{x}$):
2      if $F = 0 \vee (F = 1 \wedge \text{tag}(F) = \perp)$ : return $F$
3      if $\vec{x} = \emptyset$ : return $F$
4      while var($\vec{x}$) < tag($F$) :
5          $\vec{x} \leftarrow \vec{x} \setminus \{\text{var}(\vec{x})\}$
6          if $\vec{x} = \emptyset$ : return $F$
7      $v \leftarrow$ min(var(F), var($\vec{x}$))
8      $F_0, F_1 \leftarrow$ cofactors ($F$, $v$, next($v$))
9      $\vec{x}' \leftarrow \vec{x} \setminus \{v\}$
10     if $v <$ var($F$) :  res $\leftarrow$ exists($F_0$, $\vec{x}'$)
11     elif $v =$ var($\vec{x}$) :  res $\leftarrow$ or(exists($F_0$, $\vec{x}'$),
           exists($F_1$, $\vec{x}'$))
12     else:  res $\leftarrow$ FoaNode($v$, exists($F_0$, $\vec{x}'$),
           exists($F_1$, $\vec{x}'$), next($v$))
13     if tag($F$) $< v$ : res $\leftarrow$ FoaNode(tag($F$), res, 0, $v$)
14     return res
```

Algorithm 4: The implementation of the algorithm `exists`.

See Alg. 3 for the implementation of the well-known `if-then-else` operation. Given three TBDDs representing f, g and h, this algorithm computes "if f then g else h". We use `tag` and `var` to obtain the tag of an edge and the variable of the root node. The algorithm first tries to apply the trivial cases (lines 2–4). We compute the topmost tag (line 5) and then we determine the pivot variable v. Variables that are in Dom_1 of all three parameters are in Dom_1 of the result. Variables that are in Dom_2 of *all* three parameters are in Dom_2 of the result. We perform recursion on the first variable that is not in Dom_1 of all parameters or in Dom_2 of all parameters. This variable is equal to the lowest variable if all tags are the same, or the lowest tag if this is not the case. The cofactors are computed at lines 9–11 and the recursion is performed at line 12, where also the result is computed. If there are variables in Dom_2 of the result (which is true if $t \neq v$) then we use `FoaNode` in a special way to compute the result. This ensures the correct application of Rule 2b when the result has variables in Dom_1.

We also implement existential quantification as in Alg. 4. This operation existentially quantifies all given variables $\vec{x}$ from the input TBDD F. We check for the trivial cases at lines 2–3. We then skip all variables in $\vec{x}$ that are in Dom_1 of F (lines 4–6). Variables that are in Dom_2 and before var($\vec{x}$) will be in Dom_2 of the result. As the pivot variable we select the first variable of F and $\vec{x}$ (line 7). We obtain the cofactors and if the chosen pivot variable is in $\vec{x}$ we remove it in $\vec{x}'$ that we use for the recursion (lines 8–9). Now there are three cases. If the pivot variable is in Dom_2, then we just compute the result based recursively on F_0, as F_1 is 0 (line 10). Otherwise, the pivot variable is the variable of the TBDD node. Now either it is also in $\vec{x}$ and we perform recursion as usual and compute the disjunction of the two results, which is how existential quantification is computed (line 11), or it is not in $\vec{x}$ and we perform recursion as usual and compute the node of the result (line 12). Finally, we update Dom_2 of the result as discussed above (line 13) and return the result (line 14).

As is clear from the above discussions, we must consider for all variables whether they are in *Dom*$_1$ or *Dom*$_2$ of the parameters. This makes these algorithms more complex, but overall improves the efficiency by exploiting the reductions.

D. Computing Successors

Besides "typical" decision diagram operations, we additionally implemented an operation dedicated for the LTSMIN application motivated in Section III, namely `relnext` which applies a transition relation to a set in order to compute the successors (combined with variable renaming). An additional challenge is that LTSMIN partitions the transitions into different transition groups. This has the advantage that each transition group only affects a part of the state vector. As a consequence, the `relnext` operation cannot assume that both parameters are defined on the same variable domain; rather, it handles various special cases introduced by the difference in the variable domains. In the interest of space, we cannot treat `relnext` here but refer the interested reader to the publicly available source code (see next section).

VI. Experimental Evaluation

This section evaluates the ideas proposed in this paper. The evaluation is performed based on the BEEM database of models [26]. In Sec. VI-A, we study the impact of different reduction rules on the size of the graphs at each iteration in state space exploration. In Sec. VI-B, we evaluate the performance of state space exploration using either BDDs or TBDDs. We use LTSMIN to perform on-the-fly state space exploration, using the FORCE algorithm for selecting a variable reordering [1] and using the standard breadth-first-search strategy to explore the state space. All experimental data and the scripts required to reproduce them are available online via http://fmv.jku.at/tbdd.

A. Impact of Reduction Rules

We modify LTSMIN to write the BDDs of the explored states and of all transition relations to disk at every iteration of the on-the-fly state space exploration. We compute the number of nodes required for the explored states and the number of nodes required for all transition relations, when removing no redundant nodes, when removing nodes according to the five rules from Sec. II-C (without using complemented edges) and when using the proposed TBDD type.

Since computing these sizes costs a lot of time, we only perform this analysis on a subset of the full benchmark set. Our hypothesis in Sec. III was that, as the state space exploration progresses, the contribution from the $(k, k) \Rightarrow k$ rule would increase. We found that this is not the case for the models that we checked. See the top row of Fig. 6 for a representative model (`at.1`). The top lines in the graphs represent rules 3 and 5 and the case where no rule is applied. They appear to have no significant effect. The bottom lines represent rule 2 (ZBDD, dots) and the TBDD representation (long dashes). These rules have the most significant effect. The middle lines represent rule 1 (BDD, dashes) and rule 4 (dots and dashes) Rules 1 and 4 have a greater impact for the transition relation than for the set of explored states.

B. On-the-fly State Space Exploration

We look at the performance of on-the-fly state space exploration on the BEEM benchmark database, using either standard BDDs (16 bits per state variable) or TBDDs. We set the timeout for the experiments to 1200 seconds, i.e., 20 minutes. The experiments are run on a 48-core AMD Opteron machine. We set the number of cores to either 1 or 48, so we can also measure the effect of the parallelism of Sylvan.

For 219 models, no experiment timed out. For these models, the results are summarized by the following table, where the time is the sum of all models, and the number of nodes is the size of the BDD that represents the set of visited states.

	BDD	TBDD
Time 1 core	24504 sec.	6453 sec.
Time 48 cores	14672 sec.	1075 sec.
#Nodes in the visited set	59,503,837	5,922,973

See further Fig. 6 for a comparison of the time and the number of nodes for the models, using just 1 worker. We see that TBDDs are approximately an order of magnitude faster and smaller. Finally, the highest parallel speedup we obtain when using TBDDs was 32.4× (with 48 cores) for the model `rushhour.3`.

VII. Conclusions

BDDs and ZBDDs offer compact representations of Boolean functions which is crucial for many applications. Due to the different reduction rules, their effectiveness strongly depends on the type of functions. The proposed TBDDs simultaneously apply reduction rules of both types which in general results in smaller graphs for a broader set of applications.

We studied the benefits of TBDDs for on-the-fly state space exploration using the BEEM database of models. We obtained a significant improvement compared to ordinary BDDs, although our analysis showed that the contribution of the reduction rule of BDDs was less than expected. Our implementation of TBDDs also resulted in a similarly high parallel scalability as has been obtained for BDDs in the past.

We expect that improvements and tuning of the TBDD operations may result in a better performance, especially considering that these operations are complex and therefore give various opportunities to be optimized. Furthermore, we believe that looking at other benchmark models and application areas would be insightful. Tagged binary decision diagrams can also be applied using other reduction rules, like those mentioned in this paper, or rules involving multiple nodes, or simply by exchanging the order of the rules that we applied here. Furthermore, implementation of other operations like dynamic variable reordering may be challenging, as the interaction between the reduction rules of TBDDs and the popular sifting algorithm is probably complex. These are several ideas that may inspire future research on TBDDs.

Acknowledgment

The authors acknowledge the support of FWF, NFN Grant S11408-N23 (RiSE) and of the COST Action IC1405.

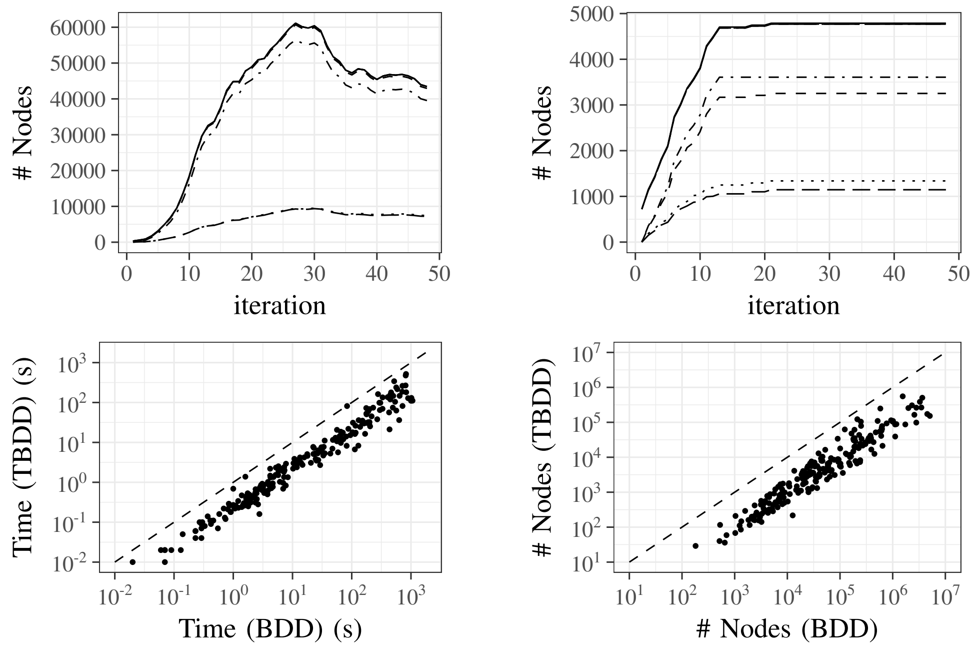


Fig. 6: Results of experimental evaluation. The first row shows the evolution of graph sizes of the set of explored states (left) and the transition relations (right) per iteration (see Sec. VI-A). The second row compares BDDs with TBDDs.

References

[1] Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: VLSI 2003. pp. 116–119. ACM (2003)

[2] Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Formal Methods in System Design 10(2/3), 171–206 (1997)

[3] Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: Automata, Languages and Programming 1997. LNCS, vol. 1256, pp. 430–440. Springer (1997)

[4] Biere, A.: μcke - Efficient μ-Calculus Model Checking. In: CAV 1997. pp. 468–471. LNCS, Springer (1997)

[5] Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: CAV. LNCS, vol. 6174, pp. 354–359. Springer (2010)

[6] Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC. pp. 40–45 (1990)

[7] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers C-35(8), 677–691 (8 1986)

[8] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (4 1994)

[9] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^20 states and beyond. Inf. Comput. 98(2), 142–170 (1992)

[10] Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design 10(1), 47–71 (1997)

[11] van Dijk, T.: Sylvan: Multi-core Decision Diagrams. Ph.D. thesis, University of Twente (7 2016)

[12] van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer pp. 1–22 (2016)

[13] Drechsler, R., Becker, B.: Ordered Kronecker functional decision diagrams-a data structure for representation and manipulation of Boolean functions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(10), 965–973 (10 1998)

[14] Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design 10(2/3), 149–169 (1997)

[15] Haji Ghasemi, M.: Symbolic model checking using Zero-suppressed Decision Diagrams. Master's thesis, University of Twente, Dept. of C.S. (11 2014)

[16] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer (2015)

[17] Knuth, D.E.: The Art of Computer Programming, vol. 4A. Addison-Wesley, Upper Saddle River, New Jersey (2011)

[18] Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)

[19] Marrero, W.: Using BDDs to decide CTL. In: TACAS 2005. pp. 222–236 (2005)

[20] Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, Write and Copy Dependencies for Symbolic Model Checking. In: HVC. pp. 204–219 (2014)

[21] Meinel, C., Somenzi, F., Theobald, T.: Linear sifting of decision diagrams and its application in synthesis. IEEE Trans. on CAD 19(5), 521–533 (2000)

[22] Meolic, R., Brezočnik, Z.: Universal decomposition rule for OBDD, OFDD, and 0-sup-BDD. Tech. rep., University of Maribor (2016), https://dk.um.si/IzpisGradiva.php?id=64403

[23] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th international Design Automation Conference. pp. 272–277. DAC '93, ACM, New York, NY, USA (1993)

[24] Minato, S.: Techniques of BDD/ZDD: Brief History and Recent Activity. IEICE Transactions 96-D(7), 1419–1429 (2013)

[25] Minato, S., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: Proceedings of the 27th ACM/IEEE Design Automation Conference. pp. 52–57. DAC '90, ACM, New York, NY, USA (1990)

[26] Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN. pp. 263–267. Springer-Verlag, Berlin, Heidelberg (2007)

[27] Shannon, C.E.: A Symbolic Analysis of Relay and Switching Circuits. Transactions of the American Institute of Electrical Engineers 57(12), 713–723 (12 1938)

[28] Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets. In: FMCAD. pp. 435–449 (1996)

First Order Temporal Logic Monitoring with BDDs

Klaus Havelund
Jet Propulsion Laboratory,
California Inst. of Technology, USA

Doron Peled
Department of Computer Science
Bar Ilan University, Israel

Dogan Ulus
Verimag/Université Grenoble-Alpes
Grenoble, France

***Abstract*—Runtime verification is aimed at analyzing execution traces stemming from a running program or system. The traditional purpose is to detect the lack of conformance with respect to a formal specification. Numerous efforts in the field have focused on monitoring so-called parametric specifications, where events carry data, and formulas can refer to such. Since a monitor for such specifications has to store observed data, the challenge is to have an efficient representation and manipulation of Boolean operators, quantification, and lookup of data. The fundamental problem is that the actual values of the data are not necessarily bounded or provided in advance. In this work we explore the use of Binary Decision Diagrams (BDDs) for representing observed data. Our experiments show a substantial improvement in performance compared to related work.**

I. Introduction

Runtime verification (RV) allows checking whether a temporal property holds during the execution of a system. The system execution can be considered as emitting an execution trace, a sequence of events, which is then consumed and checked by a monitor. A monitor performs for each received event some incremental computation that is aimed at detecting and warning as soon as the temporal property is violated. The field of model checking has mostly focused on propositional logics [18]. Very early RV systems, were also based on specifications given in some form of propositional temporal logic. A propositional temporal logic formula can for example be translated into a finite automaton, where the incremental computation updates the automaton state based on the recent input reporting information captured from the monitored system. However, the state of the art in RV has for some time focused on monitoring so-called parametric specifications, where events carry data, and formulas can refer to such. Since such a monitor has to store observed data, the challenge is efficient representation; manipulation such as negation, conjunction, disjunction, quantification, and lookup. The field has not settled on a single best solution. As is usually the case, there are compromises to be made with respect to the efficiency of algorithms and expressiveness of logics.

Temporal logics usually come in two variants: future and past (or mixtures). As future temporal properties depend on an infinite input, one may only provide partial information about whether the property holds; namely, if it is already violated, already achieved, or undecided yet [18]. The focus of this work is past temporal properties, which are also classified as the safety temporal properties [2], [19], and are properties for which we are capable of detecting a violation based on the monitored current prefix of the execution, as soon as it occurs [18]. As an example, consider a predicate $open(f)$, indicating that a file f is being opened, and a predicate $close(f)$ indicating that f is being closed. We can formulate that a file cannot be closed unless it was opened before with the following first order past time temporal logic formula:

$$\forall f\,(close(f) \longrightarrow \mathbf{P}\,open(f))$$

Here **P** is the "sometimes in the past" temporal operator. This property must be checked for every monitored event. Already in this very simple example we see that we need to store *all* the names of files that were previously opened so we can compare to the files that are being closed. A more refined specification would be the following, requiring that a file can be closed only if it was opened before, and has not been closed since. Here, we use the temporal operators $\ominus$ ("at previous step") and $\mathcal{S}$ ("since"):

$$\forall f\,(close(f) \longrightarrow \ominus(\neg close(f)\,\mathcal{S}\,open(f)))$$

One problem we need to solve is the unboundedness caused by negation. For example, assume that we have only observed so far one *close* event $close(\text{“ab”})$. The subformula $close(f)$ is therefore satisfied for the value $f = \text{“ab”}$. The subformula $\neg close(f)$ is satisfied by all values from the domain of f *except* for "ab". This set contains those values that we have not seen yet in the input within a *close* event. We need a representation of finite and infinite sets of values, upon which applying complementation is efficient.

We present a first order past time temporal logic, named QTL (Quantified Temporal Logic), and an implementation, named DejaVu based on a BDD (Binary Decision Diagram) representation of sets of assignments of values to the free variables of subformulas. Instead of storing the values assigned to variables, we enumerate input values as soon as we see them and use Boolean encodings of this enumeration. We use BDDs to represent sets of such enumerations. For example, if the runtime verifier sees the input events $open(\text{“a”})$, $open(\text{“b”})$, $open(\text{“c”})$, it will encode them as 000, 001 and 010 (say, we use 3 bits b_0, b_1 and b_2 to represent each enumeration, with b_0 being the most significant bit). A BDD that represents the set of values $\{\text{“a”}, \text{“c”}\}$ would be equivalent to a Boolean function $(\neg b_0 \wedge \neg b_2)$ that returns 1 for 000 and 010 (the value of b_1 can be arbitrary). This approach has the following benefits:

The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the second author was partially funded by Israeli Science Foundation grant 2239/15: "Runtime Measuring and Checking of Cyber Physical Systems".

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

- It is highly compact. With k bits we can represent 2^k values. The BDD can grow up to a maximal number of $2^k + 1$ nodes; but BDDs usually compact the representation very well [9]. In fact, we are expected to pay very little for keeping surplus bits, as the BDD will compact away most of their effect. Thus, we can start with an overestimated number of bits k such that it is unlikely to see more than 2^k different values for the domain they represent. We can also incrementally extend the BDD with additional bits.
- Complementation (negation) is efficient, by just switching the 0 and 1 leaves of the BDD. Moreover, even though at any point we may have not seen the entire set of values that will show up during the execution, we can safely (and efficiently) perform complementation: values that have not appeared yet in the execution are being accounted for and their enumerations are reserved already in the BDD before these values appear.
- Our representation of sets of assignments as BDDs allows a very simple algorithm that naturally extends the dynamic programming monitoring algorithm for propositional past time temporal logic shown in [14].

We first define the semantics of a predicate linear temporal logic property for an assignment of values to its free variables after a given execution prefix. Then we redefine it as a function that returns the set of assignments satisfying the property at that prefix. There, we use the union and intersection set operators. For the final algorithm, we replace union and intersection by BDD disjunction and conjunction operators, respectively. We only have to keep values to represent the current and previous state in the execution. The remaining part of the paper is organized as follows. Section II discusses related work. Section III presents the syntax and semantics of the QTL temporal logic. Section IV presents the BDD-based algorithm for monitoring a trace against a QTL formula. Section V outlines the implementation, and Section VI presents an evaluation of the implementation. Finally, Section VII concludes the paper.

II. Related Work

There are several systems that allow monitoring temporal properties with data. The system closest to our presentation, in monitoring first order temporal logic is the MonPoly system [7]. As in the current work, it monitors first order temporal properties. In fact, it is also has the additional capabilities of asserting and checking properties that involve arithmetic relations among the data elements, progress of time, and a limited capability of reasoning about the future. The main difference between our system and MonPoly is in the way in which data are represented and manipulated. MonPoly exists in two versions. The first one models unbounded sets of values using regular expressions (see, e.g., [16] for a simple representation of sets of values). This version allows unrestricted complementation of sets of data values. Another version of MonPoly is based on representing finite sets of assignments. This is based on using algebraic database operators. For example, intersecting between two sets of assignments that are possibly over non identical sets of variables is done using the *join* operator. In that implementation complementation is restricted, to account for finite sets. Our system is based on representing sets of enumerations of data values as BDD functions, and does not restrict negation.

An important volume of work on data centric runtime verification is the set of systems based on trace slicing. These include TraceMatches [1], Mop [20], and Qea [21]. Trace slicing is based on the idea of mapping variable bindings to propositional automata relevant for those particular bindings. This results in very efficient monitoring algorithms, although with limitations w.r.t. expressiveness. Qea is an attempt to increase the expressiveness of the trace slicing approach. It is based on automata, as is the Orhcids system [11].

Other systems include BeepBeep [12] and TraceContract [5], which are based on future time temporal logic using formula rewriting. Very different kinds of specification formalisms can be found in systems such as Eagle [4], Ruler [6], LogFire [13] and Lola [3]. The system Mmt [10] represents assignments as constraints solved with an SMT solver. An encoding of BDD functions over enumerations of values appears in [22] in the context of datalog programs. However, that work does not deal with unbounded domains.

III. Syntax and Semantics

We define here the syntax and semantics for the QTL logic. Assume a finite set of domains $D_1, D_2, \ldots$. Assume further for now that the domains are infinite, e.g., they can be the integers or strings. (In Section IV it is explained how to deal with finite domains.) Let V be a finite set of *variables*, with typical instances x, y, z. In an *assignment*, each variable x can be assigned a value from its associated domain $domain(x)$, where multiple variables (or all of them) can be related to the same domain. For example $[x \rightarrow 5, y \rightarrow \text{“abc”}]$ is an assignment of the values 5 and “abc” to the variables x and y, respectively. Let T a set of *predicate names* with typical instances p, q, r. Each predicate name p is associated with some domain $domain(p)$. (Notice that *domain* is used both with a predicate name and with a variable.) A predicate is constructed from a predicate name and a variable or a constant of the same type. Thus, if the predicate name p and the variable x are associated with the domain of strings, we have predicates like $p(\text{“gaga”})$, $p(\text{“baba”})$ and $p(x)$. Similarly, if q and y are associated with the domain of integers, then we can have the predicates $q(3)$ and $q(y)$. We refer to predicates over constants as *ground predicates*. A *state* is a finite set of ground predicates, where each predicate name may appear at most once. An *execution* $\sigma = s_1 s_2 \ldots$ (observed at any time) is a finite sequence of *states*. For example, if $T = \{p, q, r\}$, then $\{p(\text{“xyzzy”}), q(3)\}$ is a possible state. Although during monitoring we always at any point in time only have observed a finite trace so far, the trace can grow unbounded, as the system being monitored keeps executing. In a monitoring context such as this, however, we will never observe an infinite trace.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Syntax. The formulas of the core QTL logic are defined by the following grammar, where a is a constant in *domain*(p). (For simplicity of the presentation, we define here the logic with unary predicates, but this is not due to any principle limitation, and, in fact, our implementation supports predicates with multiple arguments, including zero arguments, which correspond to propositions.)

$$\varphi ::= true \mid false \mid p(a) \mid p(x) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \mathcal{S} \varphi) \mid \ominus \varphi \mid \exists x\, \varphi \mid \forall x\, \varphi$$

At a given state the formula $p(\text{“a”})$ means that $p(\text{“a”})$ happened, more formally, that $p(\text{“a”})$ is among the ground predicates of the state. Consider now the formula $p(x)$, for a variable $x \in V$. We interpret it such that x is assigned any value “a” where $p(\text{“a”})$ appears in the current state. Thus, for interpreting $p(x) \wedge q(y)$ in a state that has the predicates $p(\text{“a”})$ and $q(3)$, we have the assignment $[x \mapsto \text{“a”}, y \mapsto 3]$. The formula $(\varphi_1 \mathcal{S} \varphi_2)$ (reads φ_1 *since* φ_2) means that φ_2 occurred in the past (including now) and since then (beyond that state) φ_1 has been true. This is the past dual of the common future time *until* modality [19]. The property $\ominus \varphi$ means that φ is true in the previous state. This is the past dual of the common future time *next* modality. We can also define the following additional temporal operators: $P\,\varphi = (true\, \mathcal{S}\, \varphi)$ (“previously”), and $H\varphi = \neg P \neg\varphi$ (“always in the past”). The operator $[\varphi_1, \varphi_2)$, borrowed from [17], has the same meaning as $(\neg\varphi_2 \mathcal{S} \varphi_1)$, but reads more naturally as an interval.

In the following we present the semantics of QTL, formulated in two alternative ways. First using predicates on variable assignments, and subsequently using sets of such assignments. In Section IV the algorithm is introduced which encodes such sets of assignments as BDDs.

Semantics. Let γ be an assignment to the variables that appear free in a formula φ. Then $(\gamma, \sigma, i) \models \varphi$ if φ holds for the prefix $s_1 s_2 \ldots s_i$ of the trace σ with the assignment γ. This is a standard definition, agreeing, e.g., with [7]. Note that by using past operators, the semantics is not affected by states s_j for $j > i$. Let *vars*(φ) be the set of free (i.e., unquantified) variables of a subformula φ. We denote by $\gamma|_{vars(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ. Let ε be an empty assignment. In any of the following cases, $(\gamma, \sigma, i) \models \varphi$ is defined when γ is an assignment over *vars*(φ), and $i \geq 1$.

- $(\varepsilon, \sigma, i) \models true$.
- $(\varepsilon, \sigma, i) \models p(a)$ if $p(a) \in \sigma[i]$.
- $([v \mapsto a], \sigma, i) \models p(v)$ if $p(a) \in \sigma[i]$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{vars(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{vars(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi \mathcal{S} \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{vars(\psi)}, \sigma, j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{vars(\varphi)}, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x\, \varphi$ if there exists $a \in domain(x)$ such that[1] $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

[1]$\gamma[x \mapsto a]$ is the overriding of γ with the binding $[x \mapsto a]$.

The definition of the *since* operator S can be simplified in a standard way such that it refers only to the positions i and $i-1$ in the sequence σ. This is based on the fact that according to the semantics of *since*, $(\varphi \mathcal{S} \psi) = (\psi \vee (\varphi \wedge \ominus(\varphi \mathcal{S} \psi)))$. This will serve in the implementation to work with only two versions of the sets of assignments, for the current and previous state:

- $(\gamma, \sigma, i) \models (\varphi \mathcal{S} \psi)$ if $(\gamma|_{vars(\psi)}, \sigma, i) \models \psi$ or $i > 1$, $(\gamma|_{vars(\varphi)}, \sigma, i) \models \varphi$, and $(\gamma, \sigma, i-1) \models (\varphi \mathcal{S} \psi)$.

The rest of the operators are defined as syntactic sugar using the operators defined in the above semantic definitions: $false = \neg true$, $\forall x\, \varphi = \neg\exists x \neg\varphi$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$.

Set Semantics. We now refine the semantics of the logic. Under the new definition, $I[\varphi, \sigma, i]$ is a function that returns a set of assignments such that $\gamma \in I[\varphi, \sigma, i]$ iff $(\gamma, \sigma, i) \models \varphi$. This redefinition will later lead to a simple implementation using BDDs, where each set of assignments will be represented as a BDD, and the Boolean operators will correspond directly to Boolean operators on BDDs.

In order to deal with subformulas with different sets of free variables (hence, different domains for assignments), we apply a projection and an extension operator to assignments over a subset of the variables. Let Γ be a set of assignments over the variables W, and $U \subseteq W$. Then $hide(\Gamma, U)$ (for “projecting *out*” or “hiding” the variables U) is the largest set of assignments over $W \setminus U$, each agreeing with some assignment of Γ on all the variables in $W \setminus U$. Let $U \cap W = \emptyset$, then $ext(\Gamma, U)$ is the largest set of assignments over $W \cup U$, where each such assignment agrees with some assignment in Γ on the values assigned to the variables W. This means that we extend Γ by adding arbitrary values to the variables in U from their domains. We have that $hide(ext(\Gamma, U), U) = \Gamma$. We define the union and intersection operators on sets of assignments, even if they are defined over non identical sets of variables. In this case, the assignments are extended over the union of the variables. Thus, if Γ is a set of assignments over W and Γ' is a set of assignments over W', then $\Gamma \bigcup \Gamma'$ is defined as $ext(\Gamma, W' \setminus W) \cup ext(\Gamma', W \setminus W')$ and $\Gamma \bigcap \Gamma'$ is $ext(\Gamma, W' \setminus W) \cap ext(\Gamma', W \setminus W')$. Hence, both are defined over the set of variables $W \cup W'$.

We denote by $A_{vars(\varphi)}$ the set of all possible assignments of values to the variables that appear free in φ. Thus, $I[\varphi, \sigma, i] \subseteq A_{vars(\varphi)}$. To simplify definitions, we add a dummy position 0 for sequence σ (which starts with s_1), where every formula is interpreted as an empty set. Observe that the value $\emptyset$ and $\{\varepsilon\}$, behave as the Boolean constants 0 and 1, respectively. The set semantics is defined as follows, where $i \geq 1$.

- $I[\varphi, \sigma, 0] = \emptyset$.
- $I[true, \sigma, i] = \{\varepsilon\}$.
- $I[p(a), \sigma, i] =$ if $p(a) \in \sigma[i]$ then $\{\varepsilon\}$ else $\emptyset$.
- $I[p(v), \sigma, i] = \{[v \mapsto a] | p(a) \in \sigma[i]\}$.
- $I[(\varphi \wedge \psi), \sigma, i] = I[\varphi, \sigma, i] \bigcap I[\psi, \sigma, i]$.
- $I[\neg\varphi, \sigma, i] = A_{vars(\varphi)} \setminus I[\varphi, \sigma, i]$.
- $I[(\varphi \mathcal{S} \psi), \sigma, i] = I[\psi, \sigma, i] \bigcup (I[\varphi, \sigma, i] \bigcap I[(\varphi \mathcal{S} \psi), \sigma, i-1])$.
- $I[\ominus\varphi, \sigma, i] = I[\varphi, \sigma, i-1]$.
- $I[\exists x\, \varphi, \sigma, i] = hide(I[\varphi, \sigma, i], \{x\})$.

As before, the interpretation for the rest of the operators can be obtained from the above using the connections between the operators. For example, $I[P\,\varphi,\sigma,i] = I[(true\,\mathcal{S}\,\varphi),\sigma,i]$. The correspondence between this set based semantics and the previous semantics, namely that $\gamma \in I[\varphi,\sigma,i]$ iff $(\gamma,\sigma,i) \models \varphi$ can be proved by a simple structural induction on the size of the formulas.

IV. An efficient Algorithm using BDDs

Representation of sets of assignments as BDDs

Our last refinement is to represent sets of assignments using Ordered Binary Decision Diagrams (OBDDs, although we write simply BDDs) [8]. A BDD is a compact representation for a Boolean tree representing a Boolean function. Because of compaction, however, the BDD forms a directed acyclic graph rather than a tree. Each internal node is marked with a Boolean variable. The left edge from a node represents that this variable has the Boolean value 0, while the right edge represents that it has the value 1. The nodes in the tree have the same order along all paths from the root, although some of the nodes may be missing, where the result of the Boolean function does not depend on the value of the corresponding variable. The leaves have the Boolean values 0 and 1. Thus, following a path in this graph, moving left or right corresponding to choosing 0s or 1s, respectively, leads to a leaf node that is marked by either a 0 or 1, representing the Boolean value returned by the function for the Boolean values on the path. The graph is compacted in such a way that isomorphic subtrees are "glued" together. Instead of keeping a node b with left or right edges that lead to the same subgraph, the node and its outgoing edges are removed from graph representation of the BDD. (previous edges point directly to a successor node). This means that for the Boolean values on the prefix of the path so far, the BDD value does not depend on the value of b. This compaction can be quite significant. BDDs have been instrumental in achieving a tremendous improvement in the size of systems that can be automatically verified [9].

When a new value of some domain D_i appears in a predicate in the current state, we add it to a list of values of that domain that were seen. In order to search efficiently if this value already appeared, in time linear with its representation, we can use e.g. a hash table. Thus, if we see p("ab"), p("de"), p("af") and q("fg") in subsequent states, where p and q are over the domain of strings, then we obtain a list of values ["ab","de","af","fg"].

Each new value that appears in the monitored sequence is enumerated as a binary number. We use BDDs to represent sets of values. The BDDs are over Boolean representations of *enumerations* of the observed values, according to the order in which they appear in the input, rather than a direct representation of the actual domain values. Thus, using two bits, "ab" can be represented as the bit string 00 (we start to enumerate from 00), "de" as 01, "af" as 10 and "fg" as 11. A BDD returns a 1 for each bit string representing an enumeration of a value in the set, and 0 otherwise. Then a BDD for a set containing the values "de" and "af" (2nd and 3rd values) will return 1 for 01 and 10. If the Boolean function is over b_0 (for most significant bit) and b_1 (for least significant), then this is the Boolean function $(\neg b_0 \wedge b_1) \vee (b_0 \wedge \neg b_1)$.

We can now represent *sets of assignments to variables* as required by our set semantics. We use a partition of the BDD bits according to the variables. Say, we want to represent a set S of assignments to the variables x and y, each expected to assume no more than 8 values. Then we can use the bits $y_0 y_1 y_2 x_0 x_1 x_2$, where x_0, x_1 and x_2 represent the enumerations of values of x, and y_0, y_1 and y_2 represent the enumerations of values of y. The BDD over these 6 bits will return 1 for each pair of enumerations that represent an assignment of values to x and y in the set S.

A subset of a set of k values can therefore be represented as function on $\lceil \log_2(k) \rceil$ bits. It can be represented as a Boolean tree of size $O(k)$. If we have m variables, $z^1, \ldots z^m$, where the number of values from the domain of the variable z^i is of size k_i, then we can represent any encoding of an assignments to the m variables with $\Sigma_{i=1..m} \lceil \log_2(k_i) \rceil$ bits. With this number of bits, the BDD graph can grow up to size $O(\Pi_{i=1..m} k_i)$. However, representing this function as a BDD can often be quite more compact.

The Algorithm

Given some value a observed in the trace as an argument to a ground predicate, let **lookup**(a) return a bit string that represents the occurrence order of appearance of a (among other values of the same domain) in the trace. Thus, if a is the first value occurring for that domain, **lookup**(a) will return $00\ldots00$. If it is the 2nd value occurring, $00\ldots01$, and so forth. We update this representation for each new state that appears.

We use a function called **build**(x,a) for building a BDD function that represents an assignment of a to the variable x, independent of the other variables. For example, if **lookup**$(a) = 011$, assuming we use only 3 bits, b_0, b_1 and b_2 to represent values, then **build**(x,a) will obtain a BDD representation of the function $\neg b_0 \wedge b_1 \wedge b_2$. There may be other bits, representing other variables, but the BDD function is independent of them (which leads to a large compaction).

Union and intersection of sets of assignments are translated simply to disjunction and conjunction of their BDDs representation, respectively, and complementation becomes negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential (universal, respectively) operators, as in the interpretation of $\exists x\, \varphi$, we use the BDD existential (universal, respectively) operator over the bits that represent (the enumeration of) values of x. Thus, we translate $\exists x$, where x is represented using the bits $x_0, x_1, \ldots x_{k-1}$ into $\exists x_0 \ldots \exists x_{k-1}$. We use the following BDD function to perform existential quantification over bits: **exists**$(\langle x_0, \ldots, x_{k-1} \rangle, bdd)$. Finally, BDD$(0)$ and BDD(1) are the BDDs that return always 0 or 1, respectively.

The algorithm uses these standard BDD operators, and is almost a direct translation of the semantics using sets of assignments. The structure of the algorithm is similar to that of [14]. Namely, there are only two vectors (arrays) of values indexed by subformulas: for the current state (now) and for the previous state (pre). However, while in [14] the vectors contain Boolean values, here the values are BDD functions. The algorithm follows.

1) Initially, for each subformula φ, $\mathsf{now}(\varphi) = \text{BDD}(0)$.
2) Observe a new state (as set of ground predicates) s as input.
3) Let $\mathsf{pre} := \mathsf{now}$.
4) Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) = \text{BDD}(1)$
   - $\mathsf{now}(p(a)) = \text{if } p(a) \in s \text{ then BDD}(1) \text{ else BDD}(0)$
   - $\mathsf{now}(p(x)) = \text{ if } \exists a\, p(a) \in s \text{ then } \mathbf{build}(x,a) \text{ else BDD}(0)$
   - $\mathsf{now}((\varphi \wedge \psi)) = \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$
   - $\mathsf{now}(\neg\varphi) = \mathbf{not}(\mathsf{now}(\varphi))$
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) = \mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\ominus\, \varphi) = \mathsf{pre}(\varphi)$
   - $\mathsf{now}(\exists x\, \varphi) = \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathsf{now}(\varphi))$
5) Goto step 2.

We can define the number of bits per domain to a large enough number k such that we anticipate no more than 2^k different values. For example, if $k = 20$, this will allow more than a million different values. In fact, large part of the BDD that is related to bits that are not used is mostly compacted away. To see this, recall that the BDD functions, obtained during the runtime verification for representing the sets of assignments for the subformulas, are functions from the enumeration of values, according to the order in which they appear in the input. We start enumerating from $00\ldots00$, and then continue with $00\ldots01$, $00\ldots10$, etc. The Boolean operators, including the negation and applying quantification, maintain invariantly, that as long as only $m < 2^k$ values appeared, then the values for the binary representation of the $m+1st$ to the $2^k th$ enumerations of values are the same (for any combination of values of the other variables). In fact, it can be shown by induction on the length of temporal formulas and the input sequence that these enumerations, and in particular the enumeration $11\ldots11$, correctly represent all the values that were not seen so far in the input[2].

Suppose now that only $l < k$ bits are needed for storing the current set of enumerations, where the other (most significant bits) are 0. Then the maximal enumeration that is assigned to input values is no larger than binary $00\ldots0011\ldots11$, with $k-l$ times 0s, and l times 1s. The BDD function will return the same Boolean values for larger enumerations of the same domain (these will be enumerations that have at least a single 1 in the most significant $k-l$ digits). This by itself leads to a significant compaction.

It is important to maintain that for infinite domains there is at least one such unused enumeration for the allocated k bits. In particular, the largest possible enumeration $11\ldots11$ would play this role (but as discussed above, possibly also some smaller enumerations). To see why this is important, consider the case where *all* 2^k enumerations are used (i.e., they were seen in the execution so far) for predicate $g(x)$. Then $P g(x)$ will be represented as BDD(1), returning constantly a 1. Thus, $\neg P g(x)$ will be calculated to BDD(0), returning constantly 0. Now, $\exists x \neg P g(x)$ will be translated into $\exists x_0 \exists x_1 \ldots \exists x_{k-1}$BDD(0), and will return BDD(0) (*false*). However, checking $\exists x \neg P g(x)$ should have returned BDD(1) (*true*), since it claims that there are values that did not occur so far within a g predicate; indeed, for an infinite domain we could have never seen all the possible values during a finite execution.

Dynamic expansion of the BDDs. In case we did not allocate in advance enough bits, it is possible to extend the number of bits we use for representing values for a variable. As explained above, the enumeration $11\ldots11$ of length k represents for every variable "all the values not seen so far in input the sequence". Consider the following two cases:

- When the added (most significant) bit has the value 0, the enumeration still represents the same value. Thus, the updated BDD needs to return the same values that the original BDD returned without the additional 0.
- When the added bit has the value 1, we obtain enumerations for values that were not seen so far in the input. Thus, the updated BDD needs to return the same values that the original BDD gave to $11\ldots11$.

Suppose we have three variables, x, y and z, represented using 3 bits each, i.e., x_0, x_1, x_2, y_0, y_1, y_2, z_0, z_1, z_2, and we want to add a new most significant bit y_{new} for representing y. Let B be the BDD before the expansion. The case where the value of y_{new} is 0 is the same as for a single variable. For the case where y_{new} is 1, the new BDD needs to represent a function that behaves like B when all the y bits are set to 1. Denote this by $B[y_0 \setminus 1, y_1 \setminus 1, y_2 \setminus 1]$. This function returns the same Boolean values independent of any value of the y bits, but it may depend on the other bits, representing the x and z variables. Thus, to expand the BDD, we generate a new one as follows:

$$((B \wedge \neg y_{new}) \vee (B[y_0 \setminus 1, y_1 \setminus 1, y_2 \setminus 1] \wedge y_{new}))$$

The generalization of this formula to any number of variables is clear.

Finite domains. We now show how to deal with the case of variables that are defined over finite domains. Say we have a BDD over enumerations of variables x, y and z, where y has a domain of size m. Then we need $k = \lceil \log_2(m) \rceil$ bits, $y_0, \ldots y_{k-1}$, for representing y. We need to relativize the use of existential quantifier to m. We can encode a fixed BDD function $\mathit{smaller}(y,t)$ that expresses that the bits that

[2]Formally, let ψ be a subformula, for which a BDD B_ψ was constructed so far. Then B_ψ will return 1 for exactly the following bit strings. Let γ be some assignment satisfying ψ after the current input. Construct the following concatenation of bit strings, according to the given order on variables: for each variable, if its value under γ has appeared in the input, concatenate its binary enumeration, otherwise, concatenate some enumeration larger than the number of its domain values seen so far.

represent y have a binary value that is *smaller* than the binary representation of t. (Note that we start enumerating from $00\ldots00$, hence we check *smaller* rather than *smaller or equal*). For example, if we use 2 bits y_0 and y_1 then $smaller(y,3) = \neg(y_0 \wedge y_1)$, as any binary number smaller than 3 will have at least $y_0 = 0$ or $y_1 = 0$. Now, we need to replace each subformula of the form $\exists x\,\varphi$ where x appears free in φ by $\exists x\,(smaller(x,m) \wedge \varphi)$. This limits the quantification on the bits that represent x to values that are in the finite domain with m values. We implement universal quantification $\forall x\,\varphi$ using negation (twice) and existential quantification; effectively, it is translated into $\forall x\,(smaller(x,m) \rightarrow \varphi)$.

Quantifying over values seen so far. We can also extend our logic with a construct $seen(y)$ for a variable y. This construct will be translated, in a similar way as explained in the previous paragraph, into a BDD that encodes the binary values of the bits representing y that are no bigger than the maximal enumeration used (seen) so far for the domain of y. We saw earlier that $\exists x\,\neg P\,g(x)$ should always return a *true* in an infinite domain, as it says that there is a value in the domain of x that did not appear within a g predicate name. However, we may intend to mean that the existential quantification is restricted to be only over the values that were seen. In this case, we can write $\exists x\,(seen(x) \wedge \neg P\,g(x))$. This can be *true* if there is some value in the domain of x that appeared in the execution so far within a predicate name other than g, but not within g.

Comparing variables. Another important extension is to be able to compare different variables, i.e., $x = y$ (or $x \neq y$). This can also be coded as a fixed BDD over the bit representation of enumerations of values of x and y. This is encoded as a BDD representing $x_0 = y_0 \wedge x_1 = y_1 \wedge \ldots x_{k-1} = y_{k-1}$.

V. Implementation

We implemented a monitoring tool for the QTL logic, called DejaVu. Let $\mathbb{E}$ be the type of n-ary predicate symbols, and the $\mathbb{B}$ be the type of Boolean values. The implementation of the monitoring algorithm presented in Section IV consists of a program $translate : Spec \rightarrow (\mathbb{E}^* \rightarrow \mathbb{B}^*)$, which, when provided a specification generates a *monitor* program; the monitor takes as input a trace (a sequence of events), and returns a verdict, effectively a Boolean value for each event in the trace. In the following we outline the format of the generated monitor program. The tool is implemented in Scala, using the standard approach where a parser parses the specification and produces an abstract syntax tree, which is then traversed and translated into the monitor program. The parser is written using Scala parser combinators. The generated monitor program uses the JavaBDD package [15] for generating and operating BDDs. Log files in CSV format are parsed using the Apache Commons CSV (Comma Separated Value format) parser. The tool can be used for online (observing a program as it executes) as well as offline (analyzing log files) monitoring. We shall illustrate the monitor generation using an example. Consider the following variation of the first property from Section I (using syntax supported by the implementation):

```
prop p: forall f . close(f) → exists m . P open(f,m)
```

It states that if a file f is closed, it should have been opened in the past with some access mode (*read*, *write*, ...). The generated monitor relies on an enumeration of the subformulas of the original formula in order to evaluate the subformulas bottom up for each new event. Figure 1 (bottom) shows the decomposition of the original formula into subformulas (an Abstract Syntax Tree - AST), indexed by numbers from 0 to 5, satisfying the invariant that if a formula φ_1 is a subformula of a formula φ_2 then φ_1's index is bigger than φ_2's index. The monitor generated from the property is shown in Figure 1 (top). Specifically two arrays are declared, indexed by subformula indexes: pre for the previous state and now for the current state. A BDD here represents a predicate on bit strings, effectively representing a set of bit strings (those for which the BDD evaluates to true). Actual values in the trace are uniquely mapped to such bit strings, and the BDD therefore indirectly represents a set (set membership function) of the actual values.

In each step the evaluate function re-computes the now array from highest to lowest index, and returns true (ok) iff

```
class Formula_p extends Formula {
  var pre: Array[BDD] = Array. fill (6)(False)
  var now: Array[BDD] = Array. fill (6)(False)
  var tmp: Array[BDD] = null
  val var_f :: var_m :: Nil = declareVariables("f", "m")

  override def evaluate(): Boolean = {
    now(5) = build("open")(V("f"),V("m"))
    now(4) = now(5).or(pre(4))
    now(3) = now(4).exist(var_m)
    now(2) = build("close")(V("f"))
    now(1) = now(2).not().or(now(3))
    now(0) = now(1).forAll(var_f)
    tmp = now; now = pre; pre = tmp
    !tmp(0).isZero
  }
}
```

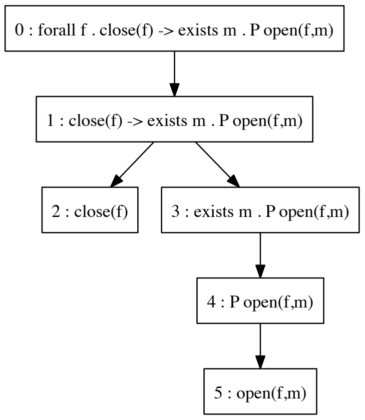

0 : forall f . close(f) -> exists m . P open(f,m)
1 : close(f) -> exists m . P open(f,m)
2 : close(f)
3 : exists m . P open(f,m)
4 : P open(f,m)
5 : open(f,m)

Fig. 1: Monitor (top) and AST (bottom) for the property

now(0) is not the zero-BDD. Assume for example that an event *close*(*out*) is observed. At the leaf node 2 representing the close(f) event, the function call build("close")(V("f")) builds a new BDD for *out* unless one has previously been computed, in which case that is used. At composite subformula nodes, BDD operators are applied. For example for subformula 4, the new value is now(5).or(pre(4)), which is the interpretation of the formula **P** open(f,m). Quantification is solved by performing quantification over the relevant bits of the BDD corresponding to the variable in question.

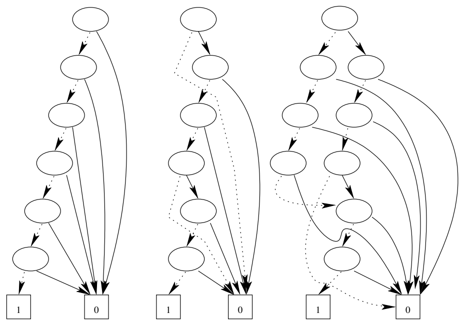


Fig. 2: BDDs for (left) subformula 5 on the first event, (mid) subformula 5 on the second event, (right) subformula 4 on the second event

Example

Assume that each variable f and m is represented by three bits. Consider the input trace, consisting of three events: $\langle open(input, read), open(output, write), close(out)\rangle$. When the monitor processes the first event for subformula number 5 (all subformulas here are numbered according ot Figure 1), it will create a bit string composed of a bit string for each parameter f and m. As previously explained, bit strings for each variable are allocated in increasing order: 000, 001, 010,..., hence the first bit string representing the assignment [f↦*input*,m↦*read*] becomes 000000 where the three rightmost bits represent *input* and the three leftmost bits represent *read*. Figure 2 (left) shows the corresponding BDD. (Note that most significant bits are implemented lower in the BDD.) For each bit (node) in the BDD, the dotted arrow corresponds to this bit being 0 and the full drawn arrow corresponds to this bit being 1. In this BDD all bits have to be zero in order to be accepted by the function represented by the BDD.

Upon the second event, new values $(output, write)$ are observed as argument to the *open* event. Hence a new bit string for each variable f and m is allocated, in both cases 001 (the next unused bit string). The new combined bit string for the assignments satisfying subformula 5 then becomes 001001, again forming a BDD representing a single assignment, appearing in Figure 2 (mid). Subformula 4 now becomes the union of the two BDDs, resulting in the BDD on Figure 2 (right). The existential quantification in subformula 3 causes the BDD to be reduced to only the first 3 bits. However since subformula 2 is still *false* the whole formula evaluates to *true*.

```
prop access : forall u . forall f .
  access(u,f) → [login(u),logout(u)) & [open(f),close(f))

prop file : forall f .
  close(f) → exists m . @ [open(f,m),close(f))

prop fifo : forall x .
  (enter(x) → ! @ P enter(x)) &
  (exit(x) → ! @ P exit(x)) &
  (exit(x) → @ P enter(x)) &
  (forall y . (exit(y) & P (enter(y) & @ P enter(x))) →
    @ P exit(x))
```

Fig. 3: Evaluation properties in QTL

Finally, on observation of the third and last event *close*(*out*), a new value *out* for f is observed, and allocated the bit pattern 010, represented by the corresponding BDD for subformula 2. As end result, for the formula in node 1 we end up with a BDD that is neither constantly *true* nor constantly *false*, and hence universally quantifying over it yields *false* since it is not the case that for all bit assignments it yields *true*.

VI. Evaluation

DejaVu performance is evaluated by comparing against MonPoly, the tool that seems to have most similarities to DejaVu as previously discussed. We specifically evaluated the three temporal properties shown in Figure 3, on different sizes of traces, while varying the number of bits allocated to represent variables in BDDs. The properties were encoded in MonPoly in a 1-1 manner. The Access property states that if a file f is accessed by a user u, then the user should have logged in and not yet logged out, and the file should have been opened and not yet closed. The File property states that if a file is closed, then it must have been opened (and not yet closed) with some mode m (e.g. read or write). Finally, the Fifo property is a conjunction of four subformulas about data entering and exiting a queue. The first two subformulas state that a datum can at most enter and exit once. The last subformula states the Fifo principle of queues.

TABLE I: Evaluation of DejaVu and MonPoly

Property	Trace length	MonPoly (sec)	DejaVu (sec) bits per var.: 20 (40, 60)
Access	11,006	**1.9**	**3.1** (3.3, 3.2)
	110,006	**241.9**	**6.1** (9.1, 10.9)
	1,100,006	**58,455.8**	**36.8** (61.9, 88.8)
File	11,004	**61.1**	**2.8** (2.8, 3.0)
	110,004	**7,348.7**	**6.3** (6.5, 8.6)
	1,100,004	**DNF**	**30.3** (43.9, 59.5)
Fifo	5,051	**158.3**	**195.4** (OOM, -)
	10,101	**1140.0**	**ERR** (-, -)

Table I shows the result of the evaluation, which was performed in the Mac OS X 10.7.5 operating system on a 2 × 2.93 GHz 6-Core Intel Xeon with 32 GB of memory. The properties were evaluated with each tool on traces of sizes spanning from (approximately) 5 thousand to 1 million events (see table for exact numbers). Traces have the general form that initially numerous opening events (login, open, enter) occur, in order to accumulate a large amount of data stored in the monitor, after which a smaller number of corresponding closing events (logout, close, exit) occur. In addition, for each trace we experimented with three different sizes of bit vectors: 20, 40 and 60 bits, corresponding to the ability to store respectively approximately a million, a trillion, and a quintillion different values for each variable (the latter two are not needed for these traces). The following abbreviations are used: DNF = Did Not Finish (during 16 hours), OOM = Out of Memory, and ERR = an error occurred, in this case an array index out of bound problem in the JavaBDD package. The important numbers to compare are in bold font.

Table I demonstrates clearly that w.r.t. the first two properties, DEJAVU outperforms MONPOLY by a factor up to 3000. Those are substantial differences, and demonstrates that BDDs may be an interesting way to refer to stored data. However, for the last FIFO property, the two systems are somewhat comparable, although MONPOLY seems to do better on this particular property. The complexity lies in the last of the four subformulas in the conjunction, the actual FIFO property. Increasing the number of bits allocated per variable, from 20 to 40 and 60, does not seem to have a substantial impact on the performance, except for the FIFO property, where it causes an OOM result for 40 bits.

VII. CONCLUSION

We described a BDD based runtime verification algorithm for checking the execution of a system against a first order past time temporal logic property. The challenge is to provide a compact representation that will grow slowly and can be updated quickly with each incremental calculation that is performed per each new monitored event, even for very long executions.

We used a BDD representation of sets of assignments for the variables that appear (free) in the monitored property. Each value observed in the trace is represented by a BDD encoding the value's enumeration in appearance order. While the size of the BDD can grow linearly with the number of represented values, it is often much more compact, and the BDD functions of a standard BDD package are optimized for speed. Our representation allows assigning a redundantly large number of bits for representing the encoding of values, so that even extremely long executions can be monitorable. For example, if the encoding for each variable uses 64 bits, the BDD can hold up to 2^{64} different values for each variable. Redundant bits, used pessimistically for representing encodings in the expectation that the number of values encountered during the execution will grow considerably, do not cause a large explosion in the size of the BDD. Alternatively, we showed how to dynamically expand the BDD when the number of values exhausts the allocated size.

Our experiments provide an optimistic view on the benefit of using BDDs. The implementation was written in SCALA, an object-oriented and functional programming language with active garbage collection. We expect that using an iterative programming language such as C will result an even quicker runtime verification monitor. A limitation of our approach is that the encoding of assignment sets does not blend well with using various relations between values. While we can easily compare variables to be equal or not equal, we are not capable of comparing, say, whether one value is smaller than another value in an efficient way. This remains a challenge for future work.

REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, OOPSLA 2005, 345-364.
[2] B. Alpern, F. B. Schneider, Recognizing Safety and Liveness. Distributed Computing 2(3), 117-126, 1987.
[3] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna: LOLA: Runtime Monitoring of Synchronous Systems, TIME 2005, 166-174.
[4] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-Based Runtime Verification, VMCAI, LNCS Volume 2937, Springer, 2004.
[5] H. Barringer, K. Havelund, TraceContract: A Scala DSL for Trace Analysis, Proc. of the 17th International Symposium on Formal Methods (FM'11), LNCS Volume 6664, Springer, 2011.
[6] H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: from Eagle to RuleR, Proc. of the 7th Int. Workshop on Runtime Verification (RV'07), LNCS Volume 4839, Springer, 2007.
[7] D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 45, 2015
[8] R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318 (1992).
[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, LICS 1990, 428-439.
[10] N. Decker, M. Leucker, D. Thoma, Monitoring modulo theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 2016.
[11] J. Goubault-Larrecq, J. Olivain, A Smell of ORCHIDS, Proc. of the 8th Int. Workshop on Runtime Verification (RV'08), LNCS Volume 5289, Springer, 2008.
[12] S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
[13] K. Havelund, Rule-based runtime verification revisited, Journal of Software Tools for Technology Transfer, Volume 17 Number 2, Springer, 2015.
[14] K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS 2002, 342-356.
[15] JavaBDD, http://javabdd.sourceforge.net.
[16] J. G. Henriksen, J. L. Jensen, M. E. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, A. Sandholm, Mona: Monadic Second-Order Logic in Practice, TACAS 1995, 89-110.
[17] M. Kim, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: a Run-time Assurance Tool for Java, Proc. of the 1st Int. Workshop on Runtime Verification (RV'01), Elsevier, ENTCS 55(2), 2001.
[18] O. Kupferman, M. Y. Vardi, Model Checking of Safety Properties, Formal Methods in System Design 19(3): 291-314, 2001.
[19] Z. Manna, A. Pnueli, Completing the Temporal Picture, Theoretical Computer Science 83, 91-130, 1991.
[20] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, J. Software Tools for Technology Transfer, Springer, 2011.
[21] G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at Runtime with QEA, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), Springer, 2015.
[22] J. Whaley, D. Avots, M. Carbin, M. S. Lam, Using Datalog with Binary Decision Diagrams for Program Analysis, APLAS 2005, 97-118.

Factored Boolean Functional Synthesis

Lucas M. Tabajara
Department of Computer Science, Rice University
Houston, USA
lucasmt@rice.edu

Moshe Y. Vardi
Department of Computer Science, Rice University
Houston, USA
vardi@cs.rice.edu

***Abstract*—Boolean functional synthesis allows the automated construction of Boolean functions from declarative specifications. BDD-based techniques for this problem can be very efficient when the specification can be compactly represented by a BDD, but this is not always possible. In model checking, a way around this problem has been found by using factored representations, where formulas are represented as a conjunction of subformulas, each encoded individually as a BDD. We show how techniques and heuristics for quantifier elimination on factored formulas can also be lifted to perform synthesis, and show that this approach allows the synthesis of many problem instances that are intractable when represented by a single BDD. We compare our approach to other tools for Boolean synthesis that are not BDD-based. Our empirical evaluation shows that, while no approach dominates across the board, our tool outperforms other tools on several problem instances.**

***Index Terms*—Binary Decision Diagrams, Boolean synthesis, factored representation**

I. Introduction

The problem of synthesizing Boolean functions from relations is central to many areas of formal methods. Boolean functions can represent both logical circuits and programs over finite data types, and Boolean synthesis is also an essential component of synthesis from temporal specifications [1].

In the Boolean synthesis problem, we are given as specification a Boolean formula $f(\vec{x}, \vec{y})$, where $\vec{x}$ is a vector of input variables $x_1, \ldots, x_m$ and $\vec{y}$ is a vector of output variables $y_1, \ldots, y_n$. Our goal is twofold: first, to characterize the set of valid inputs by a formula $p(\vec{x})$ that is satisfied exactly by those inputs for which there is an output that satisfies f; second, to construct a Boolean function $g : \mathbb{B}^m \to \mathbb{B}^n$ such that $p(\vec{x}) \Rightarrow f(\vec{x}, g(\vec{x}))$. In other words, for every input $\vec{x}$ that satisfies $p(\vec{x})$, setting $\vec{y} = g(\vec{x})$ satisfies $f(\vec{x}, \vec{y})$.

Early approaches to this problem have used techniques based on Binary Decision Diagrams (BDDs) [2], [3]. The efficiency of BDDs, however, is highly dependent on finding a good variable ordering, which is a hard optimization problem. Furthermore, it is well-known that there are interesting Boolean formulas that cannot be represented by a polynomial-sized BDD. Because of this, more recent works have avoided BDDs in favor of approaches using SAT solvers [4].

Nevertheless, BDD-based techniques have been shown to be very competitive when the specification can be efficiently represented by a BDD and a good variable ordering is known [5]. This raises the question of whether it is possible, instead of discarding BDDs, to find a way to employ BDD-based techniques even in cases when a BDD for the specification cannot be constructed. In other applications where the use of BDDs is common, the solution to this problem came in the form of *factored representations* of formulas, which allow a much wider range of instances to be effectively computed [6].

Factored representations are based on the fact that it is common for Boolean formulas of practical importance to be represented by conjunctions of constraints. In other words, a Boolean formula $f(\vec{x}, \vec{y})$ might be written in the format $f_1(\vec{x}, \vec{y}) \wedge \ldots \wedge f_k(\vec{x}, \vec{y})$. In this case, rather than constructing a single *monolithic* BDD B for f, we can instead represent the formula as a collection of BDDs $B_1, \ldots, B_k$ for each of the *factors* $f_1, \ldots, f_k$, implicitly interpreted as a conjunction. Since conjoining multiple BDDs can lead to a combinatorial explosion, the factored representation is usually significantly more compact.

In symbolic model checking, where the idea of factored BDD representations originated, this approach was able to reduce the size of representations of transition relations by an order of magnitude [6]. Since then, different techniques have been developed for further improving performance, including heuristics for clustering and reordering factors [7]. Similar techniques have been used for processing factored formulas in the context of symbolic satisfiability [8]. In this approach to the satisfiability problem, a CNF formula is encoded by partitioning the set of clauses and representing each partition as a BDD. Then, symbolic quantifier elimination is used to find if there is a satisfying assignment to the formula. In this paper we show how techniques and heuristics used in these applications can be adapted to perform synthesis from factored specifications.

Other approaches have been developed for synthesis of factored formulas that do not employ BDDs. A recent work [9] uses And-Inverter Graphs (AIGs) for representing Boolean formulas and a counterexample-guided abstraction refinement (CEGAR) loop for synthesizing the function. A downside to this approach is that the CEGAR loop requires repeated calls to a SAT solver, which can have a high cost in running time. Furthermore, BDDs can be very compact for small formulas, which poses the question of whether they can produce smaller functions than AIGs when using factored representation.

A different synthesis approach is based on the close relation between Boolean synthesis and QBF solving. The CNF formulas given as input to QBF solvers are special cases of factored formulas, and a number of modern solvers are capable of computing Skolem functions for the existential variables

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

in terms of the universal variables [10] [11]. Therefore, by writing a specification $f(\vec{x}, \vec{y})$ in CNF, the synthesis problem can be encoded as a QBF $\forall\vec{x}.\exists\vec{y}.f(\vec{x}, \vec{y})$.

Although QBF solvers can be very efficient in solving these formulas, they are able to synthesize Skolem functions only when the QBF $\forall\vec{x}.\exists\vec{y}.f(\vec{x}, \vec{y})$ evaluates to *true*. This corresponds to the case when the specification $f(\vec{x}, \vec{y})$ is *realizable*, that is, when every input $\vec{x}$ has an output $\vec{y}$ that satisfies f, and consequently $p(\vec{x}) \equiv 1$. In many applications of Boolean synthesis we are interested, however, in unrealizable specifications as well. One such a case is LTL$_f$ synthesis using DFA games [1], in which a winning strategy might not exist for every state of the automaton, but we would like to synthesize this strategy for all states for which it exists.

In our experimental evaluation, we first compare our implementation using factored representation with the monolithic approach, allowing us to confirm that indeed factoring the specification allows us to synthesize a number of instances that would be otherwise intractable. We then compare our implementation using BDDs to two other tools: CEGARSKOLEM [9] [12], which uses the CEGAR-based approach, and the QBF solver CADET [11]. The results show that no approach is universally better, and every tool outperforms the others in some subset of the benchmarks. Although the QBF approach has a clear advantage for realizable specifications, being unable to handle unrealizable instances limits its applicability in a number of practical cases.

Beyond performance, an advantage of using BDDs is that this makes the approach easier to integrate in temporal synthesis applications, such as [1]. This is because such applications usually employ some kind of fixpoint computation, for which BDDs are particularly suited due to the ease of checking if two BDDs are equivalent. Using other representations for Boolean formulas, such as AIGs or CNF, it becomes harder to perform such computations.

II. PRELIMINARIES

A. Boolean Formulas and Functions

We denote by $\mathbb{B} = \{0, 1\}$ the set of Boolean values. We use the notation $\vec{x}$ to represent a Boolean vector $(x_1, \ldots, x_m) \in \mathbb{B}^m$, for some m. We identify a Boolean formula f over Boolean variables $x_1, \ldots, x_m$ with the Boolean function $f : \mathbb{B}^m \to \mathbb{B}$ such that $f(\vec{x}) = 1$ if and only if $\vec{x}$ is a satisfying assignment of formula f.

We use $\neg$, $\wedge$, $\vee$ to denote the usual Boolean operators of negation, conjunction and disjunction, and $\equiv$ to denote logical equivalence of two Boolean formulas. Given two formulas $f(x_1, \ldots, x_m)$ and $f'(y_1, \ldots, y_n)$, we use $f[x_i \mapsto f']$ to denote the formula $f(x_1, \ldots, x_{i-1}, f'(y_1, \ldots, y_n), x_{i+1}, \ldots, x_m)$. We say that a variable x_i is in the *support* of a formula f if x_i determines the value of f, that is, $f[x_i \mapsto 0] \not\equiv f[x_i \mapsto 1]$.

We also use $\forall$ and $\exists$ to denote universal and existential quantification over Boolean variables. Given a quantified Boolean formula, we can use the following lemma to obtain a logically equivalent quantifier-free formula:

Lemma 1 (Self-Substitution [5])**.** *Let $f(\vec{x}, y)$ be a Boolean formula over variables $\vec{x} = (x_1, \ldots, x_m)$ and y. Then*

- $\forall y.f(\vec{x}, y) \equiv f(\vec{x}, f(\vec{x}, 0))$
- $\exists y.f(\vec{x}, y) \equiv f(\vec{x}, f(\vec{x}, 1))$

Given a formula $f(\vec{x}, y)$ with $\vec{x} = (x_1, \ldots, x_m)$, and a function $g : \mathbb{B}^m \to \mathbb{B}$, if $f(\vec{x}, g(\vec{x})) \equiv \exists y.f(\vec{x}, y)$, we say that g is a *witness* for y in f. It is clear from Lemma 1 that, for every formula $f(\vec{x}, y)$, $f(\vec{x}, 1)$ is a witness for y.

B. Boolean Synthesis

We use the following formulation of the Boolean synthesis problem:

Problem 1 (Boolean Synthesis)**.** *Given a Boolean formula $f(\vec{x}, \vec{y})$ where $\vec{x} = (x_1, \ldots, x_m)$ and $\vec{y} = (y_1, \ldots, y_n)$, called the* specification*, compute a Boolean formula $p(\vec{x})$, called the* precondition*, and a Boolean function $g(\vec{x}) : \mathbb{B}^m \to \mathbb{B}^n$, called the* implementation*, such that $\exists\vec{y}.f(\vec{x}, \vec{y}) \equiv p(\vec{x}) \equiv f(\vec{x}, g(\vec{x}))$.*

In this context, we call $\vec{x}$ the *input variables* and $\vec{y}$ the *output variables*. Intuitively, f specifies a relation between inputs and outputs of the desired Boolean function g, and p identifies valid inputs for g. If there is an output $\vec{y}$ that satisfies f for input $\vec{x}$, then a) $p(\vec{x})$ is true, and b) $\vec{y} = g(\vec{x})$ satisfies f. The implementation g can be represented by a sequence of functions $\langle g_1, \ldots, g_n\rangle$, $g_i : \mathbb{B}^m \to \mathbb{B}$. In this work we focus on specifications of the form $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge \ldots \wedge f_k(\vec{x}, \vec{y})$.

In the context of Boolean synthesis, we say that a specification $f(\vec{x}, \vec{y})$ with input variables $\vec{x}$ and output variables $\vec{y}$ is *realizable* if for every assignment to $\vec{x}$ there exists an assignment $\vec{y}$ such that $f(\vec{x}, \vec{y})$ evaluates to *true*. In other words, if a specification f is realizable then $\exists\vec{y}.f(\vec{x}, \vec{y}) \equiv p(\vec{x}) \equiv f(\vec{x}, g(\vec{x})) \equiv 1$. In this case, every input $\vec{x}$ is a valid input for $g(\vec{x})$.

C. Binary Decision Diagrams

A *[Reduced Ordered] Binary Decision Diagram*, or BDD, is a data structure that represents a Boolean function as a directed acyclic graph [13]. BDDs can be seen as a reduced representation of a binary decision tree of a Boolean function. We require that variables are ordered the same way along every path of the BDD ("ordered") and that the BDD is minimized to eliminate duplication ("reduced"). For a given variable order, the reduced BDD is *canonical*. The variable order used can have a major impact on BDD size, and two BDDs representing the same function but with different orders can have an exponential difference in size. Since BDDs represent Boolean functions, they can be manipulated using standard Boolean operations. We overload the notation of the operators $\neg$, $\wedge$, $\vee$ and functional composition (e.g. $B[x_i \mapsto B']$) with equivalent semantics to their counterparts for Boolean formulas.

III. SYNTHESIS FROM FACTORED FORMULAS

In this section, we start by formally defining the notion of factored representations and present some of their properties.

We then describe a method for performing synthesis over factored representations.

A. Factored Representation of Boolean Formulas

Let the specification f for an instance of the Boolean synthesis problem be of the form $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge f_2(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$. Each formula f_i is called a *factor* of f. The sequence of BDDs $\langle B_1, B_2, \ldots, B_k\rangle$, where B_i is the BDD encoding of f_i, is called the *factored representation* of f. In contrast, the representation of f as a single BDD B is called the *monolithic representation*.

Note that it is possible for a formula to have an exponential monolithic representation and a polynomial factored representation. In particular, the factored representation of a formula in CNF can always be linear, since the BDD of a single clause is linear in size.

Although factored representations can be exponentially more compact than monolithic representations, they introduce complications into the synthesis procedure. To understand why, first note from the definition of Boolean synthesis that there is a close connection between Boolean synthesis and quantifier elimination. In fact, substituting the implementation $g(\vec{x})$ in the specification $f(\vec{x},\vec{y})$ is equivalent to existentially quantifying $\vec{y}$, and the precondition $p(\vec{x})$ is exactly the result of this quantification. Then, recall that existential quantifiers do not distribute over conjunction. That is, in general $\exists y_1,\ldots,y_n.\bigwedge_{i=1}^{k} f_i(\vec{x},\vec{y}) \not\equiv \bigwedge_{i=1}^{k} \exists y_1,\ldots,y_n.f_i(\vec{x},\vec{y})$.

More precisely, as pointed out in [9], the right-hand side is an over-approximation of the left-hand side, meaning that every assignment of $\vec{x}$ that satisfies the left-hand side satisfies the right-hand side, but not vice-versa.

As a consequence, if we are given a factored representation of a Boolean formula, it is not clear how to perform existential quantifier elimination, and consequently synthesis, without conjoining the factors. However, the insight first employed in [6] is that it is possible to move conjuncts outside an existential quantifier if the quantified variable does not appear in the support of the conjunct. Formally, let $F_j \subseteq \{1,\ldots,k\}$ be the set of indices i such that y_j is in the support of f_i. Then,

$$\begin{aligned}&\exists y_1,\ldots,y_n.\bigwedge_{i=1}^{k} f_i(\vec{x},\vec{y})\\ &\equiv \exists y_1,\ldots,y_{n-1}.\left(\exists y_n.\bigwedge_{i\in F_n} f_i(\vec{x},\vec{y})\right)\wedge\bigwedge_{i\notin F_n} f_i(\vec{x},\vec{y})\end{aligned}$$

Using the relation between synthesis and existential quantification, we obtain the following result:

Lemma 2. *Let $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge f_2(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$ be a specification and $g_j(\vec{x})$ be a witness to y_j in $\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$. Then, $g_j(\vec{x})$ is a witness to y_j in $f(\vec{x},\vec{y})$.*

Proof. Since $g_j(\vec{x})$ is a witness to y_j in $\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$, then by definition $\left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})] \equiv \exists y_j.\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$.

To prove that $g_j(\vec{x})$ is also a witness of $f(\vec{x},\vec{y})$, it needs to be shown that $f(\vec{x},\vec{y})[y_j \mapsto g_j(\vec{x})] \equiv \exists y_j.f(\vec{x},\vec{y})$. But

$$\begin{aligned}&f(\vec{x},\vec{y})[y_j \mapsto g_j(\vec{x})]\\ &\equiv \left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y}) \wedge \bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})]\\ &\equiv \left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})] \wedge \bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\\ &\equiv \left(\exists y_j.\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right) \wedge \bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\\ &\equiv \exists y_j.\left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y}) \wedge \bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\right)\\ &\equiv \exists y_j.f(\vec{x},\vec{y})\end{aligned}$$

Therefore, $g_j(\vec{x})$ is a witness of $f(\vec{x},\vec{y})$. □

From Lemma 2 we have that a witness for a variable in a factored formula can be constructed from only the factors in which that variable appears. Since in practice each variable will only be in the support of a small subset of the factors, this insight means that it is possible to perform synthesis without converting entirely from the factored to the monolithic representation. Instead, we can design a strategy for synthesis directly over factored formulas.

B. Synthesis from Factored Specifications

Algorithm 1 presents a synthesis framework that takes advantage of the factored representation of the specification, using the insight from Lemma 2 to avoid conjoining all factors at once. Instead, we conjoin the factors one-by-one, and after each conjunction synthesize and eliminate the variables that do not appear in the support of any of the remaining factors. This strategy is similar to the ones followed in model checking [6] and symbolic satisfiability [8] from factored representations.

We assume the existence of a monolithic Boolean synthesis procedure, denoted by $synth(B, X, Y)$, which receives a BDD B, a set of input variables X and a set of output variables Y, and returns a BDD P representing the precondition and a sequence of BDDs $(W_j)_{y_j\in Y}$ representing the implementation.

We start, in line 2, by partitioning the output variables into sets $Y_1,\ldots,Y_k$ such that $y_j \in Y_i$ if and only if B_i is the last factor where y_j appears. In other words, $y_j \in Y_i$ if and only if $\max F_j = i$. We maintain a BDD B which accumulates the factors. In line 3, B is initialized to the empty conjunction, which is equivalent to the constant 1. We then iterate over the factors, conjoining the next factor to B at every iteration in line 5. Once B_i is conjoined, none of the output variables in Y_i appear in any of the remaining factors. The monolithic synthesis procedure is then called in line 6 to synthesize witnesses for every variable in Y_i, in terms of the input variables $x_1,\ldots,x_m$ and the output variables in $Y_{i+1},\ldots,Y_k$. Then, in line 7, B is updated to the precondition

Fig. 1. Synthesis from Factored Specifications

```
Input: Factored specification $\langle B_1, \ldots, B_k \rangle$.
Output: Precondition BDD $P$, witness BDDs $\langle W_1, \ldots, W_n \rangle$.
1:  $X \leftarrow \{x_1, \ldots, x_m\}$
2:  $Y_i \leftarrow \{y_j \mid B_i$ is the last factor where $y_j$ appears$\}$
3:  $B \leftarrow 1$
4:  for $i \leftarrow 1 \ldots k$ do
5:      $B \leftarrow B \wedge B_i$
6:      $P_i, (W_j)_{y_j \in Y_i} \leftarrow synth(B, X \cup Y_{i+1} \cup \ldots \cup Y_k, Y_i)$
7:      $B \leftarrow P_i$
8:  end for
9:  for $i \leftarrow 1 \ldots k$, $i' \leftarrow (i+1) \ldots k$ do
10:     $W_\ell \leftarrow W_\ell[y_j \mapsto W_j]$, for all $y_\ell \in Y_i$, $y_j \in Y_{i'}$
11: end for
12: $P \leftarrow B$
13: return $P, W_1, \ldots, W_n$
```

P_i, which corresponds to the conjunction of the first i factors with the output variables in $Y_1 \cup \ldots \cup Y_i$ existentially quantified.

After the end of the loop, every witness W_j for $y_j \in Y_i$ has the variables from $Y_{i+1}, \ldots, Y_k$ in its support set. In the last step, performed by the loop in lines 9-11, these extra variables are eliminated by substituting their respective witnesses, making every W_j dependent only in the input variables $x_1, \ldots, x_m$.

The following theorem states the correctness of Algorithm 1, which follows from the correctness of the monolithic synthesis procedure and Lemma 2.

Theorem 1. *If $P, W_1, \ldots, W_n$ are computed according to Algorithm 1, then $\exists y_1, \ldots, y_n.(B_1 \wedge \ldots \wedge B_k) \equiv P \equiv (B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n]$.*

Proof. We assume the correctness of the monolithic synthesis procedure $synth$, meaning that $synth(B, X, Y)$ returns a precondition P, and a witness W_j for each variable $y_j \in Y$ in terms of the variables in X, such that $\exists Y.B \equiv P \equiv B[y_j \mapsto W_j]_{y_j \in Y}$.

Consider the loop in lines 4-8. We will first prove that if at the start of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$, then at the end of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$. We will use this to prove that $P \equiv \exists y_1, \ldots, y_n.(B_1 \wedge \ldots \wedge B_k)$

Assume that at the start of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$. Then, after line 5, $B \equiv (\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})) \wedge B_i$. Since $Y_1, \ldots, Y_{i-1}$ do not appear in B_i, the quantifier can be moved outside the conjunction, so $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$.

Then, in line 6 the monolithic synthesis procedure is called on B, with input variables $X \cup Y_{i+1} \cup \ldots \cup Y_k$ and output variables Y_i. By the correctness of the monolithic procedure, $P_i \equiv \exists Y_i.B \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$. Then, after line 7, when B is updated to P_i, $B \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$.

Therefore, if $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$ at the start of the i-th iteration, $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$ at the end of the i-th iteration. Taking $i = 1$, this means that if $B \equiv 1$ (the empty conjunction) before the loop then at the end of the first iteration $B \equiv \exists Y_1.B_1$. Since the invariant $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$ is maintained, at the end of the last iteration $B \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$. Therefore, after line 12, $P \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$, as desired.

We now prove that $(B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n] \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$. In iteration i, we construct W_j for every $y_j \in Y_i$. Since at this time $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$, by the correctness of the $synth$ procedure, $(\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$. Then, since no variables in $Y_1, \ldots, Y_{i-1}$ appear in B_i,

$$\begin{aligned}
&\exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i) \\
&\equiv (\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \\
&\equiv ((\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})) \wedge B_i)[y_j \mapsto W_j]_{y_j \in Y_i}
\end{aligned}$$

Applying this transformation recursively to $\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$ results in $(\ldots(B_1[y_j \mapsto W_j]_{y_j \in Y_1} \wedge B_2)[y_j \mapsto W_j]_{y_j \in Y_2} \wedge \ldots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_k}$. Applying Lemma 2, we can move the composition operators outside the conjunction, giving

$$\begin{aligned}
&\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k) \\
&\equiv (B_1 \wedge \ldots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_1} \ldots [y_j \mapsto W_j]_{y_j \in Y_k}
\end{aligned}$$

Recall that each W_j for $y_j \in Y_i$ might contain variables from $Y_{i+1}, \ldots, Y_k$ in its support set. Because of this, we cannot change the order of the composition operators. However, the loop in lines 9-11 performs the composition of each witness with the ones that succeed it, making every W_j dependent only on $x_1, \ldots, x_m$. This allows the compositions to be performed in any order, so that $\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k) \equiv (B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n]$. □

A problem with Algorithm 1 is that performance will be very dependent on the order of the factors. Consider for example a specification in which for every i, the output support of f_i is $\{y_1, \ldots, y_i\}$. Then, $Y_1 = Y_2 = \ldots = Y_{k-1} = \{\}$ and $Y_k = \{y_1, \ldots, y_n\}$. Processing the factors in order will result in all factors being conjoined before any witness can be synthesized, thus degenerating into the monolithic synthesis procedure. On the other hand, processing the factors in the reverse order would allow one variable to be synthesized immediately after each conjunction. Therefore, it is clear that the algorithm can benefit from reordering the factors before starting the synthesis. Finding the optimal order is a combinatorially hard problem, but a number of heuristics can be used instead. Another possible improvement in the algorithm is clustering, a technique that has been employed in other applications which use factored representations of formulas [8], [14], [15]. In clustering, the set of factors is first partitioned, and the factors in each partition are conjoined into monolithic clusters. The algorithm is then applied over the clusters rather than the individual factors. The next section explores different heuristics for clustering and reordering.

C. Clustering and Reordering

As noted in [14], if the individual BDDs for each factor are small, it is often better to combine different factors into monolithic clusters. If the clusters are constructed so that they remain of reasonable size, clustering reduces the number of iterations while not excessively increasing the cost in space.

Formally, given a factored formula $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge f_2(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$ a *clustering heuristic* partitions the set of factors $\{f_1, f_2, \ldots, f_k\}$ into κ disjoint non-empty subsets $C_1, \ldots, C_\kappa$, called the *clusters*. In practice, each cluster C_ι is represented by a BDD $\mathcal{B}_\iota$ encoding the formula $\bigwedge_{f_i \in C_\iota} f_i(\vec{x},\vec{y})$. Since conjunction is associative and commutative, $\langle \mathcal{B}_1, \ldots, \mathcal{B}_\kappa \rangle$ is itself a factored representation of the original formula f. Therefore, Algorithm 1 can be applied normally to this representation.

The goal of clustering is to create a balance between the number of factors and size of the factors. An example of clustering strategy is *rank-based clustering*, employed in [8]. In this strategy, for every variable y_j, cluster $C_j = \{f_i \mid rank(f_i) = j\}$, where $rank(f_i)$ is the highest index among the variables in the support of f_i.

Rank-based clustering naturally gives rise to some reordering heuristics, in which clusters are ordered either by increasing or decreasing rank. Two more options for reordering factors appear in the context of model checking in [7]. In that work, factored formulas are used to represent transition relations, and different reordering heuristics are used in the forward and backward simulation steps. The following are the four heuristics used in this work:

a) Bouquet's method: [8] Order by increasing rank.

b) Bucket elimination: [8] Order by decreasing rank.

c) Forward: [7] Greedily order factors by number of variables that can be eliminated once the factor is conjoined. In other words, at every step choose the factor that has the greatest number of output variables that do not appear in any of the remaining factors.

d) Backward: [7] Order factors such that at every step the next factor will be the one that has the fewest new variables, that is, variables that have not appeared in any of the previous factors. This heuristic tries to avoid as much as possible increasing the size of the conjoined BDD.

All of the above heuristics for clustering and reordering can be applied to synthesis from factored representations, but it is unclear which would give better results. Section IV describes an experimental evaluation of the different techniques.

D. BDD Variable Ordering

The size of BDDs is strongly influenced by the ordering of the variables. Part of the goal of using factored representations is to be able to represent specifications for which a good variable ordering is not known beforehand. Rather than using an arbitrary variable ordering for these cases, it would be good to be able to compute one by analyzing the structure of the formula. Similarly to clustering, finding the optimal variable ordering is a hard combinatorial problem, but numerous heuristics have been developed to find good enough approximations.

One such heuristic is the inverse *maximum cardinality search* (MCS) ordering [16]. This variable ordering is constructed based on the *Gaifman graph* of the formula $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$, defined as $G = (V, E)$, where $V = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ and $E = \{(v_1, v_2) \mid$ there exists an i such that v_1 and v_2 are in the support of $f_i\}$. In other words, the Gaifman graph of a factored formula has one vertex for each variable and has an edge between every pair of variables that share a factor.

The inverse MCS order can be computed from the Gaifman graph by the following procedure: 1) initialize an empty list L; 2) at each step, select the vertex $v \in V$ not in L with the largest number of neighbors in L, and add v to L; 3) after all vertices have been added, reverse L, so that vertices added later come first in the ordering.

Other heuristics for variable ordering were studied in [8], but among them the inverse MCS heuristic had the best results in that work. Therefore, this heuristic was chosen for the experiments in this paper.

IV. Experimental Evaluation

We performed the experiments using QBF benchmarks taken from the QBFLIB collection [17]. All benchmarks selected were of the form $\forall \vec{x}. \exists \vec{y}. f(\vec{x},\vec{y})$, where $f(\vec{x},\vec{y})$ is a CNF formula. In this case, synthesis corresponds to finding a Skolem function to the existential variables. Every clause in $f(\vec{x},\vec{y})$ can be considered one factor.

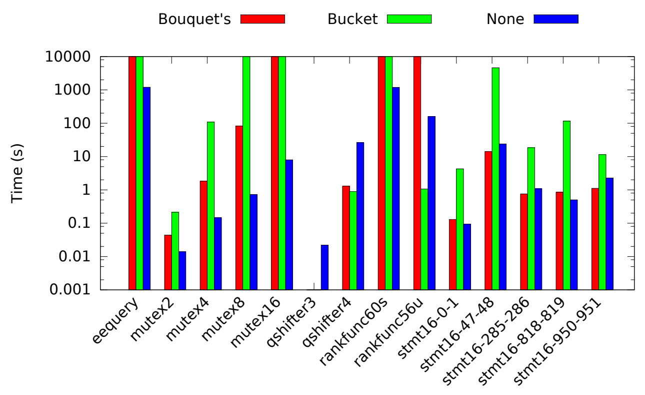
We implemented the factored algorithm from Section III and the various heuristics for clustering and reordering factors in our tool RSynth, in C++11 and using the CUDD [18] package for manipulating BDDs. As of version 3.0.0, CUDD includes a monolithic Boolean synthesis procedure `SolveEqn`, which we used in our implementation as the *synth* subroutine.

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. The algorithm has not been parallelized, so the cluster was solely used to run multiple experiments simultaneously.

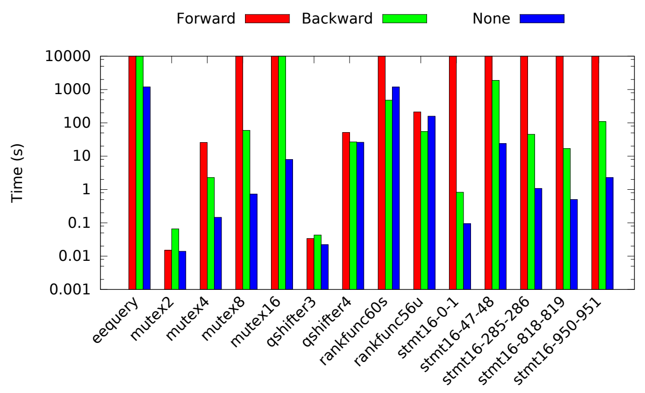
Besides comparing the monolithic and factored algorithms and evaluating different reordering heuristics, we also compare our tool RSynth with two existing tools for Boolean synthesis. The first is the CegarSkolem tool from [9], which uses a SAT-based CEGAR loop and AIGs to perform synthesis from factored formulas. The second is the 2QBF solver CADET [11].

All plots[1] in this section are shown in log scale. Each benchmark was given a time limit of two hours. Only a subset of the total set of benchmarks is included in the plots. Benchmarks for which the results were similar to already-included benchmarks were omitted, as well as benchmarks for which all or almost all of the methods timed out.

[1]Plots are best viewed online for ease of reading.



(a)



(b)

Fig. 2. Performance of the factored algorithm using different reordering heuristics, in log scale. The values include both the time spent reordering the factors and time running the algorithm. Bars of maximum height indicate instances that timed out. Bars not displayed mean that the instance took less that 1ms.

A. *Heuristics for Factor Reordering*

We first measure the performance of the factored algorithm using different reordering heuristics. The bar plots on Figure 2 show the running time of each heuristic on different benchmarks. Figure 2(a) shows the results for *Bouquet's Method* and *Bucket Elimination*, and Figure 2(b) shows the results for the *Forward* and *Backward* heuristics. The bars labeled *None* show the running time when no heuristic is used and the factors are simply processed in the order they are given in the input file.

Surprisingly, the results show that using no reordering is often preferable. In most of the instances, the best running time was achieved with no reordering. In fact, some benchmarks were able to be synthesized in the time limit only when no reordering was used. What this result suggests is that the process by which CNF formulas are generated already produces clauses in a good order. This makes sense because, when constructing a CNF formula, clauses with the same variables will generally be close to each other.

To confirm this point, we also ran experiments where the clauses were reordered randomly. In this case, regardless of the benchmark, the synthesis almost always timed out. We conclude that we can generally assume that the input is given in a good order, but, if this is not the case, using a moderately good heuristic, such as *Bouquet's Method*, already improves significantly over an arbitrary ordering. The performance of the other heuristics varied depending on the type of benchmark. Every heuristic outperformed the others on at least one case. Overall, the *Forward* heuristic seems to have the worst scalability, timing out for most of the instances. This is likely due to it being a greedy heuristic which tries to synthesize as many variables as possible at each step, causing the size of the BDDs to quickly increase.

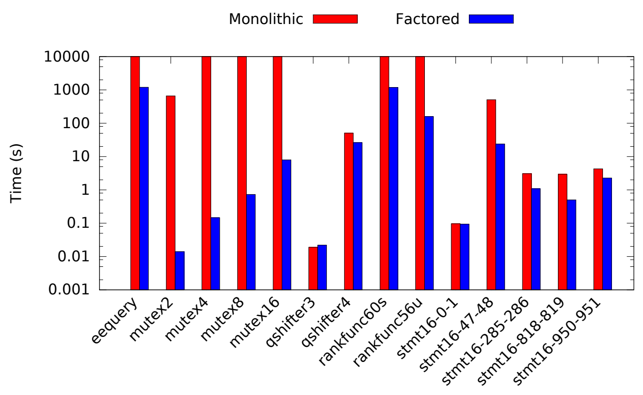


Fig. 3. Performance of the monolithic and factored synthesis algorithms, in log scale. Bars of maximum height indicate instances that timed out.

B. *Factored vs. Monolithic*

Next, we compare the running time of the factored algorithm with synthesis using the monolithic procedure. In the latter, the running time includes the time necessary to conjoin all the factors to create the monolithic representation. Given the previous results, no reordering was used for the factored approach. Results are shown in the bar plot on Figure 3.

It is immediately noticeable that the monolithic approach in most cases displays a much poorer performance compared with the factored one. In the few cases where the monolithic algorithm outperforms the factored algorithm, it is only by a small margin. On the other hand, there are several cases where the factored algorithm outperforms the monolithic one by an order of magnitude or more. There are additionally a number of cases synthesized by the factored algorithm which the monolithic algorithm is not able to solve in the time limit. This indicates that it is worthwhile to take advantage of factored representation for synthesis, and that it allows a number of instances to become feasible compared to a monolithic representation.

C. *Comparison with* CEGARSKOLEM *and* CADET

We compare the performance of RSYNTH with the CEGAR-based tool CEGARSKOLEM and the QBF solver CADET. Given the results of previous experiments, we select the factored algorithm with no reordering for the comparison.

Figure 4 shows a comparison of running time between RSYNTH, CEGARSKOLEM and CADET on the same benchmarks used in the previous experiments. All of the benchmarks

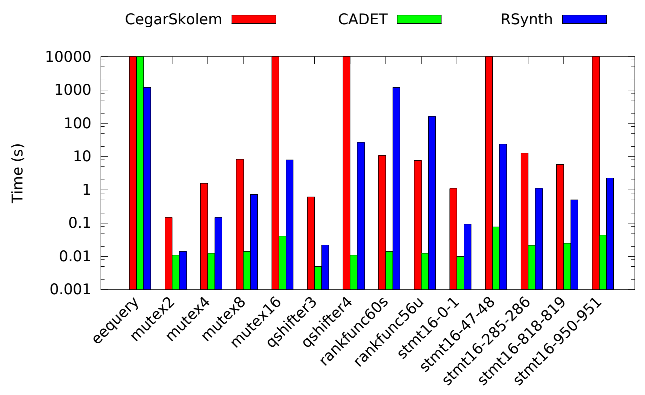


Fig. 4. Comparison of running time between RSYNTH, CEGARSKOLEM and CADET, in log scale. Bars of maximum height indicate instances that timed out.

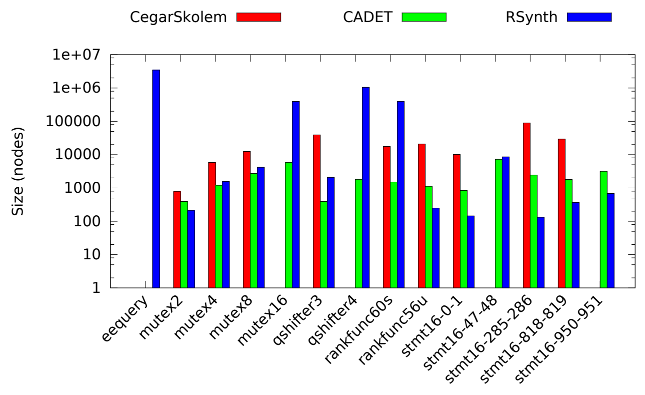


Fig. 5. Comparison of function size, in number of nodes, between RSYNTH, CEGARSKOLEM and CADET, in log scale. The size is not displayed for those instances in which the tool timed out.

are realizable, allowing CADET to be used for them. Out of 161 total benchmarks, RSYNTH was able to synthesize 87 and CEGARSKOLEM 52. There were only 6 benchmarks in which CEGARSKOLEM outperformed RSYNTH, all from the *rankfunc* class. However, CADET had by far the best performance in almost all instances, usually by orders of magnitude, and was able to synthesize all but one of the 161 benchmarks. This leads to the conclusion that the QBF approach is preferable when the specification is realizable.

Figure 5 shows a comparison of the size of the synthesized functions between the three tools. RSYNTH produces functions in the form of BDDs, while CEGARSKOLEM and CADET produce functions in the form of AIGs, therefore the comparison is in number of nodes of these data structures. Missing bars mean that the tool timed out for that particular instance. RSYNTH produced smaller functions for about half of the benchmarks, while CADET had smaller functions for the other half. This demonstrates that in many cases BDDs are indeed able to produce a more compact representation than the one obtained by AIGs.

The main conclusion that we can draw from this comparison is that, for realizable specifications, synthesis approaches

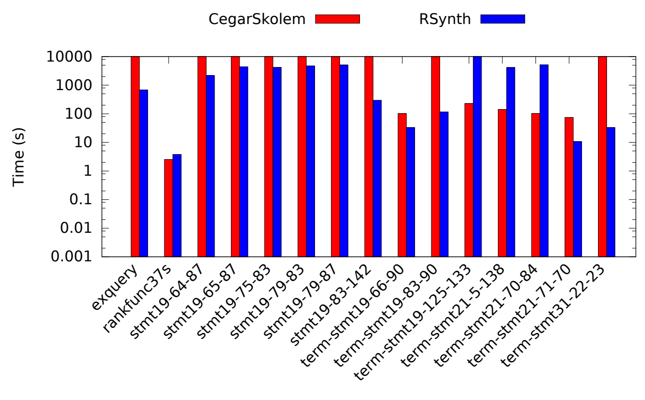


Fig. 6. Comparison of running time between RSYNTH and CEGARSKOLEM tools, in log scale, over unrealizable benchmarks. Bars of maximum height indicate instances that timed out.

based on QBF will likely dominate in terms of running time. In general, however, QBF solvers do not support the generation of Skolem functions when the formula $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$ evaluates to *false*, i.e., $f(\vec{x},\vec{y})$ is unrealizable. Therefore, for unrealizable specifications it becomes necessary to turn to other synthesis approaches. This brings up the question of how RSYNTH and CEGARSKOLEM perform in synthesizing unrealizable instances. The next section presents an evaluation dedicated to answering this question.

D. Unrealizable Specifications

For this comparison, we also used QBF benchmarks of the form $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$ from QBFLIB. This time, however, the quantified formulas evaluate to *false*, meaning that $f(\vec{x},\vec{y})$ is unrealizable. Since CADET is unable to handle such cases, we only perform a comparison between RSYNTH and CEGARSKOLEM for these formulas.

Figure 6 shows the running time of each tool in a set of unrealizable benchmarks. Comparing RSYNTH and CEGARSKOLEM, we see that the results vary depending on the instance, with either tool outperforming the other on a subset of the benchmarks. There are also many cases which one of the tools is able to synthesize while the other times out. In total, 227 benchmarks were solved by at least one of the tools, with CEGARSKOLEM performing best in 118 cases, and RSYNTH performing best in the remaining 109. This result suggests that no approach is strictly better than the other, and the best choice will likely depend on the specific instance of the problem.

The performance of QBF solvers when the specification is realizable invites the question of whether we can find a way to exploit them in synthesizing unrealizable formulas as well. It turns out that it is possible to transform an unrealizable formula into a realizable one with the same witnesses by adding an additional quantifier alternation. This idea is well known in the context of arithmetic realizability [19]. In our case, given a quantified formula $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$, we can construct a formula $\forall\vec{x}.\exists p.(p \leftrightarrow \exists\vec{y}.f(\vec{x},\vec{y}))$, which is always true. By a few simple transformations, we obtain $\forall\vec{x}.\exists p.(\neg p \vee \exists\vec{y}.f(\vec{x},\vec{y})) \wedge (p \vee \forall\vec{y}.\neg f(\vec{x},\vec{y}))$, and by renaming variables and moving the quantifiers to the front, the resulting formula is $\forall\vec{x}.\exists p.\exists\vec{y}.\forall\vec{z}.((\neg p \vee f(\vec{x},\vec{y})) \wedge (p \vee \neg f(\vec{x},\vec{z}))$. Because of the additional quantifier, the formula is no longer in 2QBF, and therefore can no longer be handled by CADET, but other certifying QBF solvers might be able to synthesize it. Note additionally that the Skolem function for the additional existentially quantified variable p now corresponds exactly to the realizability precondition. This might be a promising approach, but a number of factors will have to be taken into consideration. Besides having to deal with the additional quantifiers, if f is originally in CNF, its negation in the second conjunct is now in DNF. Dealing with this may impose another computational challenge. We leave to future work to explore the possibilities of this transformation and the resulting synthesis approach.

V. Discussion

In this paper, we adapted techniques for processing factored representations of Boolean formulas using BDDs to the problem of Boolean functional synthesis. We show that these techniques allow synthesis from a number of specifications which cannot be handled when using a monolithic representation.

We performed an experimental comparison of our tool RSynth with other tools for Boolean synthesis, namely the CEGAR-based tool CegarSkolem [9] and the QBF solver CADET [11]. Our experiments show the QBF approach to be very efficient when the specification is realizable, significantly outperforming the others. However, QBF solvers are not generally able to synthesize functions for unrealizable specifications, which motivates the use of alternative approaches such as the one presented in this paper. For unrealizable specifications, the results of the comparison between RSynth and CegarSkolem vary, with the best tool depending on the specific instance. Therefore, we conclude that there is no single approach that dominates over all cases, rather every tool is able to handle some specifications that the others cannot.

An advantage of BDD-based techniques lies on their ease of applicability to synthesis from temporal specifications, in which Boolean synthesis is a subproblem. The use of partitioned transition relations is a common technique in these problems, and BDDs are a popular representation due to being canonical. Furthermore, these applications usually require synthesis from unrealizable formulas, where these formulas represent the subset of winning states in a game. This suggests that a synthesis approach based on a factored representation using BDDs might be a good choice for this problem. Therefore, we are also interested in pursuing forms of integrating the techniques presented here in frameworks for temporal synthesis.

Acknowledgments

Work supported in part by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering" and by the Brazilian agency CNPq through the Ciência Sem Fronteiras program. We thank Supratik Chakraborty's group, from IIT Bombay, and Markus Rabe, from UC Berkeley, for their help with the tools used in the experiments and for valuable discussions.

References

[1] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, "Symbolic LTL$_f$ Synthesis," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017 (to appear).

[2] J. Kukula and T. Shiple, "Building Circuits from Relations," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds., vol. 1855. Springer, 2000, pp. 113–123.

[3] E. Tronci, "Automatic Synthesis of Controllers from Formal Specifications," in *ICFEM*, 1998, pp. 134–143.

[4] J. Jiang, H. Lin, and W. Hung, "Interpolating Functions From Large Boolean Relations," in *2009 International Conference on Computer-Aided Design (ICCAD'09), November 2-5, 2009, San Jose, CA, USA*. IEEE, 2009, pp. 779–784.

[5] D. Fried, L. M. Tabajara, and M. Y. Vardi, "BDD-Based Boolean Functional Synthesis," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, S. Chaudhuri and A. Farzan, Eds. Springer, 2016.

[6] J. Burch, E. Clarke, and D. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proc. 28th ACM/IEEE Design Automation Conference*. ACM, 1991, pp. 403–407.

[7] D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," in *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, 1994, pp. 299–310.

[8] G. Pan and M. Vardi, "Symbolic Techniques in Satisfiability Solving," *J. Autom. Reasoning*, vol. 35, no. 1-3, pp. 25–50, 2005.

[9] 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, Austin, Texas, USA, September 27-30, 2015.*, 2015, pp. 73–80.

[10] M. Heule, M. Seidl, and A. Biere, "Efficient Extraction of Skolem Functions from QRAT Proofs," in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, 2014, pp. 107–114.

[11] M. N. Rabe and S. A. Seshia, "Incremental Determinization," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, 2016, pp. 375–392.

[12] 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, 2017, pp. 337–353.

[13] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[14] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, "Efficient BDD Algorithms for FSM Synthesis and Verification," in *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV*, 1995.

[15] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, 2002, pp. 359–364.

[16] R. E. Tarjan and M. Yannakakis, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs," *SIAM J. Comput.*, vol. 13, no. 3, pp. 566–579, 1984.

[17] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella, "Quantified Boolean Formulas satisfiability library (QBFLIB)," 2005, www.qbflib.org.

[18] F. Somenzi, *CUDD: CU Decision Diagram Package Release 3.0.0*, University of Colorado at Boulder, 2015.

[19] U. Kohlenbach, *Applied Proof Theory - Proof Interpretations and their Use in Mathematics*, ser. Springer Monographs in Mathematics. Springer, 2008.

Property Directed Reachability with Word-Level Abstraction

Yen-Sheng Ho, Alan Mishchenko, Robert Brayton
University of California, Berkeley
{ysho, alanmi, brayton}@eecs.berkeley.edu

***Abstract*—SAT-based Property Directed Reachability (PDR) has become the key algorithmic development for unbounded model checking of gate-level sequential circuits, but it can be inefficient when applied to word-level problems with heavy arithmetic logic. To address this issue, word-level abstraction is often performed by replacing a whole set of signals with unconstrained new primary inputs. This paper introduces PDR-WLA, a word-level abstraction-refinement algorithm integrated into a modified PDR implementation. The algorithm uses efficient refinement and re-uses reachability information across iterations of refinement. PDR-WLA was implemented in ABC and evaluated on a large set of industrial Verilog designs. Experimental results show significant speedups on hard problems compared to the original PDR and to a naive word-level abstraction-refinement method.**

I. Introduction

Unbounded model checking (UMC) on a Register-Transfer-Level (RTL) circuit is hard but has important applications in the IC design industry:

1) **Sequential equivalence checking (SEC)**. An RTL circuit is sequentially synthesized by retiming, clock-gating, pipelining etc., and UMC is required for proving the correctness of the synthesis.
2) **Property checking**. UMC is used to prove that a circuit always satisfies a set of given properties.

UMC is challenging at the bit level, and even more so at the word level, where complex arithmetic operators, such as multipliers, adders, and variable shifters, are involved.

IC3 [3] or *Property Directed Reachability* (PDR) [8] is considered the best algorithm for bit-level UMC. Abstraction has been a key development and is widely used. Different methods of abstraction include the following. *Word-level abstraction* [12], [1], [5], [4], [14], [11] can be effective by abstracting away heavy arithmetic logic. *Localization abstraction* [19] is a method where gates or signals are replaced by new unconstrained primary inputs. *Counterexample guided abstraction and refinement* (CEGAR) [7] is a framework for iterating abstraction and refinement, where refinement is based on the analysis of *spurious* counterexamples.

We propose *PDR-WLA*, an efficient CEGAR-based word-level localization algorithm integrated with PDR. Given a word-level design, PDR-WLA starts with the extreme abstraction with all *hard signals* (e.g., outputs of multipliers, adders, etc.) abstracted (i.e. replaced by new primary inputs). Next, the resulting word-level abstraction is bit-blasted and given to a modified PDR algorithm. If a counterexample (CEX) is found, PDR-WLA simulates it on the original design to check if it is *real*. If so, PDR-WLA reports it and terminates; otherwise, the CEX is *spurious* and is used to refine the current abstraction. Then a new iteration begins with the refined abstraction.

The main contributions embodied in PDR-WLA are that it

- integrates word-level abstraction with PDR efficiently,
- uses a new refinement strategy that takes advantage of *structural* and *proof-based* analysis of spurious counterexamples, and
- re-uses *reachability information* (reachability clauses) derived in previous iterations.

PDR-WLA is implemented and available in the public verification tool ABC [6] (command *%pdra*). It was evaluated on a set of 195 industrial Verilog RTL benchmarks. PDR-WLA is capable of solving 18 hard problems not solved by PDR. The results also show that 1) reusing previously derived reachability clauses improves performance significantly and 2) the new refinement strategy is the most effective compared to several others proposed and tested.

This paper starts with background material in Section II. PDR-WLA is presented in Section III. Various refinement strategies are given in Section IV. Related work is discussed in Section V. Experiments are presented in Section VI. Conclusions and future work are discussed in Section VII.

II. Preliminaries

A. The UMC problem

The input is a word-level circuit given in *structural Verilog* containing bit-vector (BV) signals, including primary inputs (PIs), primary outputs (POs), flip flops (FFs), and internal signals. Flip flops have reset values as initial states[1]. A design is modeled as a finite state machine (FSM).

Definition 1. An *FSM* is a tuple $M = (I, O, S, Init, T)$ where I is the set of PIs, O is the set of POs, S is the set of FFs, $Init$ is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, there exists a transition from s to s' under i.

The input word-level circuit is assumed to contain a single FSM and a single output, *out*, representing a property to be checked. If the problem is to prove equivalence between two designs, it is assumed that a *miter* circuit, M, has been created by merging all PIs and merging FFs if their correspondences are known. The miter's output, *out*, is a Boolean signal, which

[1]Reset values are either constants or free variables (unknown value X).

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

is the OR of the pairwise XORs of the corresponding outputs of the two designs. Thus $out = 1$ if the two designs are different. Similarly for property checking, out is the output of a monitor, and $out = 1$ if the property fails. In terms of linear temporal logic (LTL), the UMC problem is formulated as $M \models \mathbf{G}\neg out$, i.e. out is never 1 if the property holds.

A UMC solver either reports a *counterexample* (CEX) that falsifies the property or produces an *inductive invariant* proving that the property holds globally.

Definition 2. A *counterexample (CEX)* is a sequence of PI assignments driving the design from an initial state into a state falsifying the property.

Definition 3. An *inductive invariant* (Inv) proving a property $P(s)$ is a predicate function satisfying the properties below.

1) $Init(s) \implies Inv(s)$
2) $Inv(s) \wedge T(i, s, s') \implies Inv(s')$
3) $Inv(s) \implies P(s)$

B. Property Directed Reachability

It is assumed that the reader is familiar with the basic ideas underlying the PDR [8]. Algorithm 1 outlines a high-level view of PDR. It maintains a list of sets of clauses, called the *PDR trace*: $\Omega = (R_0, R_1, \ldots, R_N)$. Every R_j is a set of clauses that over-approximates the set of states reachable from the initial states within j steps. These clauses in a PDR trace are called *reachability clauses*.

Definition 4. Given an FSM, $M = (I, O, S, Init, T)$, and a property P, a *PDR trace* is a sequence of predicate functions, $\Omega = (R_0, R_1, \ldots, R_N)$, such that

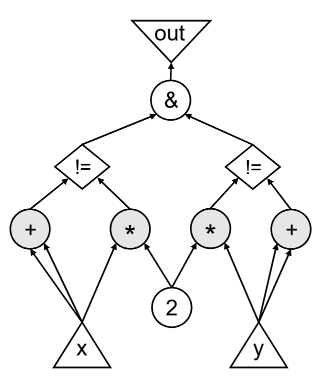
1) $R_0(s) = Init(s)$
2) $R_j(s) \implies R_{j+1}(s)$ for $0 \leq j < N$.
3) $R_j(s) \wedge T(i, s, s') \implies R_{j+1}(s')$ for $0 \leq j < N$.
4) $R_j(s) \implies P(s)$ for $0 \leq j < N$. [2]

Algorithm 1 PDR

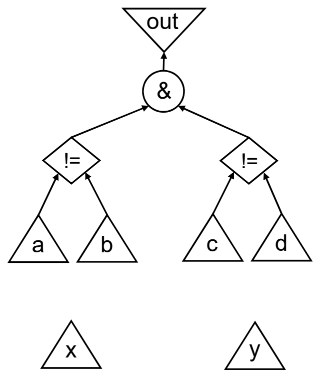
```
Input: $G_M$                         ▷ $G_M$: the bit-level input circuit
Output: status ∈ { SAT, UNSAT }
 1: $\Omega \leftarrow \{Init\}$                 ▷ $\Omega$: the PDR trace
 2: $k \leftarrow 0$                       ▷ $k$: the PDR depth
 3: while true do
 4:     $\Omega$, cex ← RECBLOCKCUBE($G_M$, $\Omega$, $k$)
 5:     if cex $\neq \emptyset$ then
 6:         return SAT                  ▷ Found a real CEX
 7:     $k \leftarrow k + 1$
 8:     $\Omega \leftarrow \Omega \cup \{\top\}$          ▷ Open a new frame
 9:     $\Omega$ ← PROPAGATEBLOCKEDCUBES($G_M$, $\Omega$)
10:     if $\Omega$ contains a fixed point then
11:         return UNSAT
```

PDR starts with the trace Ω with only one element $R_0 = Init$. It then tries to strengthen the trace by recursively blocking bad cubes[3] (line 4). If a bad cube intersects with the initial states, then a CEX is returned. Otherwise, the last element R_k of the trace now satisfies the property P. PDR then adds a new element $\top$ (empty set of clauses) to Ω, and tries to propagate clauses (using induction) from R_1 to R_k (line 9). If a fixed point ($R_j = R_{j+1}$) is found, the problem is declared UNSAT and the inductive invariant (R_j) is returned.

[2] $R_N(s)$ does not necessarily imply $P(s)$, i.e. $R_N(s)$ can contain bad states. Recursive blocking (line 4) tries to remove bad states from $R_N(s)$.



(a) The original circuit with four arithmetic operators, where x and y are primary inputs, 2 is a constant, $!=$ is the complement of a comparator, & is a bit-wise AND, and out is the negation of the property.



(b) An abstraction derived from the original by replacing the 4 arithmetic operators with 4 new primary inputs, a, b, c, and d.

Fig. 1: A combinational circuit illustrating word-level abstraction. $out \equiv 0$, UNSAT, since $2 \times x \equiv x + x$, which forces out to be constant 0.

The details of procedures RECBLOCKCUBE and PROPAGATEBLOCKEDCUBES can be found in [8].

C. Word-level abstraction

In this paper, localization abstraction [19] is used. Given a word-level circuit and a set of target signals (e.g., outputs of arithmetic operators), an abstraction is created by replacing the target signals with free variables called *pseudo PIs* (PPIs). Localization is not necessarily restricted to flip flops; *any* signal can be abstracted, similar to GLA [16].

Example 1. Consider the circuits in Figure 1. The PO, out, in Figure 1a is constant-0, since both $2 \times x \equiv x + x$ and $2 \times y \equiv y + y$ are true. Figure 1b is the result of abstracting all 4 arithmetic operators by replacing their outputs with PPIs. Note that while the example is combinational for illustration purposes, the abstraction scheme applies generally to sequential circuits and UMC problems.

Definition 5. Given an original circuit M and an abstraction A of M, a CEX of A is *real* if it can falsify the property on M (make $out = 1$). Otherwise, it is *spurious*.

[3] A cube of states containing one where the property fails (a bad state).

D. Simple CEGAR (S-CEGAR)

Algorithm 2 (S-CEGAR) is an example of a simple integration of CEGAR and PDR at the word level. The algorithm starts by abstracting *all* signals in the set $\mathcal{S}$ (e.g., outputs of all specified arithmetic operators). Next, an abstraction-refinement loop is entered where each iteration begins by creating a word-level abstraction based on the current set $\mathcal{B}$, the set of signals to be abstracted away. The abstraction is then bit-blasted and solved by a bit-level PDR. If the solver returns UNSAT, the property is proved. Otherwise a CEX to the abstraction, cex, exists and is then *simulated* on the original circuit (W_M) to check if it is *real*. If yes, the property is falsified and cex is returned; otherwise cex is analyzed to derive a set of signals ($\Delta\mathcal{B}$) that, if un-abstracted, can block cex. A new abstraction, with $\Delta\mathcal{B}$ un-abstracted, is then created and a new iteration begins.

Algorithm 2 Simple CEGAR (S-CEGAR)

```
Input: $W_M$                    ▷ $W_M$: the word-level input circuit
Input: $\mathcal{S}$                  ▷ $\mathcal{S}$: the initial set of targeted signals
Output: status ∈ { SAT, UNSAT }
 1: $Iterations \leftarrow 1$
 2: $\mathcal{B} \leftarrow \mathcal{S}$                  ▷ $\mathcal{B}$: the set of abstracted signals
 3: while true do
 4:     $W_A \leftarrow$ CreateAbstraction($W_M$, $\mathcal{B}$)
 5:     $G_A \leftarrow$ BitBlast($W_A$)
 6:     cex ← PDR($G_A$)
 7:     if cex $\neq \emptyset$ then
 8:         if IsRealCEX($W_M$, cex) then
 9:             return SAT
10:         else
11:             $\Delta\mathcal{B} \leftarrow$ Refine($W_M$, $G_A$, $\mathcal{B}$, cex)
12:             $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
13:             $Iterations \leftarrow Iterations + 1$
14:     else
15:         return UNSAT
```

In each iteration of S-CEGAR, a new PDR solver is used and reachability clauses are recomputed from scratch. This is inefficient when the algorithm needs many iterations to find a *final* abstraction, i.e. one that proves the property.

III. PDR with Word-Level Abstraction

A. The algorithm

PDR-WLA uses an important insight; *PDR traces* can be re-used between iterations if abstractions are *monotone*. The idea is similar to previous work of PDR with abstraction [18], [10], extending it to the word level.

Similar to PDR, PDR-WLA starts with the trace Ω containing only $R_0 = Init$. One difference is that PDR-WLA works on an abstraction instead of the original circuit. Similar to S-CEGAR, it begins by abstracting all targeted signals $\mathcal{S}$, resulting in a word-level abstraction (W_A), which is then bit-blasted into a circuit (G_A). As with PDR, PDR-WLA tries to recursively block bad cubes at depth k with the abstract model G_A and the trace Ω. If a bad cube intersects with the initial states, then a CEX, cex, is returned and checked on the original circuit (W_M). If cex is also a CEX on W_M, the property is falsified; otherwise cex is used to compute a subset ($\Delta\mathcal{B}$) of $\mathcal{B}$ to refine the current abstraction ($\Delta\mathcal{B}$ will be un-abstracted). Note that a *nonempty* $\Delta\mathcal{B}$ exists because cex can always be blocked by un-abstracting some signals. Set $\mathcal{B}$ is updated by removing $\Delta\mathcal{B}$. A new abstraction is derived for the next iteration of recursive blocking. If PDR-WLA successfully blocks bad cubes at the current depth k, then it increments the depth by one and adds a new element ($\top$) to Ω. It then tries to propagate the clauses in Ω using induction. If a fixed point is found, then the property holds; otherwise, blocking bad cubes at the new depth will be tried (line 10).

Note that PDR-WLA can be viewed as a PDR algorithm with on-the-fly word-level abstraction. The same trace Ω is re-used throughout the computation, even though the current abstraction is continuously refined. Thus, important reachability information derived in previous iterations is re-used, resulting in a significant speedup over S-CEGAR.

Algorithm 3 PDR with Word-Level Abstraction (PDR-WLA)

```
Input: $W_M$                    ▷ $W_M$: the word-level input circuit
Input: $\mathcal{S}$                  ▷ $\mathcal{S}$: the initial set of targeted signals
Output: status ∈ { SAT, UNSAT }
 1: $Iterations \leftarrow 1$
 2: $\Omega \leftarrow \{Init\}$                        ▷ $\Omega$: the PDR trace
 3: $k \leftarrow 0$                               ▷ $k$: the PDR depth
 4: $\mathcal{B} \leftarrow \mathcal{S}$                  ▷ $\mathcal{B}$: the set of abstracted signals
 5: $W_A \leftarrow$ CreateAbstraction($W_M$, $\mathcal{B}$)
 6:                              ▷ $W_A$: the word-level abstraction
 7: $G_A \leftarrow$ BitBlast($W_A$)
 8: while true do
 9:     while true do
10:         $\Omega$, cex ← RecBlockCube($G_A$, $\Omega$, $k$)
11:         if cex $\neq \emptyset$ then
12:             if IsRealCEX($W_M$, cex) then
13:                 return SAT
14:             else
15:                 $\Delta\mathcal{B} \leftarrow$ Refine($G_A$, $\mathcal{B}$, cex)
16:                 $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$    ▷ Un-abstract some signals
17:                 $W_A \leftarrow$ CreateAbstraction($W_M$, $\mathcal{B}$)
18:                 $G_A \leftarrow$ BitBlast($W_A$)
19:                 $Iterations \leftarrow Iterations + 1$
20:         else
21:             break
22:     $k \leftarrow k + 1$
23:     $\Omega \leftarrow \Omega \cup \{\top\}$                  ▷ Open a new frame
24:     $\Omega \leftarrow$ PropagateBlockedCubes($G_A$, $\Omega$)
25:     if $\Omega$ contains a fixed point then
26:         return UNSAT
```

B. Analysis of PDR-WLA

PDR-WLA represents a general framework for word-level abstraction. It is complementary to other abstraction techniques. The only requirement for soundness is that the derived sequence of abstractions (line 17) is *monotone*:

Definition 6. Let $\{A_j\}$ be a sequence of abstractions, let $\{T_j\}$ be their transition relations, and let $\{Init_j\}$ be their initial states. $\{A_j\}$ is *monotone* if $T_{j+1}(i,s,s') \implies T_j(i,s,s')$ and $Init_{j+1}(s) \implies Init_j(s)$.

Theorem 1. *Let M and A be FSMs where $T_M \implies T_A$ and $Init_M \implies Init_A$. Given a property P, if $\Omega = (R_0, R_1, \ldots, R_N)$ is a PDR trace of A with P, then $\Omega' = (Init_M, R_1, \ldots, R_N)$ is a PDR trace of M with P.*

Proof. Since Ω is a PDR trace of A with P, we have

$$\begin{aligned} R_j(s) &\implies R_{j+1}(s) && \text{for } 0 \le j < N \\ R_j(s) \wedge T_A(i,s,s') &\implies R_{j+1}(s') && \text{for } 0 \le j < N \\ R_j(s) &\implies P(s) && \text{for } 0 \le j < N \end{aligned}$$

Note that Ω' is the same as Ω, except that R_0 is replaced by $Init_M$. Since $Init_M \implies R_0$ and $T_M \implies T_A$, we have

$$\begin{aligned} Init_M(s) &\implies R_1(s) \\ Init_M(s) \wedge T_M(i,s,s') &\implies R_1(s') \\ R_j(s) \wedge T_M(i,s,s') &\implies R_{j+1}(s') \text{ for } 1 \le j < N \\ Init_M(s) &\implies P(s) \end{aligned}$$

Therefore by Definition 4, Ω' is a PDR trace of M with P. □

Theorem 2. *Algorithm 3 is sound and complete.*

Proof. **Soundness.** It is sound to start a new iteration with the previous trace (line 10) because each iteration makes the current abstraction tighter by removing signals from $\mathcal{B}$. Note that R_0 is the initial states of the original circuit (W_M) and is shared by all abstractions. Similarly any state variable in clauses from a previous abstraction must remain in the next abstraction because abstractions are monotone. Thus, a trace can be safely copied over to the next abstraction (Theorem 1). Finally, Algorithm 3 is sound because it returns UNSAT only if it finds an inductive invariant proving the property.

Completeness. The algorithm returns SAT only if a CEX is real. Convergence follows because, in each iteration, the size of $\mathcal{B}$ decreases by at least one (otherwise the CEX must be real). The number of iterations is bounded by $|\mathcal{S}|$. □

IV. Refinements

Given a spurious CEX, cex, the goal of refinement is to identify a subset of signals $\Delta\mathcal{B}$ in $\mathcal{B}$, such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, then cex is blocked in the next iteration. We say that $\Delta\mathcal{B}$ is *un-abstracted*.

A. Simulation-based refinement (SBR)

A simple refinement strategy is to simulate cex on the original circuit (W_M) and compare the PPI values (in cex) with their counterparts in W_M. If the values of a signal s do not match, then s is a refinement *candidate*, i.e. a candidate for un-abstraction. If *all* such candidates are un-abstracted, the property must hold; thus cex is blocked. However, this approach often results in too many candidates being un-abstracted, and thus is not a good strategy.

A more advanced way is to use a *minimized* CEX [15], in which some inputs are assigned to X (don't care) while the minimized CEX still falsifies the property using ternary simulation. Those remaining concrete assignments are called *care-set* signals, meaning that, if any assignment in the set is changed, the output would be changed also. This provides a set of good candidates for refinement. If *all* signals in the care set are un-abstracted, then cex is very likely to be blocked[4].

B. Limitations of simulation-based refinement

While simulation-based methods are often good enough in many applications [16], [10], frequently they do not find a minimal set to un-abstract.

Example 2. Consider the original circuit and its abstraction in Figure 1. Suppose a CEX to the abstraction is found (Fig. 1b), where the assignments of PIs and PPIs are

$$(x,y,a,b,c,d) = (0,0,0,1,0,1).$$

For this example, the care set C returned by counterexample minimization would be all PPIs, $C = \{a,b,c,d\}$. If any PPI is assigned an X, the PO would become X as well; thus all PPIs are in the care set. However, it is clear that the set is not *minimum* because only $\{a,b\}$ or $\{c,d\}$ needs to be un-abstracted to get a final abstraction that results in UNSAT.

Therefore, a more effective proof-based strategy for refinement is proposed.

C. Proof-based refinement (PBR)

The proposed refinement is an enhanced version of the one used in UFAR [11]. The procedure is presented, followed by an analysis and comparison with other proof-based methods in the next subsection.

The main idea is that if cex is spurious, then if the original circuit (M) is simulated with cex, the property holds in all time frames. This implies that the BMC Formula (1) below is UNSAT, where i_t is the input i at time t, s_t is the state variable at time t, k is the depth of cex, $\beta(\cdot)$ denotes the assignment function of cex, and out is the output signal which, in general, can depend on the input i and the current state.

$$\begin{aligned} &Init_M(s) \wedge \bigwedge_{t=0}^{k-1} T_M(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(i_t, s_t) \\ &\wedge \bigwedge_{t=0}^{k} (i_t = \beta(i_t)) \end{aligned} \tag{1}$$

Next, multiplexers are introduced to select between the concrete version and the abstracted version of a signal. If assumptions are made such that all the concrete versions are selected initially, then the resulting BMC formula is still UNSAT and

[4]It is possible that each care-set signal is fed by a tree, without overlaps with the trees of other care-set signals. Even if all the care-set signals are un-abstracted, this will not provide enough constraints, and therefore cex is not blocked.

a modern SAT solver, such as MiniSat [9], returns the final conflict clause. This contains a subset of the assumptions sufficient for UNSAT. This is an efficient variation of finding an *unsat core*, and the subset returned is a candidate for $\Delta\mathcal{B}$.

The procedure operates in four steps:

1) Starting with the original circuit (W_M), for each signal s in $\mathcal{B}$, introduce two new PIs, sel and ppi, where sel is a Boolean signal and ppi is a bit-vector signal *consistent*[5] with the signal s. Replace s with $s' = ITE(sel, s, ppi)$ where ITE is the *if-then-else* operator. Depending on the value of sel, either the concrete signal (s) or the abstracted one (ppi) becomes the new signal s'.
2) Denote the circuit created in Step 1 as N and unroll it with the values of cex plugged in, and keep sel and ppi as the remaining PIs. The cex values plugged in are initial states and PIs at each time frame.
3) Solve the BMC query (2) below, which is guaranteed to be UNSAT. Note that $\beta(\cdot)$ is the assignment function of cex, pi_t is the original PIs at time t, X_t is the set of sel inputs at time t, and x_{tn} is the sel input for the n-th replaced signal at time t. By propagating $x_{tn} = 1$ for all t and n, Query (2) is reduced to (1) by construction ($sel = 1$ means that the concrete version is chosen).

$$\begin{gathered} Init_N(s) \wedge \bigwedge_{t=0}^{k-1} T_N(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(i_t, s_t) \\ \wedge \bigwedge_{t=0}^{k} (pi_t = \beta(pi_t)) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=1}^{|X_t|} x_{tn} \end{gathered} \tag{2}$$

4) Derive a subset ΔX of X using the assumption interface of a modern SAT solver, and determine $\Delta\mathcal{B}$ from ΔX[6].

This procedure is different from conventional proof-based methods. Details will be discussed in Section IV-D.

Example 3. Consider the circuits in Figure 1. Suppose a CEX to the abstraction (Fig. 1b) is obtained, where the assignments of PIs and PPIs are

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 1).$$

Circuit (N), derived by introducing ITEs for each PPI, is shown in Figure 2. If all *sel* PIs $\{s_1, s_2, s_3, s_4\}$ are 1, then the circuit is reduced to the original. Next, PI values ($x = 0$, $y = 0$) are plugged in, and PPIs $\{a, b, c, d\}$ are left unconstrained. The SAT solver is called to determine if *out* can be 1. The result must be UNSAT with the assumptions of the sel PIs being all 1. In this case, the subset returned would be either $\{s_1, s_2\}$ or $\{s_3, s_4\}$, which is the *minimum* set needed. This example demonstrates that PBR can pinpoint a precise set for refinement while a simulation-based approach only gives a rough approximation.

[5]Signals are consistent if they have the same widths and signedness.

[6]In our implementation, there is only one free variable x_n associated with the replaced signal, i.e. $x_n \equiv x_{0n} \equiv x_{1n} \equiv \ldots \equiv x_{kn}$ for $1 \leq n \leq |\mathcal{B}|$. This way, we have $|\mathcal{B}|$ assumptions (instead of $(k+1)|\mathcal{B}|$) and the returned ΔX is exactly our candidate for $\Delta\mathcal{B}$.

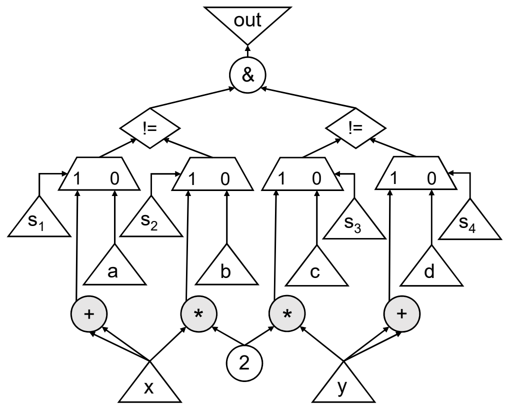


Fig. 2: Example for proof-based refinement, where x and y are original PIs, a-d are pseudo PIs, s_1-s_4 are *sel* PIs. If the assignments of x and y in cex are plugged in, and assumptions are made that s_1-s_4 are all 1, then out is constant-0 (UNSAT).

D. *Comparison of refinement strategies*

Two additional proof-based refinement strategies, PBR-A and PBR-B, are presented compared with SBR (Sec. IV-A) and PBR (Sec. IV-C).

Given a spurious CEX, cex, there are at least two more ways to formulate an UNSAT query that can be used for proof-based refinements. $\beta(\cdot)$ is the assignment function of cex.

PBR-A. This considers Formula (3) below. The idea is that if the values in cex are plugged into the *abstraction* T_A, then out must be 1 at some time frame t. Therefore, the formula asserting that out is 0 for all time frames, with cex plugged in, must be UNSAT. One can then compute the subset of PPIs sufficient for UNSAT, deriving a refinement. Note that PBR-A does not use the information of the original circuit and can be considered as a proof-based version of SBR.

$$\begin{gathered} Init_A(s) \wedge \bigwedge_{t=0}^{k-1} T_A(i_t, s_t, s_{t+1}) \wedge \bigwedge_{t=0}^{k} \neg out(i_t, s_t) \\ \wedge \bigwedge_{t=0}^{k} (i_t = \beta(i_t)) \end{gathered} \tag{3}$$

PBR-B. This uses Formula (4) below. Let pi_t and ppi_t be the original PIs and the PPIs at time t, respectively. Similar to PBR (Formula 2), it takes the original circuit into account by introducing MUXes selecting between PPIs and the original signals, creating a circuit N. The only difference with PBR is that PBR-B also plugs in the assignments of the PPIs in cex into the formula.

$$\begin{gathered} Init_N(s) \wedge \bigwedge_{t=0}^{k-1} T_N(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(i_t, s_t) \\ \wedge \bigwedge_{t=0}^{k} (pi_t = \beta(pi_t) \wedge ppi_t = \beta(ppi_t)) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=1}^{|X_t|} x_{tn} \end{gathered} \tag{4}$$

The four refinement strategies are compared using the two examples below.

Example 4. Consider the circuits in Figure 1. Suppose a CEX is obtained with the assignments of PIs and PPIs

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 1).$$

SBR and PBR-A would refine all PPIs $\{a, b, c, d\}$. PBR-B and PBR would refine only either $\{a, b\}$ or $\{c, d\}$ to obtain a final abstraction. This shows that PBR can get a smaller final abstraction by refining fewer PPIs compared to using SBR and PBR-A.

Example 5. Consider slightly different circuits from those in Figure 1: the AND gates (&) are now replaced by OR gates (|) in both the original circuit and its abstraction. Suppose a CEX is obtained with the assignments of PIs and PPIs

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 0).$$

SBR, PBR-A, and PBR-B all would refine $\{a, b\}$, which is not a final abstraction, requiring another iteration. PBR would refine all PPIs $\{a, b, c, d\}$, which is a final abstraction. This shows that PBR is able to converge with less iterations than the other three. The insight is that PBR refutes *all* spurious CEXes under the same assignments of original PIs in cex, while the others only refute CEXes with the same values of *both* PIs and PPIs.

E. Proposed refinement (PBR and MFFC)

From previous analysis, PBR provides a good set of *candidate* signals that, if un-abstracted, would block CEXes. However, we observed that in many cases, the signals in the fanin cones of those candidate signals would appear in the next iteration of refinement, implying that an additional *structural analysis* can further improve the speed of convergence.

The main idea is to use the *maximum fanout free cones* (MFFC) of those candidate signals. The MFFC of a signal s is a subset of its fanin cone, where each path from a signal in the MFFC to the POs passes through s, i.e. the MFFC of a signal contains all the logic used exclusively by the signal. If a signal is abstracted, its MFFC would be abstracted. On the other hand, if a signal is un-abstracted, its MFFC is better un-abstracted also; otherwise, additional iterations may be needed.

In our experience, un-abstracting all candidate signals as well as those in their MFFCs often converges faster, i.e. reaching a final abstracion after fewer iterations. Thus, the proposed refinement operates in three steps:

1) Compute $\Delta\mathcal{B}_{PBR}$, a set of candidate signals, using PBR.
2) Compute $\Delta\mathcal{B}_{MFFC}$, the set of signals in the intersections of the MFFCs of $\Delta\mathcal{B}_{PBR}$ and $\mathcal{B}$.
3) Derive set $\Delta\mathcal{B}$: $\Delta\mathcal{B} = \Delta\mathcal{B}_{PBR} \cup \Delta\mathcal{B}_{MFFC}$.

V. Related work

A. Word-level abstraction and model checking

Most previous work is *bounded* in that it requires unrolling a circuit to a certain depth k, and then they use SMT solvers [12], [1], [5], [4], [13]. These methods rely on Bounded Model Checking (BMC) [2] and/or k-induction [17]. This becomes inefficient when deep unrolling is needed. In practice, BMC- and induction- based approaches are efficient in finding CEXes, but often incapable of producing an inductive invariant, which is required for UMC problems. PDR-WLA addresses unbounded problems and does not require unrolling.

Welp and Kuehlmann proposed a generalization of PDR to the theory of quantifier free formulas over bit-vectors (QF_BV) [21], [20]. Hybrid simulation and mixed types of atomic reasoning units are used for inductive and CEX generalization. However, they do not re-use PDR traces nor do they perform word-level abstractions.

The closest work to PDR-WLA is AVERROES [14], a word-level algorithm integrating CEGAR and PDR. It abstracts wide data-paths into uninterpreted predicates, constants, terms, and functions, and solves the abstraction with an SMT-based PDR (where SMT solvers are used instead of SAT). The main differences between PDR-WLA and AVERROES are

- PDR-WLA re-uses PDR traces derived in previous iterations; AVERROES does not.
- PDR-WLA uses PBR and MFFC as the main refinement strategy; AVERROES uses strategies similar to SBR, PBR-A, and PBR-B.

UFAR [11] is a word-level algorithm that combines CEGAR and bit-level model checking. It abstracts arithmetic operators with black boxes as well as uninterpreted function constraints, and solves the abstraction with a portfolio of tools, including BMC and PDR. However, UFAR does not reuse PDR traces nor does it perform MFFC refinement.

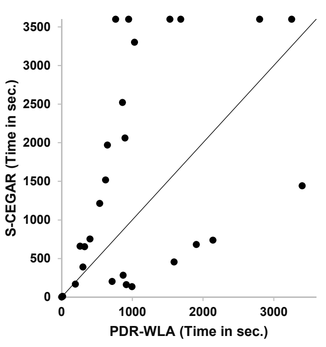
B. PDR with abstraction

Vizel et al. proposed L-IC3 [18], a bit-level IC3 with localization abstraction, where state variables are the targeted signals and different abstractions are used in different time frames. Fan et al. showed that gate-level abstraction (GLA) [16] can be integrated with PDR [10]. However, both approaches consider only bit-level problems. At the word-level, abstracting only state variables may result in aggressive refinement where the entire logic cone of a flip flop would be refined, limiting scalability. On the other hand, GLA cannot be applied directly to the word level. In particular, it mainly uses SBR without considering MFFC, which could be ineffective as discussed in Section IV-D.

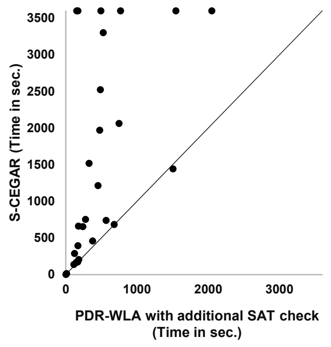
In contrast, PDR-WLA considers not only flip flops, but any type of signals, resulting in a finer-grained abstraction and refinement. Also, it uses specific procedures, PBR and MFFC, to find a final abstraction faster than the bit-level GLA.

VI. Experimental results

Experiments were done to evaluate PDR-WLA using different settings. PDR-WLA is part of the public verification tool, ABC [6] (command *%pdra*), which can parse word-level Verilog and transform the resulting design into a bit-level circuit by bit-blasting. For comparison, S-CEGAR (Sec. II-D, Algorithm 2) was implemented in ABC (command *%abs*).



(a) Running with the default settings, PDR-WLA outperforms S-CEGAR in many cases but not all of them.



(b) With appropriate additional SAT checking, PDR-WLA is able to outperform S-CEGAR in all but one case.

Fig. 3: Comparison of PDR-WLA (*%pdra*) and S-CEGAR (*%abs*). This shows the effectiveness of re-using PDR traces. Note that PDR-WLA and S-CEGAR would be the same if no PDR traces can be re-used. Therefore, only 29 cases with non-zero re-used PDR traces are shown.

The benchmarks used for evaluating PDR-WLA were a set of 195 industrial Verilog RTL designs. Large arithmetic operators and multiplexers were the signals targeted for possible abstraction (set $\mathcal{S}$). A workstation with Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM was used. A time-out of 3600 seconds was used on all experiments.

First, we compare PDR-WLA to the original PDR [8], in which the input Verilog circuit is immediately bit-blasted. Given a 1-hour time-out, PDR-WLA solves 22 fewer test-cases than PDR (89 vs. 111), but PDR-WLA manages to solve 18 hard cases not solved by PDR. It is likely that many of the 22 cases can't be abstracted, so trying such is a waste of time. Together they can solve 129 out of 195 benchmarks. Thus PDR-WLA complements PDR and would work well in a portfolio-based word-level model checker like [11].

To demonstrate the importance of re-using PDR traces in PDR-WLA, it was compared with S-CEGAR, which uses a fresh PDR solver in each iteration and does not preserve the reachability clauses across PDR runs. The results are shown in Figure 3, where the x and y axes represent the solving times of PDR-WLA and S-CEGAR, respectively. In Figure 3a, PDR-WLA outperforms S-CEGAR in all but eight cases. After investigation, it turns out that after several iterations of refinement, an abstraction can become *combinationally UNSAT*, implying that the circuit output can be proved UNSAT with all FFs un-initialized. In those cases, PDR-WLA would work hard to get a non-trivial inductive invariant while S-CEGAR proves that the problem is UNSAT after just one SAT call. To address this problem, PDR-WLA was enhanced to always check if the problem is combinationally UNSAT when an iteration begins. The results are shown in Figure 3b, where PDR-WLA beats S-CEGAR in all but one case.

20 out of the 195 designs were chosen in Table I to give an idea of details such as expected ranges of iterations needed, clauses in PDR traces re-used, and the sizes of $\mathcal{B}$ (signals to be abstracted way) in the final abstractions. All are UNSAT; each is characterized by the number of *hard signals*.

Definition 7. A *hard signal* is the output of

1) an adder, subtractor with width of at least 8, or
2) a multiplier, divider, modulus with width of at least 4, or
3) a multiplexer with width of at least 8.

The initial set of targeted signals ($\mathcal{S}$) is chosen from hard signals with an upper bound of 50 for each of the three categories (e.g., there can be at most 50 adders in $\mathcal{S}$). For each test-case, we show the runtime of six solvers: a) one PDR, b) one S-CEGAR (*%abs*), and c) four PDR-WLA versions (*%pdra*) with different refinement strategies (Sec. IV).

Observations from Table I are given below.

1) **PDR-WLA vs. PDR**. PDR-WLA generally is more efficient when proving hard problems for which small abstractions can be derived. On the other hand, if a problem cannot be abstracted well (e.g., case 20), PDR performs better.
2) **S-CEGAR vs. PDR-WLA**. An important factor in the comparison is the number of *re-used clauses* in all previous PDR traces. If the number is high, a high speedup in PDR-WLA is usually observed. Case 20 is an exception to this, where the re-use number is non-trivial but PDR-WLA is still slower. The reason is that the design becomes combinationally UNSAT after 3 iterations. This problem can be fixed by additional SAT calls as shown in Fig. 3. Note that there can be 0 re-used clauses (e.g., cases 16-19), since all refinements occur at $k = 0$ and no bad states are blocked at $k = 1$. If the trace Ω contains only $R_0 = Init$, no clause can be re-used in the next iteration.
3) **SBR (S2) vs. PBR (S5)**. PBR uses more iterations and derives smaller final abstractions (large $|\mathcal{B}|$) in most cases, implying that PBR leads to more fine-grained and focused refinements.
4) **PBR-B (S3) vs. PBR (S5)**. PBR uses less iterations to find a final abstraction, while PBR-B takes more iterations, which can be avoided by a proper analysis (see Example 5). PBR-B can derive a small final abstraction, but large numbers of iterations can cause poor performance. Note: comparison with PBR-A was not done due to its similarity to SBR.
5) **Without MFFC (S4) vs. with MFFC (S5)**. MFFC can be crucial in preventing unnecessary refinement iterations. This is critical in cases 12, 13, 16, 18, and 19.

VII. Conclusions and future work

PDR-WLA efficiently integrates PDR with word-level abstraction. It re-uses PDR traces, or reachability clauses, derived in previous iterations of refinement. An effective refinement strategy, PBR with MFFC, was developed which was shown capable of deriving small final abstractions using fewer iterations. PDR-WLA was implemented in the public verification

TABLE I: Detailed experimental results for 20 unsatisfiable word-level test-cases. #HardSignals is the number of hard signals (Definition 7). $|S|$ and $|B|$ are sizes of the set of the initial targeted signals ($\mathcal{S}$) and the set of signals to be abstracted away for each iteration ($\mathcal{B}$) in Algorithm 3. #ReusedClauses is the number of clauses in PDR traces re-used by PDR-WLA. The number is 0 if all refinements occur at $k = 0$. The details of SBR, MFFC, PBR, and PBR-B can be found in Section IV.

ID	#Hard Signals	|S|	CPU Time (seconds)						Iterations					#ReusedClauses				|B| in the last iteration				
			pdr	%abs (S1) SBR +MFFC	%pdra (S2) SBR +MFFC	%pdra (S3) PBR-B +MFFC	%pdra (S4) PBR	%pdra (S5) PBR +MFFC	S1	S2	S3	S4	S5	S2	S3	S4	S5	S1	S2	S3	S4	S5
1	1252	100	479.32	170.50	196.46	369.95	**145.23**	164.67	2	2	3	4	4	11	110	181	181	88	88	98	92	92
2	1437	100	1759.97		3253.29	956.76	931.43	**914.51**	4	4	11	4	4	7438	8129	1493	1493	79	79	81	81	81
3	1437	100	1201.74	653.70	326.80	308.24	**306.83**	335.50	2	2	3	3	3	17	100	155	155	87	87	97	94	94
4	1437	100	1800.60		1529.84	1299.36	**583.27**	597.26	4	5	11	5	5	4981	11061	1732	1732	82	79	82	77	77
5	1437	100	931.73	753.11	401.78	272.46	**169.87**	170.91	2	2	3	3	3	19	84	114	114	87	87	96	90	90
6	1437	100	2531.39		2799.57	1128.25	**672.48**	686.62	4	4	11	6	6	3661	6804	2694	2694	78	78	81	78	78
7	1437	100	1383.61	2521.83	862.04	925.89	**410.96**	415.29	5	4	11	6	6	2080	7241	3317	3317	78	78	85	79	79
8	1252	100	925.41	1213.75	538.48	472.20	**225.96**	227.98	4	4	11	6	6	2518	10122	3889	3889	78	78	83	79	79
9	1437	100	1984.69		949.32	2573.81	387.51	**366.87**	4	4	8	5	5	2304	9868	2198	2198	80	79	82	77	77
10	1437	100	850.77	391.41	302.57	766.21	242.32	**225.09**	2	2	5	5	5	113	625	693	693	90	90	95	94	94
11	1437	100	1151.91	2060.92	896.89	958.62	372.40	**349.45**	4	4	10	5	5	2456	7017	1776	1776	78	78	80	79	79
12	133	101	13.61			675.78		**10.38**	4	4	17	17	10	0	2486	0	0	11	11	21	27	30
13	133	101	15.00			624.42		**8.99**	4	4	16	19	10	0	1713	0	0	15	15	21	26	30
14	94	75			763.06	197.19	295.68	**112.98**	6	6	8	11	6	135	228	551	139	3	3	2	21	3
15	95	75			1685.29	745.23	**259.12**	816.54	7	7	8	11	6	147	115	475	151	3	3	1	21	3
16	82	82		545.37	507.48			**417.23**	3	3	4	12	2	0	0	0	0	12	12	0	0	33
17	72	72	353.69	124.98	128.85	132.70	**77.52**	113.61	9	9	14	18	9	0	0	0	0	16	16	16	14	17
18	58	58	1684.21	1343.36	1237.67	1270.25		**861.53**	3	3	4	9	2	0	0	0	0	13	13	13	13	13
19	2150	103	1731.26	**731.82**	732.24	1544.19		789.06	3	3	18	18	12	0	0	0	0	76	76	77	74	77
20	1132	100	**414.30**	739.13	2138.99	3045.19	2191.30	1296.62	3	3	35	40	33	481	5510	10307	4520	13	13	17	15	9

system ABC and evaluated on industrial benchmarks. PDR-WLA solves more hard problems and offers speedups, compared to PDR and S-CEGAR.

Future work.

- Integrate BMC into Algorithm 3. The idea is that BMC can help PDR-WLA find spurious CEXes faster. Early prototypes suggest speedups in some benchmarks.
- Develop a good way to *shrink* abstractions. A shrinking procedure can be useful as shown in GLA [16]. One of the main challenges is that PDR traces cannot be re-used if abstractions are no longer monotone.
- Enhance the refinement strategies with *constraints*. For example, uninterpreted function constraints are known to be effective for SEC problems; partial interpretation constraints can also be useful. The challenge is to derive and apply constraints efficiently and automatically.

VIII. Acknowledgements

This work was supported in part by SRC contract 2710.001 "SAT-based methods for scalable synthesis and verification" and NSF/NSA grant "Enhanced equivalence checking in crypto-analytic applications".

References

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proc. of LPAR '08*.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS'99*.

[3] A. R. Bradley. Sat-based model checking without unrolling. In *Proc. of VMCAI'11*.

[4] B. A. Brady, R. E. Bryant, and S. A. Seshia. Learning conditional abstractions. In *Proc. of FMCAD'11*.

[5] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proc. of MEMOCODE'10*.

[6] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Proc. of CAV'10*.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*.

[8] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proc. of FMCAD'11*.

[9] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*.

[10] K. Fan, M.-J. Yang, and C.-Y. Huang. Automatic abstraction refinement of TR for PDR. In *Proc. of ASP-DAC'16*.

[11] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton. Efficient uninterpreted function abstraction and refinement for word-level model checking. In *Proc. of FMCAD'16*.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In *Proc. of DAC'05*.

[13] D. Kroening and M. Purandare. Ebmc: The enhanced bounded model checker. `www.cprover.org/ebmc`.

[14] S. Lee and K. A. Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *Proc. of CAV'14*.

[15] A. Mishchenko, N. Eén, and R. Brayton. A toolbox for counter-example analysis and optimization. In *Proc. of IWLS'13*.

[16] A. Mishchenko, N. Eén, R. K. Brayton, J. Baumgartner, H. Mony, and P. K. Nalla. GLA: Gate-level abstraction revisited. In *Proc. of DATE'13*.

[17] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of FMCAD'00*.

[18] Y. Vizel, O. Grumberg, and S. Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In *Proc. of FMCAD'12*.

[19] D. Wang, P.-H. Jiang, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Prof. of DAC'01*.

[20] T. Welp and A. Kuehlmann. Property directed reachability for qf_bv with mixed type atomic reasoning units. In *Proc. of ASP-DAC'14*.

[21] T. Welp and A. Kuehlmann. QF BV model checking with property directed reachability. In *Proc. of DATE'13*.

Learning Support Sets in IC3 and Quip: the Good, the Bad, and the Ugly

Ryan Berryhill
Dept. of ECE
University of Toronto
Toronto, Canada
ryan@eecg.toronto.edu

Alexander Ivrii
IBM Research
Haifa, Israel
alexi@il.ibm.com

Neil Veira
Dept. of ECE
University of Toronto
Toronto, Canada
neil.veira@mail.utoronto.ca

Andreas Veneris
Depts. of ECE and CS
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

Abstract*—In recent years, IC3 has enjoyed wide adoption by academia and industry as an unbounded model checking engine. The core algorithm works by learning lemmas that, given a safe property, eventually converge to an inductive proof. As such, its runtime performance is heavily dependent upon "pushing" (or "promoting") important lemmas, possibly by discovering additional supporting lemmas. More recently, Quip has emerged to be a complementary extension behind the reasoning capabilities of IC3 as it allows it to target particular lemmas for pushing. This also raises the following question: which lemmas should be promoted? To that end, this paper extends the reasoning capabilities of IC3 and Quip using special SAT queries to find *support sets* that represent fine-grained information on which lemmas are required to push other lemmas. Further, this paper presents an IC3-based algorithm called Truss (Testing Reachability Using Support Sets) that uses support sets to identify sets of lemmas that may be close to forming an inductive proof. The set is targeted for promotion as a cohesive unit. If any of the lemmas cannot be promoted, the entire set is abandoned and a new set excluding that lemma is found. In the presented framework, there are two reasons why a lemma cannot be promoted: either because it blocks a known reachable state (in which case, the lemma is permanently marked as *bad*), or because lemma promotion exceeds a specified amount of effort (in which case the lemma is temporarily marked as *ugly*). Intuitively, the proposed approach allows the algorithm to construct a proof more quickly by focusing on the important yet easily-pushed lemmas. Experiments on the HWMCC'15 benchmark set show a significant improvement against existing practices. Compared to Quip, our algorithm solves 17 more problem instances and it offers an impressive 1.77x speedup.

I. Introduction

Formal verification remains one of the fastest growing segments in verification [1]. Unbounded model checking, which determines if particular states are reachable in a circuit, is a problem of fundamental importance in this area. `IC3` [2], [3] has established itself as a state-of-the-art unbounded model checker and has seen wide adoption in industry [4]. More recently, the closely related technique of `Quip` [5] has emerged with better run time performance and greater reasoning capabilities. Any improvements to `IC3` or `Quip` can therefore have wide-reaching impact in formal verification.

Both `Quip` and `IC3` aim to construct an inductive invariant proving a given safety property. They share similar core functionality. A Boolean Satisfiability (SAT)-based procedure is used to detect states that can reach a property violation, which are referred to as counter-examples-to-induction (CTIs). When a CTI is detected, the algorithm tries to learn a lemma that explains why the state is not reachable in a bounded number of steps called the lemma's *level*. Through this procedure, the algorithm learns and refines a sequence of over-approximations of the states reachable at each level. This sequence is known as the *inductive trace*, and the approximation at each level is called a *frame*. An additional pushing step promotes lemmas from one level to the next if their current frame is strong enough. The run time performance of these algorithms is dependent on their ability to learn and promote relevant lemmas.

This paper presents an algorithm called `Truss` (Testing Reachability Using Support Sets) that leverages the features of modern incremental SAT solvers to compute lemma supports and the reasoning capabilities of `Quip` to encourage the promotion of highly relevant lemmas that are easy to promote. In a nutshell, special SAT queries are used to identify a set of lemmas that participate in a bounded proof of the property, and thus may potentially appear in a safe inductive invariant. This set is targeted for promotion as a cohesive unit. If any lemma in the set fails to promote, the entire set is abandoned and one or more lemmas are temporarily or permanently blacklisted from appearing in future sets. A new set is found and targeted for promotion. This process repeats until it can be determined that no such set is available, at which point the algorithm falls back to the usual recursive blocking approach used in `Quip`.

More specifically, a special SAT query identifies a set of lemmas sufficient to support the promotion of a particular lemma or the property itself, called its *support set* [6]. The query is similar to the one normally used to check for relative induction, but requires the addition of a unique activation literal for each lemma. When the property is being promoted to a new level, a support set is computed from the frame one level below the property. If one is found, the algorithm attempts to promote the supporting lemmas first. This is accomplished by recursively computing their support sets, and attempting to promote those lemmas, and so on. Lemmas at the bottom level do not have support sets; they are promoted using the usual recursive blocking method.

When supporting lemmas cannot be promoted, the algorithm is allowed to expend a configurable amount of effort to promote the lemma with recursive blocking. If a lemma still cannot be promoted, it is marked as ugly and is temporarily blacklisted from appearing in support sets. As is the case in `Quip` this process can reveal reachable states. When a lemma blocks a known reachable state, it is marked as bad. Bad lemmas are permanently blacklisted, as they can never

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

appear in an inductive invariant. Similarly to Quip, absolute invariant lemmas are classified as good. Not all lemmas are classified, some are left in the category of unknown. Ugly lemmas also fit into the unknown class as it isn't known whether they are good or bad. Hence the algorithm partitions lemmas into classes containing the unknown, the good, the bad, and the ugly lemmas. This contrasts with the work of [6], where support sets are used to compute heuristic functions for each lemma. A special step after pushing attempts to learn new lemmas in order to promote those that have high heuristic values. Effort limits are not used and the heuristics require support sets for every lemma, making the approach presented in that work computationally expensive.

Experiments on HWMCC'15 circuits demonstrate the effectiveness of Truss. Compared to Quip, it offers a 1.77x speedup on the HWMCC'15 benchmark set and solves 17 more problem instances. Further, while Truss processes a similar number of must-proof obligations, it processes only one third as many may-proof obligations as Quip. This comes with the cost of additional overhead to compute support sets, which is found to represent less than 5% of the total runtime.

The rest of this paper is organized as follows. Section II presents notation and background material on unbounded model checking. Section III presents the technique used to compute support sets. Section IV presents the main algorithm. Section V presents an empirical evaluation of the approach. Section VI examines alternative approaches and related work. Finally, section VII concludes the paper.

II. Preliminaries

A. Notation

The following terminology and notation is used throughout this paper. A literal is either a variable or its negation. A cube is a conjunction of literals. A clause is a disjunction of literals. A Boolean formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. A clause or a cube can be treated as a set of literals, and a CNF formula as a set of clauses. For a CNF formula F, $c \in F$ means that the clause c appears in F. Similarly $l \in c$ means that the literal l occurs in c.

Consider a finite transition system, and let $\mathcal{V}$ be the state variables of the system. The primed versions $\mathcal{V}' = \{v' | v \in \mathcal{V}\}$ represent the next-state functions. That is, for each $v \in \mathcal{V}$, v' is a binary function of the current state and input defining the next state for v. For any formula F over $\mathcal{V}$, the primed version F' represents the same formula with each free variable $v \in \mathcal{V}$ replaced by v'. A model checking problem is a tuple $P = (\mathit{Init}, \mathcal{T}, \mathit{Bad})$ where $\mathit{Init}(\mathcal{V})$ and $\mathit{Bad}(\mathcal{V})$ are CNF formulas over $\mathcal{V}$ representing the initial states and the unsafe states, respectively. States that are not unsafe are called safe states. The transition relation $\mathcal{T}(\mathcal{V}, \mathcal{V}')$ is a CNF formula over $\mathcal{V} \cup \mathcal{V}'$. It is encoded such that $\mathcal{T}(\vec{v}, \vec{v}\,')$ is satisfiable iff state $\vec{v}$ can transition to state $\vec{v}\,'$. States are called *i-step reachable* if they can be reached in i or fewer steps from an initial state under $\mathcal{T}$. States that are i-step reachable for some value of i are *reachable*.

For any formula F over $\mathcal{V}$, a state $\vec{v}$ that satisfies F (*i.e.,* $F(\vec{v}) = 1$) is called an F-state. Given two formulas $F(\mathcal{V})$ and $G(\mathcal{V})$, F is inductive relative to G if:

$$G(\vec{v}) \wedge F(\vec{v}) \wedge \mathcal{T}(\vec{v}, \vec{v}\,') \Rightarrow F(\vec{v}\,')$$

If F is inductive relative to itself, it is simply *inductive*.

A problem instance P is Unsafe iff there exists a natural number N such that the following formula is SAT:

$$\mathit{Init}(\vec{v_0}) \wedge \Big(\bigwedge_{i=0}^{N-1} \mathcal{T}(\vec{v_i}, \vec{v_{i+1}})\Big) \wedge \mathit{Bad}(\vec{v_N}) \tag{1}$$

A problem instance P is Safe iff there exists an inductive formula $\mathit{Inv}(\mathcal{V})$ that also meets the following conditions:

$$\mathit{Init}(\vec{v}) \Rightarrow \mathit{Inv}(\vec{v}\,') \tag{2}$$

$$\mathit{Inv}(\vec{v}) \Rightarrow \neg \mathit{Bad}(\vec{v}) \tag{3}$$

A formula satisfying Eq. 2 satisfies *initiation*, meaning that it contains all initial states. An inductive formula that satisfies initiation contains all reachable states and is called an *inductive invariant*. A formula satisfying Eq. 3 is *safe*, meaning that it represents a superset of the safe states. A safe inductive invariant represents a superset of the reachable states and a subset of the safe states. Each of these properties can be checked using a single query to a SAT solver, so a safe inductive invariant is a proof that P is Safe.

B. Overview of Quip

This section gives an overview of Quip [5], which is itself based on IC3 [2], [3]. Given an unbounded model checking problem, it either returns an inductive invariant proving the property or a counter-example trace that reaches an unsafe state. It works by maintaining a sequence of CNF formulas $F_0, F_1, ...$ called the *inductive trace*. Each F_i is a *frame*, and its index i is called its *level*. Each clause $c \in F_i$ is called a *lemma*. The frame F_0 is identical to Init. The algorithm maintains two invariants for all $i \geq 0$:

$$F_i \wedge \mathcal{T} \Rightarrow F'_{i+1}$$

$$F_{i+1} \subseteq F_i$$

That is, each frame is inductive relative to the frame below it and each frame contains a subset of the lemmas in the frame below it. These invariants imply that F_i is an over-approximation of the i-step reachable states. The algorithm also maintains a special frame F_∞ containing lemmas that over-approximate all reachable states. In addition, Quip maintains a set R of known reachable states, which is used in various optimizations.

Algorithm 1 presents pseudocode for the top-level procedure of Quip. Certain bookkeeping details are omitted for succinctness, including the details needed to construct counter-example traces. Line 2 checks if any initial states are unsafe, as the main loop does not handle this case. Lines 3 through 7 contain the main loop. The loop calls the recursive blocking procedure on line 4, which strengthens the inductive trace such that $\neg \mathit{Bad}$ is inductive relative to F_{k-1}. After handling all proof obligations, line 5 performs the pushing procedure. For each non-bad lemma φ in each non-empty F_i, the algorithm checks if $F_i \wedge \mathcal{T} \Rightarrow \varphi'$. If so, the lemma is promoted to level $i + 1$. If at any point $F_i = F_{i+1}$ for some value of i then F_i is inductive and can be added to F_∞. Finally, line 6 checks if $F_\infty \Rightarrow \neg \mathit{Bad}$. If this holds, F_∞ is a safe inductive

Algorithm 1 `Quip` $(Init, \mathcal{T}, Bad)$

```
1: $R = \emptyset,\ F_0 = Init,\ k = 1$
2: if SAT? $(F_0 \wedge Bad)$ then return UNSAFE
3: loop
4:     if $\neg$Quip_Block$(k)$ then return UNSAFE
5:     Quip_Push()
6:     if $\neg$SAT?$(F_\infty \wedge Bad)$ then return SAFE
7:     $k = k + 1$
```

Algorithm 2 `Quip_Block` (k)

```
1: $Q = \emptyset$
2: Add$(Q, \langle Bad, k, must \rangle)$
3: while $\neg$Empty$(Q)$ do
4:     $\langle m, i, p \rangle$ = Pop$(Q)$
5:     if $(i = 0) \vee$ Match$(R, m)$ then
6:         if $p = must$ then return false
7:         else AddReachable$(R)$
8:     else if SAT?$(F_{i-1} \wedge \mathcal{T} \wedge m')$ then
9:         $u$ = Predecessor$(m)$
10:        Add$(Q, \langle$Lift$(u), i-1, p \rangle)$
11:        Add$(Q, \langle m, i, p \rangle)$
12:    else
13:        $(\varphi, g)$ = Generalize$(\neg m, i)$
14:        AddLemma$(\varphi, g)$
15:        if $g < k - 1$ then Add$(Q, \langle \neg\varphi, g+1, may \rangle)$
16: return true
```

invariant and the algorithm terminates, otherwise the algorithm continues to the next iteration.

The recursive blocking procedure is described in detail in Algorithm 2. `Quip` maintains a queue of proof obligations of the form $\langle m, i, p \rangle$, where m is a cube over $\mathcal{V}$ or $m = Bad$, i is a level, and $p \in \{must, may\}$ is the type of obligation. A must-proof obligation represents a cube that must be blocked in order for the problem to be SAFE. A may-proof obligation represents a cube that may be useful to block, but its failure does not necessarily imply the problem is UNSAFE. Line 2 initializes the queue to contain the obligation to block all unsafe states at level k.

At each step, the algorithm attempts to discharge an obligation $\langle m, i, p \rangle$ with the lowest level by proving that no m-state is i-step reachable (*i.e.,* blocking m at level i). This process has three potential outcomes.

The first (lines 5–7) occurs when either $i = 0$ or R contains an m-state. In this case, the obligation cannot be discharged and a counter-example is found (if $p = must$) or new reachable states are discovered. The `AddReachable` procedure called on line 7 adds any newly-discovered reachable states to R by traversing the chain of obligations that includes $\langle m, i, p \rangle$. All lemmas in the inductive trace are checked against R at this point, and any lemma that blocks a reachable state is marked bad.

A second possibility (lines 8–11) occurs when $F_{i-1} \wedge \mathcal{T} \not\Rightarrow \neg m'$, meaning that m has a predecessor state u in F_{i-1}. The obligation $\langle m, i, p \rangle$ is returned to the queue and $\langle u, i-1, p \rangle$ is added. The `Predecessor` procedure called on line 9 extracts a predecessor state u of m from the SAT solver. A proof obligation for u is added on line 10. However, in practice, u is lifted to a smaller cube that represents more states, all of which are predecessors of m. This is handled by the `Lift` procedure.

The final possibility (lines 13–15) occurs when $F_{i-1} \wedge \mathcal{T} \Rightarrow \neg m'$. In this case, the obligation is successfully discharged and the algorithm learns a new lemma φ such that $Init \Rightarrow \varphi$, $\varphi \Rightarrow \neg m$, and $\varphi \wedge F_{i-1} \wedge \mathcal{T} \Rightarrow \varphi'$. The lemma over-approximates the i-step reachable states and demonstrates why m is not i-step reachable. It is added to all frames F_j for $j \leq i$. When blocking m, the lemma $\varphi = \neg m$ is sufficient. However, key to the performance of `Quip` and `IC3` is the `Generalize` procedure on line 13, which may find a stronger clause that is inductive relative to F_{g-1} for some $g \geq i$. The generalized lemma is added at level g on line 14. In addition, on line 15 a new obligation $\langle \neg\varphi, g+1, may \rangle$ can be added to the queue. This forces the algorithm to push φ forward, thereby blocking m at higher levels.

Algorithm 3 `Support` $(F, \mathcal{T}, \varphi)$

```
1: $F_{en} = \emptyset$, $assumps = \emptyset$, $\Gamma(\varphi) = \emptyset$
2: for all $c_i \in F$ do
3:     $l_i$ = ActivationLit$(c_i)$
4:     $assumps = assumps \cup \{\neg l_i\}$
5:     $c_{en} = c_i \cup \{l_i\}$
6:     $F_{en} = F_{en} \cup \{c_{en}\}$
7: $\Phi = F_{en} \wedge \mathcal{T} \wedge \neg\varphi'$
8: if SAT?$(\Phi, assumps)$ then return NULL
9: $conflicts$ = ConflictAssumptions$(\Phi)$
10: for all $(\neg l_i) \in conflicts$ do
11:     $\Gamma(\varphi) = \Gamma(\varphi) \cup \{c_i\}$
12: return $\Gamma(\varphi)$
```

III. Computing Support Sets

Computing support sets, first introduced in [6], is an integral aspect of this work. This section defines the concept and describes a method to compute them as a matter of practical interest.

A support set for a lemma φ is a set of lemmas relative to which φ is inductive. `IC3` and `Quip` use this concept implicitly when executing SAT queries of the form `SAT?`$(F_i \wedge \mathcal{T} \wedge \neg\varphi')$ relative to various frames F_i. This query asks if φ is inductive relative to F_i, or equivalently, if F_i is a support set for φ. In practice, it is often the case that only a small subset of F_i is actually needed to support φ [6]. Various methods can be used to compute small support sets, but in this work only one method is considered. It takes as input a CNF formula F, a transition relation $\mathcal{T}$ also given in CNF, and a clause φ. It returns a subset of F relative to which φ is inductive. In other words, it returns a support set $\Gamma(\varphi) \subseteq F$. Note that the support of φ is not necessarily unique.

The basic version of the method is shown in Algorithm 3. In that description, `ActivationLit`(c) is a procedure that returns a new activation literal unique to clause c. A unique activation literal is added to each clause of F to construct a new formula F_{en}. A SAT query is constructed from the following formula:

$$F_{en} \wedge \mathcal{T} \wedge \neg\varphi' \bigwedge_{c_i \in F} \neg l_i$$

where l_i is the activation literal for clause c_i. The clauses forcing the activation literals to 0 are passed to the solver as assumptions. If the formula is satisfiable, then F is not a support set for φ and the algorithm returns NULL. Otherwise, the activation literals in the conflicting assumption set are mapped back to their corresponding clauses, each of which is added to $\Gamma(\varphi)$. Intuitively, this is equivalent to intersecting a clausal UNSAT core of $F_i \wedge \mathcal{T} \wedge \neg\varphi'$ with F_i. However, Algorithm 3 is useful in practice as it may offer better performance and can be applied when using SAT solvers without support for efficient generation of clausal cores.

In the context of `Truss`, Algorithm 3 is used to compute supports of various lemmas in the inductive trace. Given a lemma $\varphi \in F_{i+1}$ for some i, we need to compute its support relative to a subset of lemmas in frame F_i. More precisely, on each invocation we may also have a set $B \subset F_i$ of *blacklisted* lemmas, and a call to `Support` $(F_i \setminus B, \mathcal{T}, \varphi)$ asks if F_i contains a support set for φ consisting only of non-blacklisted lemmas.

Furthermore, the SAT queries in Algorithm 3 can be executed incrementally. To this end, each time `AddLemma`(φ, g) is called, φ is also added to the incremental solver with its unique activation literal. When constructing *assumps* in Algorithm 3, each activation literal corresponding to a lemma in $F_i \setminus B$ is added with negative polarity. Those corresponding to all other lemmas are added with positive polarity, effectively removing the corresponding clause from the resulting formula.

When $\varphi \in F_{i+1}$ has a support $\Gamma(\varphi) \subseteq F_i \setminus B$, what we will really need is the *critical* part of the support set, defined as $\Gamma(\varphi) \cap (F_i \setminus F_{i+1})$. This critical part of the support set represents lemmas that would be sufficient to promote in order to promote φ. In particular, when $\Gamma(\varphi) \cap (F_i \setminus F_{i+1})$ is empty, φ can be immediately added to a higher frame. In practice we do not introduce activation literals for lemmas in F_∞, as in our algorithm these lemmas are never blacklisted and will never be part of a critical support set.

IV. Safety Checking With Support Sets

This section presents the `Truss` algorithm, which solves the safety checking problem using support sets to guide its search for an inductive proof. We first explain the classification of lemmas into categories of good, bad, and the novel classification of ugly. We then present the algorithm itself. Finally, its strategy is contrasted against `Quip` and `IC3`.

A. Classifying Lemmas

A key aspect of the algorithm is its classification of lemmas into the categories of unknown, good, bad, and ugly. This subsection explains the criteria that lead to these classifications. The next subsection explains the algorithm and how it treats lemmas based on their classification.

The first three categories are also present in `Quip`, and we briefly describe them here. Unknown is the default classification, and simply means that the lemma is not known to belong to the other categories. Good lemmas are those that have been promoted to the frame F_∞. They are intuitively good because the algorithm terminates when $\neg Bad$ is inductive relative to F_∞.

On the other hand, bad lemmas are those that are known to be non-inductive. This is detected through the discovery of reachable states. Letting the cube r represent a known reachable state, a lemma φ is marked bad if the formula $\varphi \wedge r$ is unsatisfiable, which indicates that φ does not over-approximate the set of reachable states. These lemmas are undesirable for the algorithm. It is forced to spend time pushing them despite the fact that they cannot appear in a proof. As mentioned earlier, non-inductive lemmas may also make it harder for the algorithm to discover a proof. In `Quip` and in our approach, no attempt is made to push bad lemmas forward.

The novel classification used in `Truss` is *ugly*. Unlike good and bad, which are applied to a lemma permanently, ugly may be a temporary classification. In that sense, ugly is a sub-class of unknown, since ugly lemmas are also not known to be good or bad. Informally, an ugly lemma is one that appears difficult to push to higher levels. This could happen if the lemma is non-inductive, or if the algorithm simply needs to learn more supporting lemmas to push it forward. A lemma φ may be marked as ugly when considering a may-proof obligation of the form $\langle \neg\varphi, i, may \rangle$. If promoting φ to level i requires adding more than one lemma to F_{i-1}, it appears difficult to push and is therefore ugly. Different criteria could be applied instead. This criterion ignores many aspects of the true cost of supporting φ, but is simple to implement and works well in practice.

In addition, an ugly lemma may be reclassified into any of the other classifications. If the lemma is promoted to F_∞, it is marked as good. If it is found to exclude a reachable state, it becomes bad. Finally, if it is pushed forward during `Quip_Push`, it ceases to be ugly and becomes unknown. Intuitively, this is because the lemma was marked ugly as a result of insufficient support at its current level. The fact that it was successfully pushed indicates that the algorithm has learned enough supporting lemmas.

B. The Algorithm

This subsection describes the `Truss` algorithm. It uses the outer loop from `Quip` as described in Algorithm 1. It also uses a similar pushing procedure, the only difference being the re-classification of ugly lemmas as noted in the previous subsection. However, it uses a novel blocking procedure `Truss_Block` that discharges proof obligations using support sets where possible. This section describes the new blocking procedure.

As is the case for Algorithm 2, `Truss_Block` takes as input a natural number k representing the level at which to block all unsafe states. As mentioned earlier, it identifies a set of lemmas that could be close to forming a safe inductive invariant. This is accomplished by computing a critical support set for the property, and then computing critical support sets for the supporting lemmas, and so on. This continues until reaching lemmas for which no suitable support set can be found. Those lemmas are promoted using an approach similar to that used by `Quip_Block`. When a lemma cannot be promoted, the set is abandoned and a new one identified.

This procedure is integrated with the proof obligation processing scheme. Before explaining the algorithm, we first

Algorithm 4 `Truss_Block` (k)

```
1:  $Q = \emptyset$
2:  $E[\varphi] = 0 \ \forall$ lemmas $\varphi$
3:  $\texttt{Add}(Q, \langle Bad, k, must \rangle)$
4:  while $\neg\texttt{Empty}(Q)$ do
5:      $\langle m, i, p \rangle = \texttt{Pop}(Q)$
6:      if $(i = 0) \vee \texttt{Match}(R, m)$ then
7:          if $p = must$ then return false
8:          else
9:              $\texttt{AddReachable}(R)$
10:             $Q = \{\langle Bad, k, must \rangle\}$; continue
11:     $Ind = \neg\texttt{SAT?}(F_{i-1} \wedge \mathcal{T} \wedge m')$
12:     if $(i > 2) \wedge \neg Ind \wedge \texttt{IsEligible}(\langle m, i, p \rangle)$ then
13:         $\Gamma(\neg m) = \texttt{Support}(F_{i-2} \setminus (B \cup U), \mathcal{T}, \neg m)$
14:         if $\Gamma(\neg m) \neq$ NULL then
15:             for all $\varphi \in (\Gamma(\neg m) \setminus F_{i-1})$ do
16:                 $\texttt{Add}(Q, \langle \neg\varphi, i-1, may \rangle)$
17:             $\texttt{Add}(Q, \langle m, i, p \rangle)$; continue
18:         else
19:             if $p = may \wedge E[\neg m] \geq 1$ then
20:                 $U = U \cup \{\neg m\}$
21:                 $Q = \{\langle Bad, k, must \rangle\}$; continue
22:             $E[\neg m] = E[\neg m] + 1$
23:     if $\neg Ind$ then
24:         $u = \texttt{Predecessor}(m)$
25:         $\texttt{Add}(Q, \langle \texttt{Lift}(u), i-1, p \rangle)$
26:         $\texttt{Add}(Q, \langle m, i, p \rangle)$
27:     else
28:         $(\varphi, g) = \texttt{Generalize}(\neg m, i)$
29:         $\texttt{AddLemma}(\varphi, g)$
30: return true
```

introduce two different methods by which proof obligations are processed in `Truss`. The first is the *fallback behavior*. When processing an obligation using the fallback behavior, the algorithm behaves identically to Algorithm 2, except that it does not add may-proof obligations *i.e.,* line 15 is not executed. Alternatively, an obligation can be processed using support sets. In this case, the obligation $\langle m, i, p \rangle$ is processed by computing a support set $\Gamma(\neg m)$ and enqueuing may-proof obligations $\langle \varphi, i-1, may \rangle$ for all $\varphi \in \Gamma(\neg m)$. If a suitable support set cannot be found, then the obligation is processed using the fallback behavior.

For some obligations, the algorithm skips the support set computation and proceeds directly to the fallback behavior. Obligations that are processed using support sets are called *eligible*. Different eligibility criteria could be considered, and one alternative is discussed in Section VI-C. In this section, eligible obligations are $\langle Bad, k, must \rangle$ and may-proof obligations $\langle \neg\varphi, i, may \rangle$ where φ is a lemma already present in the inductive trace.

Pseudocode for the procedure is shown in Algorithm 4. `Truss_Block` is only called at level k when $\neg Bad$ is inductive relative to F_{k-2}. The algorithm's goal is to strengthen F_{k-1} until $\neg Bad$ is inductive relative to that frame. It begins by processing the obligation $\langle Bad, k, must \rangle$, which is added on line 3. We discuss the handling of this obligation first, and later move on to general obligations. The first step is to check if $\neg Bad$ is already inductive relative to F_{k-1} (line 11). If so, lines 28–29 are executed, adding a lemma $\neg Bad$ at a level $g \geq k$, and the algorithm terminates. Otherwise, it tries to compute a critical support set $\Gamma(\neg Bad)$ from F_{k-2}, excluding any blacklisted (*i.e.,* bad or ugly) lemmas (line 13). This may not succeed, as the property may be supported by those lemmas. In this case, the algorithm uses the fallback behavior, which results in adding a new must-proof obligation at level $i-1$ (lines 24–26).

Now, assume a support set is found. The algorithm adds obligations $\langle \neg\varphi, k-1, may \rangle$ for each $\varphi \in \Gamma(\neg Bad)$ (lines 15–16). The original obligation is also returned to the queue. Note that, since the lemmas in $\Gamma(\neg Bad)$ are in F_{k-2}, they are inductive relative to F_{k-3}. Since the obligations are enqueued at level $i = k-1$, each φ is inductive relative to F_{i-2}. The added may-proof obligations at level i are handled similarly, by computing a support set from F_{i-2} and enqueuing obligations at level $i-1$. This is the reasoning behind Lemma 1 below.

Lemma 1 *In* `Truss`*, for every proof obligation* $\langle \neg\varphi, i, p \rangle$ *with* $i \geq 2$*,* φ *is inductive relative to* F_{i-2}.

Proof: For $\langle Bad, k, must \rangle$, the proof is trivial. For obligations added on line 25, the proof follows from the properties of `Quip` and `IC3`. For obligations added on line 16, the obligation $\langle \neg\varphi, j, may \rangle$ is added at level $j = i-1$. We have $\varphi \in F_{i-2}$ by the behavior of `Support`. Since $\varphi \in F_{i-2}$ it is inductive relative to F_{i-3} *i.e.,* F_{j-2}. The lemma follows immediately. ■

We now describe the processing of an eligible obligation $\langle \neg\varphi, i, may \rangle$. Note that must-proofs other than $\langle Bad, k, must \rangle$ are not eligible, so the obligation is assumed to be a may-proof. The first step on lines 6–10 is to check if the obligation fails. This step is the same in `Quip_Block` and is also part of the fallback behavior. The next step is to check if φ is inductive relative to F_{i-1} (line 11). If so, the obligation is successfully discharged. Otherwise, by Lemma 1, φ is inductive relative to F_{i-2}. The algorithms tries to compute a support set for φ of the form:

$$\Gamma(\varphi) \subseteq F_{i-2} \setminus (B \cup U) \qquad (4)$$

where B and U represent the set of bad and ugly lemmas, respectively. This occurs on line 13.

If a support set is found, obligations are added for the supporting lemmas on lines 15–16. Note that due to the subtraction of F_{i-1} on line 15, only the lemmas in the critical support set are added.

Conversely, if a support set is not found, the algorithm tries to learn new lemmas to support φ. The rationale behind this behavior comes from the following corollary of Lemma 1.

Corollary 1 *If no support set of the form from Eq. 4 exists, all support sets* $\Gamma(\varphi) \subseteq F_{i-2}$ *include some lemma in* $B \cup U$.

Proof: Immediate from Lemma 1. ■

Corollary 1 does not imply that φ is non-inductive. However, promoting φ to level $i-1$ requires promoting blacklisted lemmas or learning new ones. The former is undesirable, so the algorithm uses its fallback behavior to learn new lemmas (lines 23–29). However, an effort limit is applied that restricts the algorithm to learn only one new lemma towards the goal of supporting φ. When a support set cannot be found, line 19 checks if the effort limit for φ has been exceeded. If so, the entire obligation queue is abandoned and φ is marked as ugly (lines 19–21). In other words, the second time $\langle \neg\varphi, i, may\rangle$ is popped from the queue, if φ is not inductive relative to F_{i-1} then φ is marked as ugly and the queue is abandoned.

Throughout this procedure, obligations may not be discharged due to a counter-example or due to effort limits. Consider the case where an obligation intended to push a lemma $\psi \in \Gamma(\varphi)$ forward is abandoned. The algorithm has no reason to continue pushing the other lemmas in $\Gamma(\varphi)$ either, since $\Gamma(\varphi) \setminus \{\psi\}$ is not necessarily a support set for φ. Intuitively, the corresponding obligations should be abandoned. However, it may also be the case that φ has no suitable support set, since ψ is now blacklisted. This means φ should be abandoned. These cascading failures can end up requiring a large number of obligations to be abandoned.

Rather than checking each individual obligation, the algorithm simply abandons all may-proof obligations when any one of them cannot be discharged. This leaves only $\langle Bad, k, must\rangle$ in the queue. The algorithm simply repeats all of the steps of computing support sets for the property, then for the supporting lemmas, and so on. In practice, re-computing support sets would be costly, so the most recently-computed support set for each lemma is cached. When the computations are repeated, the cached result is used unless it contains a blacklisted lemma. Therefore, only the support sets that have been invalidated are re-computed.

A corner case occurs when $i < 2$, as F_{i-2} does not exist. The algorithm resorts to the fallback behavior in this case. Additionally, for performance reasons, the fallback behavior is used when $i = 2$. This is because F_0 represents the initial states, and as such contains lemmas that are given as input rather than learned by the algorithm. These lemmas are expected to be non-inductive, so pushing them forward is undesirable. Therefore, the fallback behavior is used when $i \leq 2$. Note that in this case proof obligations are not subjected to effort limits.

C. Comparison with Quip and IC3

`Quip`, `IC3`, and `Truss` all repeatedly attempt to discharge $\langle Bad, k, must\rangle$ for increasing values of k. This represents an obligation to construct a bounded proof of the property at level k. All three algorithms repeat this process until converging to an inductive proof.

The algorithms differ in the additional reasoning they apply to accelerate convergence. In `IC3`, when an obligation $\langle m, i, must\rangle$ is discharged, $\langle m, i+1, must\rangle$ may be returned to the queue. This ensures that the CTI represented by m is blocked at higher levels, but causes the algorithm to expend effort learning multiple lemmas to block the same CTI. In `Quip`, upon discharging the same obligation by learning a lemma φ, a new obligation $\langle \neg\varphi, i+1, may\rangle$ may be added to the queue. This forces the algorithm to try to promote φ to block m at higher levels, even if it requires learning additional lemmas to support φ. This can be an expensive process and may only result in the algorithm discovering reachable states instead of blocking m at higher levels. The algorithms apply these consistently without regard to the particular lemmas or CTIs being considered.

`Truss` instead uses support sets to guide these decisions. Rather than consistently trying to push forward every lemma, support sets are used to identify a set of lemmas that might be close to forming an inductive invariant. In fact, the set of lemmas identified represents a portion of a bounded proof excluding any bad or ugly lemmas. It may be the case that this set of lemmas is only useful together *i.e.,* if one cannot be promoted, the entire set is not useful. Therefore, when a targeted lemma is not promoted, the algorithm detects and abandons obligations that are only valuable in conjunction with that lemma. In essence, the algorithm tries to identify the portion of the existing bounded proof that is likely to be inductive and support it until it becomes inductive. However, effort limits are used to limit this procedure.

All of these algorithms are forced to construct bounded proofs at higher and higher levels in order to "escape" the non-inductive lemmas they have learned and to learn new lemmas to replace them. In `IC3`, once a lemma is learned, no effort will ever be made to learn new lemmas to support it and push it forward. It will only be pushed forward if such lemmas are learned by chance. In `Quip`, an effort is made immediately upon learning a lemma to learn its supporting lemmas using may-proof obligations. However, after that process finishes, no further effort is made. `Truss` is able to identify important lemmas and then learn new lemmas to support them at any time. We believe this represents a significant extension of the algorithm's reasoning capabilities.

V. Experimental Results

All results presented in this section are executed on a single core of a Linux workstation with an i5-3570K 3.4 GHz CPU and 16 GB of RAM. We provide an experimental evaluation of `IC3`, `Quip`, and `Truss`. We have implemented both `Quip` and `Truss`[1] in `IImc` [7], [8]. For all algorithms, the backend SAT solver is `Glucose` [9], [10] as it was found to give a substantial runtime improvement over the other solvers available in `IImc`. Experiments are timed out after one hour.

We present results for problem instances in the HWMCC'15 benchmark set, excluding the proprietary circuits from Intel. The 126 benchmarks that were not solved by any solver in the competition are not considered, leaving 387 circuits. A further 122 circuits that were not solved by any of the evaluated algorithms are excluded. After pruning those circuits, the benchmark set contains 265 circuits.

In order to demonstrate that the "baseline" `Quip` approach is reasonable, it is compared against the `IC3` implementation provided by `IImc`, referred to as `IImc-IC3`. However, we disable several features that are not present in our implementations, including expansion of the initial states using forward Binary Decision Diagrams [11], equivalence propagation, and counter-examples to generalization [12]. All of the disabled features are applicable to `Quip` and `Truss`, but are not present in our implementation. These features are disabled so

[1]Source code and detailed results for each circuit are available at: ryanmb.bitbucket.io/truss

TABLE I
SUMMARY OF RESULTS

	SAFE SOLVED	SAFE TIME	UNSAFE SOLVED	UNSAFE TIME	TOTAL SOLVED	TOTAL TIME
IImc-IC3	175 (6)	77632	66 (5)	30131	241 (11)	107765
Quip	174 (3)	83450	56 (1)	66067	230 (4)	149517
Truss	183 (8)	57394	64 (5)	44147	247 (13)	101541

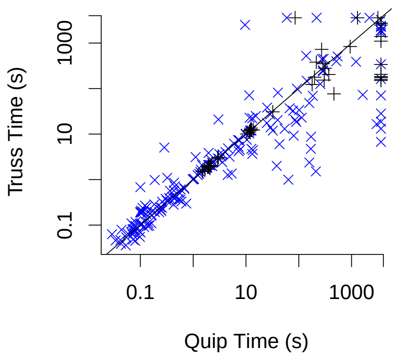


Fig. 1. Runtime comparison of Quip and Truss

as to limit the impact that the unrelated optimizations present in IImc-IC3 have on the results.

A summary of the results is shown in Table I. The columns SAFE SOLVED and UNSAFE SOLVED show the number of safe and unsafe instances solved by each algorithm, respectively. The number in parenthesis shows the number of unique instances solved. The TOTAL SOLVED column shows the total number of instances solved by each algorithm. The SAFE TIME and UNSAFE TIME columns show the total time spent by each algorithm on safe and unsafe instances respectively. The TOTAL TIME column shows the total time spent processing all instances. All times are in seconds.

The experiments show that our Quip implementation, while weaker in comparison to IImc-IC3, is competitive and represents a reasonable baseline for comparison. Since Quip is expected to outperform IC3 in typical cases, we expect this is due to other unrelated optimizations present in IImc-IC3 that could not readily be disabled. The experiments demonstrate that Truss offers a substantial improvement over Quip for both SAFE and UNSAFE instances. It also outperforms the highly-tuned IImc-IC3 implementation, especially on SAFE instances where it achieves a 1.35x speedup. Out of 265 circuits, Truss solves 247 instances compared to the 230 solved by Quip and processes the entire set 1.47x faster. Not counting the 11 instances uniquely solved by IImc-IC3, the speedup increases to an impressive 1.77x.

Figure 1 shows a detailed comparison of the runtime for each approach. It plots the runtime of Truss versus that of Quip for each of the benchmark circuits on a log-log scale. The blue marks indicate SAFE instances while the black marks indicate UNSAFE instances. It includes the 254 circuits that were solved by at least one of Quip or Truss. Points under the solid line indicate that Truss is faster, while points above it indicate that Quip is faster. It can be seen that Truss is faster than Quip in most cases. Indeed, 145 of the 254 points fall below the line. However, this

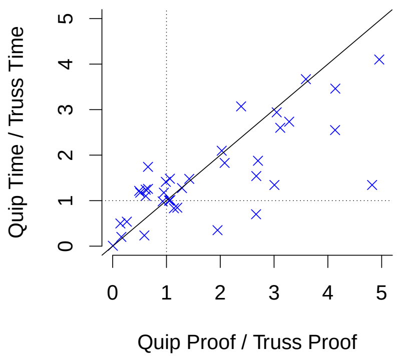


Fig. 2. Speedup versus reduction in proof size for challenging instances

significantly understates the benefits offered by Truss, as it truly shines on the more challenging problem instances. The benchmark sets contains 139 “easy” instances that were solved by both algorithms in under 12 seconds. Excluding the easy instances, Truss outperforms Quip on 81 out of 115 of the remaining “challenging” instances. Apparently, Truss introduces overhead for easy benchmarks, but pays off substantially for challenging ones.

The intuition behind Truss is that by promoting important lemmas, the algorithm is able to more quickly discover a proof. Naturally, it is expected to learn smaller proofs as a result. It may not always do so, as random factors can significantly impact the runtime and change the final proof. For instance, Quip may learn a very important lemma by chance that Truss never learns due to having a different inductive trace. Truss does not address the problem of learning better ones, so this problem is unavoidable.

To examine the relationship between runtime and proof size, Figure 2 plots the speedup for Truss versus the proof size reduction. It includes the 49 SAFE instances in the challenging set that were solved by both algorithms. The solid line indicates a 1:1 correlation between the two axes. Across these 49 instances, Truss finds proofs that are an average of 73.3% as large as those found by Quip. It can be seen that higher speedups tend to occur when Truss computes smaller proofs than Quip. This is unsurprising, as Truss tends to find a smaller proof by processing fewer obligations for more important lemmas.

To further examine this point, Table II reports the runtime of various operations for Quip and Truss. It considers the 83 challenging instances solved by both algorithms. The first column reports the total runtime. The PUSH TIME, MAY TIME, MUST TIME, and SUPPORT TIME columns report the time spent pushing, processing may-proofs (excluding calls to Support), and processing must-proof obligations, and computing support sets, respectively. The MAY PROOFS and MUST PROOFS columns report the number of may-proof and must-proof obligations processed, respectively. The SUPPORT CALLS column reports the number of calls to Support.

The data explains how Truss achieves better runtime performance. It spends similar amounts of time pushing lemmas and processing must-proofs. However, it processes fewer than one third as many may-proofs as Quip. Additionally, the overhead of computing support sets is small, accounting for

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE II
TIME SPENT PER OPERATION

	TOTAL TIME	PUSH TIME	MAY TIME	MAY PROOFS	MUST TIME	MUST PROOFS	SUPPORT TIME	SUPPORT CALLS
`Quip`	14000	2940	7108	1826812	2940	88983	0	0
`Truss`	10878	3317	3372	554233	2935	87616	517	135736

only 4.7% of the total runtime. Indeed, computing a support set is expected to cost substantially less than processing a proof obligation. Processing a proof obligation often results in learning a new lemma, thereby running generalization which may involve numerous SAT queries. Conversely, computing a support set involves only one SAT query. `Truss` avoids a large number of expensive generalization operations by performing a smaller number of less expensive support set computations.

VI. ALTERNATIVES AND FUTURE WORK

This section presents alternative implementations of `Truss` that combine the novel aspects in different ways. We also discuss ways that the implementation could be improved and aspects that may be applicable to other algorithms.

A. No Ugly Lemmas

An alternative version of `Truss` could use different criteria to define ugly lemmas. The simplest alternative is to increase the effort limit from 1. In the most extreme form, infinite effort could be allowed, thereby eliminating ugly lemmas altogether. A preliminary evaluation of this approach found it to perform poorly. The algorithm expends much more effort pushing forward lemmas that ultimately end up being marked as bad. Even increasing the effort limit slightly had a similar but less dramatic effect. Intuitively, it appears as though valuable lemmas tend to be easy to support in most cases, since `Truss` achieves better results when using a low effort limit.

B. Re-Enqueuing Obligations

`Truss` does not re-enqueue obligations in the manner `Quip` does on line 15 of Algorithm 2. The algorithm can be modified to accommodate this operation, though several variations are reasonable. For instance, it's unclear if re-enqueued lemmas should be subject to effort limits or not. A preliminary experimental evaluation found that several variations of this approach performed worse than `Truss`. One possible explanation is that the re-enqueue operation is to ensure important lemmas are available at higher levels. This is exactly the same reasoning behind adding lemmas from support sets. However, support sets contain more fine-grained information about which lemmas are important. Therefore the extra proof obligations from the re-enqueue operation may be less helpful than those added by `Truss`.

C. Using Support Sets for Non-Lemmas

Another alternative implementation could use different eligibility criteria, such as using support sets to discharge every proof obligation. In `Truss`, support sets are only used when an obligation represents a lemma in the inductive trace. However, Lemma 1 holds for every proof obligation, so it would be a reasonable to use support sets for every proof obligation. This procedure takes the goal of re-using of existing lemmas rather than learning new ones to the extreme.

D. Improved Solving under Assumptions

Algorithm 3 for computing lemma supports is based on incremental SAT solving with *many* assumptions, which may significantly slow down SAT-queries, see for example [13]. However, on problems where all assumptions can be cleanly separated into original problem literals and activation literals various solutions are possible [13], [14]. In the future we are planning to investigate the precise effect of assumptions on Algorithm 3 and adjust the back-end SAT solver accordingly. In addition, most state-of-the-art SAT solvers tend to propagate assumptions in the order they are received and it seems a good idea to experiment with different assumption orders. For example, by putting the activation literals in decreasing order of level for the corresponding lemma, the algorithm is likely to find a smaller critical support set.

VII. CONCLUSION

This work presents an `IC3`-based unbounded model checking algorithm called `Truss`. The algorithm detects a subset of lemmas in the inductive trace that participate in a bounded proof of the property and targeting the set for for pushing as a cohesive unit. Experiments on HWMCC'15 designs show a substantial speedup against the state-of-the-art.

REFERENCES

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *DAC 2015*, June 2015, pp. 1–6.

[2] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011.

[3] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," ser. FMCAD 2011, Austin, TX, 2011, pp. 125–134.

[4] A. R. Bradley, "Incremental, inductive model checking," in *2013 International Symposium on Temporal Representation and Reasoning*, Sept 2013, pp. 5–6.

[5] A. Ivrii and A. Gurfinkel, "Pushing to the top," in *FMCAD 2015*, Sept 2015, pp. 65–72.

[6] R. Berryhill, N. Veira, A. Veneris, and Z. Poulos, "Learning lemma support graphs in quip and IC3," in *Proceedings of the 2017 International Verification and Security Workshop (IVSW) 2017*, 2017.

[7] A. R. Bradley and F. Somenzi and Z. Hassan, "IImc: an Incremental Inductive model checker." [Online]. Available: https://github.com/mgudemann/iimc

[8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Incremental, inductive CTL model checking," in *CAV 2012*, 2012, pp. 532–547.

[9] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Joint Conference on Artifical Intelligence (IJCAI)*, San Francisco, CA, USA, 2009, pp. 399–404.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, 2003, pp. 502–518.

[11] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[12] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in ic3," in *FMCAD 2013*, Oct 2013, pp. 157–164.

[13] J. Lagniez and A. Biere, "Factoring out assumptions to speed up MUS extraction," in *SAT 2013*, 2013, pp. 276–292.

[14] G. Audemard, J. Lagniez, and L. Simon, "Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction," in *SAT 2013*, 2013, pp. 309–317.

K-Induction without Unrolling

Arie Gurfinkel
University of Waterloo
http://ece.uwaterloo.ca/~agurfink

Alexander Ivrii
IBM Research
alexi@il.ibm.com

***Abstract*—We present a flexible algorithmic framework `KIC3` that combines `IC3` and k-induction. The key underlying observation is that k-induction can be easily simulated by existing `IC3` implementations by following a slightly different counterexample-queue management strategy.**

I. Introduction

The principle of k-induction is the first successful technique for unbounded SAT-based model checking [1]. It is based on the following generalization of the usual one-step induction principle: a safety property φ is invariant (i.e., holds in all the reachable states of the system) if (a) φ holds in all states reachable in up to k steps, and (b) φ holds for k consecutive steps implies that it holds for $k+1$-steps. Interestingly, k-induction is complete when restricted to loop-free (or simple) paths. That is, any invariant φ is k-inductive for some k, when only loop-free paths of length k are considered. This gives rise to an unbounded model checking algorithm that repeatedly tries to prove that φ is k-inductive for increasing values of k.

Today, k-induction [1] remains a very important technique for unbounded model checking in both hardware and software domains [2]. A classical implementation of k-induction uses a SAT-solver to check a k-step unrolling of a transition relation, and ensures loop-freedom of counterexamples via additional unique state constraints. However, the scalability of the technique is limited by the depth k of the required unrolling. While combining k-induction with additional invariant synthesis (e.g., [3]) is beneficial, applicability of k-induction remains limited to properties that can be established with a small value of k.

`IC3`/`PDR` [4], [5] is currently the dominant SAT-based unbounded model checking technique. Pioneered by Bradley [4], `IC3` has became the definitive framework for developing SAT- and SMT-based model checking algorithms for both hardware and software verification. Given a safety property φ, `IC3` computes an inductive strengthening F of φ. That is, a formula F such that $\varphi \rightarrow F$ and F is inductive. Furthermore, when φ is not an invariant, then `IC3` produces a counterexample. One of the distinguishing features of `IC3` is that it does not explicitly build an unrolling of the transition relation: all reasoning is done over a single step.

As was pointed out in [6], the strengths of k-induction and `IC3` are complementary. Properties that are k-inductive for a small value of k (e.g., 3 or 4) and, therefore, are "easy" for k-induction, are not necessarily easy for `IC3`. More specifically, `IC3` is not guaranteed to terminate after exploring all k-depth counterexamples even when a property is k-inductive. Based on this observation, Jovanovic and Dutertre [6] presented an alternative model checking approach using the insights from both algorithms. However, their approach requires a significant modification of `IC3` and an unrolling-based check for k-induction. In this paper, we explore an alternative solution that tighter and more elegantly integrates k-induction within `IC3`.

The paper makes two contributions. First, we introduce a new algorithm, called `K-Ind`, to decide whether a given safety property φ is k-inductive. The algorithm is based on the insights from `IC3`, and does not explicitly unroll the transition relation. Whenever φ is k-inductive, `K-Ind` returns an inductive strengthening of φ; otherwise, it returns a counterexample to k-induction. Perhaps the most interesting feature of `K-Ind` is that it does not rely on an expensive unique-states constraint to guarantee that only loop-free paths are considered. Furthermore, since it is embedded in the `IC3` framework, it benefits from all the usual `IC3` optimizations such as inductive generalization and generalization of predecessors.

Second, we introduce a framework, called `KIC3`, that combines `IC3` and k-induction in a single `IC3`-like algorithm. Our key insight is that k-induction can be simulated by a specialized counterexample-queue management strategy. This enables `KIC3` to immediately be compatible with all known `IC3`-optimizations and extensions (e.g., [5], [7]–[9]). The algorithm is parameterized by the degree of k-inductive reasoning, where k-induction can be used to simply validate k-inductive conjectures, construct k-inductive strengthening, or recursively block counterexamples to induction.

The rest of the paper is organized as follows. In Section II, we review the necessary background about k-induction and `IC3`. In Section III, we present the `K-Ind` algorithm, and in Section IV, we present the `KIC3` framework. Finally, we conclude the paper with an overview of related work in Section V, an experimental evaluation in Section VI, and conclusion in Section VII.

II. Background

A. Propositional Satisfiability

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

This work is partially funded by an IBM Faculty Award.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathit{Init}(\mathcal{V})$ and $\mathit{Bad}(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $\mathit{Tr}(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that Init and Tr are in CNF.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$\mathit{Init}(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \wedge \mathit{Bad}(\vec{v}_N) \qquad (1)$$

P is UNSAT (or SAFE) iff there exists a formula $\mathit{Inv}(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$\mathit{Init}(\vec{v}) \to \mathit{Inv}(\vec{v}) \qquad \mathit{Inv}(\vec{v}) \wedge \mathit{Tr}(\vec{v}, \vec{v}') \to \mathit{Inv}(\vec{v}') \qquad (2)$$

$$\mathit{Inv}(\vec{v}) \to \neg \mathit{Bad}(\vec{v}) \qquad (3)$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*.

B. k-invariants and k-induction

An invariant over-approximates all the reachable states of the transition relation; however, there is no efficient way to check that a formula is an invariant. An inductive invariant is an invariant that is easy to validate.

A formula φ is called a *k-invariant* if it over-approximates all states reachable up to k-steps. That is,

$$\forall 0 \leq N \leq k \cdot \left(\mathit{Init}(\vec{v}_0) \wedge \bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \to \varphi(\vec{v}_N) \quad (4)$$

Note that any formula F that over-approximates the initial states is a 0-invariant. A formula φ is *k-inductive invariant* if it is a k-invariant and

$$\left(\bigwedge_{i=0}^{k} \varphi(\vec{v}_i) \wedge \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \to \varphi(\vec{v}_{k+1}) \qquad (5)$$

The definition of k-induction naturally extends to k-induction relative to some 0-invariant formula F, by replacing all but the last occurrence of φ in eq. (5) with $(\varphi \wedge F)$.

Neither induction nor k-induction are complete. That is, there are a transition system Tr and an invariant φ such that φ not k-inductive for any k. However, as shown in [1], k-induction is complete when restricted to loop-free (or simple) paths. That is, the antecedent of eq. (5) is strengthened to ensure that the sequence $\vec{v}_0, \ldots, \vec{v}_{k+1}$ is loop free.

C. Description of IC3

We give a brief description of IC3 that highlights some steps, but omits many crucial optimizations. We refer the reader to [10] for an overview of available optimizations and their possible implementations.

IC3 maintains a sequence of sets of clauses $F_0, F_1, \ldots$ called an *inductive trace*. Each set of clauses F_i in a trace is called a *frame*, each clause $c \in F_i$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to Init and that $\mathit{Init} \to \neg \mathit{Bad}$. IC3 maintains the following invariant:

$$F_i \to \neg \mathit{Bad} \qquad F_{i+1} \subseteq F_i \qquad F_i \wedge \mathit{Tr} \to F'_{i+1}$$

That is, each element of the trace is safe, the trace is syntactically monotone, and each F_{i+1} is inductive relative to F_i.

```
Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
       $(f_0 - 1)$-inductive
1  Add($Q, \langle s_0, f_0 \rangle$)
2  while $\neg$Empty($Q$) do
3      $\langle s, f \rangle \leftarrow$ Pop($Q$)
4      assert $\neg s$ is $(f-1)$-invariant
5      if $f = 0$ then
6          return CEX
7      if SAT?($\neg s \wedge F_{f-1} \wedge Tr \wedge s'$) then
8          $t \leftarrow$ ExtractPredecessor($s$)
9          Add($Q, \langle t, f-1 \rangle$)
10         Add($Q, \langle s, f \rangle$)
11     else
12         $\langle c, g \rangle \leftarrow$ Generalize($\neg s, f$)
13         AddLemma($c, g$)
14         if $g < f_0$ then
15             Add($Q, \langle s, g+1 \rangle$)
16 return BLOCKED
```

Fig. 1. IC3 Blocking (IC3_Block).

Fig. 1 presents the blocking procedure of IC3. The inputs to IC3_Block are a state s_0 and a level f_0, with $\neg s_0$ already known to be $(f_0 - 1)$-invariant. The procedure either strengthens the inductive trace and returns BLOCKED indicating that $\neg s_0$ is f_0-invariant, or finds a counterexample trace witnessing that s_0 is reachable from Init and returns CEX.

IC3_Block maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle s, f \rangle$ where s is a cube over state variables and f is a *level*. At each point of the execution, it considers a proof obligation $\langle s, f \rangle$ with the smallest level f, and attempts to prove that s is reachable in f steps. If $f = 0$ then there is a real counterexample. Otherwise, it makes a *predecessor* query $SAT?(\neg s \wedge F_{f-1} \wedge Tr \wedge s')$ that checks whether a state in s can be reached from a state in F_{f-1}. If the result is satisfiable, it adds a predecessor of s as a new proof obligation at level $f - 1$. If the result is unsatisfiable, it learns a new lemma c, such that $\mathit{Init} \to c$, $c \to \neg s$ and $c \wedge F_{f-1} \wedge \mathit{Tr} \to c'$, and adds c to F_j, for all $j \leq f$. In other words, the lemma c represents a new over-approximation, and in particular demonstrates why the state s cannot be reached in up to f steps from the initial states. An important optimization is to re-enqueue s at the lowest unknown frame.

Each time that IC3 blocks Bad for one additional level, it enters a propagation phase, in which for each level f and for each lemma $c \in F_f \setminus F_{f+1}$, it executes the following SAT query: $SAT?(c \wedge F_f \wedge Tr \wedge \neg c')$. Whenever this query is unsatisfiable, the lemma c can be added to frame F_{f+1}.

IC3 terminates if at any point of the execution $F_{f-1} = F_f$ and $F_f \to \neg \mathit{Bad}$. In this case F_f represents an inductive invariant establishing the correctness of the property.

```
Input: A number $k$, a $k$-invariant $\ell$, a 0-invariant $F_0$
1  $F \leftarrow F_0$
2  Add$(Q, \langle \neg\ell, k \rangle)$
3  while $\neg$Empty$(Q)$ do
4      $\langle s, f \rangle \leftarrow$ Pop$(Q)$
5      assert $\neg s$ is $(f-1)$-invariant
6      if $f = 0$ then
7          if $s \cap Init \neq \emptyset$ then return CEX
8          else return K-CTI
9      $F \leftarrow F \wedge \neg s$
10     if SAT?$(F \wedge Tr \wedge s')$ then
11         $t \leftarrow$ ExtractPredecessor$(s)$
12         assert $t \rightarrow F$
13         Add$(Q, \langle t, f-1 \rangle)$
14         Add$(Q, \langle s, f \rangle)$
15     else
16         $c \leftarrow$ Generalize$(\neg s)$
17         $F \leftarrow F \wedge c$
18         $G \leftarrow G \wedge c$
19 return $(\text{BLOCKED}, G)$
```

Fig. 2. k-induction without unrolling (K-Ind).

III. K-INDUCTION WITHOUT UNROLLING

In this section we present K-Ind, an algorithm for deciding whether a given k-invariant formula is k-inductive. Unlike the traditional approach that reduces k-induction to BMC by unrolling the transition relation, our algorithm is based on IC3, and maintains only a single copy of the transition relation. In addition, unlike the traditional approach, K-Ind guarantees loop-free paths without introducing expensive unique-state constraints. In the rest of this section, we present the algorithm, argue for its correctness, and illustrate it on an example.

A. The Algorithm

The pseudo-code of K-Ind is shown in Fig. 2. The inputs to K-Ind are a number k determining the depth of induction, a k-invariant formula ℓ, and a 0-invariant F_0. For simplicity of presentation, we require that ℓ is a clause. The algorithm returns one of three values: CEX to indicate that ℓ is not $(k+1)$-invariant, K-CTI to indicate that ℓ is not k-inductive relative to F_0, and a tuple $(\text{BLOCKED}, G)$ to indicate that ℓ is k-inductive relative to F_0, and G is an inductive strengthening of ℓ (i.e., G contains ℓ and is inductive relative to F_0.

K-Ind closely follows the blocking procedure of IC3 (shown in Fig. 1) with several important differences that are highlighted next. First, all SAT-queries are made relative to the single frame F. This can alternatively be explained as IC3_Block in which all frames are the same and do not necessarily form an inductive trace. Second, a lemma learned at any stage of the algorithm holds for all frames. Hence, discharged proof obligations are not re-enqueued to higher levels. In particular, the priority queue Q acts as a LIFO stack. Third, the level of a proof obligation represents how much remaining budget it has rather than the frame on which it should be blocked. Fourth, the assertion on line 5 holds. Fifth, the negation of the predecessor to be blocked is assumed during blocking of all of it descendants (line 9). Lastly, the generalization of predecessors is done with respect to the frame F so that the assertion on line 12 holds.

B. Correctness

In the following, we establish the correctness of K-Ind. Our correctness argument is partitioned into two cases: (1) K-Ind returns CEX or K-CTI, and (2) it returns BLOCKED.

Lemma 1 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be a 0-invariant. If* K-Ind(k, ℓ, F_0) *returns* CEX, *then ℓ is not a $(k+1)$-invariant, and if it returns* K-CTI *then ℓ is not k-inductive relative to F_0.*

Proof: Similarly to IC3_Block, whenever K-Ind reaches line 6, the queue Q contains a loop-free sequence of $k+1$ states consistent with F_0 and satisfying the transition relation. This sequence witnesses that ℓ is not k-inductive. Furthermore, if it intersects with the initial state, then ℓ does not hold after $(k+1)$ steps of the tranistion relation. Hence, ℓ is not $(k+1)$-invariant. ■

Lemma 2 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be a 0-invariant. If* K-Ind(k, ℓ, F_0) *returns a tuple* $(\text{BLOCKED}, G)$*, then G is an inductive strengthening of ℓ relative to F_0.*

Proof: By construction, G contains ℓ and is inductive relative to F. Whenever K-Ind terminates with BLOCKED, every state that has ever been added to Q is blocked. Hence, for every clause c in $F \setminus F_0$ there is a stronger clause d in G. Thus, G is also inductive relative to F_0. ■

Theorem 1 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be 0-invariant. Then, assuming that* Generalize$(\neg s)$ *always returns $\neg s$,* K-Ind(k, ℓ, F_0) *terminates and returns* BLOCKED *iff ℓ is k-inductive relative to F_0,* CEX *iff ℓ is not $(k+1)$-invariant, and* K-CTI *iff ℓ is not k-inductive but $(k+1)$-invariant.*

Proof: We only need to show termination. The rest follows from Lemma 1 and Lemma 2. The number of iterations of the outer loop is bounded by the number of clauses (or cubes). At every iteration of the loop, either a new predecessor is added to the queue Q, or a new clause is added to frame F. No predecessor is added more than once. All the clauses in F are distinct. ■

The assumption that Generalize$(\neg s)$ always returns $\neg s$ in Theorem 1 is needed only to guarantee that K-CTI is returned whenever ℓ is not k-inductive. Removing this assumption makes the algorithm stronger by allowing it to find an inductive strengthening of ℓ even when ℓ is not k-inductive (of course, this is only possible when ℓ is an invariant).

Interestingly, K-Ind is complete in the sense that if ℓ is an invariant, then there is a k such that K-Ind$(k, \ell, \top)$ returns BLOCKED. This follows from the fact that K-Ind only considers loop-free counterexamples to k-induction. Note that restriction to loop-free paths follows from assuming the negation of a state to be blocked (line 9). This is a simpler alternative to a traditional approach of encoding loop-freedom of a path via an explicit unique-states constraint.

C. An Example

In this section, we illustrate K-Ind and highlight the difference with IC3_Block on an example. Consider the following transition system P

$$\begin{aligned}\mathcal{V} &= \{x, a, b, c\}\\ Init &\equiv x \wedge a \wedge b \wedge c\\ Tr &\equiv (x' = \neg x \vee a \vee b \vee c) \wedge\\ &\qquad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0)\\ Bad &\equiv \neg x\end{aligned}$$

Note that $a = 1$ and $x = 1$ are invariants of P, while $b = 1$ and $c = 1$ are not. In fact, $c = 1$ only holds on the initial cycle, and $b = 1$ only holds on the first two cycles. Consider the property $\ell \equiv \neg Bad = (x = 1)$. ℓ is not 1-inductive but is 2-inductive.

We begin by illustrating a run of K-Ind with inputs: $\ell = \{x = 1\}$, $k = 2$ and $F_0 = \top$. On the first iteration of the loop, F is updated to (x), and the predecessor query for s_0 relative to F yields a SAT query:

$$\begin{aligned}&(x) \wedge (x' = \neg x \vee a \vee b \vee c) \wedge\\ &\qquad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0) \wedge (x' = 0)\end{aligned}$$

This query is satisfiable. This, in particular, shows that ℓ is not 1-inductive. The corresponding predecessor is

$$t = \{x = 1, a = 0, b = 0, c = 0\}$$

On the second iteration of the loop, F is updated to $(x) \wedge (\neg x \vee a \vee b \vee c)$, and the predecessor query for t relative to F yields a SAT query

$$\begin{aligned}&(x) \wedge (\neg x \vee a \vee b \vee c) \wedge (x' = \neg x \vee a \vee b \vee c) \wedge\\ &\quad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0) \wedge (x' = 1) \wedge\\ &\qquad\qquad (a' = 0) \wedge (b' = 0) \wedge (c' = 0)\end{aligned}$$

This query is unsatisfiable. Assuming that Generalize does not remove any literals, K-Ind learns the lemma $(\neg x \vee a \vee b \vee c)$ and adds it to F. With the proof obligation t being blocked, the algorithm re-examines ℓ, and makes the predecessor query

$$\begin{aligned}&(x) \wedge (\neg x \vee a \vee b \vee c) \wedge\\ &\quad (x' = \neg x \vee a \vee b \vee c) \wedge (a' = a \vee b) \wedge (b' = c) \wedge\\ &\qquad\qquad (c' = 0) \wedge (x' = 0)\end{aligned}$$

This query is also unsatisfiable. The algorithm outputs BLOCKED, with the constructed 1-inductive strengthening of φ being $\psi = x \wedge (\neg x \vee a \vee b \vee c)$. Finally, we note that generalization relative to F (the procedure Generalize) could have also yielded the lemma $(a = 1)$ (resulting in the strengthening $(x \wedge a)$), but could *not* have yielded lemma $(b = 1)$ since $(b = 1)$ is not inductive relative to $(x) \wedge (\neg x \vee a \vee b \vee c)$.

Next, consider IC3 on the same example. The blocking procedure of IC3 similarly finds t as a predecessor of s, but makes the next predecessor query relative to $F_0 = Init$. More importantly, it calls Generalize on $(\neg x \vee a \vee b \vee c)$ relative to F_0, possibly learning the lemma $(b = 0)$ instead. Then, IC3 concludes that s_0 is blocked on level 2. However, since $(x) \wedge (b)$ is not an inductive invariant, IC3 needs to unfold the trace for an additional frame and continue blocking s on frame 3.

This example shows that K-Ind guarantees to strengthen a k-inductive property to an inductive one, while IC3 does not provide any such guarantees.

```
Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
       $(f_0 - 1)$-inductive
1 $res \leftarrow$ UNKNOWN
2 while $res =$ UNKNOWN do
3     $strategy \leftarrow$ AdjustStrategy()
4     $res \leftarrow$ BlockUsingStrategy($s_0, f_0, strategy$)
5 return $res$
```

Fig. 3. KIC3 Top-level blocking (KIC3_Block).

IV. KIC3: K-INDUCTIVE IC3 ALGORITHM

In this section, we describe the KIC3 framework that unifies IC3 and k-induction model checking algorithms. The core of KIC3 is a blocking procedure that integrates IC3_Block with a variant of k-induction. This procedure is then incorporated into a flexible approach for blocking proof obligations and for pushing existing lemmas forward.

A. Top-level blocking in KIC3

A pseudo-code for the top-level blocking procedure, KIC3_Block of KIC3 is shown in Fig. 3. The procedure takes as input a state s_0 and a level f_0 and assumes $\neg s_0$ is an $(f_0 - 1)$-invariant. The procedure outputs either BLOCKED to indicate that $\neg s_0$ is f-invariant, or CEX to indicate that s_0 is reachable from *Init*. As in IC3, KIC3 maintains an inductive trace $F_0, F_1, \ldots$ that is updated throughout the blocking process. Internally, KIC3_Block implements a portfolio approach, delegating lower-level blocking to other procedures such as IC3_Block (shown in Fig. 1), or KIC3_Block_Kind (shown in Fig. 3 and described later in this section). Note that the internal blocking procedure might return UNKNOWN to indicate that it has given up before finding a solution.

B. k-induction blocking in KIC3

Fig. 4 presents the k-inductive blocking procedure KIC3_Block_Kind of KIC3. In addition to a state s_0 and a level f_0, KIC3_Block_Kind requires two parameters: k_0 – the induction depth, and m_0 – the maximum number of predecessors to k-induction to be blocked by an external blocking procedure. The output of KIC3_Block_Kind is one of BLOCKED, CEX or UNKNOWN. As s_0 is only known to be $(f_0 - 1)$-inductive, the actual induction depth k is set to the smaller of the two values k_0 and f_0 (line 1). Our algorithm is strongly reminiscent of both IC3_Block and K-Ind, with several important differences that are described below.

First, all SAT queries (line 16) are performed relative to the same frame F_{f_0-1}. The level f of a proof obligation $\langle s, f\rangle$ has a slightly different interpretation than in IC3_Block: s is guaranteed to be unreachable from *Init* in $f - 1$ steps or less (assertion on line 6). The same queue management strategy as in IC3_Block is used. That is, proof obligations with lowest levels are chosen first. As in K-Ind, all the learned lemmas hold up to level f_0. The discharged proof obligations do not need to be re-enqueued, and the priority queue Q acts as a LIFO stack. However, unlike K-Ind, the query on line 16 does not fully implement loop-free paths, and instead a more relaxed condition of IC3_Block is used.

```
Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
       $(f_0 - 1)$-inductive; parameters $k_0$ and $m_0$
1  $k \leftarrow \texttt{min}(k_0, f_0)$
2  $m \leftarrow 0$
3  $\texttt{Add}(Q, \langle s_0, f_0\rangle)$
4  while $\neg\texttt{Empty}(Q)$ do
5      $\langle s, f\rangle \leftarrow \texttt{Pop}(Q)$
6      assert $\neg s$ is $(f-1)$-invariant
7      if $f = f_0 - k$ then
           // Found cex to $k$-induction
8          if $m < m_0$ then
9              $m \leftarrow m + 1$
10             if $\texttt{BlockInRange}(s, f_0 - k, f_0) =$ CEX
               then
11                 return CEX
12         else if $(f = 0) \wedge (s \cap Init \neq \emptyset)$ then
13             return CEX
14         else
15             return UNKNOWN
16     if SAT?$(\neg s \wedge F_{f_0 - 1} \wedge Tr \wedge s')$ then
17         $t \leftarrow \texttt{ExtractPredecessor}(s)$
18         $\texttt{Add}(Q, \langle t, f - 1\rangle)$
19         $\texttt{Add}(Q, \langle s, f\rangle)$
20     else
21         $\langle c, g\rangle \leftarrow \texttt{Generalize}(\neg s, f_0)$
22         $\texttt{AddLemma}(c, g)$
23 return BLOCKED
```

Fig. 4. `KIC3` blocking using k-induction (`KIC3_Block_Kind`).

```
Input: A state $s_0$, levels $f_0$ and $f_1$ s.t. $s_0$ is
       $(f_0 - 1)$-invarant
1  for $f = f_0, \ldots, f_1$ do
2      if $(f = 0) \wedge (s_0 \cap Init \neq \emptyset)$ then
3          return CEX
4      if $(f \neq 0) \wedge (\texttt{KIC3\_Block}(s_0, f) =$ CEX$)$ then
5          return CEX
6  return BLOCKED
```

Fig. 5. `KIC3` Top-level blocking in a range of frames (`BlockInRange`).

Second, when a proof obligation $\langle s, f\rangle$ at level $f = f_0 - k$ is examined, and consequently a K-CTI is discovered, the algorithm may attempt to recover by blocking this counterexample to k-induction. Since $\neg s$ is $(f_0 - 1)$-invariant, s needs to be blocked in every level in the range $[f_0 - k, f_0]$ (see line 10). The implementation of `BlockInRange` is shown in Fig. 5. As usual, we require that $\neg s_0$ is $(f_0 - 1)$-invariant. The procedure iterates over levels from f_0 to f_1 and calls `KIC3_Block` to block s_0 at the corresponding level. It returns BLOCKED if s_0 is blocked on all levels in $[f_0, f_1]$ (and hence $\neg s_0$ is f_1-invariant), and CEX otherwise.

Third, there is a parameter to limit the number of K-CTIs considered. When the number of K-CTIs reaches the maximum number m_0, `KIC3_Block_Kind` returns UNKNOWN. Note that whenever $m_0 = 0$, the external blocking procedure is not used at all, and `KIC3_Block_Kind` returns UNKNOWN (or possibly CEX) as soon as the first K-CTI is discovered. This limits the algorithm to only learn "high-quality" lemmas that hold up to level f_0, at the risk of eventually returning UNKNOWN sooner. On the other hand, when $m_0 = \infty$, all the K-CTIs are blocked with an external blocking procedure. In this case, `KIC3_Block_Kind` is also guaranteed to return either BLOCKED or CEX (and to never return UNKNOWN).

C. Correctness and Termination

In this section, we argue the correctness of `KIC3_Block_Kind`. First, the assertion on line 6 holds: the top-level proof obligation s_0 satisfies the assertion by assumption, and other proof obligations are added on line 18 and satisfy the assertion due to the following lemma.

Lemma 3 *Let s be a state, and $f \geq 0$ be a natural number, such that $\neg s$ is f-invariant. Let t be a predecessor of s. Then $\neg t$ is $(f-1)$-invariant.*

Proof: By contradiction. Assume $\neg t$ is not $(f-1)$-invariant. Then, there is a counterexample trace π of length at most f that reaches t from the initial states. Since t is a predecessor of s, there is a one-transition extension of π that shows that $\neg s$ is not f-invariant. ■

Second, whenever `KIC3_Block_Kind` reaches line 13, the queue Q contains a sequence π of $k+1$ states satisfying the transition relation, with the first state in the sequence intersecting the initial states, and the last state intersecting s_0. The path π shows that s_0 is reachable from *Init* in $k+1$ steps.

Third, whenever the algorithm reaches line 23, correctness is argued as in `IC3`: each lemma c added in line 22 satisfies $Init \rightarrow g$ and $g \wedge F_{f_0 - 1} \wedge Tr \rightarrow g$, and, hence, is inductive relative to $F_{f_0 - 1}$.

Fourth, whenever `KIC3_Block_Kind` calls `BlockInRange` recursively with a state s, by construction `BlockInRange` always calls an internal blocking procedure at the lowest level at which s is not yet blocked. Thus, the pre-conditions of the internal blocking procedure are satisfied.

Finally, the recursion of `BlockInRange` is well founded. Suppose that `KIC3_Block_Kind`(s_0, f_0) calls `BlockInRange` which, in turn, calls `KIC3_Block_Kind`(s_1, f_1). Then either $f_1 < f_0$, or $f_1 = f_0$ and, thus, `BlockInRange` has already blocked s_1 at level $f_0 - 1$ and learned a new lemma. Thus, in both cases the recursion makes a progress and must terminate.

D. Discussion

It is interesting to contrast the blocking strategies in `IC3` and `KIC3`. `IC3_Block` blocks each proof obligation at the lowest level it is yet unknown. Thus, if $\langle s, f\rangle \in Q$ is a proof obligation and t is a predecessor of s, then `IC3_Block` recursively attempts to block t at level $f-1$, and, if successful, blocks t at level f as well. On the other hand, `KIC3_Block_Kind` attempts to directly block t at level f, without blocking it at level $f-1$ first. Thus, from a high-level perspective, `KIC3_Block_Kind` is a variant of `IC3_Block`, with a different counterexample-queue management strategy.

By always making SAT queries relative to the frame $F_{f_0 - 1}$, `KIC3` essentially ignores all lemmas not in $F_{f_0 - 1}$. On the one hand, this may force it to spend more effort on blocking the top-level proof obligation $\langle s_0, f_0\rangle$. On the other hand, all learned lemmas automatically hold up to level f_0. Intuitively, since the lemmas are true for more steps of the transition relation they are of a "higher-quality", i.e., more likely to be part of the final inductive invariant.

Note that while KIC3_Block_Kind is similar to K-Ind, it does not fully incorporate the search for loop-free paths. This might be important when proof obligations are not enqueued at their lowest unknown levels. In particular, KIC3_Block_Kind may fail to find an inductive strengthening of $\neg s_0$, even when s_0 is k-inductive relative to F_{f_0-1}. Addressing this deficiency requires an ability to remove lemmas from frames in an IC3 framework. Developing support for this feature is an interesting direction for future work.

Another interesting technical dilemma reflects predecessor generalization on line 17 of KIC3_Block_Kind. More precisely, when the SAT query SAT?$(\neg s \wedge F_{f_0-1} \wedge Tr \wedge s')$ is satisfiable, with $\bar{t}$ being the predecessor of s, it is customary to generalize $\bar{t}$ to a larger set of states t such that any state in t leads to s [5]. In practice, it is not clear whether it is desirable to additionally enforce that $t \rightarrow F_{f_0-1}$ and $t \rightarrow \neg s$. On the one hand, generalizing predecessors with respect to $\neg s \wedge F_{f_0-1}$ avoids spurious K-CTIs. On the other hand, it significantly increases the sizes of proof obligations considered. We do not take these additional constraints into account in our experiments.

E. Portfolio blocking strategies

In this section, we describe the portfolio blocking strategies used in the experimental evaluation. Given a state s_0 to be blocked at level f_0, the $B(k_0, m_0)$-strategy with $1 \leq k_0 \leq \infty$ and $0 \leq m_0 \leq \infty$ is defined as follows:

$B(k_0, m_0)$

1) Block (s_0, f_0) using KIC3_Block_Kind with the induction depth k_0, the maximum number of K-CTIs m_0, and the procedure BlockInRange realized by IC3_Block;
2) If the previous procedure returns UNKNOWN, block (s_0, f_0) using IC3_Block.

There are several important special cases of this strategy. First, when $k = \infty$, KIC3_Block_Kind is called with the largest induction depth (in other words, f_0) allowed for blocking (s_0, f_0). Second, when $m_0 = 0$, KIC3_Block_Kind returns UNKNOWN as soon as the first K-CTI is discovered, in the process only learning lemmas at levels at least f_0. Third, when $m_0 = \infty$, KIC3_Block_Kind blocks all counterexamples to k-induction using IC3_Block, and IC3_Block is never called directly.

F. Pushing in KIC3

Our generic framework for blocking a state at a specific level can also be used in the pushing stage of IC3. The pseudocode of KIC3_Push is shown in Fig. 7. Lines 1–5 implement the traditional pushing procedure: when a lemma $c \in F_f \setminus F_{f+1}$ is inductive relative to F_f, it is also added to the frame F_{f+1}. Interestingly, this process can be reinterpreted as calling KIC3_Block_Kind with parameters $k_0 = 1$ and $m_0 = 0$. On lines 6–7, we propose to additionally push lemmas at the last level of the inductive trace using KIC3_Block_Kind with a larger value of k_0 (and m_0 still equal to 0).

```
1 $N \leftarrow$ Level$((\neg Bad))$
2 for $f = 1, \ldots, N$ do
3     for all lemmas $c \in F_f \setminus F_{f+1}$ do
4         if $F_f \wedge c \wedge Tr \Rightarrow c'$ then
5             $F_{f+1} \leftarrow F_{f+1} \cup \{c\}$
6 for all lemmas $c \in F_N$ do
7     KIC3_Block$(\neg c, f+1)$
```

Fig. 6. KIC3 Pushing (KIC3_Push).

There are many alternative ways to integrate KIC3_Block_Kind into the pushing stage of IC3. However, as in general KIC3_Block_Kind$(\neg c, f+1)$ can add new lemmas to the level $f+1$, additional care must be taken to guarantee termination of the pushing stage.

V. Related and Future Work

There is a large body of work on automating k-induction using SAT-based reasoning and on unbounded model checking using IC3. We focus only on the most closely related work.

It is well-known that k-induction principle is stronger than induction. In fact, k-induction is complete when restricted to loop-free paths while induction is not. Bjørner et al. [11] show that any k-inductive property can be converted into an inductive property by interpolation. Thus, the size of an inductive property is linear in the size of the resolution proof of the corresponding k-inductive property. Our K-Ind algorithm is a constructive proof of this fact. Given a k-inductive property, K-Ind constructs an inductive certificate in CNF.

K-Ind is built on top of the blocking procedure of IC3 and shares many similarities with it. When the property φ under consideration is k-inductive, both K-Ind (instantiated with the induction depth at least k) and IC3 are guaranteed to terminate and to discover a suitable inductive strengthening of φ. However, K-Ind is guaranteed to only learn k-inductive lemmas, while IC3 provides no such guarantees. In particular, the convergence depth of IC3 might be significantly larger than k. We believe that this is an important theoretical advantage of K-Ind over IC3. Unfortunately, our KIC3_Block_Kind in the KIC3 framework does not guarantee to converge in k steps, making it closer to IC3_Block than to K-Ind. Addressing this deficiency is an interesting topic for future work.

In Quip [9], a variant of IC3, a maximal inductive subset of all the lemmas is explicitly computed and maintained in a separate frame. This guarantees that Quip converges as soon as the trace contains an inductive subset. It is interesting to extend KIC3 to guarantee convergence as soon as the trace contains a k-inductive subset. Using KIC3_Block_Kind for pushing, as suggested in Section IV-F, is a step in that direction.

The PD-Kind algorithm of Jovanovic and Dutertre [6] is closest to ours, and has inspired our work. The main difference is that we have tried to integrate k-induction into IC3 with the fewest modifications of the IC3 framework. For example, PD-Kind requires unrolling the transition relation for validating k-induction queries, while KIC3 does not. Unfortunately, a direct experimental comparison of KIC3 and PD-Kind is difficult, as PD-Kind is implemented at the level

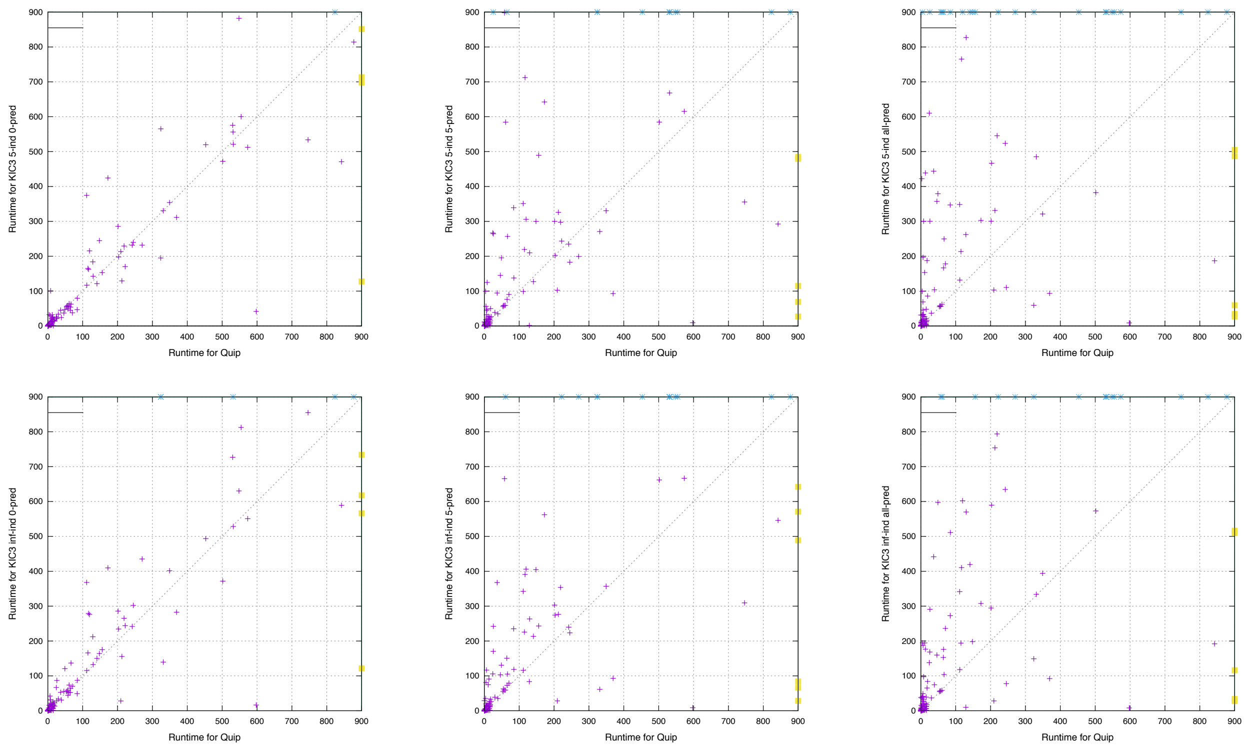


Fig. 7. Summary of experimental results comparing `KIC3` (y-axis) with `Quip` (x-axis) on benchmarks from HWMCC'15.

of SMT and does not target hardware benchmarks. Further, in our experience, a direct implementation of PD-KIND for hardware does not scale, as unrolling the transition relation has a huge negative effect on many benchmarks. At the same time, using k-induction for pushing clauses on the last frame, as we suggest in `KIC3_Push` in Section IV-F, while blocking all predecessors to k-induction using `IC3_Block`, is closely related to the `PUSH` procedure in `PD-Kind`. Overall, adapting the ideas of `PD-Kind` to hardware is a non-trivial task, and to some extent our algorithm can be viewed as a step in this direction.

Recall that `KIC3_Block_Kind` can be seen as `IC3_Block` with a specialized counterexample-queue management. Alternatively, `KIC3_Block_Kind` can also be seen as a form of *abstraction*. Whenever a proof obligation $\langle s, f \rangle$ should be blocked, traditionally, we check whether $\neg s$ is inductive relative to F_{f-1}. However, any abstraction of F_{f-1} can be used as well. For example, using only lemmas that are also in F_f as the abstraction closely corresponds to `KIC3`.

VI. Experiments

The techniques presented in this paper are implemented on top of `Quip` (a variant of `IC3` presented in [9]) in the IBM formal verification tool *IBM RuleBase SixthSense Edition* [12], [13]. All experiments were performed on a 2.13Ghz Linux-based machine with Intel Xeon E7-4830 processor and 16GB of RAM. We have used all single property designs from the HWMCC'15 benchmark set. Each design is initially simplified using standard logic synthesis techniques (similar to the `&dc2` command in ABC [14]). We used a timeout of 900 seconds.

TABLE I. Summary of experimental results.

Technique	Solved	Time (seconds)
`Quip`	230	52,776
$B(5,0)$	**233**	**51,695**
$B(5,5)$	224	57,864
$B(5,\infty)$	212	68,012
$B(\infty,0)$	229	53,757
$B(\infty,5)$	223	57,551
$B(\infty,\infty)$	219	62,397

We omit a direct experimental comparison to the original k-induction algorithm, as k-induction solves tremendously fewer properties than any of the `IC3`-based techniques. On the other hand, on most of the (few) properties that k-induction is able to solve, the value of k is small. In these cases, k-induction (based on unrolling the transition relation) usually outperforms `IC3`-based techniques, in the same manner as `BMC` usually outperforms `IC3` when searching for counterexamples.

We focus the experimental evaluation on the comparison of `Quip` and `KIC3` with different blocking strategies. The experiments are presented for 238 designs – which are all of the designs that remain after removing all instances solved by logic synthesis alone and all instances not solved by any of the techniques. Recall that the blocking strategy $B(k_0, m_0)$ means that `KIC3_Block_Kind` is called with induction depth k_0 and at most m_0 K-CTIs. We say that $k_0 = \infty$ whenever a proof obligation is always blocked with the largest possible induction depth. In what follows, we report the results for 6 different configurations of `KIC3`, obtained by setting k_0 to either 5 or ∞, and setting m_0 to either 0, 5, or ∞. A summary of the overall results is shown in Table I. The columns **Solved** and **Time** represent the total number of instances solved and the cumulative time in seconds, respectively. A detailed comparison between Quip and each KIC3 variant is shown in the scatter plots in Fig. 7. For each of the plots the horizontal axis measures the runtime of Quip, while the vertical axis measures the runtime of KIC3 with the corresponding k-induction blocking strategy. Thus, points below the diagonal represent wins for the pure Quip approach, and vice versa.

According to the experiments, the blocking strategy $B(5, 0)$ performs the best, slightly outperforming Quip by solving 3 more instances in less time. Furthermore, from the plot on the top-left, we can see that the total runtimes of Quip and KIC3 with $B(5, 0)$ are fairly well correlated, as most points are in the vicinity of the diagonal. However, a more detailed analysis shows that about 30% of the total time to block a proof obligation is spent in the KIC3_Block_Kind part of the procedure, so the actual profiles of the two algorithms are significantly different.

We have also experimented with other blocking strategies $B(k_0, 0)$, and in general all the results are highly consistent. For example, the "extreme" configuration $B(\infty, 0)$ solves 4 instance less than $B(5, 0)$ (and 1 instance less than Quip), and the plot on the bottom-left still shows high correlation with Quip.

At the same time, increasing the number m_0 of K-CTIs has a clear negative effect on the algorithm's performance for almost every k_0. The three top plots demonstrate this for $k_0 = 5$, while the three bottom plots demonstrate this for $k_0 = \infty$. However, we can also note that Quip and KIC3 become less correlated as m_0 is increased, while some instances get solved faster than before.

VII. Conclusions

In this work, we present an algorithm to decide whether a given safety property is k-inductive. This algorithm is based on the insights from IC3, and does not explicitly unroll the transition relation or add unique-state constraints to guarantee simple paths. In addition, we show how k-induction can be integrated into IC3 with minor modifications of the IC3-framework. On the practical side, a preliminary experimental evaluation shows a potential benefit of the suggested methods.

References

[1] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 108–125. [Online]. Available: http://dx.doi.org/10.1007/3-540-40922-X_8

[2] M. Brain, S. Joshi, D. Kroening, and P. Schrammel, "Safety verification and refutation by k-invariants and k-induction," in *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, 2015, pp. 145–161. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48288-9_9

[3] T. Kahsai and C. Tinelli, "PKind: A parallel k-induction based model checker," in *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011.*, 2011, pp. 55–62. [Online]. Available: https://doi.org/10.4204/EPTCS.72.6

[4] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18275-4_7

[5] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," 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. 125–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157675

[6] D. Jovanovic and B. Dutertre, "Property-directed k-induction," 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. 85–92. [Online]. Available: http://dx.doi.org/10.1109/FMCAD.2016.7886665

[7] A. Cimatti and A. Griggio, "Software model checking via IC3," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. A. Seshia, Eds., vol. 7358. Springer, 2012, pp. 277–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_23

[8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 157–164. [Online]. Available: http://ieeexplore.ieee.org/document/6679405/

[9] A. Gurfinkel and A. Ivrii, "Pushing to the top," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, 2015, pp. 65–72. [Online]. Available: http://ieeexplore.ieee.org/document/7542254/

[10] A. Griggio and M. Roveri, "Comparing different variants of the IC3 algorithm for hardware model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 1026–1039, 2016. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2015.2481869

[11] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. 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*, 2015, pp. 24–51. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23534-9_2

[12] "RuleBase SixthSense Edition," https://www.research.ibm.com/haifa/projects/verification/Formal_Methods-Home/.

[13] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 159–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30494-4_12

[14] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_5

Designing Parallel PDR

Matteo Marescotti*, Arie Gurfinkel†, Antti E. J. Hyvärinen*, Natasha Sharygina*
*Università della Svizzera italiana, Switzerland
†University of Waterloo, Canada

***Abstract*—Property Directed Reachability (PDR) is an efficient model checking technique. However, the intrinsic high computational complexity prevents PDR from meeting the challenges of real world verification. To address this problem, this paper introduces the parallel algorithm P3 based on: 1) partitioning of the input problem, 2) exchanging of learned reachability information, and 3) using algorithm portfolios. The generic nature of the proposed techniques makes them immediately suitable for software verification. This paper investigates the benefits of these techniques while taken individually and when combined together, implemented using distributed computing environment on top of the SMT-based software model checker SPACER. In our experiments over SV-COMP benchmarks we observe up to an order of magnitude speedup with respect to the sequential implementation with twice as many instances solved within a timeout.**

I. Introduction

Applying model checking to realistic, complex systems is highly non-trivial in part due to the computational complexity of the underlying decision problem. This paper studies how parallel computing can help in scaling model checking to such problems. We concentrate on parallelizing the execution of the Property directed reachability (PDR/IC3) algorithm [1], [5] using algorithm portfolios and approaches based on divide-and-conquer together with sharing of information learned in the model checking process. In particular, we study how a computational cluster can speed up model checking of software represented as sets of Constrained Horn Clauses (CHC) over Satisfiability modulo theories (SMT) constraints.

The PDR/IC3 algorithm (PDR in brief) is a relatively recent procedure that, given a transition system and a safety property, computes a safe inductive invariant or finds a counterexample for the safety of the system. During the computation, the algorithm maintains an increasingly long sequence of *frames* F_i representing symbolically safe over-approximations of states reachable from the initial state of the transition system in at most i steps. The frames are constructed as sets of PDR-*lemmas* that are computed to block spurious counterexamples. The algorithm terminates either by finding a concrete counterexample or by showing that the set of states described by F_{i+1} is a subset of the set of states described by its immediate predecessor frame F_i. In this case, the frame F_{i+1} is a safe inductive invariant for the program. Constructing the frame sequence is a heuristic process which can employ several different strategies within the PDR algorithm.

In this paper we introduce, to the best of our knowledge, the first divide-and-conquer technique for PDR (called partitioning throughout the paper), combine it with a portfolio of PDR solvers running different strategies, and allow the solvers to share PDR-lemmas. We combine these techniques in our new algorithm called P3 (Parallelly Performed PDR).

The P3 algorithm enables efficient parallel model-checking using PDR with the aim of improving the current state-of-the-art of parallel software verification. The approach is based on three concepts that expand the sequential PDR algorithm as follows: (i) P3 applies a portfolio of sequential PDR implementations parameterized to compute the frame sequence in different ways through different search strategies and by randomizing the search heuristic of the underlying SMT solver; (ii) P3 implements the novel partitioning approach introduced here by using the transition function of the program to compute, based on the negation of the safety property, its pre-images which are then distributed over the solvers in the portfolio as new safety properties; and (iii) the PDR implementations in the portfolio may share the PDR lemmas stored in the frames among each other. To the best of our knowledge, P3 enables parallel PDR for software verification for the first time.

We implemented the P3 algorithm using the sequential SPACER model checker [12] as a basis. We performed a thorough experimental analysis processing over 1000 instances from the Software Verification Competition 2016 (SV-COMP) with different configurations. The experiments were run in a computational cluster of 60 CPU cores. Our results show that a combination of divide-and-conquer, lemma sharing, and algorithm portfolio is capable of solving twice as many instances within our timeout of 1000 seconds and provides up to an order of magnitude speedup compared to the best sequential SPACER configuration. Our experimentations furthermore reveals that the choice of the heuristic for selecting which clauses to share is important, and that especially the partitioning technique benefits the most from it. The results are interesting also in the sense that they report the first experiments on running PDR on a computing cluster targeting in particular software verification with SMT-based Constrained Horn Clauses.

The paper is structured as follows. We compare with related work in Section II, and in Section III we review the basics of the PDR algorithm. In Section IV, we introduce our notion of distributed PDR and give details about our new parallelisation strategies. In Section V, we describe our implementation used for empirical evaluation presented in Section VI. Finally, we conclude the paper in Section VII.

II. Related work

The first attempt to parallelize PDR is mentioned in the original PDR paper [1], where the experimented parallel setting is based on sharing all the frames among different

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

computing threads on the same machine. This work is further improved in [3] where they study different parallel approaches, all of them focused on multi-threaded portfolios and PDR-lemma sharing, addressing propositional PDR.

Both [1], [3] are limited to propositional PDR, making them suitable mainly for hardware verification. In contast to [1], [3], we propose and thoroughly evaluate different lemma sharing strategies by differentiating between k-invariants and ∞-invariants inside the frames. We introduce the novel partitioning technique for PDR, together with a concise algorithm capable of combining all these techniques in a sound manner. Moreover our implementation is based on the more scalable distributed computing, thus exploiting the much bigger computational power offered by cloud-computing environments compared to single-machine threads.

A first investigation of PDR in the setting of software verification is done in [4]. A different approach based on CHC for software verification is presented in [12]. We extend this work with the aim of improving the current state-of-the-art of parallel software verification.

There is a substantial amount of work on parallel SMT solving that can help software verification model checking techniques in general. The parallelization tree framework for combining divide-and-conquer and portfolio directly on SMT formulas is introduced in [10] and augmented with clause sharing in [14]. A parallel approach for model checking of concurrent programs is given in [9], while [15] presents a parallel symbolic execution involving several sequential SMT solvers.

III. Preliminaries

We assume that the reader is familiar with the basic notation and semantics of First Order Logic (FOL), and theories of Linear Integer Arithmetic (LIA) and Arrays. For a set of variables X, we write X' to denote the set of primed variables $X' = \{x' \mid x \in X\}$. We extend the notation to formulas, and write φ' for a formula obtained from φ by replacing all variables in φ with the corresponding primed variables. Furthermore we denote by $X^{[i]}$ the set of variables obtained by adding i primes to each $x \in X$.

A. Safety Properties

A program can be expressed as a transition system consisting of variables X, a formula $Init(X)$ describing the program's initial states, and a formula $Tr(X, X')$ describing the program's transition relation. Given a transition system, a *safety property* $\neg Bad(X)$ is a formula over the variables of the system. A set of states described by a formula F is *safe* if $F \wedge Bad$ is unsatisfiable. A transition system satisfies a safety property, i.e., is *safe* with respect to $\neg Bad$, if all its states reachable from the initial state with the transition relation are safe. The transition system is safe up to k steps if its states reachable by i applications of the transition relation, for all $0 \leq i \leq k$, are safe. In this paper, we are interested in determining whether a given program satisfies a given safety property. An instance of this problem is expressed as a triple $\mathcal{S} = \langle Init(X), Tr(X, X'), Bad(X)\rangle$. For simplicity, we assume that $Init(X) \implies \neg Bad(X)$, otherwise, $\mathcal{S}$ is unsafe and the counterexample is a trivial model over X that satisfies $Init(X) \wedge Bad(X)$.

Definition 1 (Post Image). *Given a transition relation Tr and a set of states represented by F, the predicate $post^n_{Tr}(F)$ is the set of states reachable from any state in F after taking exactly n transitions of Tr. It is defined as follows:*

$$post^n_{Tr}(F) = \begin{cases} F & \text{if } n = 0,\\ \exists X' \cdot post^{n-1}_{Tr}(F)(X') \wedge Tr(X', X) & \text{if } n \geq 1. \end{cases}$$

$post^*_{Tr}(F)$ is the transitive closure of Tr:

$$post^*_{Tr}(F) = \bigvee_{n \geq 0} post^n_{Tr}(F)$$

The set of reachable states for a program $\mathcal{S}$ is $post^*_{Tr}(Init)$. The PDR algorithm constructs an approximation of the reachable states by computing modularly overapproximations of states reachable by $\mathcal{S}$ in a certain number of steps. The algorithm presents these overapproximations as PDR-lemmas, formulas over variables X describing reachability information learned by PDR.

Definition 2 (Relatively Inductive and Invariant Lemmas). *Given initial states represented by Init, transition relation Tr and a set of lemmas F, a* PDR*-lemma φ is* inductive relative *to F if and only if*

$$Init \implies \varphi \qquad\qquad \varphi \wedge F \wedge Tr \implies \varphi'$$

Whenever φ is inductive relative to true, we say that φ is an inductive lemma*. A* PDR*-lemma φ is an* invariant lemma *if it is true in all the reachable states, i.e., $post^*_{Tr}(F) \implies \varphi$. Every inductive* PDR*-lemma is invariant, but the converse is not true in general.*

An instance $\mathcal{S}$ is safe if there exists a safe inductive invariant $Inv(X)$ such that $Inv(X) \implies \neg Bad(X)$. $\mathcal{S}$ is unsafe if there exists an $n \in \mathbb{N}$ such that $post^n_{Tr}(Init) \wedge Bad$ is satisfiable. For an unsafe $\mathcal{S}$, a satisfying assignment for

$$Init(X^{[0]}) \wedge Bad(X^{[n]}) \wedge \bigwedge_{i=0}^{n-1} Tr(X^{[i]}, X^{[i+1]})$$

is called a *feasible counterexample*. The satisfying assignment corresponds to a sequence of states where the first state satisfies $Init$, each consecutive pair of states satisfies Tr, and the final state satisfies Bad, and can, therefore, be considered as an evidence for a programming error.

*B. Property Directed Reachability (*PDR*)*

In this section, we give a high-level overview of IC3/PDR algorithm. We refer the reader to [1], [5], [8], [6], [12] for the details of the original algorithm and its extensions to SMT.

Definition 3 (PDR Trace). *Given an instance of the safety problem $\mathcal{S}$, a* PDR trace *for $\mathcal{S}$ is a sequence of frames $\mathcal{F} = \langle F_0, F_1, \ldots, F_N, \ldots\rangle$ such that each frame $F_i \in \mathcal{F}$ is a set* *of PDR-lemmas. Furthermore, the trace satisfies the following properties for $i \geq 0$:*

$$F_0 \equiv \mathit{Init} \tag{1}$$
$$F_i \wedge \mathit{Tr} \implies F'_{i+1} \tag{2}$$
$$F_i \implies F_{i+1} \tag{3}$$
$$i < N \implies (F_i \implies \neg \mathit{Bad}) \tag{4}$$

Intuitively, each frame $F_i \in \mathcal{F}$ over-approximates all the states reachable in at most i steps of the transition relation Tr from Init. Moreover, the trace proves that $\mathcal{S}$ is safe up to $N-1$ steps of Tr from Init.

PDR uses the trace to compute an increasing bound of steps from Init up to which $\mathcal{S}$ is safe. The algorithm works by iteratively adding an initially empty frame F_N at the end of the trace. PDR then tries to either prove safety of F_N by strengthening it, or to find a feasible counterexample based on it.

Definition 4 (Proof Obligation)**.** *Given an instance $\mathcal{S}$ of the safety problem and a* PDR *trace $\mathcal{F}$ for $\mathcal{S}$, a* proof obligation *is the pair $\langle \sigma, i \rangle$ where σ is a conjunction of predicates over state variables and $i \leq N$. In addition, the proof obligation satisfies the following:*

- *$\sigma \wedge F_i$ is satisfiable, and*
- *for all models m such that $m \models \sigma$, $post^*_{Tr}(m) \wedge \mathit{Bad}$ is satisfiable.*

The conjunction σ represents a set of the states consistent with a frame $F_i \in \mathcal{F}$, containing states that can reach Bad with a feasible path.

Given an instance $\mathcal{S}$, PDR computes a trace $\mathcal{F}$ of increasing length for $\mathcal{S}$ until either a fixed point is found for Tr or the algorithm determines a feasible counterexample. In the process, PDR constructs candidate counterexamples, proof obligations, that are stored in an *obligation queue* $\mathcal{Q}$. The proof obligations $\langle \sigma, i \rangle$ are propagated towards the initial state by computing their pre-image with respect to Tr, resulting in $\langle \sigma^-, i-1 \rangle$ which is then inserted to $\mathcal{Q}$. If a counterexample candidate is not feasible, a proof obligation will at some point be blocked by a frame. This happens if for a proof obligation $\langle \sigma, i \rangle$ it holds that $F_{i-1} \wedge \mathit{Tr} \wedge \sigma'$ is unsatisfiable. The clause $\neg\sigma$ is then simplified to a PDR-lemma and inserted to F_i.

PDR proves $\mathcal{S}$ safe if:

$$\exists i < N \cdot F_{i+1} \implies F_i$$

This simplifies Equation (2) to $F_i \wedge \mathit{Tr} \implies F'_i$. Thus together with Equations (1) and (4) F_i is proved to be both a fixed point for Tr and a safe inductive invariant for $\mathcal{S}$.

The way PDR computes the fixed point leaves room for some flexibility in how the lemmas are organized. In particular [6] suggests to separate the inductive lemmas to a distinct frame F_∞. Hence the frame F_∞ is initially empty and always consists of those lemmas $\varphi \in \bigcup F_i$ inductive relative to F_∞. Thus, $\mathcal{S}$ is safe when

$$F_\infty \implies \neg \mathit{Bad}.$$

PDR proves $\mathcal{S}$ unsafe whenever a proof obligation $\langle \sigma, 0 \rangle$ is added to the obligations queue. By Definition 4, σ represents a set of states in Init from which there is a feasible path leading to a state in Bad.

Definition 5 (PDR Configurations)**.** *Given an instance of the safety problem $\mathcal{S}$, a* PDR configuration *is the quadruple $\mathcal{C} = (N, \mathcal{F}, F_\infty, \mathcal{Q})$ where:*

- *$N \in \mathbb{N}$,*
- *$\mathcal{F} = \langle F_0, \ldots, F_N, \ldots \rangle$ is a trace of $\mathcal{S}$,*
- *F_∞ is the inductive frame of $\mathcal{F}$,*
- *$\mathcal{Q} = \{\langle \sigma, i \rangle, \ldots\}$ is the obligation queue, where $i \leq N$.*

The Initial gonfiguration *of* PDR *for $\mathcal{S}$ is $\mathcal{C}_0 = (1, \langle \mathit{Init}, \emptyset, \ldots \rangle, \emptyset, \emptyset)$*

Given a PDR configuration $\mathcal{C}$ of a safety problem $\mathcal{S}$, each of the following operation performed on $\mathcal{C}$ updates its components resulting in a new configuration $\mathcal{C}'$.

Candidate. If there exists σ such that $\sigma \implies F_N \wedge \mathit{Bad}$, then the proof obligation $\langle \sigma, N \rangle$ is added to $\mathcal{Q}$.

Predecessor. Given $\langle \sigma, i \rangle \in \mathcal{Q}$ with $i > 0$, if there exists a conjunction of predicates δ such that for all consistent m such that $m \models \delta$, $m \wedge \mathit{Tr} \models \sigma'$ holds, then $\langle \delta, i-1 \rangle$ is added to $\mathcal{Q}$.

Blocking. Given $\langle \sigma, i \rangle \in \mathcal{Q}$ with $i \geq 1$, if *Predecessor* is not applicable then remove $\langle \sigma, i \rangle$ from $\mathcal{Q}$ and add the PDR-lemma φ to all F_j with $1 \leq j \leq i$ such that

$$\mathit{Init} \implies \varphi \qquad \varphi \implies \neg\sigma \qquad F_{i-1} \wedge \mathit{Tr} \implies \varphi'$$

Unfold. If $\mathcal{Q} = \emptyset$ and *Candidate* is not applicable then $N := N + 1$.

Inductive. Given a subset of lemmas $\varphi \subseteq F_i$ with $0 \leq i < N$ s.t. $\varphi \wedge F_\infty \wedge \mathit{Tr} \implies \varphi'$, add φ to F_∞ (and to each F_i).

In addition PDR has the following two rules that guarantee the termination of the algorithm when always taken when they are applicable:

Safe. If $F_\infty \implies \neg \mathit{Bad}$ report *safe*.

Unsafe. If any $\langle \sigma, 0 \rangle \in \mathcal{Q}$ report *unsafe* and generate a counterexample.

In PDR with theories, and, therefore, in our implementation, the operation *Predecessor* employs Model-Based Projection [12] to ensure termination, while *Blocking* uses interpolation [8] to build the lemma.

Definition 6 (PDR Strategy)**.** *Given an intance $\mathcal{S}$ of the safety problem and a* PDR *configuration $\mathcal{C}$, a* PDR *strategy $\mathcal{T}_\mathcal{S}$ is a function that maps $\mathcal{C}$ to one of the possible* PDR *operations applicable for $\mathcal{C}$, based on $\mathcal{S}$.*

A PDR execution is a sequence of configurations $\langle \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_T \rangle$ such that for every $i \in \{1, \ldots, T\}$, $\mathcal{C}_i$ is the result of the operation $\mathcal{T}_{\mathcal{S}}(\mathcal{C}_{i-1})$ on $\mathcal{C}_{i-1}$, $\mathcal{C}_0$ is the initial PDR configuration for $\mathcal{S}$, i.e. $(1, \langle \mathit{Init}, \emptyset, \ldots \rangle, \emptyset, \emptyset)$, and $\mathcal{T}_{\mathcal{S}}(\mathcal{C}_T) \in \{\mathit{Safe}, \mathit{Unsafe}\}$.

IV. The P3 Algorithm

In this section we introduce the P3 (Parallelly Performed PDR) algorithm for parallel model-checking with PDR.

P3 implements three parallelisation techniques for PDR: *portfolio*, *partitioning*, and *lemma sharing*. These techniques can be combined in order to exploit each one strengths.

Portfolio takes advantage by using different PDR strategies while partitioning focuses the search by constraining the problem. In general, partitioning means dividing a problem into several sub-problems. The PDR partitioning technique introduced in this paper partitions the problem by restricting the paths leading to the bad states.

Finally, lemma sharing provides each solver with useful information arising from search diversification, and possibly not derivable locally. The intuition is that PDR-lemmas express what is learned by each PDR execution. Since different executions employs different strategies, PDR-lemmas that are easily found by one strategy can be difficult to find by another.

In the rest of the section, we formalize portfolio, partitioning, and lemma sharing strategies and provide details of P3.

A. Portfolio

The most naïve parallel technique is a *portfolio* – concurrent and independent execution of multiple sequential PDR strategies on the same problem. A portfolio terminates as soon as one of its instances terminates successfully.

We define a notion of a distributed PDR configuration to model a PDR portfolio with any combination of lemma sharing and partitioning.

Definition 7 (Distributed PDR Configuration)**.** *Given an instance $\mathcal{S}$ of the safety problem, a distributed* PDR *configuration is a set of tuples:*

$$\mathcal{D}^n = \{(\mathcal{T}^i, \mathcal{C}^i)\}$$

where for each $i \in \{1, \ldots, n\}, n \in \mathbb{N}$:

- $\mathcal{T}^i$ *is a* PDR *strategy from Definition 6,*
- $\mathcal{C}^i = \left(N^i, \mathcal{F}^i = \langle F_0^i, F_1^i, \ldots, F_{N^i}^i, \ldots, \rangle, F_\infty^i, \mathcal{Q}^i\right)$ *is a* PDR *configuration from Definition 5.*

A distributed PDR configuration expresses a PDR portfolio when for every $i \in \{1, \ldots, |\mathcal{D}|\}$, $\mathcal{T}^i$ is a strategy for the input problem $\mathcal{S}$. That is, every strategy evolves the corresponding PDR-configuration independently, performing asynchronous and arbitrary choices based on $\mathcal{S}$.

A PDR portfolio $\mathcal{D}$ terminates when there exists $\mathcal{T}_{\mathcal{S}}^i(\mathcal{C}^i) \in \{\mathit{Safe}, \mathit{Unsafe}\}$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$. Termination and soundness of this setting follows trivially from PDR because every execution is independent.

B. Partitioning

In this section, we define partitioning strategy and argue for its soundness.

Definition 8. *Given a safety problem $\mathcal{S}$, $\mathit{partition}(\mathcal{S})$ is a set of problems $\{\mathcal{S}_{p_1}, \ldots, \mathcal{S}_{p_n}\}$, where each instance $\mathcal{S}_{p_i} = \langle \mathit{Init}(X), \mathit{Tr}(X, X'), p_i(X)\rangle$ is called a partition of $\mathcal{S}$, and such that*

$$\bigvee_i^n p_i \iff \exists X' \cdot \mathit{Tr}(X, X') \wedge \mathit{Bad}(X')$$

From Definition 8, it follows that $\mathcal{S}$ is safe if and only if all of its partitions are safe. A distributed PDR configuration $\mathcal{D}$ expresses partitioning if for each partition $\mathcal{S}_p \in \mathit{partition}(\mathcal{S})$ there is a pair $(\mathcal{T}_{\mathcal{S}_p}^i, \mathcal{C}^i) \in \mathcal{D}$. The result of a distributed configuration with partitioning is Unsafe if there exists $\mathcal{T}_{\mathcal{S}_p}^i(\mathcal{C}^i) = \mathit{Unsafe}$, and the result is Safe if for each $\mathcal{S}_p \in \mathit{partition}(\mathcal{S})$ there exist $\mathcal{T}_{\mathcal{S}_p}^i(\mathcal{C}^i) = \mathit{Safe}$.

The soundness is by construction: a counterexample for $\mathcal{S}_p$ is also valid for $\mathcal{S}$, while the safety of all $\mathcal{S}_p$ ensures the safety of $\mathcal{S}$ because every state leading to Bad in one step is expressed in a partition.

C. Lemma Sharing

In this section, we give the formal definition of lemma sharing and argue for its soundness in a distributed portfolio setting.

Definition 9 ((k-)invariant)**.** *A* PDR*-lemma ψ is* k-invariant *if it is true in all the states reachable in k steps or less, i.e., $\mathit{post}_{\mathit{Tr}}^k(\mathit{Init}) \implies \psi$. If φ is invariant, then it is k-invariant for any $k \in \mathbb{N}$.*

Following Definition 9, each frame F_k, $k \in \mathbb{N}$, is a set of k-invariants for $\mathcal{S}$, while F_∞ is a set of invariants for $\mathcal{S}$.

Theorem 1 (Lemma Sharing)**.** *Given a distributed* PDR *configuration $\mathcal{D}$ for an instance $\mathcal{S}$ of the safety problem, the* PDR*-lemma $\psi \in F_k^i, k \in \mathbb{N}$ is a k-invariant for $\mathcal{S}$ and the operation of adding ψ to any F_l^j with $i \neq j$ and $l \leq k$ keeps $\mathcal{C}^j$ a valid* PDR *configuration for $\mathcal{S}$.*
The same holds for $\varphi \in F_\infty^i$ when added to any F_∞^j, $i \neq j$.

Proof. The proof follows from Definition 2. Each $\varphi \in F_k^i$ is a k-invariant if $k \in \mathbb{N}$, or an invariant if $k = \infty$ and can be used to soundly refine a different abstraction of states reachable in up to k steps.

Similarly, sharing invariants is sound and makes every F_∞^i an invariant for $\mathcal{S}$. Thus, $\mathcal{S}$ is safe whenever any F_∞^i implies $\neg \mathit{Bad}$. □

D. Parallelly Performed PDR

The P3 algorithm is shown in Algorithm 1. It combines portfolio, lemma sharing, and partitioning. The algorithm works as follows. Until there are available computing resources, the procedure **Entrust** randomly selects a partition not yet solved, creates a new strategy and allocates the necessary resources in order to run PDR.

The procedure **Exclude** at line 12 is taken exactly once for each partition in $\mathcal{P}$ whenever a corresponding sequential PDR instance terminates. This happens finitely many times because there are finitely many partitions. Therefore, the algorithm

```
Input    : Safety problem
           $\mathcal{S} = \langle Init(X), Tr(X, X'), Bad(X)\rangle$.
Output   : $\{Reachable, Unreachable\}$
Data     : A distributed PDR configuration $\mathcal{D}$, a set of
           partitions $\mathcal{P}$.
Initially: $\mathcal{D} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$.
Assume: $Init \wedge Bad$ is unsatisfiable.
1  $\mathcal{P} \leftarrow partition(\mathcal{S})$
2  while True do
3      Reachable: if $\langle \sigma, 0\rangle \in \mathcal{Q}^i$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$,
        return $Reachable$.
4      Unreachable: if $\mathcal{P} = \emptyset$, return $Unreachable$.
5      Lemma Sharing: copy a PDR-lemma $\varphi \in F_n^i$ to $F_n^j$
        with: $i, j \in \{0, \ldots, |\mathcal{D}|\}$, $i \neq j$ and $n \in \mathbb{N} \cup \{\infty\}$
6      Entrust: if computing resources are available, then:
7          select a partition $\mathcal{S}_p \in \mathcal{P}$
8          create a new PDR strategy $\mathcal{T}_{\mathcal{S}_p}$
9          create new $\mathcal{C} = (1, \langle Init, \emptyset, \ldots\rangle, \emptyset, \emptyset)$
10         set $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathcal{T}_{\mathcal{S}_p}, \mathcal{C})\}$
11         allocate computing resources for PDR($\mathcal{T}_{\mathcal{S}_p}, \mathcal{C}$)
12     Exclude: if there exists $\mathcal{S}_p \in \mathcal{P}$ such that
        $F_\infty^i \implies \neg p$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$, then:
13         $\mathcal{P} \leftarrow \mathcal{P} \setminus \{p\}$
14         release computing resources used for each
            $(\mathcal{T}_{\mathcal{S}_p}, \mathcal{C}) \in \mathcal{D}$
15 end
```

Algorithm 1: The P3 algorithm.

terminates if all corresponding PDR instances terminate. Once exclude is taken, all the computing resources available might be reallocated on other partitions by several **Entrust** calls.

Lemma Sharing does not affect termination because it only refines the frames, leading the execution toward convergence. It is not possible for any frame to get weaker after lemma sharing is applied.

If $partition(\mathcal{S}) = \{\mathcal{S}\}$ at line 1, then partitioning is disabled, and if procedure **Lemma Sharing** at line 5 is never taken then lemma sharing is disabled. If both are disabled then the algorithm corresponds to a PDR portfolio. We assume that there is a global t to handle the desired setting.

V. Implementation

We implemented our parallel PDR algorithm using the tool SMTService [13], a framework for parallel and distributed solving already used in [14] for distributed SMT. A Graphical User Interface [2] is also available for analysing the parallel executions. We upgraded it to also support distributed PDR. SMTService is a client-server based framework. The *server* implements Algorithm 1. The *client* uses the SMT-based PDR model checker Spacer [12]. We modified Spacer in order to expose the API for lemma sharing. The overview of the architecture is shown in Figure 1.

The server is written in Python and its behaviour can be controlled through a *configuration* file. The server is in charge of pre-processing the input instances, managing clients connections and solving tasks and collecting statistics sent by the client solvers. The server stores its information in an SQLite3 database called *Logs*. The database is also used to analyse the steps of the parallel solving process. Instances to be solved are provided to the server at run time through the *control socket*. The control socket allows either the user or a tool (e.g., a model checker) to interact with the server.

The *partitioner* component creates the partitions. The partitioning process is guided by the CHC syntactic structure of the input instances. When partitioning is disabled, the partitioner creates a single partition corresponding to the bad states.

The *scheduler* keeps the list of all the instances and their partitions and manages connected clients. The scheduler solves one instance at a time. The partitions of the current instance are evenly distributed among the connected clients. Once a partition is proven unsatisfiable, the client working on it is provided with a remaining unsolved partition. Spacer implements three strategies, Spacer(DEF), Spacer(IC3), Spacer(GPDR), which differ in how they manage the queue of proof obligations. The scheduler proceeds in a best-effort way assigning each partition to 3 solvers, each configured with one strategy. If there are still solvers available then the same procedure is repeated using a different random seed in the underlying SMT solver of the client. Client failures and connection of new clients are handled in a sound way so that computational power may be added or removed on request.

The *Lemma Database* is implemented as a separate server. Each client periodically pushes learned lemmas to the Lemma Database accordingly to the *sharing strategy* provided by the server. The pull of the lemmas is also periodic and each client only pulls lemmas that are yet unknown for that client.

We implemented 4 lemma sharing strategies. The server reads from the configuration file which is the desired lemma sharing strategy and forwards it to the solver together with the instance. The procedure **Lemma Sharing** at line 5 in Algorithm 1 describes the strategy $*$-*invariants*, namely the exchanging of every learned PDR-lemma. Sharing only PDR-lemmas from F_∞ (∞-*invariants*) is done by fixing $n = \infty$, while sharing only k-*invariants* is achieved by constraining $n \in \mathbb{N}$. Finally, lemma sharing is disabled when the procedure **Lemma Sharing** is never taken. Once an instance is solved its lemmas are removed to reduce memory consumption. In the case of partitioning combined with lemma sharing, our implementation shares lemmas only among solvers working on the same partition.

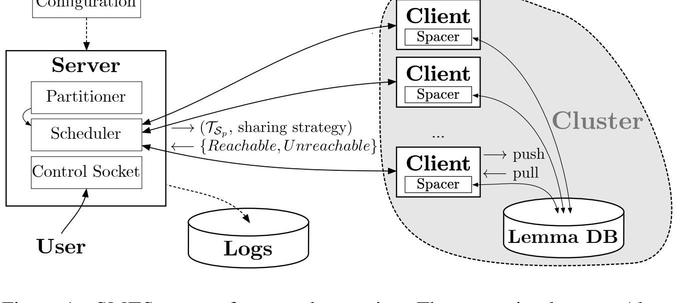


Figure 1. SMTService framework overview. The server implements Algorithm 1. Each client represents a different solver process in a computing node. Solid lines represent TCP/IP connections, while dashed lines represent disk I/O.

Table I
SUMMARY OF RESULTS SHOWING THE NUMBER OF SOLVED INSTANCES.

Technique	*less500*			*more500*		
	#reachable	#unreachable	#unknown	#reachable	#unreachable	#unknown
SPACER(GPDR)	63	175	13	0	8	317
SPACER(IC3)	64	155	32	2	9	314
SPACER(DEF)	64	155	32	2	13	310
portfolio	66	185	0	8	40	277
∞-invariants	66	185	0	7	49	269
k-invariants	66	182	3	7	90	228
$*$-invariants	66	185	0	7	90	228
partitioning	66	176	9	10	34	281
partitioning+∞-invariants	66	183	2	11	49	265
partitioning+k-invariants	66	182	3	11	115	199
partitioning+$*$-invariants	66	185	0	16	98	211
virtual best	66	185	0	18	132	175

VI. EXPERIMENTS

This section presents an extensive experimental evaluation of the P3 algorithm on instances from the Software Verification Competition. We measure separately the performance of the algorithm on instances that are known to be easy and hard for the SPACER model checker, and study the performance of combinations of lemma sharing, portfolio, and partitioning. We also experiment on different lemma sharing heuristics on these settings.

All the reported experiments are executed on a cluster where each computing node is equipped with 2×Intel E5-2650 v3 CPU, 64 GB of RAM and Intel 40Gbps Infiniband network adapter. The parallel experiments are executed using 60 solvers on 6 computing nodes, with the server running on a separate node. The timeout is set to 1000 seconds (wall-clock time) on all the experiments.

The benchmark set used in our evaluations is constructed by SEAHORN [7], a fully automated analysis framework for LLVM-based languages. Given as input the source file, SEAHORN constructs the triplet $\langle Init(X), Tr(X, X'), Bad(X)\rangle$ expressed in SMT-LIB v2 like language and representing the input safety problem ready to be provided to SPACER. Our benchmark set is based on 1,802 C problems taken from the SV-COMP 2016 Device Drivers Linux 64-bit (LDV) categories available at https://github.com/sosy-lab/sv-benchmarks/tree/master/c and preprocessed by SEAHORN.

We first evaluate the benchmarks sequentially using the different strategies available in SPACER: IC3, GPDR and DEF. We call these settings *sequential*. Those benchmarks solved in less than one second are removed from the set and we experiment over the remaining 562 benchmarks.

Based on these evaluations we create two different benchmarks sets of easy and hard instances respectively:

- *less500*: benchmarks solved in less than 500 seconds by at least one strategy. It contains 251 benchmarks.
- *more500*: benchmarks solved in more than 500 or timed out by at least one strategy. It contains 325 benchmarks.

These benchmark sets partially overlap by having 14 benchmarks in common.

Table I shows an overall evaluation over all the experiments we carried out. For each technique, we report the number of instances proven reachable, unreachable, and those unsolved, for both the experimental sets. The table is partitioned into 4 parts. Going from top to bottom: part 1 contains the results from sequential executions; part 2 contains the result for lemma sharing strategies with pure portfolio; part 3 contains results with partitioning; and part 4 presents the results of the *virtual best* solver that uses the best configuration for each problem. This *virtual best* is achievable by running in isolation a portfolio of the 8 combinations, using 8×60 CPUs. Notably, every parallel technique outperforms sequential execution.

For the *more500* set the partitioning-based techniques perform the best. For the *less500* set, especially for reachable instances, portfolio-based technique is the best, matching the virtual best solver.

In Table II, we show average time speedups between sequential executions and the respective best parallel techniques for both sets, over benchmarks that did not time out. The columns *60 CPU* show the performance of the best technique: ∞-invariants for *less500* and partitioning+k-invariants for *more500*. An overview of performance over all the techniques is shown in Figure 2. We present the runtime performance for the three sequential strategies (IC3, GPDR and DEF) and all

Table II
AVERAGE SPEEDUP COMPARED TO SEQUENTIAL SOLVING.

Sequential strategy	*less500*		*more500*	
	60 CPU	virtual best	60 CPU	virtual best
SPACER(GPDR)	8×	10×	59×	91×
SPACER(IC3)	26×	32×	56×	88×
SPACER(DEF)	27×	33×	53×	83×

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

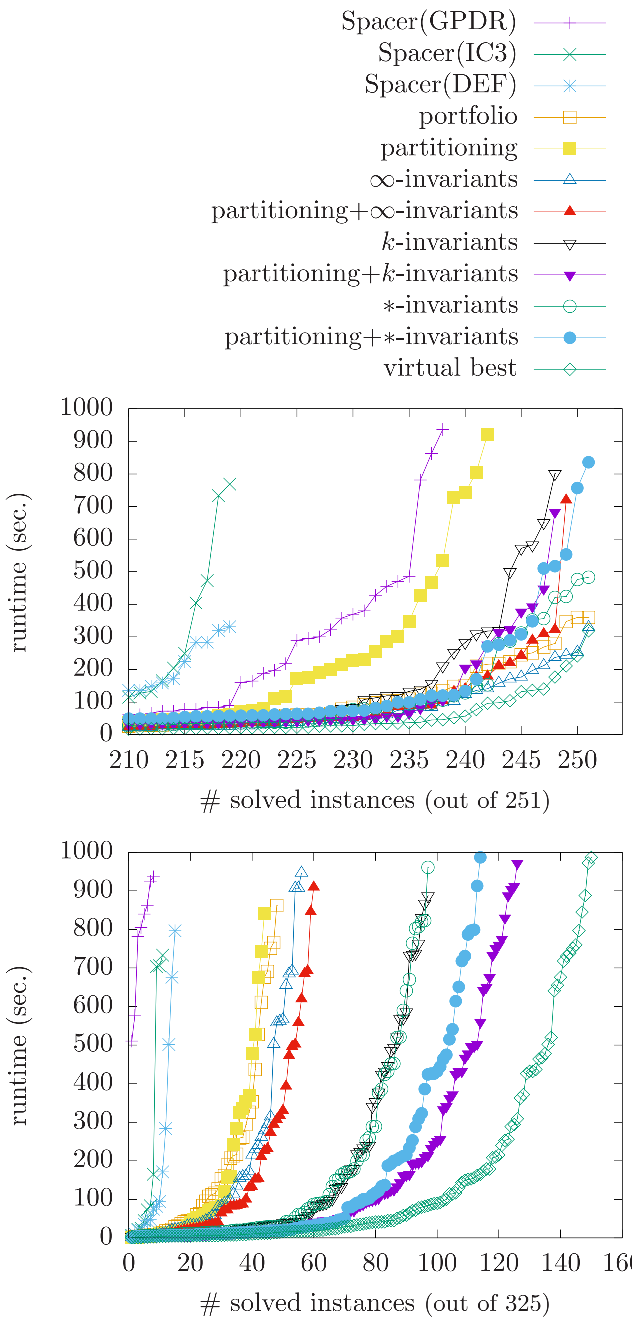


Figure 2. Comparison among all tested techniques on the sets *less500* (top) and *more500* (bottom). k-invariants refers to sharing only lemmas from the trace, ∞-invariants refers to sharing just F_∞, while $*$-invariants implements both. Finally, for each benchmark we report the virtual best runtime among all the tested techniques.

the parallel techniques tested in the cluster.

An overview for the set *less500* is given in Figure 2 (top). Sharing ∞-invariants over portfolio is the best technique for this set, already performing similarly to the virtual best. In fact, ∞-invariants solves all the benchmarks with an average 30% slowdown with respect to the virtual best, and up to 27× faster than sequential, as reported in Table II.

Figure 2 (bottom) shows a similar overview for *more500*. Regarding this set, partitioning techniques are the best choices. In fact, the best technique for this set is partitioning+k-invariants, followed by partitioning+$*$-invariants. However, 40 benchmarks are solved by only one of these two techniques, making them complementary rather than strictly better than the other.

A possible way to address the complementarity issue is by implementing a portfolio of multiple isolated parallel techniques, giving priority to the complementary ones. This approach is capable of gradually improving performance toward the virtual best results, where the best performance is reached with the highest CPU resource allocation.

Regarding *more500*, a portfolio of partitioning+k-invariants and partitioning+$*$-invariants is capable of solving 140 instances, 10 less than the virtual best and with an average 15% slowdown. Then, by adding $*$-invariants and k-invariants it is possible to solve 148 instances with 5% slowdown and half computing resources with respect to virtual best. The missing 2 instances are only solved by partitioning+∞-invariants and ∞-invariants.

Regarding *less500*, a similar setting can only decrease solving time. This is because ∞-invariants already solved all the benchmark. Figure 3 shows the comparison between partitioning and portfolio with $*$-invariant sharing being quite complementary. Thus, a portfolio of these two techniques is capable of solving the entire *less500* set using one fourth the CPU power requested by the virtual best, with a 14% slowdown.

Another important result shown in Figure 3 that partitioning often outperforms pure portfolio on reachable instances. This is because the first partition proven satisfiable also proves the entire problem satisfiability, and the focused search done in each partition helps the solvers to converge quickly.

Table III shows results about lemma sharing. The columns *time* show the average amount of time spent on lemma sharing push and pull, with respect to solving time. The columns *#lemmas* show the average number of PDR-lemmas exchanged. The amount of time spent on PDR-lemmas push and pull is about 1% and 3% of solving time respectively for portfolio based techniques, and partitioning based technique. The average amount of PDR-lemmas generated is significantly lower than in parallel SAT and SMT [10], [11]. While heuristics for clause sharing within parallel SAT and SMT are required due to the high throughput, in this settings lemma sharing

Table III
Lemma sharing statistics.

Parallel technique	*less500* time	*less500* #lemmas	*more500* time	*more500* #lemmas
portfolio +				
∞-invariants	0.35%	141	0.41%	670
k-invariants	1.24%	252	1.00%	347
$*$-invariants	1.55%	243	0.83%	348
partitioning +				
∞-invariants	1.46%	170	0.87%	403
k-invariants	3.51%	140	4.55%	238
$*$-invariants	3.27%	221	4.45%	320

heuristics are less crucial.

Overall, our experimental evaluations show that parallel techniques are highly beneficial. In particular, parallelization with portfolio combined with sharing PDR-lemmas from F_∞ is the best choice for easier instances, while partitioning with sharing PDR-lemmas from the trace is the best choice for harder instances. This demonstrate that the choice of the lemma sharing strategy is important.

More experimental results are available at http://verify.inf.usi.ch/content/p3-experimental-results-fmcad2017.

VII. Conclusions

This work introduces the first parallel approach of the IC3/PDR algorithm for software model checking based on constrained horn clauses and satisfiability modulo theories. The P3 algorithm is based on combining in a new and PDR-specific way algorithm portfolios, divide-and-conquer approaches, and sharing of information learned during the algorithm execution. We describe our algorithm and its parallel extensions in a unified framework that allows us to both reason about the correctness of the implementation, and study the effect of each component in relative isolation. In addition we identify two different types of PDR-lemmas, the k-invariants, and the ∞-invariants, and give the first results on constructing lemma sharing heuristics. To the best of our knowledge in particular the divide-and-conquer approach and the lemma sharing heuristics have not been previously used in the context of PDR. The techniques we propose improve the previously introduced powerful portfolio technique for PDR, and we believe that the contributions help in applying parallel and distributed computing both in hardware and software verification.

We implemented the P3 algorithm following the same principle of isolation between the different techniques. The implementation is based on the SPACER model checker and is adaptable to both distributed environments and multi-core computing throught our SMTSERVICE framework. Our experimental results obtained by running P3 on a representative set of software verification benchmarks from the SV-COMP 2016 competition show that the parallel approach is vastly superior to sequential SPACER configurations, solving over hundred more instances within our timeout and providing on the average super-linear speed-ups. We also show that each of the new techniques work in isolation and that they have interesting interaction with the different clause-sharing heuristics.

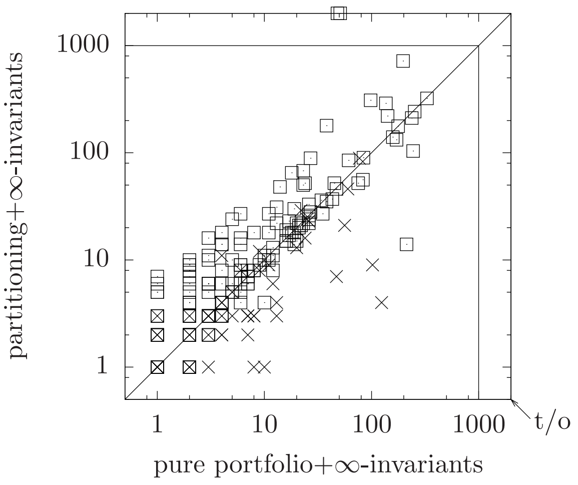


Figure 3. Comparison between sharing invariant with and without partitioning over *less500* sat (×) and unsat (⊡) instances. Compared to “best”, this combination is overall 14% slower while using 75% less CPU. Moreover it is shown that partitioning outperforms on all sat instances.

Acknowledgements. This work is supported by the Swiss National Science Foundation (SNSF) grants 153402 and 166288.

References

[1] Bradley, A.R.: SAT-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. pp. 70–87 (2011)

[2] Budakovic, J., Marescotti, M., Hyvrinen, A., Sharygina, N.: Visualising SMT-based parallel constraint solving. In: Proceedings of the 15th International Workshop on Satisfiability Modulo Theories. (2017)

[3] Chaki, S., Karimi, D.: Model checking with multi-threaded IC3 portfolios. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 517–535 (2016)

[4] Cimatti, A., Griggio, A.: Software model checking via IC3. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. pp. 277–293 (2012)

[5] Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011. pp. 125–134 (2011)

[6] Gurfinkel, A., Ivrii, A.: Pushing to the top. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 65–72 (2015)

[7] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 343–361 (2015)

[8] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. pp. 157–171 (2012)

[9] Holzmann, G.J.: Cloud-based verification of concurrent software. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 311–327 (2016)

[10] Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. LNCS, vol. 9340, pp. 369–386. Springer (2015)

[11] Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Incorporating clause learning in grid-based randomized SAT solving. Journal on Satisfiability Boolean Modeling and Computation 6(4), 223–244 (2009)

[12] Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design 48(3), 175–205 (2016)

[13] Marescotti, M.: SMTService Framework for distributed SMT and PDR. https://scm.ti-edu.ch/projects/smts (2017)

[14] Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. pp. 428–443 (2016)

[15] Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S.: Parallel SMT solving and concurrent symbolic execution. In: 2015 IEEE Trust-Com/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3. pp. 17–26 (2015)

FuseIC3: An Algorithm for Checking Large Design Spaces

Rohit Dureja and Kristin Yvonne Rozier
Iowa State University

Abstract—**The design of safety-critical systems often requires *design space exploration*: comparing several system models that differ in terms of design choices, capabilities, and implementations. Model checking can compare different models in such a set, however, it is continuously challenged by the state space explosion problem. Therefore, learning and reusing information from solving related models becomes very important for future checking efforts. For example, reusing variable ordering in BDD-based model checking leads to substantial performance improvement. In this paper, we present a SAT-based algorithm for checking a set of models. Our algorithm, FuseIC3, extends IC3 to minimize time spent in exploring the common state space between related models. Specifically, FuseIC3 accumulates artifacts from the sequence of over-approximated reachable states, called *frames*, from earlier runs when checking new models, albeit, after careful repair. It uses bidirectional reachability; forward reachability to repair frames, and IC3-type backward reachability to block predecessors to bad states. We extensively evaluate FuseIC3 over a large collection of challenging benchmarks. FuseIC3 is on-average up to 5.48× (median 1.75×) faster than checking each model individually, and up to 3.67× (median 1.72×) faster than the state-of-the-art incremental IC3 algorithm.**

I. Introduction

In the early phases of design, there are several models of the system under development constituting a *design space* [2, 19, 23]. Each model in such a set is a valid design of the system, and the different models differ in terms of core capabilities, assumptions, component implementations, or configurations. We may need to evaluate the different design choices, or to analyze a future version against previous ones in the product line. Model checking can be used to aid system development via a thorough comparison of the set of models. Each model in the set is checked one-by-one against a set of properties representing requirements. However, for large and complex design spaces, such an approach can be inefficient or even fail to scale to handle the combinatorial size of the design space. Nevertheless, model checking remains the most widely used method in industry when dealing with such systems [5, 19, 21, 23, 24].

We assume that different models in the design space have overlapping reachable states, and the models are checked sequentially. In a typical scenario, a model-checking algorithm doesn't take advantage of this information and ends up re-verifying "already explored" state spaces across models. For large models this can be extremely wasteful as every model-checking run re-explores already known reachable states. The problem becomes acute when model differences are small, or when changes in the models are outside the cone-of-influence of the property being checked, i.e., although the reachable states in the models vary, none of them are bad. Therefore, as the number of models grows, learning and reusing information from solving related models becomes very important for future checking efforts.

We present an algorithm that automatically reuses information from earlier model-checking runs to minimize the time spent in exploring the symbolic state space in common between related models. The algorithm, FuseIC3, is an extension to one of the fastest bit-level verification methods, IC3 [6], also known as *property directed reachability* (PDR) [17]. Given a set of models and a safety property, FuseIC3 sequentially checks each model by reusing information: reachable state approximations, counterexamples (cex), and invariants, learned in earlier runs to reduce the set's total checking time. When the difference between two subsequent models is small or beyond the cone-of-influence of the property, the invariant or counterexample from the earlier model may be directly used to verify the current model. Otherwise, FuseIC3 uses reachable state approximations as inputs to IC3 to only explore undiscovered reachable states in the current model. In the former, verification completes almost instantly, while in the latter, significant time is saved. When the stored information cannot be used directly, FuseIC3 repairs and patches it using an efficient SAT-based algorithm. The repair algorithm is the main strength of FuseIC3, and uses features present in modern SAT solvers. It adds "just enough" extra information to the saved reachable states to enable reuse. We demonstrate the industrial scalability of FuseIC3 on a large set of 1,620 real-life models for the NASA NextGen air traffic control system [19, 23], selected benchmarks from HWMCC 2015 [1], and a set of seven models for the Boeing AIR6110 wheel braking system [5]. Our experiments evaluate FuseIC3 along two dimensions; checking all models with the same property, and checking each model with several properties. Lastly, we evaluate the effect of model relatedness on the performance of FuseIC3.

Related Work The idea of reusing model-checking information, like variable orderings, between runs has been extensively used in BDD-based model checking leading to substantial performance improvement [3, 27]. Similarly, intermediate SAT solver clauses and interpolants are reused in bounded model checking [22, 25]. Reusing learned invariants in IC3 speeds up convergence of the algorithm [8]. These techniques enable efficient incremental model checking and are useful in *regression verification* [28] and *coverage computation* [9]. FuseIC3 is an

Artifacts for reproducibility, code, theorem proofs, and detailed experimental results can be found at http://temporallogic.org/research/FMCAD17. Thanks to NSF CAREER Award CNS-1552934 for supporting this work.

incremental algorithm and is applicable in these scenarios.

Product line verification techniques, e.g., with Software Product Lines (SPL), also verify models describing large design spaces [4, 13, 15, 16]. The several *instances* of feature transition systems (FTS) [14] describe a set of models. FuseIC3 relaxes this requirement and can be used to check models that cannot be combined into a FTS. It outputs model-checking results for every model-property pair in the design space without dependence on any *feature*. Nevertheless, SPL instances can be checked using FuseIC3. Large design spaces can also be generated by models that are parametric over a set of inputs. *Parameter synthesis* [10] can generate the many models in a design space that can be checked using FuseIC3. The parameterized model-checking problem [18] deals with infinite homogeneous models. In our case, the models in a set are heterogeneous and finite.

The work most closely related to ours is a state-of-the-art algorithm for incremental verification of hardware [8]. It extends IC3 to reuse the generated proof, or counterexample, in future checker runs. It extracts minimal inductive subclauses from an earlier invariant with respect to the current model. In our analysis, we compare FuseIC3 with this algorithm, and show that with the same amount of information storage, FuseIC3 is faster when checking large design spaces.

Contributions The contributions of our work are many-fold. We present a query-efficient SAT-based algorithm for checking large design spaces, and incremental verification. The algorithm is fully automated, general, and scalable. To the best of our knowledge, FuseIC3 is the first algorithm to reuse reachable state approximations to guide bad-state search in IC3. Our novel procedure to repair state approximations requires little computation effort and is of individual interest. We present an extensive experimental analysis using real-life benchmarks. Lastly, we make all reproducibility artifacts and source code publicly available.

Structure Section II details background information, overviews the typical IC3 algorithm, and defines the notation used throughout the paper. Section III presents the FuseIC3 algorithm. A large-scale experimental evaluation forms Section IV, and Section V concludes by highlighting future work and possible extensions.

II. Preliminaries

A. Definitions

A Boolean transition system, or model M is represented using the tuple (Σ, Q, Q_0, δ) where $Q_0 \subseteq Q$ is the set of *initial states* and δ is the *transition relation* over state variables Σ. A *safety property* is a predicate φ over Σ. A primed variable σ', such that $\sigma \in \Sigma$, represents σ in the next time step. If ψ is a Boolean formula over Σ, ψ' is obtained by replacing each variable in ψ with the corresponding primed variable.

A sequence of states $s_0, s_1, \ldots, s_n$ is a *path* in M if s_0 is an initial state, each $s_i \in Q$ for $0 \leq i \leq n$, and for $0 < i < n$, $(s_i, s_{i+1}) \in \delta$, i.e., there is a valid transition from s_i to s_{i+1}. A state t in a model is *reachable* if there exists an execution path such that $s_n = t$. A model M *satisfies* safety property φ, denoted $M \models \varphi$, when no reachable states of M intersect $\neg\varphi$.

The state variables and their negations are called *literals*. A disjunction of literals is called a *clause*. A Boolean formula containing a conjunction of clauses is said to be in *Conjunctive Normal Form* (CNF).

We assume that a Boolean formula ψ over Σ represents a set of states in M, or $\psi \subseteq Q$. Two Boolean formulas ψ_1 and ψ_2 over Σ *overlap* if $\psi_1 \cap \psi_2 \neq \emptyset$, i.e., they contain common symbolic states. Models M and N are *related* if they contain overlapping reachable states. A *set of models* is a collection of such related models.

B. Overview of IC3

IC3/PDR [6, 17, 26] is a novel verification method based on property directed invariant generation. Given a model $M = (\Sigma, Q, Q_0, \delta)$, and a safety property φ, IC3 incrementally generates an inductive strengthening of φ to prove whether $M \models \varphi$. It maintains a sequence of frames $S_0 = Q_0, S_1, \ldots S_k$ such that each S_i, for $0 < i < k$, satisfies φ and is an over-approximation of states reachable in i-steps or less. If two adjacent frames become equivalent, IC3 has found an inductive invariant and the property holds for the model. If a state violating the property is reachable, a counterexample trace is returned. Throughout IC3's execution, it maintains the following invariants on the sequence of frames:

1) for $i > 0$, S_i is a conjunction of clauses,
2) $S_{i+1} \subseteq S_i$,
3) $S_i \wedge \delta \Rightarrow S'_{i+1}$, and
4) for $i < k$, $S_i \Rightarrow \varphi$.

Each clause added to the frames is an intermediate lemma constructed by IC3 to prove whether $M \models \varphi$. The algorithm proceeds in two phases: a *blocking* phase, and a *propagation* phase. In the blocking phase, S_k is checked for intersection with $\neg\varphi$. If an intersection is found, S_k violates φ. IC3 continues by recursively blocking the intersecting state at S_{k-1}, and so on. If at any point, IC3 finds an intersection with S_0, $M \not\models \varphi$ and a counterexample can be extracted. The propagation phase moves forward the clauses from preceding S_i to S_{i+1}, for $0 < i \leq k$. During propagation, if two consecutive frames become equal, a fix-point has been found and IC3 terminates. The fix-point $\mathcal{I}$ represents the inductive strengthening of φ and has the following properties: $Q_0 \Rightarrow \mathcal{I}$, $\mathcal{I} \wedge \delta \Rightarrow \mathcal{I}'$, and $\mathcal{I} \Rightarrow \varphi$. We refer the reader to [7, 20] for lower-level details of IC3.

C. SAT with Assumptions

In our formulation, we consider SAT queries of the form $\mathsf{sat}(\varphi, \gamma)$, where φ is a CNF formula, and γ is a set of assumption clauses. A query with no assumptions is simply written as $\mathsf{sat}(\varphi)$. Essentially, the query $\mathsf{sat}(\varphi, \gamma)$ is equivalent to $\mathsf{sat}(\varphi \wedge \gamma)$ but the implementation is typically more efficient. If $\varphi \wedge \gamma$ is:

1) SAT, get-sat-model() returns a satisfying assignment.

2) UNSAT, get-unsat-assumptions() returns a unsatisfiable core β of the assumption clauses γ, such that $\beta \subseteq \gamma$, and $\varphi \wedge \beta$ is UNSAT.

We abstract the implementation details of the underlying SAT solver, and assume interaction using the above functions.

D. Notation

We reduce the task of verifying a set of models by restricting the description of our algorithm to two related models $M = (\Sigma, Q_M, Q_{0_M}, \delta_M)$ and $N = (\Sigma, Q_N, Q_{0_N}, \delta_N)$ in the set. Each model has to be checked against a safety property φ. Assume that model M is checked first. The algorithm computes frame sequence R and S for M and N, respectively. $|R|$ denotes number of frames in the sequence R.

III. Algorithm

In this section, we present the main contribution of our paper, FuseIC3. We start with the core idea behind the algorithm by giving the intuition behind recycling IC3-generated intermediate lemmas. We then provide a general overview of different sub-algorithms that help FuseIC3 achieve it's performance. We next describe the two main components: *basic check* and *frame repair* of FuseIC3.

A. Intuition

Recall that frames computed by IC3 represent over-approximated states. When M is checked with IC3, frames $R_0, R_1, \ldots, R_j$, are computed such that $R_i \wedge \delta_M \Rightarrow R'_{i+1}$ for $i < j$ (invariant 3, Section II-B). In the classical case, checking N after M requires resetting and restarting IC3, which then computes frames $S_0, S_1, \ldots, S_k$ for N. Due to to the reset, all intermediate lemmas are lost and verification for N has to start from the beginning. However, since M and N are related, the frames for M and N overlap, and therefore, frames for M can be recycled and potentially reused in the verification for N. The idea is illustrated using Venn diagrams in Fig. 1.

In Fig. 1a, the parallelogram and ellipse represent clauses c_1 and c_2, respectively, in frame R_{i+1} such that $R_{i+1} = c_1 \wedge c_2$, and the triangle represents states reachable from R_i in one step, i.e., $R_i \wedge \delta_M$. So, $R_i \wedge \delta_M \Rightarrow R'_{i+1}$. Now consider a scenario in which we recycle the clauses in R_{i+1} when verifying N. The triangle and the rectangle in Fig. 1b represent the states reachable from S_i in one step. If we were to make $S_{i+1} = R_{i+1}$, we end up with $S_i \wedge \delta_N \not\Rightarrow S'_{i+1}$ since c_1 doesn't contain some states reachable from S_i. Therefore, we have to modify c_1 such that the invariant holds. Fig. 1c and 1d show the two possible modifications of c_1. In the former case, we add states $(S_i \wedge \delta_N) \setminus c_1$ to c_1. In the latter, we over-approximate c_1 to $\hat{c}_1$ such that $S_i \wedge \delta_N \Rightarrow \hat{c}_1$ (a trivial over-approximation is to make c_1 equal to the set of all states). Irrespective of the approach used, we end up with $S_i \wedge \delta_N \Rightarrow \hat{R}'_{i+1} = S'_{i+1}$, where $\hat{R}_{i+1} = \hat{c}_1 \wedge c_2$. Then we check the $(i+1)$-th step over-approximation for intersection with $\neg\varphi$ and IC3 continues. In this way, reusing clauses from model M, saves a lot of effort in rediscovering these clauses for model N. FuseIC3 uses state over-approximations.

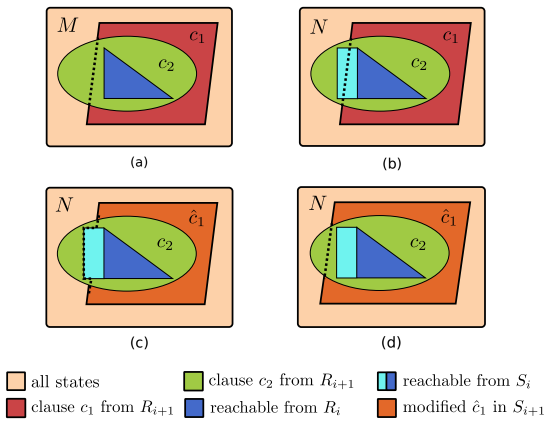

M
N
c_1
c_2
$\hat{c}_1$
(a)
(b)
(c)
(d)
all states
clause c_1 from R_{i+1}
clause c_2 from R_{i+1}
reachable from R_i
reachable from S_i
modified $\hat{c}_1$ in S_{i+1}

Fig. 1. Intuition behind repairing frames computed for one model by IC3, and reusing them for checking another related model.

```
bool FuseIC3 ($Q_0$, $\delta$, $\varphi$)
 1: if CheckInvar($Q_0$, $\delta$, last_invariant, $\varphi$) : return true
 2: if SimulateCex($Q_0$, $\delta$, last_cex, $\varphi$) : return false
 3: $k \leftarrow 0$, $S_k \leftarrow Q_0$   # first frame is initial state
 4: while true :  # main FuseIC3 loop
 5:     while sat($S_k \wedge \neg\varphi$) :  #  blocking phase
 6:         $s \leftarrow$ get-sat-model()
 7:         if not recursive_block($s, k$) :
 8:             last_cex $\leftarrow$ extract_cex(), return false
 9:     $k \leftarrow k + 1$
10:     $S_k \leftarrow$ FrameRepair($k - 1$)
11:     for $i \leftarrow 1$ to $k - 1$ :  # propagation phase
12:         for each new clause $c \in S_i$ :
13:             if not sat($S_i \wedge c \wedge \delta \wedge \neg c'$) : add $c$ to $S_{i+1}$
14:         if $S_i \equiv S_{i+1}$ :  # found fix-point invariant
15:             last_invariant $\leftarrow S_i$, return true
frame FrameRepair (int i)
 1: if not sat($S_i \wedge \delta \wedge \neg R'_{i+1}$) : return $R_{i+1}$
 2: $\mathcal{G} \leftarrow$ FindClauses($S_i$, $\delta$, $R_{i+1}$)
 3: $\hat{R}_{i+1} \leftarrow R_{i+1} \setminus \mathcal{G}$
 4: for each clause $c \in \mathcal{G}$ :
 5:     $\hat{c} \leftarrow$ ExpandClause($S_i$, $\delta$, $c$)
 6:     $\hat{c} \leftarrow$ ShrinkClause($S_i$, $\delta$, $c$, $\hat{c}$)
 7:     $\hat{R}_{i+1} \leftarrow \hat{R}_{i+1} \wedge \hat{c}$
 8: return $\hat{R}_{i+1}$  # repaired frame $R_{i+1}$
```

Fig. 2. High-level description of FuseIC3. Parts of the algorithm for typical IC3 are based on the description in [17, 20].

B. Overview

FuseIC3 is a bidirectional reachability algorithm. It uses forward reachability to reuse frames from a previously-checked related model, and IC3-type backward reachability to recursively block predecessors to bad states. The algorithm description appears in Fig. 2.

FuseIC3 takes as input the initial states Q_0 and the transition relation δ for the current model, and a safety property φ. The internal state maintained by the algorithm is last_invariant, last_cex, and the frames R computed for the last model verified. Initially, the state is empty. Lines 1–2 perform basic checks in an attempt to reuse proofs from an earlier run to verify the current model. Lines 4–15 loop until an invariant or a counterexample is found. FuseIC3 maintains a sequence of frames $S_0, S_1, \ldots, S_k$ for the current model being checked. Whenever a new frame S_k is introduced in line 10, the algorithm reuses a frame from R after repairing it with FRAMEREPAIR. The repaired frame is added to S_k, which after propagation in lines 11–15, is checked for intersection with a bad state. A typical execution of IC3 follows until a new frame is introduced. Upon termination, R is replaced with the current set of frames S, and last_invariant and last_cex are updated accordingly.

FRAMEREPAIR takes as input an integer i. It checks if R_{i+1} can be used as is in line 1. If yes, R_{i+1} is returned. Otherwise, the frame is repaired in lines 2–7. FINDCLAUSES finds violating clauses in R_{i+1}. Each of these clauses is repaired in lines 4–7 using EXPANDCLAUSE and SHRINKCLAUSE. After repair, the updated frame $\hat{R}_{i+1}$ is returned.

The models in a set are checked sequentially. When FuseIC3 is run on the first model in the set, it reduces to running typical IC3. During propagation and when $k < |R|$, only repaired clauses (from FRAMEREPAIR) and discovered clauses for the current model are propagated. When $k \geq |R|$, FRAMEREPAIR returns an empty frame and all clauses from earlier frames take part in propagation.

C. Basic Checks

It is possible that the changes in design between two models are very small, and are outside the cone-of-influence of the verification procedure. Therefore, although the models are different, they might have the same over-approximated inductive invariant with respect to the property being checked. A similar argument applies for two models that fail a property. In this case, a counterexample for the first model might be a valid counterexample for the second model. Both these checks can be carried out in very little time as explained below. For the case when M and N have different state variables, cone-of-influence with respect to variables in N is applied on the invariant/counterexample before performing the checks.

a) Inductive Invariant: If $\mathcal{I}_M$ is an inductive invariant for M with respect to φ, it satisfies the following three conditions:

1) $Q_{0_M} \Rightarrow \mathcal{I}_M$,
2) $\mathcal{I}_M \wedge \delta_M \Rightarrow \mathcal{I}'_M$, and
3) $\mathcal{I}_M \Rightarrow \varphi$.

If changes in N are outside the cone-of-influence of $\mathcal{I}_M$, then $N \models \varphi$ if the above conditions hold for N (checked using three SAT calls).

b) Counterexample Trace: If $M \not\models \varphi$, IC3 generates a counterexample trace $s_0, s_1, \ldots s_k$ such that

1) $s_0 \in Q_{0_M}$,
2) $(s_i, s_{i+1}) \in \delta_M$ for $i < k$, and
3) $s_k \in \neg\varphi$.

Simulate the counterexample trace for M on N and check if it satisfies the above three conditions (using $k+1$ SAT calls). If the conditions are satisfied, conclude that $N \not\models \varphi$.

To summarize, if changes in two subsequent models are outside the cone-of-influence of the proofs generated by IC3, verification completes almost instantly. The pseudo-code for these two basic checks is given in Fig. 3.

```
bool CHECKINVARIANT ($Q_0$, $\delta$, invariant $\mathcal{I}$, $\varphi$)
 1: if not sat($Q_0 \wedge \neg\mathcal{I}$) and not sat($\mathcal{I} \wedge \delta \wedge \neg\mathcal{I}$) and not
    sat($\mathcal{I} \wedge \neg\varphi$) : return true
 2: else return false

bool SIMULATECEX ($Q_0$, $\delta$, trace $s$, $\varphi$)
 1: if not sat($s_0 \wedge Q_0$) : return false
 2: if not sat($s_k \wedge \neg\varphi$) : return false
 3: for $i \leftarrow 0$ to len($s$) :
 4:     if not sat($s_i \wedge \delta \wedge s'_{i+1}$) : return false
 5: return true  # valid counterexample
```

Fig. 3. CHECKINVARIANT evaluates the last known invariant against the current model, and returns *true* if invariant holds, otherwise *false*. SIMULATECEX simulates the last known counterexample on the current model, and returns *true* if successful, otherwise, *false*.

```
FINDCLAUSES (frame $S$, $\delta$, frame $R$)
 1: for each clause $c_i \in R$ :  # configure solver assertions
 2:     introduce auxiliary variable $y_i$
 3:     for each literal $l \in c'_i$ :
 4:         add assertion $\neg l \vee y_i$ to solver
 5: $\mathcal{G} \leftarrow \emptyset$  # set is initially empty
 6: while sat($S \wedge \delta, (\neg y_1 \vee \neg y_2 \vee \ldots \vee \neg y_k)$) :
 7:     $\alpha \leftarrow$ get-sat-model()
 8:     for each $y_1, y_2, \ldots y_k$ :
 9:         if $\alpha(y_i)$ == $\perp$ :
10:             add $c_i$ to $\mathcal{G}$ and remove $y_i$ from sat query
11: return $\mathcal{G}$  # set of violating clauses
```

Fig. 4. FINDCLAUSES algorithm to find all clauses in R that lead to violation of $S \wedge \delta \not\Rightarrow R'$. Upon termination, $\mathcal{G}$ contains violating clauses.

D. Frame Repair

We want to expand clauses in frame R_{i+1} that are responsible for the violation of $S_i \wedge \delta_N \Rightarrow R'_{i+1}$. The satisfiability model is a pair of states (a, b) such that $a \in S_i$, $b \notin R_{i+1}$, and $(a, b) \in \delta_M$. In other words, b is missing from some, or all clauses in R_{i+1}. If all such missing states are added to clauses in R_{i+1}, resulting in $\hat{R}_{i+1}$, the condition $S_i \wedge \delta_N \Rightarrow \hat{R}'_{i+1}$ becomes valid and $\hat{R}_{i+1}$ can be reused in checking N. Adding these states one-by-one requires several calls to the underlying SAT solver and is infeasible in practice (reduces to all-SAT). Instead, the violating clauses in R_{i+1} are over-approximated. The over-approximation ends up adding several states to R_{i+1} that are in the post-image of multiple states in S_i. As the first step in repairing the frame, we want to find all such violating clauses.

Find Violating Clauses: Let's assume frame R_{i+1} is composed of clauses $C = \{c_1, c_2, \ldots c_n\}$. There are clauses $\mathcal{G} \subseteq C$ such that the assertion $S_i \wedge \delta_N \Rightarrow c'$ is violated for all $c \in \mathcal{G}$. Set $\mathcal{G}$ can be found by brute-forcing the assertion check for all clauses in C. However, such an approach doesn't scale since IC3 frames can have thousands of clauses. Algorithm FINDCLAUSES, which is inspired by the *Invariant Finder* algorithm in [8], efficiently finds such violating clauses. The pseudo-code for the algorithm is given in Fig. 4.

```
ExpandClause (frame $S$, $\delta$, clause $\hat{c}$)
 1: $v \leftarrow$ all primed variables in $\delta$
 2: $l \leftarrow$ all variables in clause $\hat{c}'$
 3: $B \leftarrow v \setminus l$  # variables not in clause $\hat{c}$
 4: while $|B| > 0$ and sat($S \wedge \delta \wedge \neg\hat{c}'$) :
 5:     $\alpha \leftarrow$ get-sat-model()
 6:     randomly pick any $b' \in B$
 7:     if $\alpha(b') == \top$ : add $b$ to clause $\hat{c}$
 8:     else if $\alpha(b') == \bot$ : add $\neg b$ to clause $\hat{c}$
 9:     remove $b'$ from $B$
10: if sat($S \wedge \delta \wedge \neg\hat{c}'$) : return $\emptyset$
11: return $\hat{c}$  # expanded clause; $S \wedge \delta \Rightarrow \hat{c}'$
```

Fig. 5. ExpandClause algorithm to add literals to clause c such that $S \wedge \delta \Rightarrow \hat{c}'$. Upon termination, an empty set is returned if expansion fails.

FindClauses takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, and frame $R = R_{i+1}$. Upon termination, it returns all violating clauses. An auxiliary variable y_i is introduced for each clause c_i in R in line 2. Lines 3–4 are equivalent to adding the assertion $c_i \Rightarrow y_i$ to the solver. Lines 6–10 loop until the query in line 6 is SAT. On every iteration of the loop, there is at least one y_i that is assigned *false*. Clauses c_i corresponding to all such y_i are added to $\mathcal{G}$ and y_i is removed from the query. When the query becomes UNSAT, $\mathcal{G}$ contains all violating clauses in R, and is returned. In practice, multiple y_i are assigned *false* which helps terminate the loop faster.

Lemma 1. FindClauses *returns all clauses in R_{i+1} that are responsible for $S_i \wedge \delta \not\Rightarrow R'_{i+1}$.*

After discovering all violating clauses, FuseIC3 attempts to expand them before reusing R_{i+1} to check model N. In the trivial case, each violating clause can be removed from R_{i+1} altogether. However, doing this is quite wasteful. For example, consider a frame in which all clauses are violating. Reusing this frame entails restarting IC3 from an empty frame, a scenario we want to avoid. Instead, we rely on efficient use of the SAT solver to over-approximate the violating clauses.

Expand Violating Clauses: A clause c is violating if none of its literals match the literals in state b (recall the model (a, b) to the SAT query $S_i \wedge \delta_N \Rightarrow R'_{i+1}$). If any literal from b is added to c, resulting in $\hat{c}$, then $b \in \hat{c}$. Fundamentally, we want to add literals to clause c without actually enumerating all such b such that the assertion $S_i \wedge \delta_N \Rightarrow \hat{c}'$ holds. A literal can be added as is, or in its negated form. Adding both makes the assertion trivially valid. For example, consider a system with variables x, y, z, and a violating clause $c = (x \vee y)$. Our aim is to add states to c. Either z or $\neg z$ can be added to c, but not both. However, deciding what to add to make the assertion valid is beyond the scope of a SAT solver. Instead, we use an efficient randomized algorithm, ExpandClause, to add literals to clause c. The pseudo-code for the algorithm is given in Fig. 5.

ExpandClause takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, and the violating clause $c \in R_{i+1}$. Initially, $\hat{c} = c$. Lines 1–3 find all variables that are missing from c and store them in set B. The loop in lines 4–9 is repeated until set B becomes empty, or the query $S \wedge \delta \Rightarrow \hat{c}'$ becomes valid. In the latter case, enough literals have been added to expand c and the algorithm can terminate. From the SAT model α, randomly pick an assignment to a variable in B. If the assignment is *true*, add the variable as is to $\hat{c}$, otherwise, negate variable and add to $\hat{c}$. The added variable is removed from B and the loop continues. When all possible variables have been added to $\hat{c}$ and the assertion is still SAT, return $\hat{c}$ to be the empty clause ($c = $ *true*, or set of all states) in line 10.

```
ShrinkClause (frame $S$, $\delta$, clause $c$, clause $\hat{c}$)
    assert(not sat($S \wedge \delta \wedge \neg\hat{c}'$))
 1: $v \leftarrow$ {literals in $\hat{c}$} $\setminus$ {literals in $c$}
 2: for each $l \in v$ :
 3:     $g \leftarrow v \setminus l$  # drop literal $l$
 4:     if not sat($S \wedge \delta \wedge \neg c', \neg g'$) :
 5:         $v \leftarrow \{$literals $j \mid j' \in$ get-unsat-assumptions()$\}$
 6: return $c \vee \bigvee\{$literals in $v\}$
```

Fig. 6. ShrinkClause algorithm to remove excess literals from clause c while maintaining $S \wedge \delta \Rightarrow c'$.

Lemma 2. ExpandClause *expands clause c to $\hat{c}$ such that the assertion $S_i \wedge \delta \Rightarrow \hat{c}'$ is valid.*

Shrink Expanded Clauses: Due to the nature of the randomized algorithm, we may end up adding more states than required to the expanded clauses. As a last step in repairing the frame, we remove the excess states added from all such clauses, albeit, maintaining the over-approximation. FuseIC3 uses UNSAT assumptions generated in the proof for $S_i \wedge \delta \Rightarrow \hat{c}'$ to shrink clause $\hat{c}$. The ShrinkClause algorithm tries dropping a subset of the newly added literals from $\hat{c}$. The pseudo-code for the algorithm is given in Fig. 6.

ShrinkClause takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, non-expanded violating clause c, and the expanded non-violating clause $\hat{c}$. Set v contains all literals that were added to c by ExpandClause. Lines 2–5 loop until enough literals have been dropped from $\hat{c}$ such that the $S_i \wedge \delta_N \wedge \neg c' \wedge \neg v'$ is valid. On each iteration of the loop, a literal l to drop from v is chosen. If the assertion is UNSAT, we can successfully drop l from v, and replace v with the UNSAT assumptions in the query. However, if the assertion is SAT, l is a required literal in v and we try dropping another literal.

Lemma 3. ShrinkClause *removes all possible literals from $\hat{c}$ such that the assertion $S_i \wedge \delta \Rightarrow \hat{c}'$ is valid.*

The violating clause may appear in future frames in R (due to the propagation phase when checking M). The modification is reflected in all occurrences of the clause. All such violating clauses in R_{i+1} are repaired.

Theorem 1. FrameRepair *returns repaired frame $\hat{R}_{i+1}$ such that $S_i \wedge \delta \Rightarrow \hat{R}'_{i+1}$ is valid.*

The repaired frame $\hat{R}_{i+1}$ is added to the set of frames for N at step $i+1$. Therefore, $S_{i+1} = \hat{R}_{i+1}$, and we continue with the normal execution of IC3. Clauses are propagated from frames

S_j, for $j \leq i$, to S_{i+1}, which is then checked for intersection with bad states and, if any are found, IC3 tries recursively blocking them at earlier steps.

IV. Experimental Analysis

We extensively experimentally analyze FuseIC3. We summarize the setup used for the experiments, briefly detail our benchmarks, and end with experimental results.

A. Setup

FuseIC3 is coded in C++ and uses MathSAT5 [11] as the underlying SAT solver. It takes SMV models or AIGER files as input. The IC3 part of FuseIC3 is based on the description in [17] and `ic3ia`.[1] We compare the performance of FuseIC3 with typical IC3 (typ), and incremental IC3 (inc). The algorithm for incremental IC3 is part of IBM's RuleBase model checker [3]. We coded inc based on the description in [8] to the best of our understanding. All experiments were performed on Iowa State University's Condo Cluster comprising of nodes having two 2.6GHz 8-core Intel E5-2640 processors, 128 GB memory, and running Enterprise Linux 7.3. Each model-checking run had exclusive access to a node.

B. Benchmarks

We evaluated FuseIC3 over a large collection of challenging benchmarks. The benchmarks are derived from real-world case studies and modified benchmarks from HWMCC 2015.

1) Air Traffic Controller (ATC) Models: are a large set of 1,620 real-world models representing different possible designs for NASA's NextGen air traffic control (ATC) system [19]. The set of models are generated from a contract-based, parameterized NUXMV model. Each model is checked against 34 safety properties. The entire evaluation consists of 34 model-sets (one for each property) containing 1,620 models.

2) Selected Benchmarks from HWMCC 2015: We considered a total of 548 benchmark models from the single safety property track [1]. Of the 548, 110 models were solved using our implementation of IC3 within a timeout of 5 minutes. To create a model-set, we generated 200 mutations of each of the 110 benchmarks. The original model was mutated to only modify the transition system, and not the safety property implicit in the AIGER file; 1% of the assignments were randomly modified. An assignment of the form $g = g_1 \wedge g_2$ was selected with probability 0.01 and changed to $g = 0$, $g = 1$, $g = \neg g_1 \wedge g_2$, $g = g_1 \wedge \neg g_2$, $g = \neg g_1 \wedge \neg g_2$, $g = g_1 \wedge g_2$, $g = g_1$, $g = \neg g_1$, $g = g_2$, or $g = \neg g_2$, with equal probability. Therefore, the full evaluation consists of 110 model-sets, each consisting of one property and 200 models.

3) Wheel Braking System (WBS) Models: are a set of seven real-world models representing possible designs for the Boeing AIR6110 wheel braking system [5]. Each model is checked against $\sim$300 safety properties. However, the properties checked for each model are not the same. We evaluate FuseIC3 using this benchmark to measure performance when a model is checked against several related or similar properties. Each model was checked using a timeout of 120 minutes.

[1]https://es-static.fbk.eu/people/griggio/ic3ia/

TABLE I
Summary of results for 34 sets of 1,620 models each for NASA Air Traffic Control System.

Algorithm	Cumulative Time in minutes	Median Speedup	
		v/s typ (avg)	v/s inc (avg)
Typical IC3 (typ)	2502.70	-	-
Incremental IC3 (inc)	2180.57	1.29 (1.3)	-
FuseIC3	**1683.53**	**1.75** (5.48)	**1.34** (3.67)

C. Results

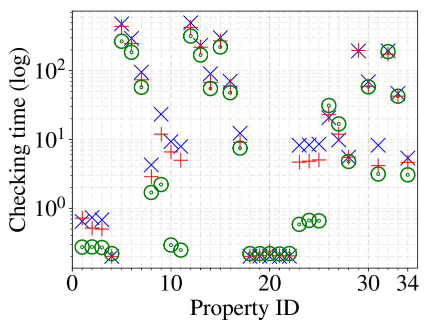
1) Air Traffic Controller (ATC) Models: Each of the 34 model-sets were checked using a timeout of 720 minutes per algorithm. The models in a set were checked in random order. Table I gives a summary of the results. FuseIC3 is median 1.75× (average 5.48×) faster compared to typical IC3, and median 1.34× (average 3.67×) faster compared to incremental IC3. On the other hand, incremental IC3 is median 1.29× (average 1.3×) faster than typical IC3.

Fig. 7a shows time taken by the algorithms on each model-set. FuseIC3 is almost always faster than typical IC3, and incremental IC3. However, for some very small instances, typical IC3 is faster; both incremental IC3 and FuseIC3 require a certain overhead in extracting information from the last checker run. FuseIC3 tries minimizing the time spent in exploring the common state space between models. In terms of the IC3 algorithm, this relates to time spent in finding bad states and blocking them at earlier steps (blocking phase). Fig. 7b shows time taken by each algorithm in blocking discovered bad states. FuseIC3 spends considerably less time in the blocking phase compared to typical IC3 and incremental IC3. Therefore, FuseIC3 is successful in reusing a major part of the already-discovered state space between different checker runs, a major requirement when checking large design spaces. Fig. 7c shows the total number of calls made to the underlying SAT solver by each algorithm. FuseIC3 makes fewer SAT calls and takes less time to check each model-set. For small models, FuseIC3 makes more SAT calls compared to typical IC3.

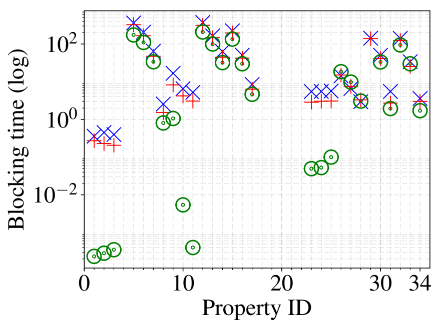
2) Benchmarks from HWMCC 2015: Each of the 110 model-sets were checked using a timeout of 120 minutes per algorithm. The models in a set were checked in random order. 91 of 110 model-sets were solved by all algorithms within the timeout. Incremental IC3 solved two more model-sets compared to typical IC3, while FuseIC3 solved five more compared to typical IC3. Table II gives a summary of results.

Fig. 8a shows time taken by the algorithms in checking each benchmark model-set. FuseIC3 is median 1.75× (average 3.18×) faster than typical IC3, and median 1.72× (average 2.56×) faster than incremental IC3. Significant speedup is achieved when checking model-sets containing large models with FuseIC3. Performance for model-sets containing small models is similar for all algorithms. Fig. 8b shows time spent by each algorithm in blocking predecessors to bad states.

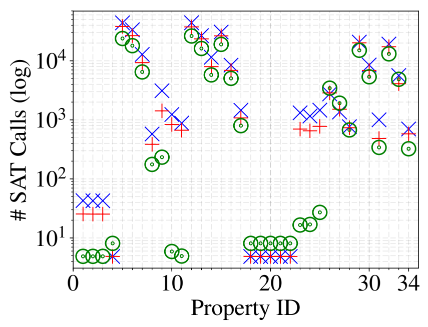
To estimate performance of FuseIC3 on model-sets with varying degree of overlap among models, we picked the `bobtuint18neg` benchmark from HWMCC 2015. 100



(a) Cumulative checking time per set (minutes)

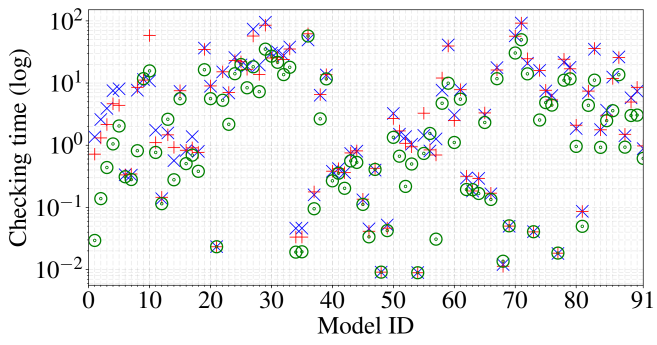


(b) Cumulative blocking time per set (minutes)

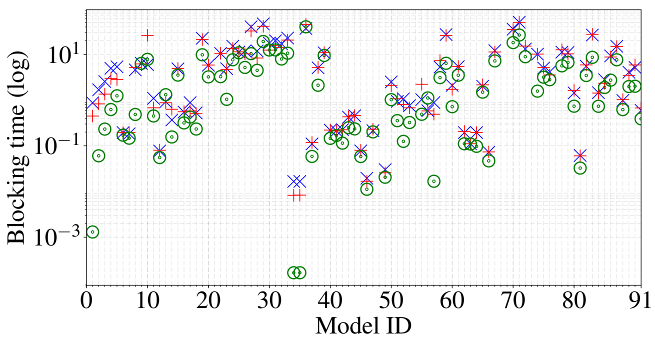


(c) Number of SAT calls per set

Fig. 7. Comparison between IC3 (×), incremental IC3 (+), and FuseIC3 (⊙) on NASA Air Traffic Control System models. There are a total of 34 properties. 1,620 models are checked per property. A point represents cumulative time taken to check all models for a property by an algorithm.



(a) Cumulative checking time per design space (minutes)



(b) Cumulative blocking time per design space (minutes)

Fig. 8. Comparison between IC3 (×), incremental IC3 (+), and FuseIC3 (⊙) on 91 benchmarks from HWMCC 2015. Each model is converted to a model-set containing 200 models, generated by 1% mutation of the original. A point represents cumulative time for checking all mutated versions of a model.

TABLE II
SUMMARY OF RESULTS FOR 91 OF 110 SETS OF 200 MODELS EACH FOR SELECTED HWMCC 2015 BENCHMARKS.

Algorithm	Cumulative Time in minutes	Median Speedup	
		v/s typ (avg)	v/s inc (avg)
Typical IC3 (typ)	1024.60	-	-
Incremental IC3 (inc)	1026.30	1.04 (1.07)	-
FuseIC3	**545.31**	**1.75** (3.18)	**1.72** (2.56)

model-sets with varying degrees of mutation, between 0.5% to 50%, of the original model were generated. Each model-set consists of 100 models each. Each set was checked using a timeout of 60 minutes per algorithm. Model-sets corresponding to higher mutation values time out (SAT solvers are tuned for practical designs and random mutations create SAT instances that don't always correspond to real designs). However, FuseIC3 is able to verify more models in a set for almost all mutation percentages. Fig. 9 gives a summary of the adjusted relative speedup between checking using FuseIC3 versus typical IC3. Even at higher mutation percentages, FuseIC3 is significantly faster.

3) Wheel Braking System Models: A model in the design space was checked against several properties, differently from the other benchmarks that checked all models in a set with the same property. Each model was checked using a timeout of 120 minutes. The properties for each model were checked

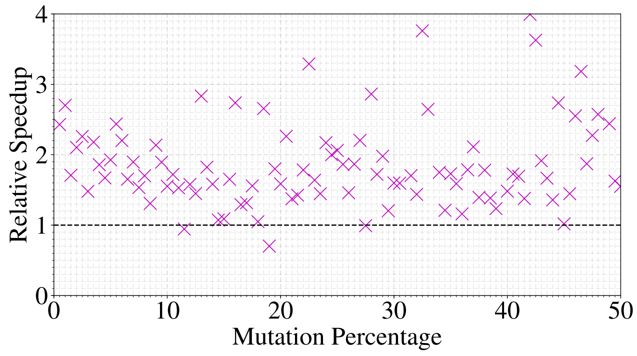


Fig. 9. Adjusted relative speedup between checking using FuseIC3 versus typical IC3. 100 models are generated for every mutation percentage between 0.5% to 50% in steps of 0.5%, and are checked against the same property.

in random order. Table III gives a summary of the results.

Compared to other benchmarks, FuseIC3 achieves a smaller speedup when checking the WBS models. Although some properties being checked for the models are similar, i.e., the bad states representing the negation of the property overlap, the order in which they are checked greatly influences the performance of FuseIC3. In the random ordering used for the experiment, FuseIC3 is able to reuse frames without any repair (the same model is being checked), however, it spends a lot of time in blocking predecessors to bad states. Nevertheless, it is faster than checking all properties on a model using typical

TABLE III
COMPARISON BETWEEN TYPICAL IC3, INCREMENTAL IC3, AND FUSEIC3 FOR AIR6110 WHEEL BRAKING SYSTEM (TIME IS IN MINUTES).

	Typical IC3	Incremental IC3		FuseIC3		
Model	Time	Time	v/s typ	Time	v/s typ	v/s inc
M_1	4.36	5.02	0.87	**3.72**	1.17	1.35
M_2	15.78	16.65	0.95	**14.80**	1.07	1.13
M_3	12.43	13.48	0.92	**11.24**	1.11	1.20
M_4	12.45	13.66	0.91	**11.09**	1.12	1.23
M_5	15.92	17.04	0.93	**14.71**	1.08	1.16
M_6	**16.85**	17.79	0.95	17.04	0.99	1.04
M_7	12.95	13.67	0.95	**12.12**	1.07	1.13
	90.73 (total)	97.31 (total)	0.95 (median)	84.72 (total)	1.11 (median)	1.20 (median)

IC3. On the other hand, incremental IC3 is slower compared to typical IC3. It is able to extract the minimal inductive invariant (invariant finder) instantly, however, suffers from the same problem as FuseIC3. Incremental IC3, and FuseIC3 will benefit if similar properties are checked in order.

V. CONCLUSIONS AND FUTURE WORK

FuseIC3, a SAT-query efficient algorithm, significantly speeds up model checking of large design spaces. It extends IC3 to minimize time spent in exploring the state space in common between related models. FuseIC3 spends less time during the blocking phase (Fig. 7b and Fig. 8b) due to success in reusing several clauses, has to learn fewer new clauses, and makes fewer SAT queries. The smallest salvageable unit in FuseIC3 is a clause; due to this granularity, FuseIC3 is able to selectively reuse stored information and is faster than the state-of-the-art algorithms that rely on reusing a coarser CNF invariant [8]. FuseIC3 is industrially applicable and scalable as witnessed by its superior performance on a real-life set of 1,620 NASA air traffic control system models (achieving an average 5.48× speedup), and benchmarks from HWMCC 2015 (achieving an average 3.18× speedup). Despite spending significant time in learning new clauses for the Boeing wheel braking system models, FuseIC3 is still faster than the previous best algorithm, typical IC3, when checking properties in random order; FuseIC3's performance will improve if similar properties are checked in order. We contribute to the available benchmarks by releasing all artifacts for reproducibility.

Ordering of models and properties in the design space improves the performance of FuseIC3, much like variable ordering in BDDs. Heuristics for optimizing model ordering are a promising topic for future work. Preprocessing the models and properties, based on knowledge about the design space, before checking them with FuseIC3 may remove redundancies in the design space. We plan to extend FuseIC3 to checking liveness properties by using it is a safety checker [12]. We also to plan to investigate extending FuseIC3 to reuse intermediate results of SAT queries, generalized clauses, and IC3 proof obligations across models. Finally, since checking large design spaces is becoming commonplace, we plan to develop more model-set benchmarks and make them publicly available.

REFERENCES

[1] "HWMCC 2015," http://fmv.jku.at/hwmcc15/.
[2] C. Bauer, K. Lagadec, C. Bès, and M. Mongeau, "Flight control system architecture optimization for fly-by-wire airliners," *J. Guidance, Control, and Dynamics*, vol. 30, no. 4, 2007.
[3] I. Beer, S. Ben-David, C. Eisner, and A. Landver, "RuleBase: An industry-oriented formal verification tool," in *DAC*, 1996.
[4] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu, "Symbolic model checking of product-line requirements using SAT-based methods," in *ICSE*, vol. 1, 2015, pp. 189–199.
[5] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, "Formal design and safety analysis of AIR6110 wheel brake system," in *CAV*, 2015.
[6] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.
[7] A. R. Bradley, "Understanding IC3," in *SAT*, 2012, pp. 1–14.
[8] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental Formal Verification of Hardware," in *FMCAD*, 2011, pp. 135–143.
[9] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," *FMSD*, vol. 28, no. 3, pp. 189–212, 2006.
[10] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *FMCAD*, 2013, pp. 165–168.
[11] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *TACAS*, 2013, pp. 93–107.
[12] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, 2012, pp. 52–59.
[13] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking software product lines with snip," *(STTT)*, pp. 1–24, 2012.
[14] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1069–1089, 2013.
[15] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE*, 2011, pp. 321–330.
[16] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE*, 2010, pp. 335–344.
[17] N. Een, A. Mishchenko, and R. Brayton, "Efficient Implementation of Property Directed Reachability," in *FMCAD*, 2011, pp. 125–134.
[18] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *CADE*, 2000, pp. 236–254.
[19] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, and K. Y. Rozier, "Model checking at scale: Automated air traffic control design space exploration," in *CAV*, 2016.
[20] A. Griggio and M. Roveri, "Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 6, pp. 1026–1039, Jun 2016.
[21] P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "On modelling and verifying railway interlockings: Tracking train lengths," *Science of Computer Programming*, vol. 96, no. 3, 2014.
[22] J. Marques-Silva, "Interpolant learning and reuse in sat-based model checking," *Theoretical Computer Science*, vol. 174, no. 3, pp. 31 – 43, 2007.
[23] C. Mattarei, A. Cimatti, M. Gario, S. Tonetta, and K. Y. Rozier, "Comparing different functional allocations in automated air traffic control design," in *FMCAD*, 2015.
[24] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Defining and model checking abstractions of complex railway models using CSP—B," in *HVC*, 2013.
[25] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental bounded model checking for embedded software," *Formal Aspects of Computing*, 2016.
[26] F. Somenzi and A. R. Bradley, "IC3: Where Monolithic and Incremental Meet," in *FMCAD*, 2011, pp. 3–8.
[27] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi, "A performance study of bdd-based model checking," in *FMCAD*, 1998, pp. 255–289.
[28] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *ICSM*, 2009, pp. 115–124.

FAR-Cubicle - A new reachability algorithm for Cubicle

Sylvain Conchon*† Amit Goel‡ Sava Krstić§ Rupak Majumdar¶ Mattias Roux*

*LRI, Université Paris Sud CNRS, Orsay F-91405
†INRIA Saclay – Ile-de-France, Orsay cedex, F-91893
‡Apple
§Intel Corporation
¶Max Planck Institute for Software Systems

Abstract—**We present a fully automatic algorithm for verifying safety properties of parameterized software systems. This algorithm is based on both IC3 and Lazy Annotation. We implemented it in Cubicle, a model checker for verifying safety properties of array-based systems. Cache-coherence protocols and mutual exclusion algorithms are known examples of such systems. Our algorithm iteratively builds an abstract reachability graph refining the set of reachable states from counter-examples. Refining is made through counter-example approximation. We show the effectiveness and limitations of this algorithm and tradeoffs that results from it.**

1. Introduction

We describe FAR (Forward Abstracted Reachability), an algorithm for fully automatic verification of parameterized software systems. A parameterized system describes a family of programs such as cache coherence protocols where the number of processes involved can change but the algorithm handling their behaviour is the same for all of them. Thus, the parameter allows to talk about these algorithms without knowing the actual number of processes that will be involved and then to prove its safety regardless of this number.

Safety properties state that "nothing bad happens" in our parameterized system. Verifying them can be reduced to finding an invariant of it. Building an invariant can be hard (and even undecidable [1]). The standard approach to do so is to find a formula Φ such that Φ is an inductive invariant of the system (*i.e.* the initial state of the system satisfies Φ and taking a transition from a state satisfying Φ leads to another state satisfying Φ).

In this paper we describe an algorithm for the automatic construction of inductive invariants for array-based systems (Section 2). This algorithm, based on both IC3 [2] and Lazy Abstraction [3], builds an inductive invariant by unwinding a graph (Section 3) building a forward abstract reachability of our system. This unwinding is described as a set of non-deterministic rules. We then provide an implementation in Cubicle [4], [5], [6] (Section 5) of these rules and test its effectiveness on several cache coherence protocols (Section 6).

2. Array-based Systems

An array-based system is described in [7] as first-order logic formulas on arrays. Such a system can be described as a set of basic types, a set X of *system variables* associated to type (built as usual with basic types and standard constructions), a formula *init* representing the initial states and a set Δ of transition rules $\tau_i(X, X')$ (X' is the set X where all the variables are primed which represents the *next state* reached after the application of a transition). Since we work on *parameterized* programs, our arrays are indexed by an infinite type *proc*.

We describe the Dekker mutual exclusion algorithm as an array-based system. Each process has two boolean variables, *want* (stating that the process wants to enter in critical section or not) and *crit* (stating that the process is in critical section or not). There is a global variable *turn* of type *proc* that tracks which process can go into the critical section. Since we work in the array-based systems fragment, we represent the local variables as arrays indexed by processes and containing booleans. The set X contains two arrays, *want[proc] : bool* and *crit[proc] : bool* and the global variable *turn : proc*. Initially, no process is or wants to be in critical section. Three transitions can be triggered, one to require an access to the critical section, one to enter in it and one to exit it. According to the previous description, we write this algorithm as in Figure 1.

Since we focus on safety problems (*nothing bad happens*), we need to define what is considered as *bad states*. For Dekker algorithm, these states would be defined with the following formula :

$$\mathcal{U} \equiv \exists p_1,\ p_2.\ p_1 \neq p_2\ \wedge\ \texttt{crit}[p_1]\ \wedge\ \texttt{crit}[p_2]$$

Our goal is then to prove that no state represented by $\mathcal{U}$ is reachable from $init$ (which can be formulated as : there exists no path $init = X_0 \xrightarrow{p_1} X_1 \xrightarrow{\cdots} \dots \xrightarrow{p_n} X_n = \mathcal{U}$ with $p_i \in \{req, enter, exit\}$). To do so on parameterized systems, one of the main algorithms came from Ghilardi et al. with MCMT [8]. It builds the set of all reachable states by *backward reachability* (starting, then, from the unsafe state) and checks if this set contains an initial state. In this paper,

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

$\texttt{turn} : proc$
$\texttt{crit}[proc] : bool$
$\texttt{want}[proc] : bool$

$init : \forall p.$	$\neg\texttt{want}[p] \wedge \neg\texttt{crit}[p]$
$req : \exists p.$	$\neg\texttt{want}[p]$ $\texttt{want}'[p]$
$enter : \exists p.$	$\texttt{want}[p] \wedge \texttt{turn} = p$ $\texttt{crit}'[p]$
$exit : \exists p_1,\ p_2.$	$\texttt{crit}[p_1]$ $\neg\texttt{want}'[p_1] \wedge \neg\texttt{crit}'[p_1]$ $\wedge\texttt{turn}' = p_2$

Figure 1. Dekker algorithm as array-based system

we implement a different algorithm which offers a wider range of possibilities in terms of reachabilty construction.

3. Program unwinding

This algorithm also starts from $\mathcal{U}$ but tries to build an invariant of the system that does not contain it. Before going into details, we give a brief explanation. This invariant is iteratively built as an inductive invariant Θ that does not contain $\mathcal{U}$:

- if $\Theta \wedge \Delta \wedge \neg\Theta'$ is unsatisfiable then we found an inductive invariant
- if $\Theta \wedge \Delta \wedge \neg\Theta'$ is satisfiable, our candidate invariant is not inductive and we try to refine it until we either discover that there is no such refinement or we find some.

For Dekker algorithm, for example, let's take $\Theta = \neg\mathcal{U} = \forall p_1 \neq p_2. \neg crit[p_1] \vee \neg crit[p_2]$:

- $\Theta \wedge \Delta \wedge \neg\Theta'$ is satisfiable (if we take, for example, the following state : $\varphi_1 = crit[p_1] \wedge want[p_2] \wedge turn = p_2 \wedge \neg crit[p_2]$, $\varphi_1 \models \Theta$ but if we apply *enter* to it we obtain the state $\varphi_2 = crit[p_1] \wedge crit[p_2] \wedge \ldots$ and $\varphi_2 \not\models \Theta$.)
- We need to create $\Theta'' = \Theta \wedge \rho$ which is a refinement of Θ that does not contain φ_1.

To do so, we build an *unwinding* of the algorithm as a quadruple $\langle V, E, \mathcal{W}, \mathcal{B}\rangle$, where:

- $\langle V, E\rangle$ is a rooted graph with edges labeled by transitions from Δ;
- $\mathcal{W}$ associates a formula (called *world of the vertex*) to each vertex;
- $\mathcal{B}$ associates a formula (called *bad part of the vertex*) to each vertex.

This graph contains three initial vertices :

ϵ	: the root vertex, $\mathcal{W}(\epsilon) = init$ and $\mathcal{B}(\epsilon) = \bot$;
β	: the unsafe vertex, $\mathcal{W}(\beta) = \top$ and $\mathcal{B}(\beta) = \mathcal{U}$;
ω	: the sink vertex, $\mathcal{W}(\omega) = \bot$ and $\mathcal{B}(\omega) = \bot$.

We define $V^\epsilon = \{v \in V,\ \epsilon \xrightarrow{*} v \in E\}$ (*i.e.* the set of vertices that are linked to the root) and $(F \models_f G) \equiv (F \wedge f \models G)$

The idea behind this unwinding it that if we manage to create a graph $\mathcal{G}$ of a system $\mathcal{S} = \langle init, \Delta\rangle$ where every vertex in V^ϵ does not contain a bad part and from which no more transitions can be taken, then the disjunction of their worlds ($\Theta = \bigvee_{v \in V^\epsilon} \mathcal{W}(v)$) is an invariant of the system ($init \models \Theta$ and $\Theta \models_\Delta \Theta$).

We now propose a set of non-deterministic rules for building this unwinding. Let $\langle X, init, \Delta, \mathcal{U}\rangle$ be an array-based system. Initially, $\mathcal{G}$ is defined as follow :

- $V = \{\epsilon, \omega, \beta\}$
- $E = \emptyset$

The unwinding works by the non-deterministic application of the following rules :

Rule 1 (Extend). *If* $\exists v \in V, \tau \in \Delta.\ \mathcal{W}(v) \models_\tau \top$ *and* $\nexists v'. v \xrightarrow{\tau} v' \in E$ *then* $E = E \cup \{v \xrightarrow{\tau} \beta\}$

Rule 2 (Refine). *If* $\exists v, v' \in V, \tau \in \Delta.\ v \xrightarrow{\tau} v' \in E$, $\mathcal{B}(v') \neq \bot$, $\exists \varphi. \mathcal{W}(v) \models_\tau \varphi$ *and* $\varphi \models \neg\mathcal{B}(v')$ *then we create a new vertex* v'' *such that* $\mathcal{W}(v'') = \mathcal{W}(v') \wedge \varphi$ *and* $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$

Rule 3 (Propagate). *If* $\exists v, v' \in V, \tau \in \Delta.\ v \xrightarrow{\tau} v' \in E$, $\mathcal{B}(v') \neq \bot$, $\exists\gamma.\ \gamma \models \mathcal{W}(v)$, *and* $\gamma \models_\tau \mathcal{B}(v')$ *then* $B(v) \leftarrow \gamma$

Rule 4 (Cover). *If* $\exists v, v' \in V, \tau \in \Delta.\ v \xrightarrow{\tau} v' \in E$, $v'' \in V$ *such that* $\mathcal{W}(v'') \models \mathcal{W}(v')$ *and* $\mathcal{W}(v) \models_\tau \mathcal{W}(v'')$ *then* $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$

Rule 5 (Sink). *If* $\exists v \in V, \tau \in \Delta.\ \mathcal{W}(v) \models_\tau \bot$ *and* $\nexists v'. v \xrightarrow{\tau} v' \in E$ *then* $E = E \cup \{v \xrightarrow{\tau} \omega\}$

$\mathcal{S}$ is safe if and only if no rule can be applied to $\mathcal{G}$, an unwinding of it and $\mathcal{B}(\epsilon) = \bot$. Intuitively, since no more transitions can be taken and all the vertices connected to the root are not bad, *init* will never be able to lead to $\mathcal{U}$.

4. Example

The example shown in Figure 2 describes the first four runs of the unwinding on the Dekker algorithm (we decided not to show ω since it just serves as a sink for the transitions that can not be taken from a vertex) :

(a) initially, the only rule that can be applied is **Extend** from ϵ (with $\mathcal{W}(\epsilon) \equiv init \equiv \forall p.\ \neg\texttt{want}[p] \wedge \neg\texttt{crit}[p]$) with req;
(b) then we can only apply **Refine** because $init \not\models_{req} \mathcal{U} \equiv \mathcal{B}(\beta)$. We create a new vertex called v_1;
(c) we can chose here to apply **Extend** from the new vertex with any transition. We chose to take the transition req;
(d) $\mathcal{W}(v_1) \not\models_{req} \mathcal{B}(\beta)$ and $\mathcal{W}(v_1) \models_{req} \mathcal{W}(v_1)$ so we can apply the cover rule;
(e) if we keep applying these rules, we reach a fixpoint at the third created vertex.

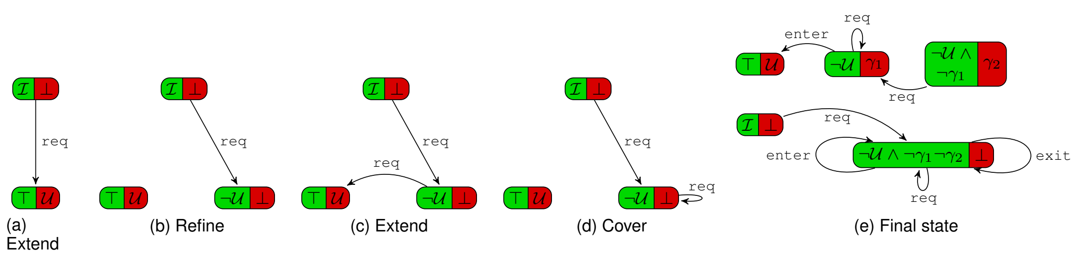


Figure 2. First four steps of the unwinding and final state of the graph (*sink rules are not shown)*

5. Implementation

We implemented the unwinding algorithm in Cubicle [1]. To do so we had to chose a deterministic strategy depending on multiple parameters :

- the order in which the rules are applied;
- which vertex and transition should be taken for the **Extend** rule;
- which formula φ should we take for the **Refine** rule;
- which formula γ should we take for the **Propagate** rule;
- which vertex v'' should we take for the **Cover** rule.

We came out with the following algorithm where we write $v = (W, B)$ to denote the fact that $\mathcal{W}(v) = W$ and $\mathcal{B}(v) = B$:

Algorithm 1 Graph unwinding - main loop

```
procedure FAR-CUBICLE($\mathcal{S} = \langle init, \Delta, \mathcal{U}\rangle$)
    $\epsilon \leftarrow (init, \bot)$
    $\beta \leftarrow (\top, \mathcal{U})$
    $\omega \leftarrow (\bot, \bot)$
    $V \leftarrow \{\epsilon, \beta, \omega\}$
    $E = \emptyset$
    PUSH($\mathcal{Q}$, $\epsilon$)                    ▷ $\mathcal{Q}$ is a priority queue
    while NOT_EMPTY($\mathcal{Q}$) do
        $v \leftarrow$POP($\mathcal{Q}$)
        for all $\tau \in \Delta$ do
            if $\mathcal{W}(v) \models_\tau \top$ then
                $E = E \cup \{v \xrightarrow{\tau} \beta\}$
                UNWIND($v \xrightarrow{\tau} \beta$)
            else $E = E \cup \{v \xrightarrow{\tau} \omega\}$
    return safe
```

This algorithm picks a vertex v from a priority queue (which initially contains only the root vertex) and for all the transitions, adds an edge to the graph from this transition to the sink vertex if the formula represented by v is inconsistent with the transition or to the unsafe vertex if the transition can be taken. If the edge goes to a vertex v' that is not the sink, the procedure UNWIND is called on it. This procedure checks if $\mathcal{B}(v) \neq \bot$ or if the $\mathcal{B}(v') = \bot$ and if both these conditions are false it tries to *close* the edge. An edge is *closed* if :

1. cubicle.lri.fr/far

Algorithm 2 Graph unwinding - unwinding procedure

```
procedure UNWIND($v \xrightarrow{\tau} v'$)
    if $\mathcal{B}(v) = \bot \ \wedge \ \mathcal{B}(v') \neq \bot$ then
        switch CLOSE($v \xrightarrow{\tau} v'$) do        ▷ See Algorithm 3
            case Covered v”
                $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$
                UNWIND($v \xrightarrow{\tau} v''$)
            case Bad $\varphi$
                if $v = \epsilon$ then return unsafe
                else
                    $\mathcal{B}(v) \leftarrow \varphi$
                    for all $u \xrightarrow{\tau'} v$ do
                        UNWIND($u \xrightarrow{\tau} v$)
            case Refined v”
                $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$
                PUSH($\mathcal{Q}, v''$)
```

- $\mathcal{W}(v) \models_\tau \mathcal{B}(v')$. In this case, all the edges coming to it must be unwinded again;
- there exists another vertex v'' such that $v \models_\tau v''$ and $\mathcal{W}(v'') \models \mathcal{W}(v')$. In this case, the edge from v to v' is deleted and a new one from v to v'' is created and unwinded;
- $\mathcal{W}(v) \not\models_\tau \mathcal{B}(v')$. A counter example φ is found a new node v'' is created with $\mathcal{W}(v'') \equiv \mathcal{W}(v') \wedge \varphi$ and pushed in the queue.

If all the edges are closed and the queue is empty, the system is safe. If the propagation of bad parts reaches the root vertex, the system is unsafe.

Algorithm 3 Graph unwinding - closing edge procedure

```
1: procedure CLOSE($v \xrightarrow{\tau} v'$)
2:     if $\exists v''.\ \mathcal{W}(v'') \models \mathcal{W}(v') \ \wedge \ \mathcal{W}(v) \models_\tau \mathcal{W}(v'')$ then
3:         return Covered $v''$
4:     else if $\mathcal{W}(v) \models_\tau \mathcal{B}(v')$ then
5:         return Bad PRE($\mathcal{B}(v'), \tau$)
6:     else
7:         $v'' \leftarrow (\mathcal{W}(v') \ \wedge \ $ GENERALIZE($\neg\mathcal{B}(v')$)$, \bot)$
8:         return Refined v”
```

As we can see on line 5 of the CLOSE procedure, the formula γ chosen for the **Propagate** rule is the pre image of the bad formula of the vertex v'. Also, on line 7 of the CLOSE procedure, the formula φ chosen for the **Refine** rule is a generalization of the negation of the bad part of the vertex v'. These are, of course, implementation choices. Other implementation could involve *model finding*, *interpolants* ... In our case, the generalization is a naive one consisting in taking the smallest part of the resulting formula that was not already taken and that still satifies the conditions of the **Refine** rule.

6. Benchmarks

We compared our implementation to the backward reachability algorithm already implemented in Cubicle (without the invariants inference implemented with BRAB [4], [6]) and obtained the following results (the timeout was set to 5 minutes and the α version uses an abstraction engine related to the approximation implemented in BRAB to get better refinement):

Protocol	Cubicle	FAR	FAR-α
`dekker`	0.04s	0.04s	0.03s
`mux_sem`	0.04s	0.05s	0.03s
`german`-ish	0.06s	0.1s	0.55s
`german`-ish2	0.13s	0.11s	0.65s
`german`-ish3	1.2s	8.3s	0.65s
`german`-ish4	3.5s	2.5s	0.75s
`german`-ish5	1.9s	8.2s	0.60s
`german`	18s	5.8s	4.25s
`szymanski_at`	TO	13s	2.60s
`szymanski_na`	TO	TO	16s

As we can see in this table, this algorithm is competitive and even better when good refinements can be found.

7. Related Works

There has been a lot of research in software model checking and Property-Driven Reachability. This type of algorithm was first introduced by Bradley in [2] and McMillan revisited his Lazy Annotation (which shares similarities with PDR algorithms) in [9] or the recent approach from Cimatti et al. [10] and Z3 with a PDR approach in [11] and [12]. Even though some of these tools are supposed to work on parameterized systems, we were either not able to find them or they were not able to prove our examples.

8. Conclusion

We presented the problem of parameterized protocol verification and gave an algorithm to automatically do it. This new algorithm was implemented in Cubicle and successfully applied to many cache coherence protocols.

This algorithm could be improved with a better generalisation engine (allowing to explore less vertices), an incremental approach (the parameterized aspect of our language makes it hard to *remember* the state of our SMT solver). Other optimizations could involve a novel way of refining our formulas (it is clear that the best refinements are inductive invariants but it is still an open problem as how to find these).

Some optimizations were not documented in this article such as

- *Set-theoretic test* : some formulas are trivially unsatisfiable and don't require call to the SMT solver;
- *relevant instantiations* : handling universally quantified formulas can lead to multiple useless instantiations that are trivially unsatisfiable or valid and do not help the SMT solver to solve the whole formula. This optimization allows to gain a significant time in the SMT solver.
- *selecting good bads* : handling bad parts from the ones with less processes involved allows to control the number of processes that have to be instantiated when checking the satisfiability of formulas. It is mandatory, if we want to have a competitive algorithm, that we handle the bad parts cleverly (this can be done in the priority queue).

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS*, 1996.

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

[3] K. L. McMillan, "Lazy abstraction with interpolants," *CAV*, pp. 123–126, 2006.

[4] A. Mebsout, "Inférence d'invariants pour le model checking de systèmes paramétrés," Ph.D. dissertation, 2014.

[5] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper," in *CAV*, 2012, pp. 718–724.

[6] ——, "Invariants for finite instances and beyond," in *FMCAD*, 2013, pp. 61–68.

[7] S. Ghilardi and S. Ranise, "Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis," *LMCS*, vol. 6, no. 4, 2010.

[8] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards SMT model checking of array-based systems," in *Automated Reasoning, 4th International Joint Conference, IJCAR 2008*, 2008, pp. 67–82.

[9] K. L. McMillan, "Lazy annotation revisited," in *Computer Aided Verification - 26th International Conference, CAV 2014*, 2014, pp. 243–259.

[10] 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.

[11] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, 2012, pp. 157–171.

[12] K. Hoder, N. Bjørner, and L. M. de Moura, "*μZ*- an efficient engine for fixed points with constraints," in *Computer Aided Verification - 23rd International Conference, CAV 2011*, 2011, pp. 457–462.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

THETA: a Framework for Abstraction Refinement-Based Model Checking

Tamás Tóth[†*], Ákos Hajdu[†‡], András Vörös[†‡], Zoltán Micskei[†] and István Majzik[†]
[†]Budapest University of Technology and Economics,
Department of Measurement and Information Systems
[‡]MTA-BME Lendület Cyber-Physical Systems Research Group
Email: {totht, hajdua, vori, micskeiz, majzik}@mit.bme.hu

***Abstract*—In this paper, we present THETA, a configurable model checking framework. The goal of the framework is to support the design, execution and evaluation of abstraction refinement-based reachability analysis algorithms for models of different formalisms. It enables the definition of input formalisms, abstract domains, model interpreters, and strategies for abstraction and refinement. Currently it contains front-end support for transition systems, control flow automata and timed automata. The built-in abstract domains include predicates, explicit values, zones and their combinations, along with various refinement strategies implemented for each. The configurability of the framework allows the integration of several abstraction and refinement methods, this way supporting the evaluation of their advantages and shortcomings. We demonstrate the applicability of the framework by use cases for the safety checking of PLC, hardware, C programs and timed automata models.**

I. INTRODUCTION

Nowadays there are several model checking tools implementing algorithms for different formalisms. Most tools focus on a specific algorithm and formalism to solve a particular verification task efficiently. However, as new tasks emerge, more generic tools are also needed since the appropriate formalism and algorithm are usually not known initially.

THETA[1] is a generic, modular and configurable model checking framework, aiming to support the development and evaluation of abstraction refinement-based algorithms for the reachability analysis of different formalisms. The main distinguishing characteristic of THETA is its architecture that allows the combination of various abstract domains, interpreters, and strategies for abstraction and refinement, applied to models of various formalisms with higher level language front-ends.

THETA primarily aims to support researchers by providing a framework where new components and combinations can easily be implemented, evaluated and compared. Concrete tools were also built for the verification of transition systems, control flow automata and timed automata, combining different abstract domains (including predicates, explicit values and zones) and refinement strategies (including interpolation and unsat cores). Measurement results show strong dependency on the models and analysis components, motivating the need for a configurable framework. Furthermore, we also used THETA for education at our university, where students implemented model checkers using components from the framework.

Related tools. Abstraction refinement is a widely used approach for model checking software. Several tools, e.g. SLAM [1], BLAST [2] and SATABS [3] are based on predicate abstraction. Lazy abstraction tools like IMPACT [4] and WOLVERINE [5] use Craig interpolation to compute abstractions over the predicate domain without expensive post-image computation. Some tools apply abstraction refinement over domains other than predicates: the tool DAGGER [6] supports refinement for octagon and polyhedra domains, and the algorithm VINTA [7] applies abstraction refinement over intervals. Frameworks CPACHECKER [8] and UFO [9] support configurability by the definition of abstract domains, post operators and refinement strategies, but only targeting software models. The LTSMIN tool supports various formalisms through its Partitioned Next-State Interface (PINS) [10]. However, its main focus is on symbolic and parallel model checking algorithms. Our THETA framework aims to combine the concept of configurability with formalism independence: the core analysis algorithms can be implemented independently of the input formalisms, and relevant combinations of them can be selected to verify models of several input formalisms.

In this paper we focus on the architecture of THETA (Section II) and the use cases demonstrating the efficient use of the tools that are derived from the framework (Section III).

II. ARCHITECTURE AND IMPLEMENTATION

Figure 1 shows the architecture of THETA. The main parts of the framework are the formalism and language front-ends, the analysis back-end and the SMT solver interface.

A. Formalisms and language front-ends

One goal of the THETA framework is to enable the analysis of several formalisms. Formalisms are usually low level, mathematical representations based on first order logic expressions and graph like structures. Each formalism supports higher level languages that can be mapped to that particular formalism by a language front-end (consisting of a specific parser and possibly reductions for simplification of the model). Currently, transition systems, control flow automata and timed automata are the supported formalisms with front-ends for higher level languages as AIGER, PLC, C programs and UPPAAL XTA

*This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

[1]http://theta.inf.mit.bme.hu

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

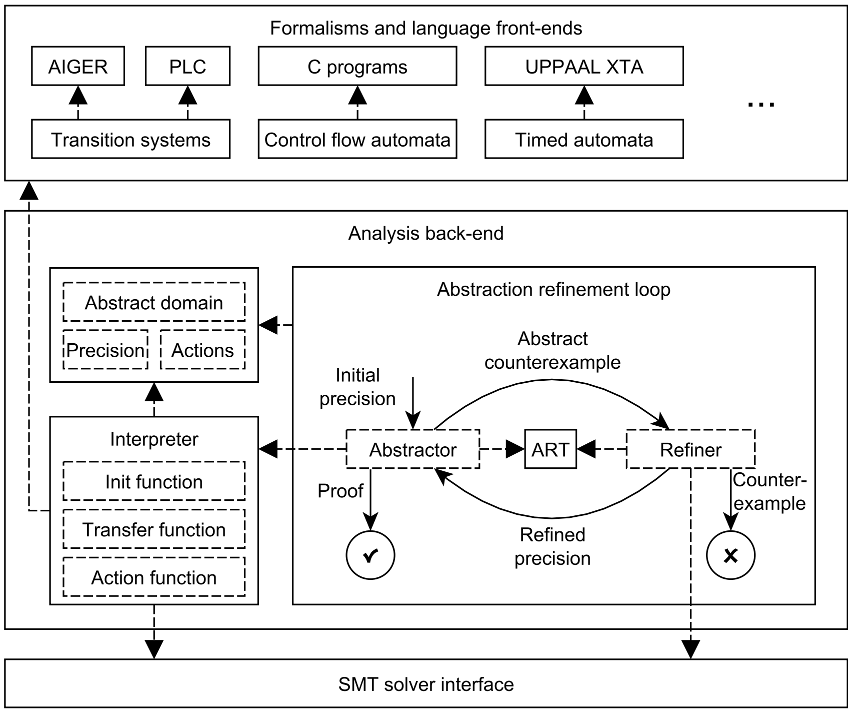


Fig. 1. Architecture of the THETA framework.

models. Section III describes instantiations of the framework for each of these formalisms.

B. Analysis back-end

The analysis back-end consists of three main parts: the abstract domain, the interpreter and the abstraction refinement loop for reachability analysis, with the interpreter being dependent on the formalisms. The basis of the analysis is an *abstract domain* with a set of *abstract states*, its bottom element and a partial order over the states. The accuracy of a given analysis is represented by an element of a set of precisions. Moreover, the formalism for which the analysis is performed defines a set of actions. Given a precision, an *interpreter* defines an abstract operational semantics over the abstract domain and set of actions. The abstract initial states are given by an *init function*. For an action, the abstract successors of a state are computed by a *transfer function*. An *action function* determines for an abstract state a set of actions that are enabled from that state.

The reachability analysis is performed by the *abstraction refinement loop*. As usual for lazy abstraction methods [4], its central data structure is an *abstract reachability tree* (ART), with nodes annotated with abstract states that represent overapproximations of reachable states along a given path, and edges annotated with actions. The ART is manipulated by the two main components of the loop. Using an interpreter, the *abstractor* constructs the ART w.r.t the current precision and an abstraction strategy, i.e. when to expand a node or cover it by an other node. If no target (i.e., unsafe) nodes are encountered, the constructed ART serves as an evidence for the safety of the input model. Otherwise, given a target node, the *refiner* is invoked to analyze the abstract path for feasibility. If the path is feasible, it is a counterexample to safety. Otherwise, the refiner carries out its refinement strategy to ensure that the analysis can continue without encountering the same spurious counterexample again (refinement progress). This can typically be achieved by pruning nodes and computing a new analysis precision (overapproximation-driven approach), or by uncovering nodes and strengthening labels (underapproximation-driven approach), both of which includes partial deconstruction of the ART.

Currently, built-in domains in THETA include predicates, explicit values, zones and their combinations. There are also interpreters provided for transition systems, control flow automata and timed automata. A default abstractor implementation is built-in that relies on the domain and the interpreter, also parameterizable with a search strategy. Interpolation and unsat core-based refinement strategies are provided for formalisms that are described with first order logic expressions.

C. SMT solver interface

The framework provides a general SMT solver interface that supports incremental solving, unsat cores, and the generation of binary and sequence interpolants. The solver interface can be used by the analysis components. Typically, the partial order over states and the transfer function are implemented in terms of queries to an SMT solver. A refiner component may use the interface to check feasibility of an abstract path and to generate interpolants or unsat cores for abstraction refinement. Currently, the interface is implemented by the SMT solver Z3 [11], but it can easily be extended with new solvers.

D. *Extending and instantiating the framework*

The framework can easily be extended with new formalisms and analyses. As an example, suppose that one wants to add support for the reachability checking of Petri nets [12]. First, the formalism has to be implemented, which is a collection of simple classes representing places, transitions and arcs of Petri nets. A possible language front-end could be the standard PNML format for Petri nets.

In order to perform reachability checking, the analysis back-end has to be extended as well. Petri nets can be described with first order logic formulas, for example by representing places (marked with tokens) with integer variables and transitions as FOL expressions adding/subtracting from places. Therefore, some abstract domains (such as predicates and explicit values) along with abstraction and refinement strategies (such as interpolation) work out of the box if the interpreter is implemented. An action of a Petri net can be implemented as the expression describing a transition and the action function as the collection of all transitions. The init and transfer functions also work out of the box for the abstract domains mentioned before.

Instantiating an executable tool from the framework (see examples in Section III) is also straightforward. A (command line or GUI) application has to be written that takes the parameters (path of the input model, domain, abstraction and refinement strategies, etc.), parses the input model using the language front-ends and instantiates and runs the analysis.

III. USE CASES

A. THETA *for transition systems*

The tool THETA-STS is an instantiation of the THETA framework for reachability analysis of (symbolic) transition systems, based on an earlier, preliminary version [13]. As input language, the tool supports the AIGER format (also used in the Hardware Model Checking Competition [14]) and an intermediate language for describing PLC models [15]. The tool relies on the built-in predicate and explicit value domains and refinement strategies based on binary interpolation, sequence interpolation and formulas from unsat cores. Some additional utilities are also implemented, for example inferring the initial precision and simplifying the input system.

Figure 2 (from [16]) shows a heatmap of the execution time of 20 analysis configurations on 12 hardware (hw) and 6 PLC models. White squares correspond to a timeout. Configurations are abbreviated with the first letter of the domain (predicate, explicit), the refinement strategy (binary interpolation, sequence interpolation, unsat cores), the initial precision (empty, property-based) and the exploration strategy (DFS, BFS). The heatmap shows that no single configuration can verify all models and the execution time is very diverse, motivating the need for a configurable framework.

B. THETA *for control flow automata*

The tool THETA-CFA is an instantiation of the THETA framework for the reachability analysis of control flow automata. As input language, the tool supports a subset of C, enhanced by various size reduction techniques such as

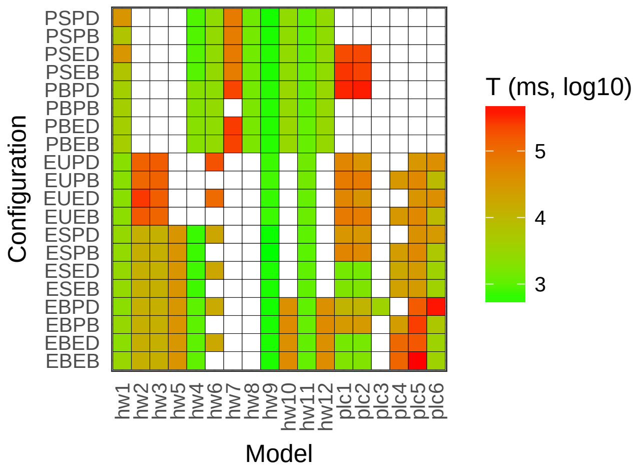


Fig. 2. Heatmap of execution time for transition systems (millisec., log. scale)

compiler optimizations and program slicing methods [17]. This tool uses the same built-in abstract domains and refinement strategies as the THETA-STS tool, only the interpreter differs.

Figure 3 (from [17]) presents a heatmap of the verification time of 16 analysis configurations on 9 models from SV-COMP [18], selected from those categories that are currently supported by our C frontend. Configurations are abbreviated with the first letter of the slicing method (none, backward, value, thin), the compiler optimizations (true, false) and the exploration strategy (DFS, BFS). Similarly to transition systems, different configurations are more suitable for different input models.

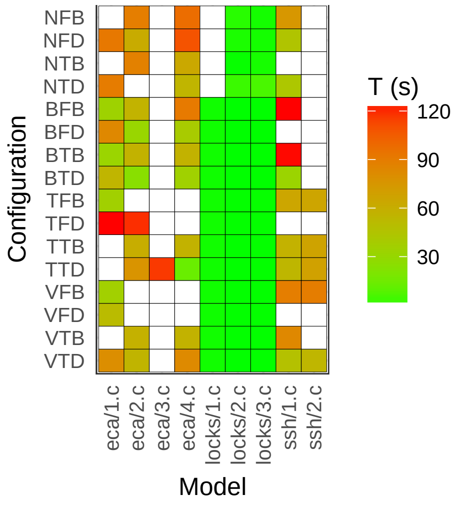


Fig. 3. Heatmap of execution time for C programs (in seconds)

C. THETA *for timed automata*

The tool THETA-XTA is an instantiation of the THETA framework for reachability checking of timed automata. As input language, the tool supports a subset of the UPPAAL

4.x XTA format[2]. The tool implements two lazy abstraction algorithms based on zone abstraction: a variant of $\langle \mathbf{a}_{\preccurlyeq LU}, \mathsf{disabled}\rangle$ [19], a non-convex lazy abstraction algorithm based on LU-bounds, and an algorithm based on interpolation for zones [20] with two different refinement strategies (BIN and SEQ). Table I (from [20]) presents some measurement results for the tool. Column *time* is the total execution time in ms, and *passed* is the number of expanded nodes in the ART. Models come from the PAT benchmarks[3].

TABLE I
COMPARISON OF ALGORITHMS FOR TIMED AUTOMATA IN THETA-XTA

Model	$\mathbf{a}_{\preccurlyeq LU}$		BIN		SEQ	
	time	*passed*	*time*	*passed*	*time*	*passed*
Critical 3	1.8	4923	1.6	3213	1.6	3157
Critical 4	65.0	130779	78.2	83686	75.2	78252
CSMA 9	6.6	30476	7.3	30476	7.9	30476
CSMA 10	21.3	78605	21.0	78605	22.8	78605
CSMA 11	61.4	198670	58.9	198670	63.8	198670
CSMA 12	167.2	493583	168.7	493583	179.1	493583
FDDI 50	1.4	402	2.0	402	2.0	402
FDDI 70	2.9	562	3.5	562	3.7	562
FDDI 90	5.9	722	6.8	722	7.1	722
FDDI 120	12.9	962	15.0	962	15.4	962
Fischer 7	1.9	7737	2.8	7737	2.8	7737
Fischer 8	5.1	25080	7.7	25080	8.7	25080
Fischer 9	21.3	81035	29.0	81035	32.4	81035
Fischer 10	94.4	260998	133.2	260998	149.7	260998
Lynch 7	2.6	9977	3.6	9977	4.0	9977
Lynch 8	7.7	30200	12.2	30200	13.9	30200
Lynch 9	32.8	92555	45.2	92555	54.2	92555

As can be seen from the data, in general, the $\mathbf{a}_{\preccurlyeq LU}$-based algorithm performs better in terms of execution time, but the interpolation based algorithms might construct a significantly smaller ART, thus easy configurability of the tool pays off.

IV. CONCLUSIONS

In this paper we introduced THETA, a configurable model checking framework for abstraction refinement-based reachability analysis for different formalisms. We described the architecture that helps to implement, evaluate and combine various algorithms in a modular way for different formalisms. We also demonstrated the applicability of the framework by use cases for the verification of hardware, PLC, software and timed automata models. Results of the evaluation with configuring and combining different analysis modules support the need for a generic framework, such as THETA.

Future work. At the moment the framework focuses on flexibility rather than performance (hence it is not yet intended to be competitive with highly optimized implementations). We are currently extending both the supported formalisms and the algorithms. We are working on supporting a wider set of elements in the C programming language and on defining hierarchical statecharts in THETA along with an interpreter. We are also working on increasing the number of input models in our experiments in order to reach stronger conclusions. This would also allow us to address the problem of selecting the most suitable configuration for a given verification task. Moreover, we also plan to experiment with novel, state-of-the-art algorithms, e.g., abstractions over data variables for timed automata.

[2]See the web help on http://www.uppaal.org for a language reference
[3]http://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html

REFERENCES

[1] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer Aided Verification*, ser. LNCS, vol. 2102. Springer, 2001, pp. 260–264.
[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, 2007.
[3] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: Sat-based predicate abstraction for ansi-c," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.
[4] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.
[5] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with WOLVERINE," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 573–578.
[6] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Automatically refining abstract interpretations," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2008, pp. 443–458.
[7] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Craig interpretation," in *Static Analysis*, ser. LNCS, vol. 7460. Springer, 2012, pp. 300–316.
[8] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190.
[9] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "UFO: A framework for abstraction- and interpolation-based software verification," in *Computer Aided Verification*, ser. LNCS, vol. 7358. Springer, 2012, pp. 672–678.
[10] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "LTSMIN:: High-performance language-independent model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 9035. Springer, 2015, pp. 692–707.
[11] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.
[12] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
[13] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A configurable CEGAR framework with interpolation-based refinements," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS, vol. 9688. Springer, 2016, pp. 158–174.
[14] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, K. Heljanko, and J. Baumgartner, "Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 135–172, 2016.
[15] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.
[16] A. Hajdu and Z. Micskei, "Exploratory analysis of the performance of a configurable CEGAR framework," in *Proceedings of the 24th PhD Mini-Symposium*. BUTE DMIS, 2017, pp. 34–37.
[17] G. Sallai, A. Hajdu, T. Tóth, and Z. Micskei, "Towards evaluating size reduction techniques for software model checking," in *Verification and Program Transformation*, ser. EPTCS. Open Publishing Association, 2017, (Accepted).
[18] D. Beyer, "Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016)," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2016, vol. 9636, pp. 887–904.
[19] F. Herbreteau, B. Srivathsan, and I. Walukiewicz, "Lazy abstractions for timed automata," in *Computer Aided Verification*, ser. LNCS, vol. 8044. Springer, 2013, pp. 990–1005.
[20] T. Tóth and I. Majzik, "Lazy reachability checking for timed automata using interpolants," in *Formal Modeling and Analysis of Timed Systems*, ser. LNCS, vol. 10419. Springer, 2017, (Accepted).

Modular SMT-Based Analysis of Nonlinear Hybrid Systems

Kyungmin Bae
POSTECH, Republic of Korea
kmbae@postech.ac.kr

Sicun Gao
University of California, San Diego, USA
sicung@ucsd.edu

***Abstract*—We present SMT-based techniques for analyzing networks of nonlinear hybrid systems, which interact with each other in both discrete and continuous ways. We propose a modular encoding method to reduce reachability problems of hybrid components, involving *continuous I/O* as well as usual discrete I/O, into the satisfiability of first-order logic formulas over the real numbers. We identify a generic class of logical formulas to modularly encode networks of hybrid systems, and present an SMT algorithm for checking the satisfiability of such logical formulas. The experimental results show that our techniques significantly increase the performance of SMT-based analysis for networks of nonlinear hybrid components.**

I. Introduction

Formal analysis of hybrid systems can be reduced to the satisfiability of SMT formulas over the real numbers. This approach can combine state-of-the-art SMT techniques with numerical methods to analyze continuous dynamics, governed by ordinary differential equations (ODEs). The satisfiability of these formulas are in general undecidable for nonlinear hybrid systems, but important advances have been made by various approaches, e.g., [1]–[6].

In principle, these methods can deal with networks of nonlinear hybrid systems by combining the SMT encodings of all components. Following SMT-based approaches for digital systems, *discrete communication* between components, such as synchronization or message passing, can be encoded using first-order variables that commonly occur in the encodings of several components. In order for this technique to work, all interactions between components must be discrete.

However, many networks of hybrid systems also include *continuous interaction* as well as discrete communication. For example, consider the problem of controlling the temperature of several adjacent rooms. The temperature of one room can continuously affect the temperature of all adjacent rooms. If we model this system as a network of several thermostat systems, it is quite clear that such continuous interactions cannot be captured only using discrete communication. This kind of *continuous I/O* is common in control systems that are composed of mechanically connected components; e.g, cars [7], airplanes [8], plants [9], etc. Indeed, formal models of hybrid systems, such as hybrid automata [10] and hybrid I/O automata [11], can precisely specify continuous interactions.

This work was partially supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1D1A1B03935275), and NSF CPS-1446675.

The analysis of networks of nonlinear hybrid systems has not been studied much in existing SMT-based approaches. To apply existing SMT algorithms, one may define a sound discrete approximation of continuous I/O. But this is very difficult for nonlinear systems because continuous I/O can involve nonlinear functions, not just single values. Or one can build a single hybrid component that is equivalent to the entire network, but at the cost of the state explosion problem.

The goal of this paper is to provide an SMT technique to analyze networks of nonlinear hybrid systems involving continuous I/O as well as discrete I/O. The contribution of this paper is twofold: (1) to directly provide a new modular SMT encoding for networks of nonlinear hybrid systems with continuous I/O, and (2) to develop an SMT solving algorithm to check the satisfiability of such logical formulas.

The basic idea is to encode continuous interaction by means of *uninterpreted real functions*, not first-order variables. We then use *universally quantified equalities over time* to encode a continuous synchronization of uninterpreted real functions. To logically decompose continuous I/O expressed as systems of ODEs, we make use of *parameterized integration operators*. We identify a syntactic subclass of formulas that is expressive enough to modularly encode continuous I/O.

We present a novel SMT algorithm to check the satisfiability of the proposed class of formulas. Existing algorithms cannot deal with uninterpreted real functions, universally quantified equalities, and parameterized integration operators at the same time. Our algorithm is based on an equisatisfiable process that removes uninterpreted real functions and parameterized integration operators. This process can easily be combined with existing DPLL($\mathcal{T}$)-based algorithms with minimal cost.

We have implemented our algorithm in the dReal solver [12], and performed experiments on a range of nontrivial networks of nonlinear hybrid systems. These case studies include both discrete and continuous interactions between hybrid components. The experimental results show that our techniques can greatly increase the performance of SMT-based analysis for networks of general nonlinear hybrid systems.

The paper is organized as follows. Section II explains networks of hybrid systems. Section III proposes a modular encoding. Section IV presents an SMT solving algorithm. Section V provides an overview of the case studies. Section VI presents the experimental results. Section VII discusses related work, and Section VIII presents concluding remarks.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

II. Network of Hybrid System Components

We consider a network of hybrid systems that interact with each other in discrete or continuous ways. As shown in Fig. 1, each component has two types of input and output; continuous I/O (denoted by bold lines) and discrete I/O (denoted by thin lines). A hybrid system can be specified by *hybrid automata* [10]. This paper uses an extended version of hybrid automata with explicit I/O, similar to one presented in [11].

A. Hybrid I/O Automata

In *hybrid I/O automata*, discrete states are given by a finite set of *modes* Q, and continuous states are specified by using a finite set of real-valued *variables* $X = \{x_1, \dots, x_l\}$. A combined state is then a pair $\langle q, \boldsymbol{v} \rangle$ of a mode $q \in Q$ and a vector $\boldsymbol{v} = (v_1, \dots, v_l) \in \mathbb{R}^l$ of real numbers. Given a (possibly infinite) set of actions Σ, a discrete transition between two states $\langle q, \boldsymbol{v} \rangle \xrightarrow{a} \langle q', \boldsymbol{v'} \rangle$, identified with an *action* $a \in \Sigma$, is specified by a *jump condition* $jump_{q,q'}(\boldsymbol{v}, a, \boldsymbol{v'})$.

Each mode $q \in Q$ of a hybrid I/O automaton defines extra conditions to specify the continuous behavior of the variables X in the mode q. An *invariant condition* inv_q defines all possible values of the variables X in mode q. A *flow condition* $flow_q$ defines *trajectories* of the variables X—describing continuous changes of X's values over time—in mode q, typically using ordinary differential equations (ODEs). An *initial condition* $init_q$ defines a set of initial states.

Discrete input and output of a component are identified by using disjoint sets of *input actions* $\Sigma_I \subseteq \Sigma$ and *output actions* $\Sigma_O \subseteq \Sigma$. Other "local" actions in $\Sigma \setminus (\Sigma_I \cup \Sigma_O)$ are called *internal actions*. Likewise, continuous input and output are identified by using disjoint sets of *input variables* $X_I \subseteq X$ and *output variables* $X_O \subseteq X$, and other "local" variables in $X \setminus (X_I \cup X_O)$ are called *internal variables*.

Definition 1. *A* hybrid I/O automaton *(HIOA) is defined as a tuple* $H = (Q, X, \Sigma, \{inv_q\}_{q \in Q}, \{flow_q\}_{q \in Q}, \{init_q\}_{q \in Q}, \{jump_{q,q'}\}_{q,q' \in Q}, (X_I, X_Y), (\Sigma_I, \Sigma_O))$.

The $init$, inv, and $jump$ conditions are often written as predicates $init_q(\boldsymbol{x})$, $inv_q(\boldsymbol{x})$, and $jump_{q,q'}(\boldsymbol{x}, a, \boldsymbol{x'})$ over the variables X. The *flow* condition is written as a system of ODEs of the form $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\iota})(t)$, where the input variables $\boldsymbol{\iota}$ in X_I appear as *free variables* in the ODEs.

A network of hybrid system components is specified by a parallel composition $\mathcal{H} = H_1 \parallel H_2 \parallel \cdots \parallel H_n$ of hybrid I/O automata. A communication between components is specified using I/O actions and variables. We assume that: (i) output actions and variables of one component are not output of any other components, and (ii) internal actions and variables of one component are not parts of any other components.

There are two types of communications in a network $\mathcal{H}$. Consider a source component H_o and target components $H_{i_1}, \dots, H_{i_m}$. A *discrete communication* is modeled by *joint synchronous actions* in $\Sigma_O^o \cap \Sigma_I^{i_1} \cap \cdots \cap \Sigma_I^{i_m}$ that are output of the source and input of the targets. A *continuous interaction* is modeled by *shared variables* in $X_O^o \cap X_I^{i_1} \cap \cdots \cap X_I^{i_m}$ that are output of the source and input of the targets.

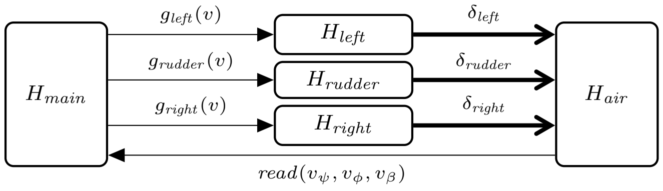


Fig. 1. Controllers for turning an airplane

A discrete state of a network $\mathcal{H}$ is defined as a vector $(q_1, \dots, q_n)$ of modes of its components. Components in $\mathcal{H}$ synchronize their discrete transitions with a joint action. When one component performs a transition labeled with an action a, every component that includes the action a must perform a transition with the same action. A component can individually perform a discrete transition with an internal action.

A continuous state of a network $\mathcal{H}$ is defined as trajectories of the variables $X_1 \cup \cdots \cup X_n$ of its components. The values of $\mathcal{H}$'s variables evolve simultaneously over time according to their flow and invariant conditions. Flow conditions of an input variable are given by those of its connected output variable. That is, an output variable and its corresponding input variables must have the same value at all times.

Consequently, a network of HIOA is semantically equivalent to a single HIOA (see [11] for a formal definition).

B. Example: Turning an Airplane

We consider a networked controller for turning an airplane, adapted from [8], [13]. An aircraft makes a turn by controlling two *ailerons* (surfaces attached to the end of the wings) and a *rudder* (a surface attached to the vertical tail). As depicted in Fig. 1, the main controller orchestrates the subcontrollers for the ailerons and the rudder to achieve a coordinated turn. The entire system is $H_{main} \parallel H_{left} \parallel H_{rudder} \parallel H_{right} \parallel H_{air}$.

A subcontroller H_M for $M \in \{left, right, rudder\}$ has three modes acc, dec, and con. Its angle δ_M and the moving rate r_M changes according to the ODEs with constant $c_M > 0$:

$$\dot{\delta}_M = r_M,$$
$$\dot{r}_M = c_M \text{ (acc)}, \quad \dot{r}_M = -c_M \text{ (dec)}, \quad \dot{r}_M = 0 \text{ (con)}.$$

H_M performs a jump transition with an input action $g_M(v)$ to determine a next mode based on the goal angle v. The sets of I/O variables and I/O actions are defined by: $X_O^M = \{\delta_M\}$, $\Sigma_I^M = \{g_M(v) \mid v \in \mathbb{R}\}$, and $X_I^M = \Sigma_O^M = \emptyset$.

The lateral dynamics of the aircraft is modeled by H_{air} that has a single mode with the nonlinear ODEs:

$$\dot{\beta} = Y(\beta, \boldsymbol{\delta})/mV - r + V \cos\beta \sin\phi/g, \quad \dot{\psi} = g/V \cdot \tan\phi,$$
$$\dot{p} = (c_1 r + c_2 p) r \tan\phi + c_3 L(\beta, \boldsymbol{\delta}) + c_4 N(\beta, \boldsymbol{\delta}), \quad \dot{\phi} = p,$$
$$\dot{r} = (c_8 p - c_2 r) r \tan\phi + c_4 L(\beta, \boldsymbol{\delta}) + c_9 N(\beta, \boldsymbol{\delta})$$

with β the yaw angle, ψ the direction, p the rolling moment, ϕ the roll angle, and r the yawing moment. Y, L, and N are linear functions of β and angles $\boldsymbol{\delta} = (\delta_{left}, \delta_{right}, \delta_{rudder})$. The sets of I/O variables and I/O actions are defined by: $X_I^{air} = \{\boldsymbol{\delta}\}$, $\Sigma_O^{air} = \Sigma_I^{main}$, and $X_O^{air} = \Sigma_I^{air} = \emptyset$.

The main controller H_{main} provides a goal angle to a subcontroller M using a discrete transition with an output action g_M, and monitors the current position from H_{air} using a discrete transition with an input action $read$. The component H_{main} has no I/O variables, but has the sets of I/O actions:

$$\Sigma_I^{main} = \{read(v_\psi, v_\phi, v_\beta) \mid v_\psi, v_\phi, v_\beta \in \mathbb{R}\}$$
$$\Sigma_O^{main} = \Sigma_I^{left} \cup \Sigma_I^{right} \cup \Sigma_I^{rudder}$$

We are interested in a reachability problem to find outputs of H_{main} to reach a goal direction in a reasonable time, while keeping the yaw angle β close to 0 during the turn.

The continuous interaction between $(H_{left}, H_{rudder}, H_{right})$ and H_{air} cannot be specified using discrete synchronization without I/O variables. We can eliminate I/O variables by building a single HIOA that is equivalent to the composition of the network, but at the cost of the state explosion problem. For example, a single hybrid automaton for $H_1 \parallel \ldots \parallel H_n$ has total $\prod_{i=1}^n m_i$ modes, if each H_i has m_i modes.

III. SMT Encoding of Hybrid Components

In this section we propose a modular method to encode a network of hybrid system components that involve *both discrete and continuous I/O* at the same time. Our technique generalizes previous SMT-based approaches that only focus on discrete communication using synchronous actions.

A. Encoding of Discrete Transitions

We use first-order logic formulas over the real numbers, called $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formulas, along with a collection $\mathcal{F}$ of Type 2 computable real functions [3]. A Type 2 computable real function can be numerically evaluated up to an arbitrary precision, such as polynomials, exponentiation, trigonometric functions, and solutions of Lipschitz-continuous ODEs. The syntax is defined in the standard way, with c real number constants, y first-order variables, and f real functions in $\mathcal{F}$:

$$t ::= c \mid y \mid f(t_1, \ldots, t_n)$$
$$\varphi ::= t > 0 \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \exists y.\varphi \mid \forall y.\varphi.$$

Consider a HIOA H. As usual, a discrete transition of H with an action can simply be encoded by using a first-order variable w that denotes the action. Discrete synchronization by the action is encoded by using the same variable w in the formula for each corresponding component.

Definition 2. *An $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding for a (synchronized) discrete transition of H from state $\langle q, \boldsymbol{y}\rangle$ to $\langle q', \boldsymbol{y}'\rangle$ by an action w is:*

$$\begin{aligned} dt(q, \boldsymbol{y}, w, q', \boldsymbol{y}') \;\equiv\; & (w \in \Sigma \;\rightarrow\; jump^H_{q,q'}(\boldsymbol{y}, w, \boldsymbol{y}')) \\ & \wedge (w \notin \Sigma \;\rightarrow\; q = q' \;\wedge\; \boldsymbol{y} = \boldsymbol{y}'). \end{aligned}$$

B. Encoding of Continuous Flows

The continuous behavior of one component is parameterized by the trajectories of its input variables, which are composed of the trajectories of the corresponding output variables in other components. Unlike actions, these (potentially nonlinear) trajectories cannot be encoded as first-order variables in $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$, because they are actually real-valued functions.

We use an *uninterpreted real function symbol* $\mathbb{R} \to \mathbb{R}$ to represent a trajectory for each input variable. Consider a flow condition for mode q, expressed as a system of ODEs:

$$\left[\tfrac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{f_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(t), \quad \tfrac{\mathrm{d}\boldsymbol{o}}{\mathrm{d}t} = \boldsymbol{g_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(t)\right]$$

with $\boldsymbol{x}$ the internal variables, $\boldsymbol{o}$ the output variables, and $\boldsymbol{\iota}$ the input variables. Mathematically, these ODE variables in the terms $\boldsymbol{f_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(t)$ and $\boldsymbol{g_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(t)$ denote unary real functions $\mathbb{R} \to \mathbb{R}$ over time t.

Given function symbols $\dot{o}_1, \ldots, \dot{o}_l$ to denote the derivatives of $\boldsymbol{o} = (o_1, \ldots, o_l)$, and function symbols $\dot{x}_1, \ldots, \dot{x}_n$ to denote the derivatives of $\boldsymbol{x} = (x_1, \ldots, x_n)$, we can express the trajectories of $\boldsymbol{x}$ and $\boldsymbol{o}$ as the $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula

$$\forall u \in [0, t].\;\; [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}](u) = [\boldsymbol{f_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(u),\; \boldsymbol{g_q}(\boldsymbol{x}, \boldsymbol{o}, \boldsymbol{\iota})(u)].$$

The trajectories of the input variables $\boldsymbol{\iota}$ (that appear as free variables in $\boldsymbol{f_q}$ and $\boldsymbol{g_q}$) are then defined by using *output derivatives of other components.*

Using these I/O derivative symbols, we can express the continuous change of H's states from initial values $\boldsymbol{y^0}$ to new values $\boldsymbol{y^t}$ for duration t as the $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula:

$$\boldsymbol{y^t} = \boldsymbol{y^0} + \textstyle\int_0^t [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}, \dot{\boldsymbol{\iota}}]\,\mathrm{d}s$$

This $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula is parameterized by the internal derivatives $\dot{\boldsymbol{x}} = (\dot{x}_1, \ldots, \dot{x}_n)$, the output derivatives $\dot{\boldsymbol{o}} = (\dot{o}_1, \ldots, \dot{o}_l)$, and the input derivatives $\dot{\boldsymbol{\iota}} = (\dot{\iota}_1, \ldots, \dot{\iota}_m)$.

The ODE solution term $\boldsymbol{y^0} + \int_0^t [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}, \dot{\boldsymbol{\iota}}]\,\mathrm{d}s$ can be considered as a computable real function $F(\boldsymbol{y^0}, t)$ in the collection $\mathcal{F}$, provided that $\boldsymbol{f_q}$, $\boldsymbol{g_q}$, and the ODEs for the input variables are all Lipschitz-continuous [14], [15]. The extra derivative function symbols are also in the collection $\mathcal{F}$. In sum, an encoding of continuous flows is defined as follows:

Definition 3. *An $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding for a continuous flow of H from $\boldsymbol{y^0}$ to $\boldsymbol{y^t}$ in mode q for duration t, that involves I/O variables $(\boldsymbol{\iota}, \boldsymbol{o})$ and satisfies the invariant condition inv^H_q, where $F(\boldsymbol{y^0}, t)$ denotes $\boldsymbol{y^0} + \int_0^t [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}, \dot{\boldsymbol{\iota}}]\,\mathrm{d}s$, is:*

$$\begin{aligned} ct(q, \boldsymbol{y^0}, \boldsymbol{y^t}, t \mid \boldsymbol{\iota}, \boldsymbol{o}, \boldsymbol{x}) \;\equiv\;\; & \boldsymbol{y^t} = F(\boldsymbol{y^0}, t) \\ & \wedge \forall u \in [0, t].\, [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}](u) = [\boldsymbol{f_q}, \boldsymbol{g_q}](u) \\ & \wedge \forall u \in [0, t].\, inv^H_q(F(\boldsymbol{y^0}, u)). \end{aligned}$$

We use different uninterpreted functions for variables in different components, even if they are "shared" variables in a network of HIOA. The use of the same uninterpreted functions for different components may cause unintended semantic effects, as explained in Sec. IV-A below. This is not a strict restriction, because we can always use "syntactic copies" to enforce that input variables $\boldsymbol{\iota}$ and output variables $\boldsymbol{o}$ follow the isomorphic system of ODEs $\boldsymbol{f}$ up to renaming, by means of *connection formulas* between $\boldsymbol{\iota}$ and $\boldsymbol{o}$ as follows.

Definition 4. *An $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding of connections between input variables $\boldsymbol{\iota}$ and output variables $\boldsymbol{o}$ is as follows (it includes internal variables $\boldsymbol{x}$ and $\boldsymbol{x}'$, since $\boldsymbol{o}$ may depend on $\boldsymbol{x}$):*

$$\begin{aligned} conn(\boldsymbol{\iota}, \boldsymbol{o}, \boldsymbol{x}, \boldsymbol{x}') \;\equiv\;\; & (\forall u \in [0, t].\;\; [\dot{\boldsymbol{o}}, \dot{\boldsymbol{x}}](u) = \boldsymbol{f}(\dot{\boldsymbol{o}}, \boldsymbol{x})(u)) \\ \leftrightarrow\;\; & (\forall u \in [0, t].\;\; [\dot{\boldsymbol{\iota}}, \dot{\boldsymbol{x}}'](u) = \boldsymbol{f}(\dot{\boldsymbol{\iota}}, \boldsymbol{x}')(u)). \end{aligned}$$

C. Encoding of Bounded Reachability

For a network of HIOA $H_1 \parallel \cdots \parallel H_N$, the reachability up to the k-th discrete step—that involves continuous I/O as well as discrete synchronization—can be encoded in $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$ as follows. This formula is defined as just a conjunction of N subformulas, each of which encodes the reachability of each individual component H_j up to the k-th discrete step.

Definition 5. *An $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding for the k-step reachability of a network of HIOA $H_1 \parallel \cdots \parallel H_N$ is the $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula of size $O(\sum_{j=1}^{N} k \cdot |Q_j|^2)$ ($\exists$-quantified at the top):*

$$\bigwedge_{j=1}^{N} \left[\begin{array}{l} \quad init^j_{m_0^j}({\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{0}}) \wedge ct(m_0^j, {\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{0}}, {\boldsymbol{y_j}}^{\boldsymbol{t}}_{\boldsymbol{0}}, t_0 \mid \boldsymbol{\iota}^{\boldsymbol{j}}_{\boldsymbol{0}}, \boldsymbol{o}^{\boldsymbol{j}}_{\boldsymbol{0}}, \boldsymbol{x}^{\boldsymbol{j}}_{\boldsymbol{0}}) \\ \wedge \bigwedge_{i=1}^{k} \left[\begin{array}{l} dt(m_{i-1}^j, {\boldsymbol{y_j}}^{\boldsymbol{t}}_{\boldsymbol{i-1}}, w_i, m_i^j, {\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{i}}) \wedge \\ ct(m_i^j, {\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{i}}, {\boldsymbol{y_j}}^{\boldsymbol{t}}_{\boldsymbol{i}}, t_i \mid \boldsymbol{\iota}^{\boldsymbol{j}}_{\boldsymbol{i}}, \boldsymbol{o}^{\boldsymbol{j}}_{\boldsymbol{i}}, \boldsymbol{x}^{\boldsymbol{j}}_{\boldsymbol{i}}) \end{array} \right] \end{array} \right]$$
$$\wedge \;\; goal(m_k^1, \ldots, m_k^N, {\boldsymbol{y_1}}^{\boldsymbol{t}}_{\boldsymbol{k}}, \ldots, {\boldsymbol{y_N}}^{\boldsymbol{t}}_{\boldsymbol{k}})$$
$$\wedge \;\; \textstyle\bigwedge_{i=0}^{k} conn(\boldsymbol{\iota}^{\boldsymbol{1}}_{\boldsymbol{i}}, \ldots, \boldsymbol{\iota}^{\boldsymbol{N}}_{\boldsymbol{i}}, \boldsymbol{o}^{\boldsymbol{1}}_{\boldsymbol{i}}, \ldots, \boldsymbol{o}^{\boldsymbol{N}}_{\boldsymbol{i}}, \boldsymbol{x}^{\boldsymbol{1}}_{\boldsymbol{i}}, \ldots, \boldsymbol{x}^{\boldsymbol{N}}_{\boldsymbol{i}})$$

For each i-th discrete step of duration t_i, component H_j is in mode m_i^j, and the values of H_j's variables begin with ${\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{i}}$ and end with ${\boldsymbol{y_j}}^{\boldsymbol{t}}_{\boldsymbol{i}}$. At the first step ($i = 0$), the initial values ${\boldsymbol{y_j}}^{\boldsymbol{0}}_{\boldsymbol{0}}$ of H_j's variables satisfy the initial condition. To begin the $(i+1)$-th step, every component synchronizes its transition with the same action w_i. For the k-th step, the goal formula holds for H_j's final state. The connection formulas $conn$ link input and output variables.

D. Example

Consider the airplane controller example in Sec. II-B. For a subcontroller M, an $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding of a continuous flow from initial values (δ_M^0, r_M^0) to new values (δ_M^t, r_M^t) with the invariant condition $-45 < \delta_M < 45$ is the formula below.

$$(\delta_M^t, r_M^t) = (\delta_M^0, r_M^0) + \textstyle\int_0^t [\dot\delta_M, \dot r_M] \,\mathrm{d}s$$
$$\wedge \left[\begin{array}{l} (q_M = acc \;\rightarrow\; \forall u \in [0,t].\ [\dot\delta_M, \dot r_M](u) = [r_M, c_M]) \wedge \\ (q_M = dec \;\rightarrow\; \forall u \in [0,t].\ [\dot\delta_M, \dot r_M](u) = [r_M, -c_M]) \wedge \\ (q_M = con \;\rightarrow\; \forall u \in [0,t].\ [\dot\delta_M, \dot r_M](u) = [r_M, 0]) \end{array} \right]$$
$$\wedge\, \forall u \in [0,t].\ -45 < \pi_1\big((\delta_M^0, r_M^0) + \textstyle\int_0^u [\dot\delta_M, \dot r_M] \,\mathrm{d}s\big) < 45.$$

The last line expresses the invariant condition $-45 < \delta_M < 45$ using the solution term $(\delta_M^0, r_M^0) + \int_0^u [\dot\delta_M, \dot r_M] \,\mathrm{d}s$ and the projection function $\pi_1(a, b) = a$.

For the airplane component H_{air}, an $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-encoding of its continuous flow is simply the single ODE solution term:

$$(\beta^t, \psi^t, \phi^t, p^t, r^t, \boldsymbol{\delta}^t) = (\beta^0, \psi^0, \phi^0, p^0, r^0, \boldsymbol{\delta}^0) + \textstyle\int_0^t \boldsymbol{F}(s) \,\mathrm{d}s,$$

that is parameterized by the component H_{air}'s input variables $\boldsymbol{\delta^{air}} = (\delta^{air}_{left}, \delta^{air}_{right}, \delta^{air}_{rudder})$, where

$$\boldsymbol{F}(s) = \left[\begin{array}{l} Y(\beta, \boldsymbol{\delta^{air}})/mV - r + V/g \cdot \cos\beta \sin\phi \\ g/V \cdot \tan\phi \\ p \\ (c_1 r + c_2 p) r \tan\phi + c_3 L(\beta, \boldsymbol{\delta^{air}}) + c_4 N(\beta, \boldsymbol{\delta^{air}}) \\ (c_8 p - c_2 r) r \tan\phi + c_4 L(\beta, \boldsymbol{\delta^{air}}) + c_9 N(\beta, \boldsymbol{\delta^{air}}) \\ \dot{\boldsymbol{\delta}}^{\boldsymbol{air}} \end{array} \right]$$

The behavior of the input variables $\boldsymbol{\delta^{air}}$ is given by connection formulas such as: $(\forall u \in [0,t].\ [\dot\delta_M, \dot r_M](u) = [r_M, c_M]) \leftrightarrow (\forall u \in [0,t].\ [\dot\delta_M^{air}, \dot r_M^{air}](u) = [r_M^{air}, c_M])$, where variable r_M^{air} is a "copy" of the internal variable r_M of M.

E. The Correctness of the Encoding

Our modular encoding is correct in the sense that it is equisatisfiable to the encoding of the composition, which is a single hybrid automaton (see [16] for the proof).

Theorem 1. *Given a network of HIOA $\mathcal{H} = H_1 \parallel \cdots \parallel H_n$, its modular encoding (of size $O(k \sum_{j=1}^{N} |Q_j|^2)$) and the encoding of its composition (of size $O(k \prod_{j=1}^{N} |Q_j|)$) are equisatisfiable.*

IV. SMT Algorithm for Hybrid Components

We syntactically identify a generic class of $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formulas that involve universal quantification for uninterpreted real functions. This class includes $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formulas for networks of hybrid I/O automata in Def. 5. We present an SMT procedure for checking the satisfiability of these $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formulas. The proofs of lemmas and theorems in this section are in [16].

A. Syntactic Classification

We explicitly decompose a collection $\mathcal{F}$ as $\mathcal{F}' \cup \mathcal{G}$. A collection $\mathcal{G}$ is composed of uninterpreted unary differentiable functions $\mathbb{R} \to \mathbb{R}$ to denote variables $x_1, \ldots, x_l$ in ODEs and their derivatives $\dot x_1, \ldots, \dot x_l$ (including I/O variables and I/O derivatives), whereas $\mathcal{F}'$ includes only interpreted real functions. For each pair $(x, \dot x)$ in $\mathcal{G}$, the function $\dot x \in \mathcal{G}$ is a derivative of the function $x \in \mathcal{G}$.

We explicitly take into account *parameterized integration operators* over ODE variables $\boldsymbol{x} \in \mathcal{G}$ and derivatives $\dot{\boldsymbol{\iota}} \in \mathcal{G}$:

$$\boldsymbol{y^0} + \int_0^t [\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\iota})(s), \dot{\boldsymbol{\iota}}(s)] \,\mathrm{d}s,$$

specifying solution functions of parameterized ODE systems of the form $\frac{\mathrm{d}}{\mathrm{d}s}[\boldsymbol{x}, \boldsymbol{\iota}] = [\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\iota})(s), \dot{\boldsymbol{\iota}}(s)]$ with initial values $\boldsymbol{y^0}$ over time t. As mentioned, we can define the behavior of each input derivative $\dot\iota$ by universally quantified $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas of the form $\forall u \in [0,t].\ \dot\iota(u) = g(\boldsymbol{z})(u)$, where $\boldsymbol{z} \in \mathcal{G}$ denote unary function symbols for ODE variables.

We identify a subclass of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas that allows a reduction to $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$ without extra uninterpreted real functions in $\mathcal{G}$. We can apply any (existing) algorithms for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas after the reduction. As a matter of fact, the entire class of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas is too expressive to have any kinds of efficient decision procedures. For example, control problems of nonlinear systems are undecidable in general [17], but can be written as $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas.[1]

We require that each uninterpreted function symbol in $\mathcal{G}$ only occurs in a single integration term. Otherwise, a "shared" function symbol in different terms leads to an unintended semantic restriction that all the integration terms have the same solution function. The decomposition using parameterized integration in Sec. III-B is therefore no longer equisatisfiable to the original formula. This is why we encode variables in different components as different functions.

[1]Consider a nonlinear system $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$ with initial values $\boldsymbol{x_0}$ and controls $\boldsymbol{u}$. The controllability to $\boldsymbol{x_t}$ can be expressed as the satisfiability of $\boldsymbol{x}_t = \boldsymbol{x_0} + \int_0^t \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \,\mathrm{d}s$ with uninterpreted real functions $\boldsymbol{u}$.

For example, consider the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-formula that involves the ODE $\frac{\mathrm{d}x}{\mathrm{d}s} = x$ with different initial values 1 and 2:

$$x_1 = 1 + \int_0^t x(s)\,\mathrm{d}s \;\wedge\; x_2 = 2 + \int_0^t x(s)\,\mathrm{d}s \;\wedge\; t = 1,$$

which is satisfiable with $x_1 = e$ and $x_2 = 2e$. Now let us replace the term $x(s)$ with an uninterpreted function $\dot{x}(s)$:

$$x_1 = 1 + \int_0^t \dot{x}(s)\,\mathrm{d}s \;\wedge\; x_2 = 2 + \int_0^t \dot{x}(s)\,\mathrm{d}s \;\wedge\; t = 1 \\ \wedge\; \forall u \in [0,t].\, \dot{x}(u) = x(u).$$

This formula is *not* satisfiable, because there are no single interpretations of $\dot{x}$ and x that denote solution functions of $\frac{\mathrm{d}x}{\mathrm{d}s} = x$ for both initial values 1 and 2 at the same time.

We restrict our attention to $\mathcal{G}$ with the following constraints. Every uninterpreted function $x \in \mathcal{G}$ that occurs in the same integration term has the same domain $[0, t_x]$. Each derivative $\dot{x} \in \mathcal{G}$ is bounded by a finite set of Lipschitz-continuous functions $G_x = \{g_1(\boldsymbol{z}), g_2(\boldsymbol{z}), \ldots, g_m(\boldsymbol{z})\}$. This *boundedness constraint* for $\dot{x} \in \mathcal{G}$ can be expressed as the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-formula: $bound_{G_x}(\dot{x}) \equiv \bigvee_{g(\boldsymbol{z})\in G_x} \forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$.

Definition 6. *An $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-formula for networks of hybrid I/O automata has the form: $\exists \boldsymbol{y}.\, \varphi \wedge \bigwedge_{\dot{x}\in\mathcal{G}} bound_{G_x}(\dot{x})$, where:*

- *each subformula of φ is quantifier-free or containing only universally quantified subformulas over time;*
- *each uninterpreted function in $\mathcal{G}$ appears in integration terms, universally quantified formulas over time of the form $\forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$, or formulas of the negated form $\exists u \in [0,t_x].\, \dot{x}(u) \neq g(\boldsymbol{z})(u)$;*
- *each uninterpreted function in $\mathcal{G}$ appears in at most one integration term (but the same integration term can appear in a formula many times).*

The formula in Def. 5 is in the syntactic class of Def. 6, if every input variable corresponds to some output variable.

B. $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-Reduction of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-Formulas

A key part of our algorithm is *the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction* that removes uninterpreted functions in $\mathcal{G}$ from $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-formulas, summarized in Alg. 1. For a conjunction μ of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals that *satisfy the boundedness constraint and the syntactic restriction in Def. 6*, the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction procedure builds an $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula that is equisatisfiable to the conjunction μ.

Definition 7. *An $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$*-literal *is an atomic $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula, a universally quantified formula $\forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$, or a negation of these $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-atoms.*

Without loss of generality, we assume that for each derivative $\dot{x} \in \mathcal{G}$, the conjunction μ includes a universally quantified $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literal of the form $\forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$ from the boundedness constraint $bound_{G_x}(\dot{x})$.

The $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction begins with choosing a universally quantified $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literal $\forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$ for each derivative $\dot{x} \in \mathcal{G}$ (line 2). We replace every occurrence of derivative $\dot{x}$ in parameterized integration terms by the term $g(\boldsymbol{z})(u)$ (line 3). This procedure preserves the satisfiability of the formula as stated in Lemma 1, since each uninterpreted function in $\mathcal{G}$ appears in at most one integration term.

Algorithm 1: $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-formulas.

```
Input: A conjunction $\mu = l_1 \wedge \cdots \wedge l_m$ of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals
Output: An equisatisfiable $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula
1 for each $\dot{x} \in \mathcal{G}$ inside $\mu$'s integration terms do
2     pick a literal $l = \forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$ in $\mu$;
3     replace each $\dot{x}$ by $g(\boldsymbol{z})$ in $\mu$'s integration terms;
4     for each literal $l'$ in $\mu$ including $\dot{x}$ other than $l$ do
5         if $l' = \forall u \in [0,t_x].\, \dot{x}(u) = g'(\boldsymbol{z})(u)$ then
6             combine $\forall u \in [0,t_x].\, g(\boldsymbol{z})(u) = g'(\boldsymbol{z})(u)$ with $\mu$;
7         else
8             combine $\exists u \in [0,t_x].\, g(\boldsymbol{z})(u) \neq g'(\boldsymbol{z})(u)$ with $\mu$;
9 return $\mu$;
```

Lemma 1. *Given a conjunction μ of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals including $\forall u \in [0,t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$, the formula μ' obtained from μ by replacing each occurrence of $\dot{x}$ in integration terms by $g(\boldsymbol{z})$ is equisatisfiable to the conjunction μ.*

As a result, every parameterized integration term in the conjunction μ becomes *concrete solution $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-term* without free ODE variables. That is, each integration term is now considered as a computable real function in $\mathcal{F}'$.

We add extra constraints to ensure the consistency between the chosen $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals and the other $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals in μ (line 4). For each pair of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-literals in μ, we must ensure that they are satisfiable at the same time. For example, we need the constraint $\forall u \in [0,t_x].\, g(\boldsymbol{z})(u) = g'(\boldsymbol{z})(u)$ for a universally quantified literal $\forall u \in [0,t_x].\, \dot{x}(u) = g'(\boldsymbol{z})(u)$ in μ. These new constraints are often *not* $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas, since they can still include ODE variables (e.g., $\boldsymbol{z}$).

We express these constraints in $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$ by adding extra ODEs to corresponding integration terms. For a constraint $\forall u \in [0,t_x].\, g(\boldsymbol{z})(u) = g'(\boldsymbol{z})(u)$, we define a new ODE $\frac{\mathrm{d}w}{\mathrm{d}s} = g'(\boldsymbol{z})(s) - g(\boldsymbol{z})(s)$ with a fresh variable w. If the constraint is true, the value of the variable w is always 0; that is, the invariant condition $\forall u \in [0,t_x].\, w(u) = 0$ must hold. We formally define this process as follows.

Definition 8. *For constraints $\forall u \in [0,t_x].\, g(\boldsymbol{z})(u) = g'(\boldsymbol{z})(u)$ or $\exists u \in [0,t_x].\, g(\boldsymbol{z})(s) \neq g'(\boldsymbol{z})(s)$, the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction of integration $F(\boldsymbol{y^0}, t) \equiv \boldsymbol{y^0} + \int_0^t \boldsymbol{f}(\boldsymbol{z})(s)\,\mathrm{d}s$ is the term:*

$$\hat{F}(\boldsymbol{y^0}, t) \equiv [0, \boldsymbol{y^0}] + \int_0^t [g'(\boldsymbol{z})(s) - g(\boldsymbol{z})(s),\; \boldsymbol{f}(\boldsymbol{z})(s)]\,\mathrm{d}s,$$

and the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reductions of the constraints are respectively:

$$\forall u \in [0,t_x].\, \pi_1(\hat{F}(\boldsymbol{y^0}, u)) = 0,\; \exists u \in [0,t_x].\, \pi_1(\hat{F}(\boldsymbol{y^0}, u)) \neq 0.$$

This process also preserves the satisfiability of the formula as stated in Lemma 2, because every corresponding integration term of a constraint is identical by assumption.

Lemma 2. *A conjunction μ with an extra constraint is equisatisfiable to the formula μ' obtained from μ by replacing corresponding integration terms by their $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reductions and replacing the constraint by its $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction.*

The time complexity of the entire $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction process is $O(n^2)$, where n denotes the number of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals in the conjunction μ.[2] The correctness of our $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction procedure follows from the fact that each $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction step in Alg. 1 preserves the satisfiability.

Theorem 2. *Given a conjunction μ that meet the boundedness constraint and the syntactic restriction in Def. 6, Algorithm 1 generates an $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula that is equisatisfiable to μ.*

C. Checking Satisfiability Using $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-Reduction

We consider the satisfiability of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas of Def. 6. We apply the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction process to have equisatisfiable $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas without extra function symbols in $\mathcal{G}$, and then employ an existing algorithm to check $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-satisfiability as a subroutine. Our algorithm terminates if the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-satisfiability subroutine is terminated. This provides a theory solver for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$, by means of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction and an $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-solver.

Algorithm 2: SMT procedure for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formulas.

```
Input: An $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-formula $\exists \boldsymbol{y}.\, \varphi \wedge \bigwedge_{\dot{x} \in \mathcal{G}} bound_{G_x}(\dot{x})$
Output: Unsat, or Sat with a satisfiable assignment
1 while $\exists$ a propositionally satisfiable set $L$ of literals do
2     $\phi \leftarrow \mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction($\bigwedge_{l \in L} l$);
3     if $\phi$ is Sat by $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-satisfiability solving then
4         return Sat and a satisfiable assignment of $\boldsymbol{y}$;
5     learn the conflicts between $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals;
6 return Unsat;
```

Algorithm 2 summarizes our algorithm. We employ the DPLL($\mathcal{T}$) framework to obtain a propositionally satisfiable set of literals by using Boolean satisfiability solving (line 1). The formula is unsatisfiable if no propositionally satisfiable set exists (line 6). Otherwise, each set imposes a conjunction of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals. Since we only consider formulas in the class of Def. 6, the conjunction satisfies the boundedness constraint.

We apply the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction procedure in Alg. 1 to obtain an *equisatisfiable $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula* ϕ (line 2). We use an SMT algorithm for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formula as a subroutine (line 3). If the resulting formula ϕ is satisfiable, then the original formula is also satisfiable (line 4).[3] If ϕ is not satisfiable, conflict clauses can be used by SAT solving for the next iteration based on conflict-driven clause learning (line 5).

D. δ-Complete SMT for $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-Formulas

Our algorithm is best suited for nonlinear hybrid systems where continuous I/O cannot be encoded as discrete actions. For example, δ-complete SMT can be applied to check the satisfiability of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas up to a given precision $\delta > 0$, called *δ-satisfiability* [3]. The satisfiability of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas is in general undecidable for nonlinear real functions, but the δ-satisfiability of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas is decidable.

[2]There can be many identical integration terms for each step, but the replacements can be performed in constant time using subformula sharing.

[3]Since the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction does not alter first-order variables, satisfiable assignments for μ and ϕ have the same values for the first-order variables.

Finding conflict $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals is very important for the performance of DPLL($\mathcal{T}$) and conflict-driven clause learning, but this process is nontrivial for δ-complete SMT. The reason is that δ-consistency actually depends on the value of δ. For example, $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals $\forall u \in [0, 0.5].\, \dot{x}(u) = 1 + \frac{1}{2}u$ and $\forall u \in [0, 0.5].\, \dot{x}(u) = \sqrt{u+1}$ are inconsistent up to precision $\delta = 0.01$, but consistent up to different precision $\delta = 0.1$.

To facilitate the process of finding conflict $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals, we use *uniqueness lemmas* for incompatible $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals. Consider two $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literals $\forall u \in [0, t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u)$ and $\forall u \in [0, t_x].\, \dot{x}(u) = g'(\boldsymbol{z})(u)$. If we know in advance that two functions g and g' are not equal, we add the formula $\neg[\forall u \in [0, t_x].\, \dot{x}(u) = g(\boldsymbol{z})(u) \wedge \forall u \in [0, t_x].\, \dot{x}(u) = g'(\boldsymbol{z})(u)]$ and the consistency checking can be performed at line 1.

For the reachability of networks of hybrid I/O automata, we apply a heuristic specialized for the encoding in Def. 5. Since each mode q can correspond to only one flow condition, if a formula $\forall u \in [0, t].\ [\dot{\boldsymbol{x}}, \dot{\boldsymbol{o}}](u) = [\boldsymbol{f_q}, \boldsymbol{g_q}](u)$ for one continuous output is true, the truths of other continuous output formulas are not relevant. In Alg. 1, we choose one universally quantified $\mathcal{L}_{\mathbb{R}_{\mathcal{F}' \cup \mathcal{G}}}$-literal for each mode, and disregard extra consistency checking in line 4 in this case.

V. Case Studies

This section shows a number of examples of networks of hybrid system components, besides the airplane example. They include discrete and continuous interactions between components, and involve nontrivial nonlinear ODEs.

A. Driving Simple Cars.

A number of cars are running in sequence, while each car follows the behavior of the car in front (the first car moves according to its own scenario). The position (x_i, y_i) and the direction θ_i of each car i of length L_i depends on its speed v_i and steering angle ϕ_i, given by the nonlinear ODEs [18]:

$$\dot{x_i} = v_i \cos\theta_i, \qquad \dot{\theta_i} = v_i / L_i \cdot \tan\phi_i,$$
$$\dot{y_i} = v_i \sin\theta_i, \qquad \dot{\phi_i} = -k_i(\phi_i - \phi_{i-1}),$$

To keep a safe distance, each car has three modes *acc*, *dec*, and *keep* for acceleration, deceleration, and following the speed of the front car, respectively: $\dot{v_i} = -K_i(v_i - v_{i-1})$ if $q_i = keep$, $\dot{v_i} = C$ if $q_i = acc$, and $\dot{v_i} = -C$ if $q_i = dec$. The goal is to find a mode change schedule for driving cars safely.

B. Network of Thermostat Controllers.

A number of rooms are interconnected by open doors (Fig. 2). The temperature x_i of each room i is separately controlled by each thermostat, depending on both the heater's mode $q_i \in \{m_{\text{on}}, m_{\texttt{off}}\}$ and the temperatures of the adjacent rooms. The value of x_i changes according to the ODEs:

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = \begin{cases} K_i\big(h_i - (c_i x_i - d_i \sum_{j \in A_i} x_j)\big) & \text{if } q_i = m_{\text{on}} \\ -K_i\big(c_i x_i - d_i \sum_{j \in A_i} x_j\big) & \text{if } q_i = m_{\texttt{off}}, \end{cases}$$

where A_i is the set of the adjacent rooms, and K_i, h_i, c_i, d_i depend on the size of the room, the heater's power, and the size of the open doors. The goal is to keep each temperature in a desired range, while the outside temperatures change.

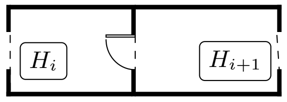


Fig. 2. Connected rooms

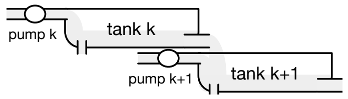


Fig. 3. Connected water tanks

C. Network of Water Tanks.

A number of water tanks are connected by pipes (Fig. 3), adapted from [9]. The water level x_i of each tank i is separately controlled by its pump, depending on the pump's mode $m_i \in \{m_{\mathtt{on}}, m_{\mathtt{off}}\}$ and the levels of the adjacent tanks. The value of x_i changes according to the nonlinear ODEs:

$$\begin{aligned} A_i\dot{x}_i &= (q_i + a\sqrt{2g}\sqrt{x_{i-1}}) - a\sqrt{2g}\sqrt{x_i} & \text{if } m_i = m_{\mathtt{on}} \\ A_i\dot{x}_i &= a\sqrt{2g}\sqrt{x_{i-1}} - a\sqrt{2g}\sqrt{x_i} & \text{if } m_i = m_{\mathtt{off}} \end{aligned}$$

($x_0 = 0$ for the leftmost tank 1), where A_i, q_i, a depend on the size of the tank, the power of the pump, and the width of the pipe, and g is the standard gravity constant. The goal is to keep each water level in a desired range.

D. Multiple battery Usage.

Given a number of fully charged batteries, a controller switches load between them to achieve longer lifetime out of the batteries, adapted from [19]. Each battery i has three modes *switchedOn*, *switchedOff*, and *dead*. The battery charge dynamics is expressed by the ODEs:

$$(\text{on}) \begin{array}{l} \dot{d}_i = L/c - kd_i \\ \dot{g}_i = -L, \end{array} \quad (\text{off}) \begin{array}{l} \dot{d}_i = -kd_i \\ \dot{g}_i = 0, \end{array} \quad (\text{dead}) \begin{array}{l} \dot{d}_i = 0 \\ \dot{g}_i = 0, \end{array}$$

with d_i its kinetic energy difference, g_i its total charge, L its load, and $c \in [0,1]$ its threshold. If $g_i > (1-c)d_i$, battery i is dead. Otherwise, it can be either on or off. When k batteries are on, load to each battery is divided by k. A goal is to find a switching schedule to achieve a desired lifetime.

VI. Experimental Results

We have implemented our algorithm in version 2 of the dReal solver [12]. It can decide the δ-satisfiability of a wide range of $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-formulas (containing universal quantification over time).[4] We base our implementation on dReal, since the tool supports nontrivial *nonlinear ODEs* in our case studies.[5] However, in principle our techniques can be combined with any other SMT-based approaches for hybrid systems.

We have compared the performance of our algorithm for the $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-encoding with one by a non-modular $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-encoding. Because our examples involve continuous I/O that cannot be encoded by discrete actions, no modular $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-encoding is possible. Therefore, we use an $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-encoding of a single hybrid automaton that is equivalent to the composition, which has been studied in many approaches [1]–[5], [14], [20].

[4]dReal uses interval constraint propagation (ICP) to numerically evaluate ODE integration functions up to a precision $\delta > 0$ [14].

[5]For example, many analysis tools for nonlinear hybrid systems only support polynomials or linear ODEs, not nonlinear ODEs.

The experimental results are summarized in Fig. 4. The case studies and the experimental results are available in [16]. We have performed reachability analysis for the five case studies up to bound $k = 5$. We consider two variants for each example, with double or triple components (for the airplane example, nonlinear ODEs or linear ODEs). We consider both *sat* (reachable) and *unsat* (unreachable) cases using different goals. All experiments were conducted on Intel Xeon 2.6 GHz with 512 GB memory. We set a timeout of 12 hours.

The results show that our approach with the new modular $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-encoding *significantly outperforms* the old approach with the non-modular encoding. For example, the analysis with the non-modular $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-encoding for three interconnected thermostats did not terminate within 12 hours for bound $k = 2$, whereas the same analysis by our algorithm with the modular $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'\cup\mathcal{G}}}$-encoding gave the result less than 15 seconds.

According to the results, the performance improvement tends to be more apparent for bigger models and for the *unsat* cases. For the *sat* cases, it is possible that a satisfiable assignment can be found in an early stage of DPLL($\mathcal{T}$), e.g., the *sat* case of two batteries where the difference is small. In a bigger model the size of the generated formula becomes larger, and thus the benefit of using the modular encoding is clearly more explicit (e.g., the cases of three batteries).

This performance improvement is due to the fact that the modular encoding allows a much compact size of the formulas, and a more efficient ODE solving by decomposition of complex systems of ODEs. Also, conflict-driven clause learning can be fully exploited for continuous connections in this way, because conflicts caused by continuous I/O can be detected and then learned in our algorithm.

VII. Related Work

One of the early studies on SMT-based analysis of nonlinear hybrid systems is [21], which proposes constraint solving algorithms for nonlinear reachability problems. There are several approaches that explicitly formulate analysis problems of nonlinear hybrid systems as SMT formulas over the real numbers, such as MathSAT/HyCOMP [5], [6], hydlogic [4], iSAT/HySAT [1], [2], and dReal/dReach [12], [20].

But a modular SMT encoding of networks of nonlinear hybrid systems has not been much investigated, which is what we aim to improve in this paper. All of these techniques assume that interactions between components can be specified through discrete synchronization (e.g., using joint actions). Hence, continuous interactions between components, which occur frequently in many networks of hybrid systems as shown in Sec. V, cannot be properly dealt with in a modular way.

In iSAT-ODE [1], the syntax allows ODE fragments to occur positively in formulas, and thus the encoding of Def. 6 can be expressed in principle. But a modular encoding has not been studied for iSAT-ODE, and the tool does not support it either. Our work formally identifies a generic class of formulas that *strictly* includes one supported by iSAT-ODE. Our algorithm is based on equisatisfiable $\mathcal{L}_{\mathbb{R}_{\mathcal{F}'}}$-reduction, whereas iSAT-ODE uses a specialized algorithm that extends DPLL.

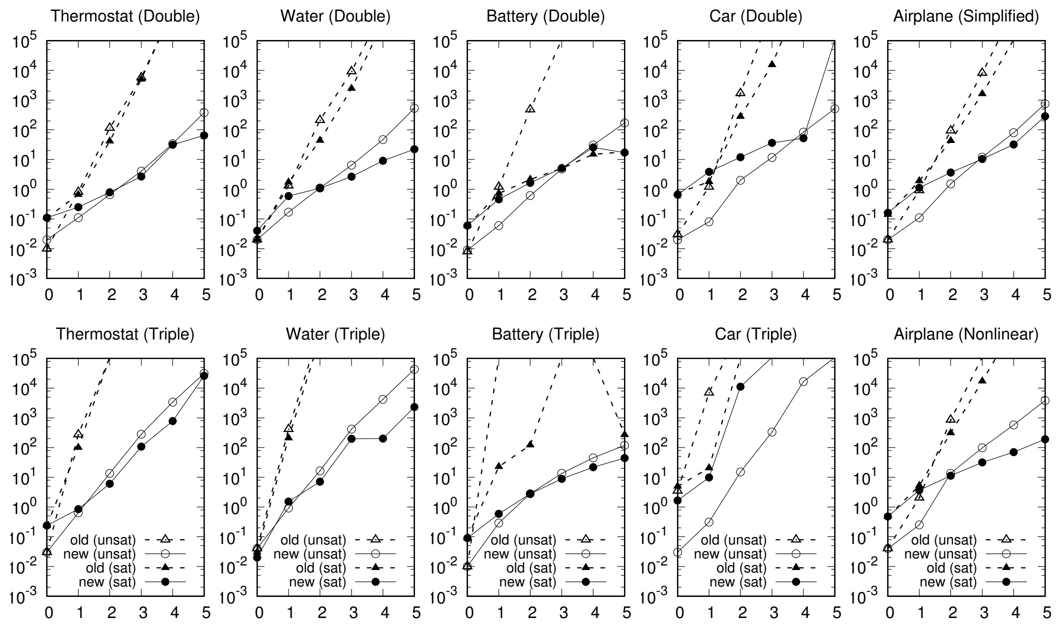


Fig. 4. Running time (in seconds) of k-step reachability analysis (solid lines for the new approach and dashed lines for the non-modular approach)

In addition to SMT-based approaches, there are many other approaches to analyze nonlinear hybrid systems by calculating a set of reachable states, e.g., [22], [23]. Our technique and such reachable-set computation techniques focus on different aspects of hybrid system analysis. Reachable-set computation can be used as an ODE solver in Alg. 2, while our technique addresses higher-level composition and modular analysis.

VIII. Conclusions

We have presented new SMT-based techniques for analyzing networks of nonlinear hybrid systems. We have shown that continuous interactions between hybrid components, which cannot be captured by discrete communication in general, can be decomposed and modularly encoded as SMT formulas. Since existing SMT algorithms cannot deal with the modular encoding, we have presented a new SMT solving algorithm, which can greatly increase the performance of SMT-based analysis for networks of nonlinear hybrid systems.

Future work will include investigating approximation and decomposition methods to achieve further scalability when analyzing networks of nonlinear hybrid systems.

References

[1] A. Eggers, N. Ramdani, N. S. Nedialkov, and M. Fränzle, “Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods,” *Softw. Syst. Model.*, vol. 14, no. 1, 2015.

[2] M. Fränzle and C. Herde, “HySAT: An efficient proof engine for bounded model checking of hybrid systems,” *Form. Methods Syst. Des.*, vol. 30, no. 3, pp. 179–198, 2007.

[3] S. Gao, J. Avigad, and E. M. Clarke, “δ-complete decision procedures for satisfiability over the reals,” in *IJCAR*. Springer, 2012, pp. 286–300.

[4] D. Ishii, K. Ueda, and H. Hosobe, “An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems,” *Int. J. Softw. Tools Technol. Transf.*, vol. 13, no. 5, pp. 449–461, Oct. 2011.

[5] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “HyComp: An SMT-based model checker for hybrid systems,” in *TACAS*, ser. LNCS, vol. 9035. Springer, 2015, pp. 52–67.

[6] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, “The MathSat5 SMT solver,” in *TACAS*. Springer, 2013, pp. 93–107.

[7] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, “Powertrain control verification benchmark,” in *HSCC*. ACM, 2014, pp. 253–262.

[8] K. Bae, P. C. Ölveczky, S. Kong, S. Gao, and E. M. Clarke, “SMT-based analysis of virtually synchronous distributed hybrid systems,” in *HSCC*. ACM, 2016, pp. 145–154.

[9] J. Raisch, E. Klein, C. Meder, A. Itigin, and S. O’Young, “Approximating automata and discrete control for continuous systems — two examples from process control,” in *Hybrid systems V*. Springer, 1999.

[10] T. A. Henzinger, “The theory of hybrid automata,” in *LICS*. IEEE Computer Society, 1996, pp. 278–292.

[11] N. Lynch, R. Segala, and F. Vaandrager, “Hybrid I/O automata,” *Inf. Comput.*, vol. 185, no. 1, pp. 105–157, 2003.

[12] S. Gao, S. Kong, and E. M. Clarke, “dReal: An SMT solver for nonlinear theories over the reals,” in *CADE*. Springer, 2013, pp. 208–214.

[13] K. Bae, J. Krisiloff, J. Meseguer, and P. C. Ölveczky, “Designing and verifying distributed cyber-physical systems using Multirate PALS: An airplane turning control system case study,” *Sci. Comput. Program.*, vol. 103, no. C, pp. 13–50, 2015.

[14] S. Gao, S. Kong, and E. M. Clarke, “Satisfiability modulo ODEs,” in *FMCAD*. IEEE, 2013, pp. 105–112.

[15] K.-I. Ko, *Complexity theory of real functions*. Birkhäuser, 1991.

[16] K. Bae and S. Gao, “Modular smt-based analysis of nonlinear hybrid systems,” manuscript, August 2017. [Online]. Available: https://kquine.github.io/hybrid/fmcad17

[17] V. D. Blondel and J. N. Tsitsiklis, “A survey of computational complexity results in systems and control,” *Automatica*, vol. 36, no. 9, 2000.

[18] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[19] M. Fox, D. Long, and D. Magazzeni, “Plan-based policies for efficient multiple battery load management,” *J. Artif. Int. Res.*, vol. 44, 2012.

[20] S. Kong, S. Gao, W. Chen, and E. M. Clarke, “dReach: δ-reachability analysis for hybrid systems,” in *TACAS*. Springer, 2015, pp. 200–205.

[21] S. Ratschan and Z. She, “Safety verification of hybrid systems by constraint propagation-based abstraction refinement,” *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, 2007.

[22] G. Frehse, R. Kateja, and C. Le Guernic, “Flowpipe approximation and clustering in space-time,” in *HSCC*. ACM, 2013, pp. 203–212.

[23] X. Chen, E. Ábrahám, and S. Sankaranarayanan, “Flow*: An analyzer for non-linear hybrid systems,” in *CAV*. Springer, 2013, pp. 258–263.

SMT-based Analysis of Switching Multi-Domain Linear Kirchhoff Networks

Alessandro Cimatti
Fondazione Bruno Kessler
Trento, IT
Email: cimatti@fbk.eu

Sergio Mover
University of Colorado Boulder
Boulder, USA
Email: sergio.mover@colorado.edu

Mirko Sessa
Fondazione Bruno Kessler and
University of Trento
Trento, IT
Email: sessa@fbk.eu

***Abstract*—Many critical systems are based on the combination of components from different physical domains (e.g. mechanical, electrical, hydraulic), and are mathematically modeled as Switched Multi-Domain Linear Kirchhoff Networks (SMDLKN). In this paper, we tackle a major obstacle to formal verification of SMDLKN, namely devising a global model amenable to verification in the form of a Hybrid Automaton. This requires the combination of the local dynamics of the components, expressed as Differential Algebraic Equations, according to Kirchhoff's laws, depending on the (exponentially many) operation modes of the network.**

We propose an automated SMT-based method to analyze networks from multiple physical domains, detecting which modes induce invalid (i.e. inconsistent) constraints, and to produce a Hybrid Automaton model that accurately describes, in terms of Ordinary Differential Equations, the system evolution in the valid modes, catching also the possible non-deterministic behaviors. The experimental evaluation demonstrates that the proposed approach allows several complex multi-domain systems to be formally analyzed and model checked against various system requirements.

I. Introduction

Complex critical systems are often formed by the interaction of components from multiple physical domains (e.g. electrical, hydraulic, and mechanical). An example from aerospace is a landing gear system [1], depicted in Fig. 1, where the pressure applied by a hydraulic circuit (including valves and pumps) operates moving components from the hydro-mechanical domain (e.g. a cylinder). Basic components (e.g. valves, accumulators, and tanks) have multiple operation modes and exhibit hybrid dynamics. These dynamics include continuous behaviors, typically described by Differential-Algebraic Equations (DAE) associated to the modes, and instantaneous changes (or switches) among modes. The connection of basic components into composite systems is often modeled as Switched Multi-Domain Linear Kirchhoff Networks (SMDLKN) [2]. Each combination of the components modes determines a (global) mode of the network. For each global mode, the continuous dynamics is represented by the system of DAE obtained by joining the equations that characterize each component in the respective mode with the equations that correspond to the Kirchhoff's connection laws.

In this paper, we investigate methods for the formal analysis of SMDLKN, tackling two key challenges. The first challenge is to convert a DAE-based network description into a formalism based on Ordinary Differential Equations (ODE) and

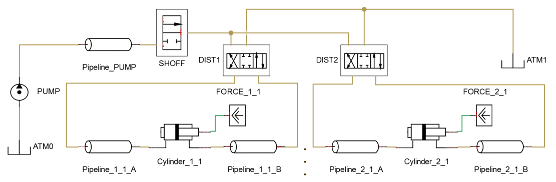


Fig. 1. Landing Gear System with $N = 2$ hydraulic cylinder lines (LGS$_{[N]}$).

that is amenable to formal verification. The existing formal verification tools for hybrid systems [3], [4], [5] take as input *hybrid automata* and, in most cases, require a description of the continuous dynamics in the form of ODE. Obtaining an ODE from a DAE is possible with a process called *reformulation* [6]. One could thus conceive an approach that iterates over the network modes, reformulates for each of them the corresponding DAE into an ODE, and recombines the resulting ODE into an automaton. Unfortunately, this iterative approach is unfeasible in practice: the number of modes of a switched network is exponential in the number of components.

The second challenge stems from the fact that the reformulation cannot always map a DAE onto an ODE. In fact, a DAE is a relational characterization deriving from a constraint-based description, while an ODE is in essence a functional description. Thus, under certain conditions, a DAE may be inconsistent (i.e. infeasible from the physical standpoint) or under-constrained (i.e. some physical quantities are undetermined). Unfortunately, inconsistencies and under-specifications may be hidden in the (exponentially many) modes of the network, and may be hard to spot.

In this paper, we propose a general method to reformulate SMDLKN into hybrid automata with ODE dynamics. In order to deal with multi-domain networks, we propose a purely algebraic, general argument, which guarantees the existence of the reformulation, generalizing the *Implicit Function Theorem* [7] for linear systems. The method is able to synthesize the modes free from inconsistencies and under-specifications, and to present them in the form of diagnostic information.

We adopt an approach based on Satisfiability Modulo Theories (SMT) [8] to reason about the algebraic representation of DAE-based networks. We build on the ability of modern SMT solvers to carry out quantifier elimination and to deal with huge sets of assignments to discrete variables. We exploit the algebraic nature of the problem, in particular the linearity principle holding for the DAE associated to each network modes,

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

to aggressively simplify the expensive quantifier elimination steps.

We perform an experimental evaluation on several multi-domain scalable real-world benchmark applications. The proposed optimizations substantially increase the scalability of the procedures, allowing us to validate and reformulate SMDLKN featuring millions of modes. We verify the hybrid automata resulting from our procedures by means of some existing SMT-based verification tools (e.g. HYCOMP [5]).

The rest of the paper is organized as follows. Section II provides the necessary background notions. Section III defines the validation and reformulation problems for SMDLKN, Section IV presents the proposed symbolic algorithms. Section V discusses the related work. Section VI presents the experimental evaluation. We conclude in Section VII.

II. Background

A. Notation

$|X|$ denotes the cardinality of the set X. Given a set of real variables X, the notation $\vec{X}$ refers to the vector that contains all the variables in X ordered lexicographically. $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, $\mathbb{B}$ denote the set of Real numbers, non-negative Real numbers, and Boolean. If X is a set of variables, then X' and $\dot{X}$ are the sets obtained by replacing each element with its primed and dotted version, resp.

We use the standard notions of theory, satisfiability, validity, and logical consequence. We restrict to formulas interpreted with the Theory of Linear Real Arithmetic (LRA) [8]. Given a first-order logic formula ψ and a set of variables X, $\psi(X)$ denotes that X is the set of free variables in ψ. $\varphi \models_{\mathcal{T}} \psi$ denotes that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$; when clear from context, we omit $\mathcal{T}$ and simply write $\varphi \models \psi$. An assignment μ for a set of variables X is the set $\{x \mapsto c \mid x \in X \text{ and } c \text{ is a constant}\}$, $\mu_{|X}$ is the projection of all the assignments in μ only onto the variables contained in X, and $\mu(x)$ is the value assigned to x in μ. Abusing the notation, we interchange the linear system notation (e.g. $\vec{X} = \vec{B}\vec{Y}$, where $|\vec{X}| = n \times 1, |\vec{B}| = n \times m, |\vec{Y}| = m \times 1$) with the conjunction of predicates in LRA corresponding to the matrix product (e.g. $\bigwedge_{i=1}^{n} \vec{X}[i] = \sum_{j=1}^{m}(\vec{B}[i][j]\vec{Y}[j])$). Given two vectors (resp. matrices) A and B, $A \cdot B$ denotes their vertical (resp. horizontal) concatenation.

B. Linear Systems

The linear system $\vec{A}\vec{W} = \vec{b}$ is *homogeneous* if $\vec{b} = \vec{0}$, and, in that case, it admits at least the solution $\vec{\vec{W}} = \vec{0}$.

Given a solvable linear system $\vec{\vec{A}}\vec{W} = \vec{b}$, its *general* solution is $\vec{\vec{W}} = \vec{\vec{W}}_p + \vec{\vec{W}}_h$, where $\vec{\vec{W}}_p$ is a *particular* solution of the *inhomogeneous* system $\vec{\vec{A}}\vec{W} = \vec{b}$ and $\vec{\vec{W}}_h$ is the *homogeneous* solution of the homogeneous system $\vec{\vec{A}}\vec{W} = \vec{0}$. The existence of the *particular* solution $\vec{\vec{W}}_p$ guarantees the *existence* of at least one solution $\vec{\vec{W}}$.

Lemma 1 (Linearity [9]): Let $\vec{\vec{A}}\vec{W} = \vec{b}_1$, ..., $\vec{\vec{A}}\vec{W} = \vec{b}_n$ be n distinct linear systems and $z_1, ..., z_n \in \mathbb{R}$ n real variables. The systems $\vec{\vec{A}}\vec{W} = \vec{b}_1$, ..., $\vec{\vec{A}}\vec{W} = \vec{b}_n$ are all solvable iff the system $\vec{A}\vec{W} = \vec{b}_1 z_1 + ... + \vec{b}_n z_n$ is solvable for all values of the variables z_1, ..., z_n. ■

C. Hybrid Automata

Hybrid automata (HA) [10] represent a system with continuous and discrete dynamics. We use a symbolic representation of hybrid automata, where the discrete locations and transitions are represented by means of SMT formulae [11].

A HA is a tuple $H = \langle V, X, Init, Invar, Trans, Flow \rangle$ where 1) V is the set of discrete variables; 2) X is the set of continuous variables; 3) $Init(V, X)$ represents the set of initial states; 4) $Invar(V, X)$ represents the set of invariant states; 5) $Trans(V, X, V', X')$ represents the set of discrete transitions; 6) $Flow(V, \dot{X}, X)$ is the flow condition. We assume that all the formulas $Init$, $Invar$, $Trans$ and $Flow$ are quantifier-free and linear.

In the above definition, $Flow$ may either define a system of Differential-Algebraic Equations (DAEs) or Ordinary Differential Equations (ODEs). We say that the automaton has an *ODE dynamics* if, for each assignment μ to V, $Flow$ is equivalent to a system of ODEs (i.e. $\dot{\vec{X}} = \vec{A}\vec{X}$). Otherwise, the automaton has a *DAE dynamics*.

A *state* of a hybrid automaton is an assignment to the variables $V \cup X$, and a *run* is a sequence of states such that the first state is in the initial states, every state belongs to the invariant, and each pair of consecutive states either satisfies $Trans$ or the solution to the differential equations described in the $Flow$ condition. The semantics of the HA is defined by the runs that it accepts. Two hybrid automata H_1 and H_2 are *equivalent* if they accept the same runs.

D. Switching Multi-Domain Linear Kirchhoff Networks

Definition 1 (Network component): A component c_i is a tuple $\langle B_i, R_i, T_i, invar_i, flow_i, input_i, trans_i \rangle$ where:

- B_i : set of *discrete variables* representing the modes.
- R_i : set of *continuous variables* representing the physical quantities of the component. We partition the set of continuous variables in three disjoint sets of *state* (X_i), *input* (U_i) and *output* (Y_i) variables.
- $invar_i : 2^{B_i} \rightarrow 2^{P_{red}}$: *invariant* conditions, where P_{red} is a set of predicates over the variables R_i.
- $input_i : 2^{U_i} \rightarrow 2^{\mathcal{F}_i}$: *input binding* assigning a continuous function of time ($\mathcal{F}_i = \{f(t) | f \text{ is continuous}\}$) to each input variable in U_i.
- $flow_i : 2^{B_i} \rightarrow 2^{P_{eq}}$: *flow* condition, where P_{eq} is a set of homogeneous linear equalities with variables from $X_i, U_i, Y_i, \dot{X}_i$.
- $trans_i(B_i, R_i, B_i')$: *mode transition* condition that represents the mode transitions (with guards) that can happen in the component.

Definition 2: A *Switched Multi-Domain Linear Kirchhoff Network* (SMDLKN) $\mathcal{N}$ is a tuple $\langle \mathcal{C}, K \rangle$, where $\mathcal{C}$ is a set of components and K is a set of equalities among continuous variables of the components, that represents the Kirchhoff conservation rules (i.e. the set of connection constraints).

We extend the notation used to specify the set of component variables to a network $\mathcal{N}$, defining the sets $B := \bigcup_{c_i \in \mathcal{C}} B_i, R := \bigcup_{c_i \in \mathcal{C}} R_i, \ldots$. Let $V = B \cup R$ be the set of all the variables of a network. A *state* of the network is given by an assignment μ to *all the variables* V. We refer to each possible (complete) assignment $\mu_b \in 2^B$ to *all the discrete variables* B as a *mode* of the network. Every different network mode induces a continuous dynamics described by a DAE.

Definition 3 (Differential-Algebraic Equation of a mode): The DAE $DAE(\mu_b)$ of a mode μ_b is defined as the set of constraints:

$$DAE(\mu_b) := \bigcup_{c_i \in \mathcal{C}} flow_i(\mu_{b|B_i}) \cup K \tag{1}$$

$DAE(\mu_b)$ can be equivalently represented as a linear system:

$$\vec{M}\vec{\dot{X}} + \vec{N}\vec{X} + \vec{O}\vec{Y} + \vec{P}\vec{U} = \vec{0} \tag{2}$$

for some coefficient matrices $\vec{M} \in \mathbb{R}^{l \times |X|}, \vec{N} \in \mathbb{R}^{l \times |X|}$, $\vec{O} \in \mathbb{R}^{l \times |Y|}, \vec{P} \in \mathbb{R}^{l \times |U|}$, and a positive integer l equal to the number of system constraints.

Definition 4 (Network semantics): The semantics of the network $\mathcal{N}$ is the hybrid automaton $H_{\mathcal{N}} = \langle V_H, X_H, Init, Invar, Trans, Flow \rangle$ where

$$V_H := B \quad X_H := X \cup U \cup Y \quad Init(V_H, X_H) := True$$

$$Invar(V_H, X_H) := \bigwedge_{\mu_b \in 2^B} (\mu_b \to \bigwedge_{c_i \in \mathcal{C}} invar_i(\mu_{b|B_i}))$$

$$Trans(V_H, X_H, V'_H, X'_H) := \bigwedge_{c_i \in \mathcal{C}} trans_i \wedge \bigwedge_{x \in X} (x' = x)$$

$$Flow(V_H, \dot{X}_H, X_H) := \bigwedge_{\mu_b \in 2^B} (\mu_b \to DAE(\mu_b)) \wedge \bigwedge_{c_i \in \mathcal{C}} input_i(U_i)$$

III. Validation and Reformulation problems

Given a network $\mathcal{N} = \langle \mathcal{C}, K \rangle$, our first goal is to automatically check if it contains inconsistencies, which represent an unwanted condition in the real system modeled by the network.

Definition 5: A mode μ_b of a network $\mathcal{N}$ is *consistent* if, for every possible assignment to the state (X) and input (U) variables, the linear system $DAE(\mu_b)$ has at least a solution. An inconsistent mode in the network represents an undesired condition in the physical system that must be avoided. Consider the electrical circuit of Fig. 2, where the voltages V_{C_1} and V_{C_2} across the capacitors C_1 and C_2 are state variables, and the current I_B imposed by the current generator B is the input variable (we use I and V to refer to currents and voltages, and we use the component's name as subscript to identify the current or voltage of that component). For the sake of brevity, all the electrical parameters take value one and have been omitted from the following formulas. The DAE associated to the discrete mode where both the switches S_1 and S_2 are open is $I_B = I_R, I_R = I_{S_1} + I_{S_2}, I_{S_1} = 0, I_{S_2} = 0, I_{C_1} = \dot{V}_{C_1}, I_{C_2} = \dot{V}_{C_2}$. The mode is not consistent when $I_B \neq 0$. Clearly, inconsistent modes in the design are undesirable, since the behavior of the real system would violate some physical laws. Thus, checking if a mode is consistent is a fundamental step in the validation of $\mathcal{N}$.

Our second goal is to verify safety properties on $\mathcal{N}$. As explained in the introduction, a requirement imposed by the symbolic verification tools for hybrid systems (e.g. HyCOMP [5]) is to express the continuous dynamics of $\mathcal{N}$ as ODEs. This means, for every discrete mode of the network, being able to rewrite the DAE $DAE(\mu_b)$ as a system of *Ordinary Differential Equations*: $\vec{\dot{X}} = \vec{A}\vec{X} + \vec{B}\vec{U}$, where $\vec{A} \in \mathbb{R}^{|X| \times |X|}$, and $\vec{B} \in \mathbb{R}^{|X| \times |U|}$. This amounts to find a function that, for every possible values of the state and input variables, returns one and unique value for the first derivative variables $\dot{X}$. Consistency is a necessary condition to ensure the existence of the ODE function, while the other necessary condition is the determinicity of the values assigned to $\dot{X}$.

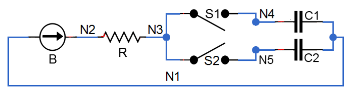


Fig. 2. Schematic of a simple electrical circuit.

Definition 6: A mode μ_b of a network $\mathcal{N}$ is *deterministic* if, for every possible value of the state X and input variables U, the linear system $DAE(\mu_b)$ admits at most one solution of the first derivative $\dot{X}$.

In the example of Fig. 2, the DAE of the discrete mode where both the switches S_1 and S_2 are closed is $I_B = I_R, I_R = I_{S_1} + I_{S_2}, I_{C_1} = I_{S_1}, I_{C_2} = I_{S_2}, I_{C_1} + I_{C_2} = I_B, I_{C_1} = \dot{V}_{C_1}, I_{C_2} = \dot{V}_{C_2}$. In this DAE, the values of the currents I_{C_1} and I_{C_2} are not uniquely identified when fixing a value of the input I_B (e.g. if $I_B = 3$ then the only constraints for I_{C_1} and I_{C_2} is that $I_{C_1} + I_{C_2} = 3$), and hence also the values of $\dot{V}_{C_1}$ and $\dot{V}_{C_2}$ is not. Due to this non-determinism, the above DAE cannot be rewritten as an ODE.

Definition 7: A mode μ_b of a network $\mathcal{N}$ is *valid* if it is both *consistent* and *deterministic*. The network $\mathcal{N}$ is *valid* if all the modes $\mu_b \in 2^B$ are valid.

Definition 8 (Validation problem): Given a network $\mathcal{N}$, the validation problem consists of deciding if $\mathcal{N}$ is valid.

Remark 1: We note that a mode is valid if it is associated to a *index-1* DAE, while it is invalid for *higher-index* DAEs.

Definition 9 (Reformulation problem): Given a valid network $\mathcal{N}$, the *reformulation problem* consists of obtaining a hybrid automaton H with **ODE dynamics** that is equivalent to $H_{\mathcal{N}}$. The reformulated automaton represents the same discrete modes as the network $\mathcal{N}$, but its continuous dynamics is expressed as a system of ODEs. Such representation exists since the network is valid.

In the electrical circuit in Fig. 2, the mode where the switch S_1 is closed and S_2 is open has the DAE $I_B = I_R, I_R = I_{S_1}, I_{S_1} = I_{C_1}, I_{C_1} = \dot{V}_{C_1}, 0 = I_{S_1}, I_{S_2} = I_{C_2}, I_{C_2} = \dot{V}_{C_2}$. The mode is valid, and the ODE representation of the DAE is $\dot{V}_{C_1} = 0V_{C_1} + 0V_{C_2} + 1I_B, \dot{V}_{C_2} = 0V_{C_1} + 0V_{C_2} + 0I_B$.

IV. Symbolic validation and reformulation

A. Basic Validation and Reformulation

Our technique performs the following steps to produce a symbolic hybrid automaton $H_{\mathcal{N}}$ amenable to verification from the network $\mathcal{N} = \langle \mathcal{C}, K \rangle$:

1) Check if all the modes of $\mathcal{N}$ are consistent. If it is the case, we proceed to the next step.

2) Check if all the modes of $\mathcal{N}$ are deterministic. If it is the case, $\mathcal{N}$ is valid and we proceed to the reformulation.
3) Reformulate all the modes of $\mathcal{N}$ and define $H_{\mathcal{N}}$.

In the case $\mathcal{N}$ is not consistent, our approach finds all the non-consistent modes, that can be used by the designer to fix the network. While we restrict the presentation to the case where $\mathcal{N}$ is consistent, our approach also performs a *partial reformulation*, that reformulates the DAEs only for the consistent modes. The partial network is necessary in the common scenario where a discrete controller is composed with the network with the goal of keeping the network *outside* the non-consistent states. In this scenario, our approach allows us to verify if such controller is correct.

Validation and reformulation steps can be done for each mode $\mu_b \in 2^B$ of $\mathcal{N}$. However, this is not feasible since the number of modes is exponential in the number of the discrete variables of $\mathcal{N}$. To scale and analyze real networks, we use a symbolic approach. In this section, we present a symbolic validation and reformulation for multi-domain networks. The idea is to express the validation and reformulation problems as a first-order logic formula.

a) SMT encodings of the network DAEs: We represent all the DAEs of the network $\mathcal{N}$ as the quantifier-free formula:

$$\psi_{\text{DAE}} := (\bigwedge_{c_i \in \mathcal{C}} \bigwedge_{\mu_b \in 2^{B} i} (\mu_b \rightarrow flow_i(\mu_b))) \wedge \bigwedge_{k \in K} k \qquad (3)$$

ψ_{DAE} predicates over the same variables of the network, so we reuse the same notation introduced in Sec. II-D for the network variables, and contains the Boolean variables B, and the Real variables $X, U, Y, \dot{X}$. The validation and reformulation problems only consider the algebraic relationships among the variables defined by the DAE, while they disregard their dependence on time. Thus, the derivative variables in $\dot{X}$ are treated as Real, and not Continuous, variables. Note that the provided encoding enumerates the components local modes in place of the network global modes, thus preventing the blow-up of the formula ψ_{DAE}.

Lemma 2: μ is a satisfying assignment of ψ_{DAE} iff $\mu_{|R}$ is a solution of $DAE(\mu_{|B})$ ■

We provide the proofs of the lemmas and theorems in an extended version of the paper available at http://es.fbk.eu/people/sessa/paper/fmcad17/main.pdf

b) Checking the network for consistency: All the modes of $\mathcal{N}$ are consistent iff the following formula is valid:

$$\psi_{con}(B) := \forall X, U. \exists Y, \dot{X}. \psi_{\text{DAE}}(B, X, U, Y, \dot{X})$$

ψ_{con} represents the set of all the consistent modes.

c) Checking the network for determinicity: All the modes of $\mathcal{N}$ are deterministic iff the following formula is valid:

$$\begin{aligned}\psi_{det}(B) := \forall X, U, \dot{X}_1, \dot{X}_2. \\ ((\exists Y. \psi_{\text{DAE}}(B, X, U, Y, \dot{X}_1) \wedge \\ \exists Y. \psi_{\text{DAE}}(B, X, U, Y, \dot{X}_2)) \rightarrow \dot{X}_1 = \dot{X}_2)\end{aligned}$$

ψ_{det} represents the set of all the deterministic modes.

d) Reformulating the network: We reformulate a valid network $\mathcal{N}$ into the hybrid automaton $H^r_{\mathcal{N}} = \langle V^r, X^r, Init^r, Invar^r, Trans^r, Flow^r \rangle$. $H^r_{\mathcal{N}}$ is defined as the hybrid automaton $H_{\mathcal{N}}$ in the Definition 4, except for $Invar^r$ and $Flow^r$. The invariant condition is given by $Invar^r := \psi_Y \wedge \bigwedge_{c_i \in \mathcal{C}} \bigwedge_{\mu_b \in 2^{B} i} (\mu_b \rightarrow invar_i(\mu_b))$, while $Flow^r := \psi_{\dot{X}} \wedge \bigwedge_{c_i \in \mathcal{C}} input_i(U_i)$. The formula $\psi_{\dot{X}}$ is the reformulation of the variables $\dot{X}$, while ψ_Y is a relation that represents the values of the output variables Y w.r.t. the state X and input U variables. While we can compute the relation for ψ_Y as $\exists \dot{X}. \psi_{\text{DAE}}(B, X, U, Y, \dot{X})$, finding the $\psi_{\dot{X}}$ is a more difficult task that requires to solve a quantified formula expressed with non-linear arithmetic terms (that synthesize the coefficients of the ODE). We know that such formula cannot be solved efficiently. We do not try to compute it and in our experiments we try to compute $\exists \dot{X}. \psi_{\text{DAE}}(B, X, U, Y, \dot{X})$. This formula *does not* reformulate the system into an ODE, but the time necessary to solve it provides a lower bound for a more complex formula (i.e. with more quantifiers and over non-linear arithmetic predicates).

B. Optimized Validation and Reformulation

We improve the basic validation and reformulation procedures by applying an extension of the *implicit function theorem* [9]. Given a system of linear equalities, the theorem gives the necessary and sufficient conditions that allow us to express the values of a subset of the system variables (the dependent variables) as a function of the remaining variables (the independent variables). For our application, the linear system is the DAE of a mode, the dependent variables are the derivatives $\dot{X}$, and the independent variables are the state X and input U. Our problem is slightly more complex, since the DAE also contains the output variables Y. One option is to consider them as dependent variables, requiring to find a function that expresses the value of all the variables in Y. However, this limits the applicability of our approach: while we have to express $\dot{X}$ as a system of ODEs, the underlying verification tool does not impose any restriction on the output variables Y that, for example, can assume a value non-deterministically. For this reason we extend the implicit function theorem as follows.

a) Implicit Function Theorem:

Theorem 1 (Implicit Function Theorem): Let m, n, l be positive integers. Let $F : \mathbb{R}^{m+n} \rightarrow \mathbb{R}^l$ be a homogeneous implicit linear function $F(\vec{W}, \vec{Z}) := \vec{A}\vec{W} + \vec{B}\vec{Z} = \vec{0}$, where $\vec{W} \in \mathbb{R}^{m \times 1}$, $\vec{Z} \in \mathbb{R}^{n \times 1}$, $\vec{A} \in \mathbb{R}^{l \times m}$, and $\vec{B} \in \mathbb{R}^{l \times n}$. Let $\vec{b}_i$ be the i-th column vector of the matrix $\vec{B}$, where $i \in \{1, ..., n\}$. Let w_j be the j-th variable of $\vec{W}$, where $j \in \{1, ..., m\}$. The following two conditions hold:

1) *consistency condition:* for all $1 \leq i \leq n$, the linear system $\vec{A}\vec{W} = \vec{b}_i$ is solvable, and
2) *determinicity condition:* the linear system $\vec{A}\vec{W} = \vec{0}$ does not admit any homogeneous solution $\vec{W}_h$ such that its j-th component w_j is different from zero

iff there exists a unique linear function $f_j : \mathbb{R}^n \rightarrow \mathbb{R}^1$ such that $w_j = f_j(\vec{Z})$ and $F(w_1, ..., f_j(\vec{Z}), ..., w_m, \vec{Z}) = \vec{0}$. ■

The condition (1) guarantees that the system $\vec{A}\vec{W} = -\vec{B}\vec{Z}$ admits at least one solution $\vec{W}$ for every assignment to the variables $\vec{Z}$, reducing the problem to a *finite* number of n checks; the condition (2) guarantees that, for every assignment to the variables $\vec{Z}$, every solution $\vec{W}$ admits a unique assignment to its j-th component w_j.

Consider the DAE $DAE(\mu_b)$ of the mode μ_b and its matrix representation $\vec{M}\dot{\vec{X}} + \vec{N}\vec{X} + \vec{O}\vec{Y} + \vec{P}\vec{U} = \vec{0}$ (see Equation 2). One can directly apply Theorem 1, just by noticing that $DAE(\mu_b)$ is indeed a linear homogeneous implicit function $F(\vec{W}, \vec{Z})$, where $\vec{W} := \dot{\vec{X}} \cdot \vec{Y}$, $\vec{Z} := \vec{U} \cdot \vec{X}$, $\vec{A} := \vec{M} \cdot \vec{O}$, and $\vec{B} := \vec{P} \cdot \vec{N}$. If the first condition of Theorem 1 holds for all the columns $\vec{b}_i$ of the concatenated coefficient matrix $\vec{B} := \vec{P} \cdot \vec{N}$, then μ_b is consistent, while if the second condition holds for all $\dot{x} \in \dot{X}$, then μ_b is deterministic. Then, if both conditions hold, the mode μ_b is valid.

b) Validation: Our goal is to check the validity of the network avoiding the universal quantification on the state and input variables introduced in the formulas in Section IV-A. We achieve this by directly checking the conditions of Theorem 1. The consistency condition (1) of the Theorem 1 is encoded as:

$$\psi_{con}(B) := \bigwedge_{z_i \in U \cup X} \exists \dot{X}, R. \left(\psi_{\text{DAE}} \left[\delta_{z_i}^{\vec{U} \cdot \vec{X}} / \vec{U} \cdot \vec{X} \right] \right)$$

where $\delta_{z_i}^{\vec{U} \cdot \vec{X}}$ represents the vector of size $|\vec{U} \cdot \vec{X}|$, whose elements are identically zero except for the one corresponding to z_i. The formula $\psi_{con}(B)$ represents all the consistent modes. The determinicity condition (2) is encoded in the formula:

$$\psi_{det}(B) := \neg \exists \dot{X}, R. \left(\psi_{\text{DAE}} \left[\vec{0} / \vec{U} \right] \left[\vec{0} / \vec{X} \right] \wedge \left(\dot{\vec{X}} \neq \vec{0} \right) \right)$$

The formula $\psi_{det}(B)$ represents all the deterministic modes of $\mathcal{N}$. We notice that the effect on ψ_{DAE} of the X and U substitutions is equivalent to symbolically "turning on and off" a subset of the columns of the coefficient matrix $\vec{B} := \vec{P} \cdot \vec{N}$ in order to symbolically check the conditions of the Theorem 1.

Lemma 3: A network $\mathcal{N}$ is consistent iff for all $\mu_b \in 2^B$, $\mu_b \models \psi_{con}(B)$, and is deterministic iff for all the modes $\mu_b \in 2^B$, $\mu_b \models \psi_{det}(B)$ ■

c) Reformulation: The algorithm PERVARIABLEREF (Fig. 3) synthesizes the formulas $\psi_{\dot{X}}$ and ψ_Y used in the reformulated automaton $H^r_{\mathcal{N}}$, by using Theorem 1 and Lemma 1.

In the algorithm, we use the SMT solver primitives *push*, *assert*, *isSat*, *pop*, *reset* (see e.g. [12]), *getModel*, to get a satisfying assignment to all the free variables of the formula, and *quantify* to eliminate the quantifiers from the formula.

PERVARIABLEREF invokes the REFORM procedure (Fig. 5) on each variable $\dot{x} \in \dot{X}$ (Line 3), computing the reformulation $Ref_{\dot{x}}$ of the variable $\dot{x}$ and the formula $\psi_{Y,\dot{x}}$. In the algorithm, we compute ψ_Y by directly substituting in ψ_{DAE} the variables $\dot{X}$ with their reformulated value. Since the reformulation of a variable $\dot{x} \in \dot{X}$ depends on the discrete modes, we store this value in a variable $\dot{x}_s$ (we add a the set of variables $\dot{X}_s = \{\dot{x}_s \mid \dot{x} \in \dot{X}\}$). $\psi_{Y,\dot{x}}$ represents the values that $\dot{X}_s$ takes depending on the discrete state of the network. At Line 6, the algorithm constructs ψ_Y, that encodes the reformulation values for $\dot{X}_s$ and the ψ_{DAE} formula where all the $\dot{X}$ variables have been substituted with the $\dot{X}_s$ variables.

REFORM works under the *validity* assumption, that ensures the existence of a reformulation, and uses the linearity Lemma 1 to synthesize the reformulation. According to Lemma 1, we know that, for a mode $\mu_b \in 2^B$ and a variable $\dot{x}$, the function $f_{\dot{x}}(\vec{U} \cdot \vec{X})$ such that $\dot{x} = f_{\dot{x}}(\vec{U} \cdot \vec{X})$ is defined as $f_{\dot{x}}(\vec{U} \cdot \vec{X}) := \bar{\vec{W}}^j_{p_1} z_1 + ... + \bar{\vec{W}}^j_{p_n} z_n$, where j is the index corresponding to the variable $\dot{x}$ in the vector $\vec{W} := \dot{\vec{X}} \cdot \vec{Y}$, $\bar{\vec{W}}^j_{p_i}$ is the element corresponding to $\dot{x}$ in the i-th particular solution $\bar{\vec{W}}_{p_i}$. Thus, we can synthesize the coefficients of the function $f_{\dot{x}}(\vec{U} \cdot \vec{X})$ by computing all the n *particular* solutions of the system and taking their j-th element. Fig. 5 shows the reformulation procedure for a single variable $\dot{x}$: each execution of the loop at Line 4 finds a mode $\mu_b \in 2^B$ (Line 5) for which the $\dot{x}$ reformulation is still unknown. Then (Line 6) the algorithm computes the coefficients D of the $\dot{x}$ reformulation in μ_b. The procedure computes (Line 7) the cluster β of **all the modes** that share the same coefficients D, and hence the same reformulation, for $\dot{x}$. At Line 8, we prune the search space removing β. Eq is created (Line 9) by computing the product of the coefficients row vector D and the variables column vector $\vec{U} \cdot \vec{X}$. At Line 10, we accumulate the reformulation (one for each cluster) in the returned formula $Ref_{\dot{x}}$. At Line 11, we construct $\psi_{Y,\dot{x}}$ that constraints the values of the additional variable $\dot{x}_s$. REFORM terminates when the reformulation of $\dot{x}$ is known for all the modes $\mu_b \in 2^B$.

GETCOEFF is shown in Fig. 6. For each variable $z_i \in U \cup X$, the condition built at Line 4 reduces the term $\vec{B}(\vec{U} \cdot \vec{X})$ of the ψ_{DAE} formula to the column vector $\vec{b}_i z_i = \vec{b}_i 1 = \vec{b}_i$ that corresponds to the i-th iteration. This formula is asserted in the solver at Line 5. At Line 6, the algorithm finds a *particular* solution μ' to the system $\vec{A}\vec{W} = -\vec{b}_i$. Then (Line 7) we assign the value $\mu'(\dot{x})$ of the $\dot{x}$ element of the solution μ' to the i-th reformulation coefficient $D[i]$.

The procedure GETEQMOD (Fig. 4) builds the condition γ that is satisfiable in every $\mu_b \in 2^B$ that shares the same reformulation coefficients for $\dot{x}$. In Line 7, we symbolically compute the set of equivalent modes β.

Theorem 2 (Correctness of the reformulation): Given a valid network $\mathcal{N}$, the hybrid automaton $H^r_{\mathcal{N}}$ is equivalent to the hybrid automaton $H_{\mathcal{N}}$ that defines the network semantics. ■

V. RELATED WORK

Multi-Domain Linear Kirchhoff Networks are widely used in various engineering applications [13], [14], [15]. Different tools support the acausal modeling phase [16], [17], also for networks with discrete switches. The main analysis tools are based on numerical simulation and use numerical integration. Although simulation provides high scalability and enables the analysis of complex dynamics [6], [18], [19], a preliminary validation of the network modes is not provided. Therefore, a hidden inconsistent mode can be discovered *only* if the user designs a simulation trace that is able to reach it. Furthermore, numerical simulators (e.g. [17]) restrict the use of components

```
PerVariableRef ($\psi_{\text{DAE}}$, $X$, $U$):
1. $(\psi_{\dot{X}}, \psi_Y) := (True, True)$
2. for each $\dot{x} \in \dot{X}$:
3.     $(Ref_{\dot{x}}, \psi_{Y,\dot{x}})$ := Reform ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$)
4.     $\psi_{\dot{X}} := \psi_{\dot{X}} \wedge Ref_{\dot{x}}$
5.     $\psi_Y := \psi_Y \wedge \psi_{Y,\dot{x}}$
6. $\psi_Y := \psi_Y \wedge \psi_{\text{DAE}}[\dot{X}_s/\dot{X}]$
7. return $(\psi_{\dot{X}}, \psi_Y)$
```

Fig. 3. Reformulation algorithm for $\mathcal{N}$.

```
GetEqMod ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$, $D$):
1. eqSolver.reset()
2. $\gamma := True$
3. for each $z_i \in U \cup X$:
4.     $rhs_{z_i} := z_i = 1 \wedge \bigwedge_{l \in (U \cup X) \setminus \{z_i\}} l = 0$
5.     $\gamma_{z_i} := \dot{x} = D[i] \wedge rhs_{z_i}$
6.     $\gamma := \gamma \wedge \exists R, \dot{X}.(\psi_{\text{DAE}} \wedge \gamma_{z_i})$
7. $\beta$ := eqSolver.quantify($\gamma$)
8. return $\beta$
```

Fig. 4. Find the cluster of modes that share the same coefficients D for $\dot{x}$.

```
Reform ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$):
1. $Ref_{\dot{x}} := True$
2. $\psi_{Y,\dot{x}} := True$
3. solver.assert($True$)
4. while solver.isSat():
       # Get a fresh mode
5.     $\mu_b$ := solver.getModel()
       # Get the row vector of coeff. that
       # contributes to $\dot{x}$ in $\mu_b$
6.     $D$ := GetCoeff ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$, $\mu_b$)
       # Get the cluster of modes that share the
       # same coeff.
7.     $\beta$ := GetEqMod ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$, $D$)
       # Prune the cluster of modes from the search
8.     solver.assert($\neg\beta$)
       # Build the reformulation equation
9.     $Eq := \dot{x} = D\ (\vec{U} \cdot \vec{X})$
10.    $Ref_{\dot{x}} := Ref_{\dot{x}} \wedge (\beta \rightarrow Eq)$
11.    $\psi_{Y,\dot{x}} := \psi_{Y,\dot{x}} \wedge \beta \rightarrow \dot{x}_s = D(\vec{U} \cdot \vec{X})$
12. return $(Ref_{\dot{x}}, \psi_{Y,\dot{x}})$
```

Fig. 5. Reformulation of a single variable $\dot{x}$.

```
GetCoeff ($\psi_{\text{DAE}}$, $X$, $U$, $\dot{x}$, $\mu_b$):
   # $D$ row vector of coeff. w.r.t. $U \cup X$
1. coeffSolver.assert($\psi_{\text{DAE}} \wedge \mu_b$)
2. for each $z_i \in U \cup X$:
3.     coeffSolver.push()
       # build the rhs corresponding to $z_i$
4.     $rhs_{z_i} := z_i = 1 \wedge \bigwedge_{l \in (U \cup X) \setminus \{z_i\}} l = 0$
5.     coeffSolver.assert($rhs_{z_i}$)
       # get a system solution
6.     $\mu'$ := coeffSolver.getModel()
       # $\mu'(\dot{x})$ is the coeff. w.r.t. $z_i$
7.     $D[i] := \mu'(\dot{x})$
8.     coeffSolver.pop()
9. return $D$
```

Fig. 6. Computes the ref. coefficients D of $\dot{x}$.

equipped with ideal behaviors, leading to the model pollution due to parasitic effects, that are hard to quantify and deviate the simulation results from the intended nominal behavior. In the following, we focus on works based on formal methods.

The closest related work is [20], that presents a method to convert Switched *Electrical* Linear Kirchhoff Networks (SELKN) into hybrid automata. The work proposed here is more general than [20] in three respects. First, we are able to deal with *multi-domain* networks, enabling mechanical, electrical and hydraulic domains, and their combination, whilst [20] is restricted to electrical networks. Second, the method in [20] is only able to produce a hybrid automaton if the electrical network fulfills the conditions of existence and determinism in *all the modes* and for *all the variables*, while here we analyze SMDLKN with *non-deterministic* output variables as well. Both extensions are made possible by the adoption of a theoretical settings that is significantly more general than the domain-specific topological approach on the network graph used in [20]. We remark that all the experiments presented in the next section are based on benchmarks that are out of reach for the method in [20]. In [21], a framework for generating hybrid automata benchmarks from a hydraulic domain is presented. This work is only seemingly related to ours. The domain knowledge in [21] (e.g. that a pump cannot draw a constant flow from an empty tank) appears to be hard-coded in the generation scripts; in our case, the detection of these conditions and the generation of the hybrid automata are direct consequence of the algebraic approach applied to the network description. As discussed in the experimental evaluation, our approach is able to deal with a significantly larger class of benchmarks than those in [21], and also to automatically identify invalid modes in the network, reasoning on its algebraic properties.

Most of the formal verification tools are unable to deal with DAE. An exception is KeymaeraX [22], a theorem prover for hybrid systems represented with Differential-Algebraic Equations. In principle, the KeymaeraX proof system can support the proof of safety properties over SMDLKN, by means of compositional reasoning. Key differences with our approach are that KeymaeraX is not fully automatic, and has no specific methods to address the validation problem.

The existing tools for formal verification of hybrid systems [23] do not directly consider Multi-Domain Linear Kirchhoff Network, but work on hybrid automata [10]. Tools like *SpaceEx* [3] or *Flow** [24] work on an *explicit* representation of the system and hence they suffer from the explosion in the number of modes of the system. Other tools [25], [5], [26], [27] reason on the symbolic representation of the system. HybridSAL [25] and HyComp[5] analyze linear hybrid systems whose continuous dynamics is specified with a linear ODE. dReach [26], [27] can be used to either perform Bounded Model Checking or apply induction to verify a system expressed with ODEs. From a DAE-based network, our reformulation step produces this kind of formal models.

Other verification techniques focus on analog-mixed-signals circuits [28], [29], [30], [31], [32]. They take the hybrid automata representation of the electrical circuit, so do not face the validation and reformulation problems. Additionally, they do not consider multi-domain networks and perform an analysis explicit in the modes that might exponentially blow-up.

Other approaches exist to generate a formal representation from Simulink and other causal component-based modeling languages [33], [34]. This causal semantics considers systems represented as a connection of input-output functional blocks, posing a major obstacle to the modeling of SMDLKN. Our work differs from those approaches since we natively accept the more suitable *acausal* component-based modeling, that, on the other side, requires to tackle the reformulation problem.

VI. Experimental evaluation

Setup: We implemented the proposed approach using the pySMT [35] library and the MathSAT5 [12] SMT-solver. At the core, we use the symbolic model checker HyCOMP [5]. The resulting workflow takes as input a SMDLKN and a safety property, and performs *validation* (Val), *reformulation* (Ref), and *verification* (Ver). The validation and the reformulation come with two variants, basic (Bas) and optimized (Opt). Bas refers to the algorithms of Section IV-A, while Opt refers to

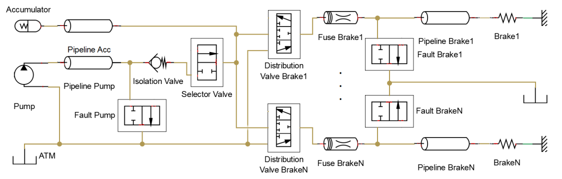


Fig. 7. Wheel Braking System, Arch.2, with N braking lines (WBSA2$_{[N]}$)

those of Section IV-B. We run the experiments on a 3.5 GHz cpu with 16GB RAM, with time out (TO) set to 3600s for VAL, 43200s for REF, and 18000s for VER. The tools and the benchmarks are available at http://es.fbk.eu/people/sessa/attachment/fmcad17/fmcad17.tar.bz2.

Benchmarks: We consider five scalable benchmarks: two variants of the Wheel Braking System (WBS) from the SAE standard AIR6110 [36], the Landing Gear System (LGS), and two variants of a hydraulic tank network (WW) inspired from [21], for a total of 29 instances. The WBS benchmarks, WBSA2$_{[N]}$ and WBSA4$_{[N]}$, are parameterized w.r.t. the number N of braking lines (see Fig. 7). The benchmarks differ in the position of the hydraulic accumulator line. Fig. 1 shows (part of) the Landing Gear System (LGS$_{[N]}$) from [1], which is parameterized w.r.t. the number N of cylinder lines. The WWLIN$_{[N]}$ and WWRING$_{[N]}$ benchmarks represent networks of N hydraulic tanks connected either linearly or in a ring through channels, composed by pipes and valves. The WW benchmarks are originally proposed in [21], with a hand-crafted technique meant for the automatic generation of hybrid benchmarks that abstracts away the mutual interactions among the liquid levels stored in the tanks. On the contrary, our work aims at faithfully representing the physics of the real system. Our SMDLKN-based models capture the physical dynamics of the (bidirectional) flow through the channels, and naturally represents the global interaction of the interconnected components, retaining the compositional structure of the physical system.

The features of the benchmarks are described in the extended version of the paper. The models contain tens of boolean variables and hundreds of real variables, resulting in up to 2 millions of modes. None of the benchmarks considered in this evaluation can be analyzed with the approach presented in [20]. There are several reasons for this. First, all the benchmarks are out of the electrical domain. Even if [20] deals with some simple hydraulic models by means of the hydraulic-electrical analogy, the cylinder component used in the LGS does not fit in the domain analogy. Second, [20] cannot deal with non-deterministic output variables. Our WBS benchmarks yield under-specified output variables that were not present in the much simpler and less complete model used in [20]. Finally, the WW benchmarks contain some inconsistent modes, and the method in [20] requires consistency for all the modes.

Also, note that our modeling of the WBS benchmarks is different than the model presented in [37], which is an abstract, discretized and causal model of the system suitable to perform a formal system safety assessment analysis. Instead, in our WBS model we capture the real continuous physics of the system.

Validation: The results of the evaluation are summarized in Tab. I. First, we consider the runtime of the basic (BAS) and the optimized (OPT) encodings for validation. We see that OPT solves all the 29 instances, while BAS times out on the 10 biggest instances. Focusing on the instances solved by both encodings, OPT outperforms BAS by two orders of magnitude and scales much better w.r.t the benchmark size. Noteworthy, the OPT method validates the two millions of modes of the WBS$_{[5]}$ instances within 327 and 252 seconds, respectively. These results provide a clear evidence that the BAS encodings is infeasible for real life systems, while OPT offers an efficient solution to solve the problem.

All the WBS and LGS benchmarks have only consistent modes. This does not hold for the WW benchmarks, where a tank cannot accept incoming [respectively, provide outgoing] liquid in mode full [resp., empty]. Notice that the full and empty modes can be seen as hazardous configurations of the network, when an actuator must pump in/out a fluid. Our validation approach is able to detect and report such bad configurations, and allows us to generate models under the assumption that the invalid modes are not entered (e.g. by the preventive action of a supervisory controller).

	VAL		REF		VER
	BAS	OPT	BAS	OPT	OPT
LGS$_{[2]}$	1	0	187	1	1
LGS$_{[3]}$	5	1	TO	5	1
LGS$_{[4]}$	29	3	TO	21	1
LGS$_{[5]}$	204	9	TO	90	7
LGS$_{[6]}$	1567	25	TO	449	9
LGS$_{[7]}$	TO	73	TO	3091	14
LGS$_{[8]}$	TO	215	TO	30269	30
WBSA2$_{[2]}$	10	0	TO	3	0
WBSA2$_{[3]}$	395	5	TO	19	4
WBSA2$_{[4]}$	TO	37	TO	204	74
WBSA2$_{[5]}$	TO	327	TO	5554	2630
WBSA4$_{[2]}$	9	0	TO	3	0
WBSA4$_{[3]}$	360	4	TO	22	5
WBSA4$_{[4]}$	TO	30	TO	223	131
WBSA4$_{[5]}$	TO	252	TO	5892	10970
WWLIN$_{[2]}$	0	0	8	0	0
WWLIN$_{[3]}$	0	0	1072	1	0
WWLIN$_{[4]}$	2	0	TO	3	2
WWLIN$_{[5]}$	21	0	TO	8	5
WWLIN$_{[6]}$	166	1	TO	19	33
WWLIN$_{[7]}$	1670	3	TO	53	62
WWLIN$_{[8]}$	TO	5	TO	419	343
WWRING$_{[2]}$	0	0	39	0	0
WWRING$_{[3]}$	4	0	TO	3	1
WWRING$_{[4]}$	74	1	TO	10	6
WWRING$_{[5]}$	1300	3	TO	30	28
WWRING$_{[6]}$	TO	7	TO	89	78
WWRING$_{[7]}$	TO	15	TO	369	848
WWRING$_{[8]}$	TO	27	TO	2465	MO

TABLE I
VALIDATION, REFORMULATION AND VERIFICATION TIME [S].

Reformulation: We consider the runtime for the BAS reformulation lower bound, and the OPT reformulation. The BAS encodings cannot deal with the benchmarks, whereas the OPT encodings successes in reformulating all the instances. Again, this happens because the OPT encodings exploits the properties of the algebraic structure of the problem to mitigate the computational complexity of the quantifier elimination in the computation of the derivative variables reformulation. Additionally, the variable substitution of the first derivative reformulation into the network DAE formula completely avoids the need for the quantifier elimination step in the reformulation of the output variables.

We notice that the reformulation of the WW benchmarks is restricted to the *valid* modes, while considering the *non-valid* modes as a macro error state of the network. The ability of representing these non-valid modes in the reformulated hybrid automaton is crucial when considering the functional verification of the network composed with a controller designed to prevent the reachability of hazardous configurations.

Verification: For both WBS benchmarks we consider the property P_1: *when the selector valve is closed, a brake command cannot actuate any brake*. Consistently with the SAE standard AIR6110 [36], that describes such design flaw, P_1 is violated for WBSA2$_{[N]}$ and is verified by WBSA4$_{[N]}$. For the LGS, we consider the (false) property L_1: *the first cylinder cannot reach its end-of-stroke*. For both WW benchmarks, we consider the (false) property W_1: *the level of the first tank cannot exceed a given threshold*, that is violated closing all the valves connected to the first tank.

The verification on the hybrid automata from the OPT reformulation completes within the time out on all the benchmarks, returning the expected results, except for WWRING$_{[8]}$ that experienced a memory out (MO). Finding the violation in WBSA2$_{[N]}$ is slightly faster than proving the property in WBSA4$_{[N]}$. Overall, these results provide empirical evidence of the applicability of our approach in the formal verification of real world hybrid system represented as a SMDLKN.

VII. Conclusion

We presented an SMT-based method for the formal analysis of Switching Multi-Domain Linear Kirchhoff Networks (SMDLKN), that is able to automatically validate and reformulate a SMDLKN into a symbolic Hybrid Automaton, amenable to be formally verified with the existing model checkers. The approach covers networks spanning multiple physical domains and exhibiting non-deterministic behaviors, achieving substantial improvements over a pure SMT-based approach by leveraging general results in linear algebra. We implemented and evaluated the SMT-based procedures to validate and reformulate the network, demonstrating the potential of complete verification workflow on real-world systems.

We plan to extend the approach to incorporate networks with discontinuous state variables [38], produce a network of HA instead of a monolithic HA, and extend the analysis towards the safety assessment for the generation of Fault Trees.

References

[1] F. Boniol and V. Wiels, *The Landing Gear System Case Study*. Cham: Springer International Publishing, 2014.

[2] K. Janschek, *Mechatronic systems design: methods, models, concepts*. Springer Science & Business Media, 2011.

[3] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable Verification of Hybrid Systems," in *CAV*, 2011, pp. 379–395.

[4] S. Gao, S. Kong, and E. M. Clarke, "dreal: An SMT solver for nonlinear theories over the reals," in *CADE-24*, 2013.

[5] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "HyCOMP: An SMT-based model checker for hybrid systems," in *TACAS*, 2015.

[6] R. Riaza, *Differential-algebraic systems analytical aspects and circuit applications*. Singapore, SG: World Scientific, 2008.

[7] J. R. Munkres, *Analysis on manifolds*. Westview Press, 1997.

[8] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009.

[9] S. J. Axler, *Linear Algebra Done Right*, ser. Undergraduate Texts in Mathematics. New York: Springer, 1997.

[10] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, 1996*, 1996.

[11] A. Cimatti, S. Mover, and S. Tonetta, "A quantifier-free SMT encoding of non-linear hybrid automata," in *FMCAD*, 2012, pp. 187–195.

[12] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *TACAS*. Springer Berlin Heidelberg, 2013.

[13] H. Yoshikawa, T. Oda, K. Nonaka, and K. Sekiguchi, "Modeling and simulation for leg-wheel mobile robots using modelica," in *The First Japanese Modelica Conferences*, no. 124, 2016.

[14] H. Elmqvist, M. Otter, and C. Schlegel, "Physical modeling with modelica and dymola and real-time simulation with simulink and real time workshop," in *Matlab User Conference*, 1997.

[15] A. M. Krishna, Arash Dizqah and B. P. Fritzson, "Modeling and simulation of a combined solar and wind systems using openmodelica," 2013.

[16] P. Fritzson, *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, 2014.

[17] T. Mathworks, "Simscape power systems." [Online]. Available: http://it.mathworks.com/help/physmod/sps/index.html

[18] P. Benner, *Large-scale networks in engineering and life sciences*. Springer, Birkhäuser Mathematics, 2014.

[19] D. L. Skaar, "Using the superposition method to formulate the state variable matrix for linear networks," *IEEE Transactions on Education*, vol. 44, no. 4, Nov 2001.

[20] A. Cimatti, S. Mover, and M. Sessa, "From electrical switched networks to hybrid automata," in *FM 2016*, vol. 9995, 2016.

[21] S. Bak, S. Bogomolov, M. Greitschus, and T. T. Johnson, "Benchmark generator for stratified controllers of tank networks," in *ARCH*, 2015.

[22] N. Fulton, S. Mitsch, J. Quesel, M. Völp, and A. Platzer, "Keymaera X: an axiomatic tactical theorem prover for hybrid systems," in *CADE*, 2015.

[23] R. Alur, "Formal verification of hybrid systems," in *EMSOFT 2011*, 2011.

[24] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *CAV*. Springer, 2013.

[25] A. Tiwari, "HybridSAL Relational Abstracter," in *CAV*, 2012.

[26] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-reachability analysis for hybrid systems," in *TACAS*, 2015.

[27] K. Bae, S. Kong, and S. Gao, "SMT encoding of hybrid systems in dReal," in *ARCH14-15*. EasyChair, 2015.

[28] M. H. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs: Survey and comparison," in *2006 IEEE North-East Workshop on Circuits and Systems*, June 2006, pp. 281–284.

[29] T. Dang, A. Donzé, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *FMCAD 2004*, 2004.

[30] G. Frehse, B. H. Krogh, R. A. Rutenbar, and O. Maler, "Time domain verification of oscillator circuit properties," *Electr. Notes Theor. Comput. Sci.*, vol. 153, no. 3, pp. 9–22, 2006.

[31] H. L. Lee, M. Althoff, S. Hoelldampf, M. Olbrich, and E. Barke, "Automated generation of hybrid system models for reachability analysis of nonlinear analog circuits," in *ASP-DAC 2015*, 2015.

[32] Y. Zhang, S. Sankaranarayanan, and F. Somenzi, "Piecewise linear modeling of nonlinear devices for formal verification of analog circuits," in *FMCAD*, 2012, pp. 196–203.

[33] K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo, "A step towards verification and synthesis from simulink/stateflow models," in *HSCC 2011*, 2011, pp. 317–318.

[34] S. Minopoli and G. Frehse, "SL2SX Translator: From Simulink to SpaceEx Models," in *HSCC 2016*, 2016, pp. 93–98.

[35] M. Gario and A. Micheli, "pySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms," in *SMT Workshop*, 2015.

[36] SAE International, "AIR 6110 - Contiguous Aircraft/System Development Process Example," 2011.

[37] M. Bozzano and et al., "Formal design and safety analysis of AIR6110 wheel brake system," in *CAV*, 2015.

[38] A. Massarini and et al., "Analysis of networks with ideal switches by state equations," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 1997.

Automatic Verification of Application-Tailored OSEK Kernels

Hans-Peter Deifel, Merlin Göttlinger,
Stefan Milius and Lutz Schröder
Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg
Email: {hans-peter.deifel, merlin.goettlinger, stefan.milius, lutz.schroeder}@fau.de

Christian Dietrich and Daniel Lohmann
Leibniz Universität Hannover
Email: {dietrich, lohmann}@sra.uni-hannover.de

***Abstract*—The OSEK industrial standard governs the design of embedded real-time operating systems in the automotive domain. We report on efforts to develop verification methods for OSEK-conformant compilers, specifically of a code generator that weaves system calls and application code using a static configuration file, producing a stand-alone application that incorporates the relevant parts of the kernel. Our methodology involves two verification steps: On the one hand, we extract an OS–application interaction graph during the compilation phase and verify that it conforms to the standard, in particular regarding prioritized scheduling and interrupt handling. To this end, we generate from the configuration file a temporal specification of standard-conformant behaviour and model check the arising formulas on a labelled transition system extracted from the interaction graph. On the other hand, we verify that the actual generated code conforms to the interaction graph; this is done by graph isomorphism checking of the interaction graph against a dynamically-explored state-transition graph of the generated system.**

1. Introduction

Embedded real-time control systems are special-purpose systems dedicated to specific, predefined tasks [1], [2]. Already now, a typical (in particular, non-autonomous) car contains up to a hundred such systems. Hence, both the hardware and the system software of each embedded control system need to be tailored to its specific needs in order to keep per-unit hardware costs as low as possible [3]. The OSEK-OS standard [4] fulfils these demands for tailorability and has been (together with its superset AUTOSAR-OS [5]) the dominant industry standard for event-triggered automotive *real-time operating systems* (RTOSs) for the last decades. What sets OSEK apart from the common POSIX-like operating systems is that it is completely statically configured. For a specific automotive application, all system objects (tasks, interrupt-service routines, resources, etc.) and their configurations have to be declared at compile-time in a domain-specific language, the *OSEK Implementation Language* (OIL) [6]. From this specification, an application-specific, highly optimized RTOS instance is derived by means of a generator.

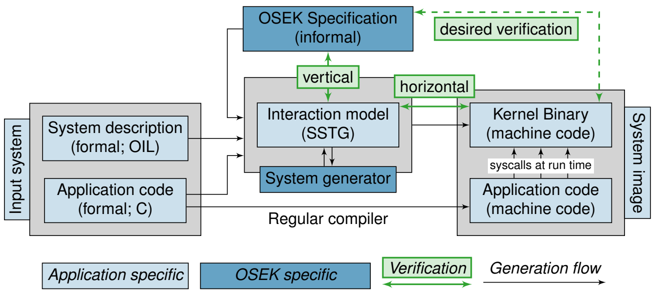


Figure 1: System generation and verification. We use the *static state-transition graph (SSTG)*, a byproduct of the OSEK system generator, as an intermediate representation for our desired kernel verification.

However, with the advent of autonomous driving features, the industry is facing new challenges with respect to functional safety; the highest safety level ISO 26262 ASIL D demands the employment of a certified RTOS, such as *RTA-OS* (ETAS), *MICROSAR OS* (Vector), or *tresos Safety OS* (EB) (vendors named in parentheses). These certified operating systems offer significantly less tailorability and thus induce higher per-unit costs. The certification of the development process of the RTOS kernel is already extremely expensive, so vendors shy away from the even higher costs of certifying a kernel generator.

Taking a step back, we argue that application developers do not need a kernel that behaves correctly in all imaginable situations. However, they are very interested in a kernel that *always* behaves correctly for their specific application and all of their kernel-usage patterns. Therefore, we replace the isolated certification of the generator with a per-instance verification of the resulting kernel binary and thus formulate our verification goal: For a given application, the generated kernel binary must expose the specified behavior when executed together with our application. This bypass of the generator in the verification process allows all kinds of highly specialized system optimizations.

We achieve the desired verification (see Figure 1) by introducing an kernel–application interaction model: Our toolchain considers not only the OIL-specified system object instances but also how these system objects actually interact with each other via the syscall interface according to the

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

```
TASK(Low) {
  if (test()) {
    ActivateTask(High);
  }
  TerminateTask();
}

TASK(Med) {
  TerminateTask();
}

TASK(High) {
  TerminateTask();
}

void ISR() {
  ActivateTask(Med);
}
```

Figure 2: Example System – Source Code

OSEK semantics [7]. We enumerate the application-specific kernel's state space, the *static state-transition graph* (SSTG), which is calculated and used by the system generator. The SSTG acts as an intermediate representation of kernel behavior and is the central data structure for our strategy. First, we statically verify (*vertically*, according to Figure 1) that the SSTG actually conforms to the OSEK standard. To this end, we formalize key aspects of the standard in CTL and model check the SSTG against this specification; this is feasible because the SSTG is of moderate size thanks to tailoring. Second, we dynamically verify the system against the SSTG (*horizontally*, according to Figure 1). To this end, we probe the system to explore a *dynamic state-transition graph* (DSTG) and check that it is isomorphic to the SSTG. We report on experiments with this methodology, both on systems from a standard test suite and on the control software of a quadrotor copter.

2. Background and Context

We give a brief overview of the single-core OSEK real-time operating system standard. Moreover, we describe the *static state-transition graph* (SSTG), which the dOSEK generator (dOSEK = dependable OSEK, our implementation of the OSEK standard) uses as an intermediate representation to model all possible interactions between application and kernel.

2.1. OSEK in a Nutshell. The tailored embedded systems that we are concerned with here are woven from application code and a tailored kernel instance. Kernel and application interact at runtime, typically subject to requirements on real-time performance. Depending on the current OS state, the kernel selects a control flow that is currently ready and dispatches it for execution. The application's control flows, as the kernel's counterpart, manipulate the OS state by invoking *system services* that influence the system behaviour (Table 1 gives a short overview). OSEK offers two main control flow abstractions: *interrupt-service routines* (ISRs) and *tasks* (i.e. threads). ISRs are activated by the hardware and fall into two classes: *category-1* ISRs, which are not allowed to call system services; and *category-2* ISRs, which are synchronized with the kernel. Tasks have a statically assigned priority, are allowed to use all system services, and are invoked according to a strict fixed-priority preemptive scheduling policy. On each new activation, tasks start from the very beginning and run until (self-)termination. Each task is configured to be either nonpreemptive (enforcing run-to-completion semantics) or fully preemptive. Preemption points can be either *synchronous*, for example caused by an explicit activation of a higher priority task (`ActivateTask`), or *asynchronous*, if a higher priority task is activated inside an ISR. Periodically or aperiodically recurring task activations can be triggered by means of statically configured *alarms*, which are driven by a hardware timer.

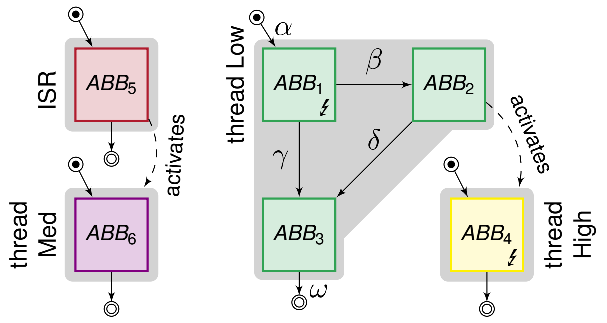
Inter-task synchronization is realized by *resource* objects. Based on a *stack-based priority-ceiling protocol*, OSEK resources ensure mutual exclusion while preventing deadlocks and priority inversion. Through the acquisition of a resource, a task raises its *dynamic* priority to the *ceiling* priority of the resource – the highest static priority of all tasks that can obtain the resource, according to the OIL file.

Figure 2 shows a small example system that consists of three tasks and one ISR. These coordinate their execution with the help of the OS, which is activated through system service invocations (syscalls). Figure 3a depicts the same system as read in by the dOSEK generator.

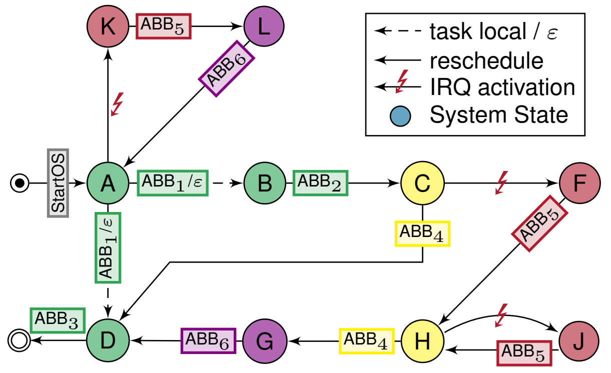
2.2. Generating a System. A dOSEK kernel instance is generated from two inputs (see also Figure 1): The application's OIL file specifies the employed RTOS objects (i.e., the tasks *Low*, *Med*, *High* and the ISR *ISR* in our example). The application's source code (Figure 2) specifies how these system objects interact according to the OSEK semantics.

Internally, we structure the application code into a set of *control-flow graphs* (CFGs) consisting of *atomic basic blocks* (ABBs) (Figure 3a). An ABB [7], [8] is a control-flow superstructure that subsumes one or more traditional *basic blocks* (BBs) forming a single-entry single-exit region; it has *exactly one* distinguished entry BB and one exit BB. The construction (see [7] for a detailed description) results in one ABB-graph for each task within the application code. By construction, every ABB either contains a single syscall or only computation code that does not interact with the OS (no syscalls). From the kernel's point of view, an ABB executes atomically, but can be interrupted by ISRs.

At build time, the dOSEK generator computes a *state transition graph* (STG) from the ABB graphs of the individual tasks and the system configuration (OIL). (We formally define STGs in Section 2.3.) This STG is the *static state-transition graph* (SSTG) already mentioned in the introduction. Starting with an initial global system state, derived from the OIL file, the generator enumerates all reachable system states explicitly (Figure 3b). Every state carries the currently running task, a block that is executed in this state (e.g. state A executes ABB_1), and other relevant scheduling data, like the list of activated tasks. A state transition is caused by either a computation block, a system call block, or an interrupt request. For the associated post-state, the currently running task and the currently executed block are calculated according to the OSEK scheduling semantics. The SSTG subsumes the interwoven application–kernel behaviour and includes all possible scheduling sequences an OSEK kernel exposes for the given application. Since *computation* blocks do not perform syscalls, their execution does not influence future scheduling decisions, and is therefore represented



(a) The application is structured into three tasks with high, medium, and low priority, respectively, and one interrupt service routine. The application code is partitioned into *atomic basic blocks* (ABBs). In ABB_2, the low-priority task activates the high-priority task via `ActivateTask`. The ISR activates the medium-priority task. (To simplify the example, the IRQ may happen only in ABB_1 and ABB_4, and the tasks issue `TerminateTask()` implicitly.)



(b) This *static state-transition graph* (SSTG) captures all possible state transitions from the start to the termination of the low-priority task according to the OSEK semantics. Every state carries and executes exactly one ABB.

Figure 3: Example system

TABLE 1: Excerpt of system services provided by the OSEK-OS API.

System Service	Arguments	Brief Description
`ActivateTask`	`TaskID`	Task `TaskID` is activated. If the current task is preemptible, immediate rescheduling takes place.
`TerminateTask`	–	The current task terminates itself and immediate rescheduling takes place.
`GetResource`	`ResID`	Acquires the resource identified by `ResID`.
`ReleaseResource`	`ResID`	Leaves the critical region associated with the resource `ResID`. The dynamic priority of the calling task is changed and a reschedule takes place for preemptible tasks.

by ε-transitions. For instance, ABB_1 is computational and, therefore, the transition between states A and D becomes an ε-transition.

Thus computed, the SSTG has node identities representing the current global system state, while the edges of the SSTG are either ε-transitions or labelled with the system calls triggering the respective state transitions. System call labels can be either system calls in the proper sense, interrupts (which are triggered by the hardware outside the system), interrupt returns (irets), or the idle system call. The idle system call is executed from an idle state when no other task is ready for execution. This ensures that all maximal paths in the SSTG are infinite. The latter three types of system call labels are artificially added by the generator and only model the implicit state transitions of an OSEK system; they are not explicitly named in the OSEK specification.

From the SSTG information, the dOSEK system generator (Figure 1) produces a kernel binary that is optimized for the actual application usage patterns. Optimizations include, for example, the avoidance of scheduler invocations for system call sites that have a known scheduling outcome.

2.3. State Transition Graphs. Our verification method is concerned with properties of and relations between STGs, which in process-theoretic parlance are essentially deterministic labelled transition systems. That is, given a set $\mathcal{A}$ of *labels* a state transition graph consists of a set S of *states* and a set $T \subseteq S \times \mathcal{A} \times S$ of *labelled transitions*. We write $s \xrightarrow{a} t$ for $(s, a, t) \in T$. The transition relation is required to be *deterministic* (but may be, and typically is, *partial*), that is, whenever $s \xrightarrow{a} t$ and $s \xrightarrow{a} t'$ then $t = t'$. We occasionally consider state transition graphs with ε-*transitions*; these additionally admit transitions of the form $s \xrightarrow{\varepsilon} t$ where ε is a special label not contained in $\mathcal{A}$. For ε-transitions, we do not require determinism. For example, the graph from Figure 3b is nondeterministic at state A w.r.t. ABB_1-transitions, which are replaced by ε-transitions as explained in Section 2.2.

To STGs with ε-transitions, we apply the usual process of ε-*elimination*, i.e. we insert an a-transition from state s to state t whenever t is reachable from s by first performing any number of ε-transitions (possibly none) and then an a-transition. We then remove all unlabelled transitions, and all states that become unreachable as a result. In general, this will produce a nondeterministic STG; we will explain in Section 3.2 why the particular STGs that appear in our verification framework do remain deterministic after ε-elimination.

3. The Formal Verification Method

Our formal verification methodology comprises a *vertical* and a *horizontal* verification process (cf. Figure 1). The central data structure for our verification is the SSTG (Section 2.2). In the *vertical verification*, we check that the SSTG adheres to key aspects of the behaviour specified by the OSEK standard, in particular regarding prioritized scheduling and interrupt handling. To this end, we formalize the corresponding parts of the OSEK standard in CTL and model check the SSTG against the arising temporal specification (Section 3.1). In the horizontal verification, we then ensure that the actual generated code conforms to the originally projected system-wide control flow; this is achieved by graph isomorphism checking of the SSTG against a judicious abstraction of the code, viz. another STG called the *dynamic state-transition graph* (DSTG) (Section 3.2).

3.1. Vertical Verification. We next describe how we formally verify that the static state transition graph (SSTG) complies with the OSEK specification. To this end, we generate a NuSMV-model and CTL formulas that formalize (parts of) the OSEK specification. Our formalization currently covers the standard roughly up to conformance class ECC1, with the exception of alarms and resource management. As input for our generator we use the (ε-eliminated) SSTG and the OIL specification, which specifies all tasks and interrupts with their respective priorities, as well as all events and resources of the system.

The SSTG already resembles a NuSMV-model, except that NuSMV-models do not have edge labels. Thus, we need to convert the edge labels into state labels, and our generator does that by pushing labels along the arrows into the next state. This leads to a multiplication of states by the number of different labels of incoming edges. Of course, this conversion produces a nondeterministic model, however with a unique edge between two states as every syscall has a unique effect on the current state. In more detail, our NuSMV-model has as global state variables (a) the variable syscall that contains the name of the system call (i.e. the label of the edge) that took the system into the current state, (b) a variable state carrying the current state in form of the node id from the SSTG, and (c) variables for all other parts of the global system state. Note that the values of the latter variables are already determined uniquely by the node id; we included them merely to make the CTL formulas and counterexample traces more readable.

All OS objects, i.e. tasks, ISRs, events, and resources are realized by means of NuSMV-modules, which are instantiated as specified in the OIL specification of the system. They do not carry internal state but are merely used to group related variables for readability in formulas and error traces. For example, the state and (dynamic) priority of a task t are referred to by t.state and t.priority. The actual state is fed into the instances through global variables that form parameters of the modules. The next value of state at any state is chosen nondeterministically as one of the successor nodes of the current SSTG node (given by the current value of state). The next(syscall) is then uniquely determined by the current value of state and the value of next(state).

Some variables, for example the resource priorities, are not explicitly contained in the OIL file and thus have to be calculated according to the OSEK specification. The latter demands that resource priorities are at least the maximum priority of all tasks using the resource but are lower than the priority of every task not using the resource but having higher priority than the ones using the resource. To avoid priority collisions between tasks due to resource occupation, we scale all priorities by a factor of two and calculate the resource priorities as the maximum of all the priorities of

```
MODULE main()
VAR
  ...
  syscall : { Start, ..., TerminateTask, ...,
              ActivateTask_High, ..., interrupt_37, ... };
  running : { Idle, Low, Med, High, ISR };
  state : { ABB_67_0, ..., ABB_4_0, ..., ABB_23_0, ABB_24_0,
            ABB_25_0, ..., ABB_63_0, ... };
  ...
ASSIGN
  init(syscall) := Start;
  next(syscall) := case
    ((state = ABB_67_0) & next((state = ABB_4_0))) :
       ActivateTask_High;
    ((state = ABB_67_0) & next((state = ABB_23_0))) :
       interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_24_0))) :
       interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_25_0))) :
       interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_63_0))) :
       TerminateTask;
    ...
    next(TRUE) : syscall;
  esac;
  init(running) := Low;
  next(running) := case
    next((... | (state = ABB_4_0))) : High;
    next((... | (state = ABB_23_0) | (state = ABB_24_0) |
          (state = ABB_25_0) | ...)) : ISR;
    next((state = ABB_63_0)) : Idle;
    ...
    next(TRUE) : running;
  esac;
  init(state) := ABB_67_0;
  next(state) := case
    ...
    (state = ABB_67_0) : {ABB_4_0, ABB_23_0, ABB_24_0,
                          ABB_25_0, ABB_63_0};
    ...
  esac;
  ...
CTLSPEC ...
```

Figure 4: Slice of the NuSMV-model for the graph in Fig. 3

the tasks using the resource plus one. This does not affect the scheduling behaviour and ensures that there can never be a situation where multiple tasks of the same dynamic priority are ready to run at the same time.

Figure 4 shows a slice of the NuSMV-model generated from the SSTG for the example system from Figure 2 (depicted in simplified form in Figure 3(b)). We include only the variables `state`, `syscall` and `running` and only the transitions from the starting state `ABB_67_0`, which corresponds to state A. The transition to `ABB_4_0` corresponds to the transition $\mathsf{A} \xrightarrow{\varepsilon} \mathsf{B} \xrightarrow{\mathsf{ABB\_2}} \mathsf{C}$ and the one to `ABB_63_0` corresponds to the transition to the final state with label ABB_3. The remaining three transitions essentially correspond to $\mathsf{A} \to \mathsf{K}$ but take into account that in reality an interrupt in the program of Figure 2 may happen before `test()` is executed or at the two points after the branch (i.e. just before `ActivateTask(High)` or just before `TerminateTask()`).

We verify that the SSTG adheres to the OSEK specification by model checking the NuSMV-model generated from the SSTG against the CTL formulas generated from the OIL file according to the OSEK specification. The latter arise by instantiating formula patterns that are parametric in the OIL configuration. Figure 5 shows an example formula, to be discussed shortly. Additional formulas specify that transitions of event states, resource states and interrupt states are correct, that ISRs may only perform their allowed system calls etc.; see the full version [9] for details. We frequently need to quantify over finite sets that are obtained from the OIL specification or are specified by OSEK (Table 2); such quantifiers are just expanded into finite conjunctions or disjunctions.

The formula in Figure 5 expresses that the scheduling for a task t is correct, i.e. each transition made in the SSTG is legal, and every transition required by the specification is in fact made by the system. Note that the transitions from Waiting and Suspended directly to Running are not technically legal according the OSEK specification; but the required intermediate state where the task in question would be Ready is not observable to the application, and we therefore opt to allow the direct transitions. The formula is instantiated for every task, and, in a slightly modified version, for every ISR, as interrupt scheduling is to some extent left up to the implementation. The formula is structured into four subformulas $\psi_1, \ldots, \psi_4$, each handling the allowed state transitions from one of the four possible starting states. To make this rather large formula readable we employ the following abbreviations for subformulas and properties:

– t.isHighestPriority specifies that control flow t *wants* to run, e.g. has state Ready or Running, and has higher priority than every other control flow that currently wants to run.
– t.isWaitingFor(e) states that the task t is waiting for the event e to occur.
– allOthersPreemptible(t) states that all tasks currently ready, other than t, are preemptible. Note that this formula needs to be used together with t.isHighestPriority to ensure that no interrupt is currently running and is needed altogether because when checking the starting transition of a non-preemptible task t, knowing t.isHighestPriority alone is not enough. For example, before the transition, t.isHighestPriority would always be false, since t was suspended, and after the transition, t would have already received the ceiling priority of the internal scheduler resource due to being non-preemptible.
– othersWillPreempt(t) denotes that task t will be preempted by another control flow due to witnessing a scheduling system call. This happens only if another control flow s currently has the highest priority, and if s is not an ISR, then t also has to be preemptible.
– waitSc(t) formalizes that task t is waiting for at least one event that is not set.

Note that the parameter e in t.isWaitingFor(e) is realized in our NuSMV-model as follows: For every task, an array containing the states of all its events is generated. The parameter e then simply is an index into this array.

Formula ψ_1 handles the case where t was previously suspended. Here, the only way for t to start up again is by a system call from the set Act (with a transition to Ready or Running depending on what else is currently running). In the case where t was previously Running, ψ_2 captures the possible transitions:

(1) There is another control flow that will preempt t (othersWillPreempt(t)) due to a system call that causes rescheduling, and t will become Ready.
(2) t issues a system call from the set TCS(s) signalling its termination, and becomes Suspended.
(3) t waits for one of its events. If the event is not set, t becomes Waiting.

In the case where t was previously Ready, ψ_3 formulates that t will either remain Ready, or become Running if it has the highest priority. Finally, in the case where t was previously Waiting, ψ_4 expresses that t keeps Waiting until one of the events it was waiting for is set and then becomes either Ready or Running, depending on whether it currently has the highest priority.

TABLE 2: Finite sets used in the OSEK formalization.

Set	Name/Description
SC	Scheduling calls, i.e. system calls that lead to a (re-)scheduling of tasks
Act(r)	System calls that activate the control flow (i.e. task or ISR) r
E(t)	Events of task t
TSC(r)	Terminating system calls, i.e. system calls that would lead to the termination of r
NPTasks	Non-preemptible tasks, i.e. tasks that cannot be preempted by higher priority tasks
RPreempt(r)	Control flows that could preempt r

```
TASK(Low) {
  print_state_hash(at: ABB1, next: interrupt);
  trigger_interrupt();
  if (read_decision(0)) {
    print_state_hash(at: ABB2, next: ActivateTask);
    ActivateTask(High);
  }
  print_state_hash(at: ABB3, next: TerminateTask);
  TerminateTask();
}
```
Figure 6: Example system – generated mockup for task Low

3.2. Horizontal Verification. To increase trust in the correctness of the actual generated code, we complement the verification that the SSTG complies with the OSEK specification (Section 3.1) with a verification procedure ensuring that the the generated code agrees with the SSTG. To this end, we extract a normalized STG, the *dynamic state transition graph (DSTG)*, from the actual binary. Interestingly, while one might expect the notion of agreement of the DSTG with the SSTG to be based on classical process-algebraic notions of equivalence such as bisimilarity [10], it turns out that the normalization process in fact guarantees agreement of the two STGs up to isomorphism. We therefore base our verification procedure on isomorphism checking, not only because we thus obtain stronger correctness guarantees but also because isomorphism checking of deterministic systems is computationally cheap, and in fact can be performed on-the-fly.

In more detail, we extract the DSTG by executing and probing the generated system binary with all possible syscall sequences that can originate from the given application. In order to facilitate exploration of the state space, we transform

$$\begin{aligned}
&\mathsf{AG}((t.\mathsf{state} = \mathsf{Suspended} \rightarrow \psi_1) \wedge (t.\mathsf{state} = \mathsf{Running} \rightarrow \psi_2) \wedge (t.\mathsf{state} = \mathsf{Ready} \rightarrow \psi_3) \wedge (t.\mathsf{state} = \mathsf{Waiting} \rightarrow \psi_4)), \text{ where}\\
&\psi_1 \equiv (\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{Act}(t) \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority})))\\
&\qquad \wedge (\neg\mathsf{allOthersPreemptible} \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{Act}(t) \leftrightarrow t.\mathsf{state} = \mathsf{Ready}))\\
&\qquad \wedge \mathsf{AX}((t.\mathsf{state} = \mathsf{Suspended} \rightarrow \neg\mathsf{syscall} \in \mathsf{Act}(t))\\
&\qquad \wedge \mathsf{AX}((t.\mathsf{state} = \mathsf{Ready} \vee t.\mathsf{state} = \mathsf{Running}) \rightarrow \mathsf{syscall} \in \mathsf{Act}(t)))\\
&\psi_2 \equiv \mathsf{AX}((t.\mathsf{state} = \mathsf{Ready} \leftrightarrow \mathsf{othersWillPreempt}(t)) \wedge (t.\mathsf{state} = \mathsf{Suspended} \leftrightarrow \mathsf{syscall} \in \mathsf{TSC}(t)) \wedge (t.\mathsf{state} = \mathsf{Waiting} \leftrightarrow \mathsf{waitSc}(t)))\\
&\psi_3 \equiv (\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{SC} \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority}))) \wedge \mathsf{AX}\,(t.\mathsf{state} = \mathsf{Running} \vee t.\mathsf{state} = \mathsf{Ready})\\
&\psi_4 \equiv \bigwedge_{e \in \mathsf{E}(t)} \big(t.\mathsf{isWaitingFor}(e) \rightarrow ((\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(e.\mathsf{set} \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority})))\\
&\qquad \wedge (\neg\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(e.\mathsf{set} \rightarrow t.\mathsf{state} = \mathsf{Ready}))\\
&\qquad \wedge \mathsf{AX}\,((t.\mathsf{state} = \mathsf{Waiting} \wedge \neg e.\mathsf{set}) \vee t.\mathsf{state} = \mathsf{Running} \vee t.\mathsf{state} = \mathsf{Ready}))\big)\\
&\mathsf{allOthersPreemptible}(t) \equiv \textstyle\bigwedge_{ot \in \mathsf{NPTasks} \setminus \{t\}} ot.\mathsf{state} \neq \mathsf{Running}\\
&\mathsf{othersWillPreempt}(t) \equiv \mathsf{syscall} \in \mathsf{SC} \setminus (\mathsf{TSC}(t) \cup \mathsf{WaitCalls}) \wedge \textstyle\bigvee_{or \in \mathsf{RPreempt}(t) \setminus \{t\}} or.\mathsf{isHighestPriority}\\
&\mathsf{waitSc}(t) \equiv \textstyle\bigvee_{e \in \mathsf{E}(t)} \neg e.\mathsf{set} \wedge t.\mathsf{isWaitingFor}(e)
\end{aligned}$$

Figure 5: Example CTL formula

the tasks' ABB graphs (see Section 2.2) into a *mock-up*, a C-program that omits the processing logic of the application and retains only the control flow, and then run an external search procedure on the mock-up that traverses the state space depth-first. We generate the mock-up (see Figure 6 for a partial mock-up of the running example) in two steps: (1) We use tools from the dOSEK framework to generate function-local ABB graphs from the generated LLVM code. (2) From these ABB graphs, we generate C-code that emulates the control flow, i.e. performs function and system calls as specified. Additionally, the mock-up:

- outputs node identifiers containing the identifier of the current ABB, as well as a hash of the actual current operating system state, where the latter includes the program counter;
- outputs identifiers for system calls containing the name of the system routine as well as the call site;
- optionally triggers any enabled interrupts;
- reads decisions on branching (including whether to trigger an interrupt) from standard input.

The mock-up is linked with the specialized kernel produced for the actual application by the dOSEK generator. It is then used by the *Dynamic State Explorer (DSE)*, a search procedure that generates the space of reachable states depth-first, steering the mock-up through the state space by feeding input to it in order to determine branching. The result of the search procedure is an STG. More precisely, some transitions in the STG are labelled with system calls as indicated above, and some are unlabelled, i.e. the STG includes ε-transitions; the latter correspond to internal transitions between computational ABBs. The ε-transitions are nondeterministic, since we omit the processing logic in the mock-up, so any form of conditional branching in the original application turns into nondeterminism; also, we cannot foresee external input. This STG is then subjected to ε-elimination as described in Section 2.3. We thus generate an STG with only labelled transitions; it is this STG that we refer to as the dynamic state transition graph (DSTG). The DSTG, as well as the SSTG, is deterministic, since every non-ε label is a tuple $(\mathit{call\ site}, \mathit{syscall\ type})$ and as such deterministically changes the system state. The same holds for interrupts, even though these are nondeterministic with regard to the activation time: Every interrupt transition label contains the interrupt number and, therefore, exactly describes its influence on the OS state, just like every other syscall-induced transition.

We check the DSTG for isomorphism with the SSTG. This is computationally unproblematic: since both LTS are deterministic after ε-elimination, we only need to check that both sides allow for the same transition labels, and then propagate this property to the states reached by the corresponding transitions. The reason that both graphs are isomorphic is that the states of the DSTG are identified by the hash value over the OS state, which is also reflected in the fields of every SSTG node.

4. Experiments

4.1. Positive Tests. To evaluate our verification method, we have run experiments on a number of OSEK systems. These systems stem from a test suite originally designed for the dOSEK implementation. In total, we have fully verified 58 test systems. We have selected eight systems that highlight key properties targeted in the verification (Table 3). The test cases "copter" and "copter-small" are the control software of a quadrotor copter and a simplified version thereof that arises by removing one asynchronous signal. For each of the systems, we list the number of system objects of the relevant types, i.e. interrupts, tasks (including the idle task), events, and resources specified in the OIL file.

Table 3 shows key parameters and the performance of the model checking tool on those systems. To qualify the generated NuSMV-model, we give the number of reachable states as well as its diameter (i.e. the length of the longest loop-free path). On a 2.4 GHz Intel Core i7-5500U machine with 8 GB of memory, the smaller experiments were completed within a fraction of a second. Only the verification of the two copter examples took longer, but finished in under three minutes.

For the horizontal verification, we probed the same 58 OSEK systems in two generator configurations (with and without system call site specialization) and established isomorphism with the respective SSTG in all cases. For most

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

TABLE 3: Performance of vertical verification.

Name	ISRs/Tasks/ Events/Res	Reachable states	Diameter	User time (sec)	Memory (MB)
bcc1-resource1j	0/6/0/4	26	16	0.10	27
bcc1-sse1c	0/6/0/4	24	14	0.08	28
ecc1-bt1g	0/6/2/2	10	10	0.05	24
ecc1-event1e	0/4/4/2	13	13	0.11	29
bcc1-isr2d	1/4/0/2	21	10	0.07	23
timing-abcomp	1/3/2/2	77	13	0.14	27
copter-small	3/11/0/3	1366	29	19.44	250
copter	4/12/0/3	4458	32	147.15	829

systems, the horizontal verification took less than 1 second. Only for the copter, the probing took 2.13s (0.81s for copter-small) and the isomorphism checking 0.17s (0.04s). When developing the hash function, we have probed examples with over 400.000 states in under 4 minutes.

4.2. Negative Tests and Fault Injection. For the vertical verification, we have performed *negative tests* by introducing faults into systems to check that these are correctly identified by our verification tool chain. To this end we have implemented a modified dOSEK generator that injects various types of faults into the input for our verification method. One can select between (a) mutations in the SSTG and (b) mutations in the input OIL specification. For (a), there is a random choice of either adding an edge or merging two states, and for (b), there is a random choice of either exchanging the priorities of two tasks or toggling the preemptability or the auto-start flag.

It should be noted here that not all faults introduced in this random way will necessarily lead to actual errors, e.g. when the change to the configuration does not influence the actual interaction between application and OS, or when additional transitions are actually valid. In our experiments, additional edges mostly did lead to errors and were detected by the formal verification; in some cases, additional edges produced legal transitions or violated parts of the specification not currently reflected in our formalization, e.g. that a task is not allowed to release resources it has not currently reserved. Merging graph nodes almost always produced errors caught by the verification, except in cases where the net effect would have been produced by ε-elimination anyway.

For the vertical verification, we injected 188 different faults into the test cases; this did not change the performance of the formal verification significantly compared to the unmodified test cases (Table 3). 177 faults lead to errors and were detected by the vertical verification. The other 11 faults were manually verified to be benign in the given usage pattern (a more detailed discussion can be found in the full version [9]). For the horizontal verification, we inserted 81 OIL-level faults: 61 lead to detected errors, while the other faults were manually checked to be benign.

4.3. Lessons Learned. Summing up the experience gained, it seems possible to achieve a fair degree of coverage in the verification of key aspects of the OSEK specification in tailored systems. (Unbounded) CTL model checking is feasible as the reachable part of the abstracted state space that we use remains within tractable range even for fairly large systems such as the quadrotor copter controller (with fewer than 4.5k states after abstraction); as stated above we attribute this fact to OS tailoring. Without wishing to get involved in the long-lasting linear-vs-branching-time war (e.g. [11]), we note that at the scale of our examples, LTL model checking does appear to reach the frontier of feasibility. For example, while in CTL, the copter-small example (Table 3) was discharged in under twenty seconds using less than 300 MB of memory, on the LTL correspondent of essentially the same specification we stopped the model checker after 30 minutes at 3.5 GB of allocated memory. This may be due to the higher formula complexity of LTL model checking (PSPACE instead of PTIME). A clear disadvantage of CTL, on the other hand, is that counterexamples are less informative, and typically stop at the first nested path quantifier.

Another somewhat surprising aspect is the fact that the notion of correspondence between the SSTG and the DSTG has turned out to be isomorphism of STGs. Actually getting this insight to work in full has required a somewhat laborious tuning process regarding the hashing of the OS state in the exploration of the DSTG, and in fact maintaining the tool chain in the future might be easier if one replaces isomorphism with strong bisimilarity.

5. Related Work

Our work is set in the highly active area of software model checking; see [12] for an (admittedly dated) overview. Our method exploits the high degree of predictability of scheduling afforded by OSEK, and in particular avoids the state space explosion caused by thread interleaving [13]. The static generation of state transition graphs from code in a somewhat similar style as featured in our approach has been used, in combination with LTL model checking, in the verification of event-condition-action systems [14].

The OSEK standard has been the subject of formal verification efforts to some degree. Waszniowski [15] modelled OSEK using timed automata within the UPPAAL model checker, and performed schedulability analyses. Huang et al. [16] modelled OSEK in CSP to verify various properties such as deadlock freedom. In this model, the internal application structure is not considered, and interrupts are excluded entirely. Vu et al. [17], [18] formalize the OSEK standard in Event-B and then verify designs of full OSs against the formalization. Where applications are considered [19], [20], these are verified in connection with an OS model rather than the actual OS implementation. In contrast, our approach avoids the verification gap between OS model and OS implementation by verifying the entire system composed of application and OS. This is made possible by focusing on the part of the OS behaviour actually relevant for the application at hand, instead of attempting to verify the full OS. We are thus able to a) work on the actual implementation, and b) fully model check the entire application/OS system including interrupts (expressly not covered in cited work on verifying OSEK applications).

Zhang et al. [21] formalize the OSEK standard in the K framework, along with the OIL and the programming language for applications. This is then used for test case generation and to verify applications by symbolic execution within the model. Interrupts are not considered.

Tigori et al. [22] use reachability checking of *extended finite automata* to remove dead code in tailored OSEK systems; the automata models involved are produced manually, while we generate STGs during code generation and from the actual code, respectively. Also, verification of OSEK adherence in [22] is by standardized testing, while we model-check the formalized standard.

On an entirely different scale, Klein et al. [23] formally verified the seL4 microkernel for functional correctness, in a project of 25 person years, and Sewell et al. [24] extended this verification from the C-Code level to the binary.

6. Conclusions

We have presented a framework for the fully automatic lightweight verification of tailored embedded systems following the OSEK industrial standard. Specifically, we have introduced a *vertical* verification process whereby the statically generated control flow of the tailored system is checked for conformance with the standard, and a *horizontal* verification method that ensures agreement between the static control and the actual generated code. Initial experiments run on a benchmark suite and on the control software of a quadrotor copter show promising results regarding the feasibility of full verification of key aspects of task interaction in OSEK systems, and in particular show that even substantial examples generate moderate-sized control flow graphs that allow fully-fledged model checking. The key to keeping state spaces small was to exploit OS tailoring as well as the particularities of scheduling in the OSEK standard. While our experiments indicate that dynamic exploration of control flow graphs scales up well, static methods that reconstruct the control flow from the compiled binary [24] may serve as a complementary approach in the future.

In further work, we plan to build more comprehensive coverage of the OSEK standard and to develop methods for validating our formalization of the standard against the informal specification, possibly building on previous work in this direction [17]. Also, we will apply our approach of model checking compiler-generated control flow graphs to application-specific verification goals beyond standard conformance.

Acknowledgments and Data. The authors thank the anonymous reviewers for their feedback. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1, LO 1719/4-1, and SCHR 1118/5-3.

Source code and data are available at
https://gitlab.cs.fau.de/dosek-verification

References

[1] P. Marwedel, *Embedded System Design*. Springer, 2006.

[2] J. Cooling, *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.

[3] M. Broy, “Challenges in automotive software engineering,” in *Proc. ICSE’06*. ACM Press, 2006, pp. 33–42.

[4] OSEK/VDX Group, “Operating system specification 2.2.3,” Tech. Rep., Feb. 2005.

[5] AUTOSAR, “Specification of operating system (version 5.1.0),” Automotive Open System Architecture GbR, Tech. Rep., Feb. 2013.

[6] OSEK/VDX Group, “OSEK implementation language specification 2.5,” Tech. Rep., 2004.

[7] C. Dietrich, M. Hoffmann, and D. Lohmann, “Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis,” *ACM Trans. Embed. Comp. Sys.*, vol. 16, pp. 35:1–35:25, 2017.

[8] F. Scheler and W. Schröder-Preikschat, “The RTSC: Leveraging the migration from event-triggered to time-triggered systems,” in *Proc. ISORC’10*. IEEE Computer Society Press, 2010, pp. 34–41.

[9] H.-P. Deifel, C. Dietrich, M. Göttlinger, D. Lohmann, S. Milius, and L. Schröder, “Automatic verification of application-tailored OSEK kernels,” full version; available at https://doi.org/10.15488/1761.

[10] R. Milner, *A Calculus of Communicating Systems*. Springer, 1980.

[11] M. Vardi, “Branching vs. linear time: Final showdown,” in *Proc. TACAS 2001*, ser. LNCS, vol. 2031. Springer, 2001, pp. 1–22.

[12] R. Jhala and R. Majumdar, “Software model checking,” *ACM Comput. Surv.*, vol. 41, pp. 21:1–21:54, 2009.

[13] L. Cordeiro and B. Fischer, “Verifying multi-threaded software using SMT-based context-bounded model checking,” in *Proc. ICSE’11*. ACM Press, 2011, pp. 331–340.

[14] M. Schordan and A. Prantl, “Combining static analysis and state transition graphs for verification of event-condition-action systems in the RERS 2012 and 2013 challenges,” *STTT*, vol. 16, pp. 493–505, 2014.

[15] L. Waszniowski and Z. Hanzálek, “Formal verification of multitasking applications based on timed automata model,” *Real-Time Systems*, vol. 38, no. 1, pp. 39–65, Jan. 2008.

[16] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, “Modeling and verifying the code-level OSEK/VDX operating system with CSP,” in *Proc. TASE’11*. IEEE Computer Society Press, 2011, pp. 142–149.

[17] D. Vu and T. Aoki, “Faithfully formalizing OSEK/VDX operating system specification,” in *Proc. SoICT’12*. ACM, 2012, pp. 13–20.

[18] D. Vu, Y. Chiba, K. Yatake, and T. Aoki, “Verifying OSEK/VDX OS design using its formal specification,” in *Proc. TASE’16*. IEEE Computer Society, 2016, pp. 81–88.

[19] H. Zhang, T. Aoki, and Y. Chiba, “Verifying OSEK/VDX applications: A sequentialization-based model checking approach,” *IEICE Transactions*, vol. 98-D, no. 10, pp. 1765–1776, 2015.

[20] H. Zhang, T. Aoki, H. Lin, M. Zhang, Y. Chiba, and K. Yatake, “SMT-based bounded model checking for OSEK/VDX applications,” in *Proc. APSEC’13*. IEEE Computer Society, 2013, pp. 307–314.

[21] M. Zhang, Y. Choi, and K. Ogata, “A formal semantics of the OSEK/VDX standard in K framework and its applications,” in *Proc. WRLA’14*. Springer, 2014, pp. 280–296.

[22] K. Tigori, J.-L. Béchennec, S. Faucou, and O. Roux, “Formal model-based synthesis of application-specific static RTOS,” *ACM Trans. Embed. Comp. Sys.*, vol. 16, pp. 97:1–97:25, 2017.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: formal verification of an OS kernel,” in *Proc. SOSP’09*. ACM, 2009, pp. 207–220.

[24] T. Sewell, M. Myreen, and G. Klein, “Translation validation for a verified OS kernel,” in *Proc. PLDI’13*. ACM, 2013, pp. 471–482.

Estimating Worst-case Latency of on-chip Interconnects with Formal Simulation

Freek Verbeek
Open University of the Netherlands
Radboud University, Nijmegen

Nikè van Vugt
Open University of the Netherlands

***Abstract*—Latency is a major issue in the design and validation of a Network-on-Chip (NoC). Various techniques for establishing latency bounds exist. Formal and mathematical methods, such as network calculus, can be used to analyze an NoC model. Simulation-based methods can be used to estimate latency bounds by exploring reachable states. Both have their advantages and disadvantages. This paper presents an approach that finds a middle ground between these two worlds. Our approach is based on simulation of high-level formal models. In contrast to traditional formal methods for worst-case latency, we do not require error-prone manual computation or the absence of cycles. In contrast to traditional simulation-based methods, we leverage the high level of abstraction to explore up to billions of states within a couple of hours. We apply our approach on an 8 core case study where a simple cache protocol runs on top of a ring-based Spidergon architecture. We show that deadlocks or starvations are easily found, and that for live networks a worst-case bound estimation can be produced within reasonable time.**

1. Introduction

Worst-case latency is a capricious matter: even a minor detail in the semantics of a minor element in a network may drastically impact whether some intricate and unexpected worst-case scenario can occur or not. Therefore, simulation is often done cycle-accurately and at a low level of abstraction (e.g., RTL). Cycle-accurate RTL simulation is a major research area in NoC validation [1], [2], [3]. However, finding the intricate scenario that causes worst-case latency can be tricky and may require simulation of many clock cycles.

This paper presents an approach that is based on the simulation of formal models of communication interconnects. Our approach is *high-level*: we purposefully do not simulate RTL but do simulation on a high level of abstraction. This enables fast simulation of large systems, e.g., an 8 core Spidergon with each core running a cache coherence protocol. Our approach is *formal*: the semantics of our model are completely formally defined in the Isabelle theorem prover. This enforces that each individual building block has concrete and executable semantics, and that the abstract blocks in our model can be extracted from RTL using the techniques described in [4], or that RTL can be generated from them. Finally, the building blocks of our model are *generic*: with various case studies we show that we can simulate routers, queues, virtual channels, credit-counters and state automata.

Simulation requires dealing with traffic patterns. Latency-Rate (LR) servers are a commonly accepted model of traffic injection and consumption [5] and are a basic concept in the *network calculus* [6]. The second part of our contribution consists of novel implementations for simulating network calculus traffic patterns in amortized cost $\mathcal{O}(1)$ per clock cycle. We use these algorithms to generate randomized traffic patterns – adhering to network calculus constraints – guided by heuristics that may quickly lead to a worst-case. All algorithms, models and Isabelle proofs are available online at http://www.cs.ru.nl/~freekver/fmcad17/.

Motivating example. Consider, e.g., Figure 4 from [7]. It presents a formal model of two communicating agents P and Q initiating requests and answering with responses. Each message type has its own virtual channel and a credit-based flow control ensures that a maximum number of packets can be en route at once. A packet that arrives at the opposite agent experiences a nondeterministic delay before it is sent back to its source.

The model can have a deadlock if the amount of credits is oversized, i.e., if sufficiently many packets can be injected to fill the cycle between the queues. That amount has to be at least $k + 2$ for such a deadlock to happen. However, for this deadlock to occur, a specific traffic pattern is necessary where the sources inject packets sufficiently fast, the sinks consume packets sufficiently slow, and the nondeterministic delay is sufficiently long.

Even though that deadlock is reachable for such a traffic pattern, for many traffic patterns it is not. If the sources are bounded in their injection rate, if the sinks minimally provide some service rate, and if the nondeterministic delay is bounded, can the deadlock still occur? Our methodology can be used to show – for example – that the deadlock may occur in a setting where the sizes of the ingress queues are 2, the credit-counters are oversized to 4, the delay is maximally 10 clock ticks, the sinks are eager, and the sources inject packets at a maximum rate of 1 packet every 10 clock ticks. If we take the exact same setting but lower the maximum delay to 9 clock ticks, then the deadlock does not occur, even though the credit-counters remain oversized. In that case, the maximum latency is 23, i.e., once a packet has been injected

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

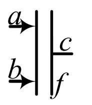
it will maximally require 23 clock ticks before it arrives as a response at the sink. Both dx queues account for a clock tick. The packet waits 9 clocks ticks in iq_1 behind another packet. In the 11th clock cycle, the packet has reached the front of iq_1. It waits for 10 clock ticks: 9 due to the delay, 1 due to the fact that if in the last clock cycle a packet is injected, the merge may add 1 clock tick to that delay. This sums to 23: since the sinks are eager a packet will not experience any delay in its final ingress queue.

To summarize, the queue sizes, the ratio between the speeds of the sources, sinks and delays and the arbitration policy of the merges *can* be such that the credit-counters are actually not necessary, since the deadlock they are meant to prevent cannot occur anyway. A minor change in any of these elements can, however, significantly increase latency or cause a deadlock.

We have simulated 10^8 clock cycles (about $5.5 \cdot 10^7$ packets) with *random* traffic that adheres to these constraints without finding this scenario. The scenario does not occur either when traffic is regular, e.g., when the sources inject *exactly* each 10th clock tick. However, by simulating a high-level formal model of the example and using traffic patterns guided by some simple heuristics, we were able to find it within a couple of hours.

2. Executable communication interconnect modeling

We model communication networks *formally*, i.e., in such a way that the semantics are defined mathematically, and *executable*. An example of a language that allows formal and executable modelling is xMAS [7]. Figure 1 presents an example of an xMAS primitive: the join. This primitive blocks its incoming packets until at both inputs a packet has arrived, at which point it will use function f to produce a packet at the output. The language xMAS provides various primitives such as merges (for arbitration), switches (for routing), sources and sinks.



(a) The xMAS join

$$\begin{aligned}
c\cdot\text{irdy} &= a\cdot\text{irdy} \wedge b\cdot\text{irdy}\\
c\cdot\text{data} &= f(a\cdot\text{data}, b\cdot\text{data})\\
a\cdot\text{trdy} &= c\cdot\text{trdy} \wedge b\cdot\text{irdy}\\
b\cdot\text{trdy} &= c\cdot\text{trdy} \wedge a\cdot\text{irdy}
\end{aligned}$$

(b) Formal semantics

Figure 1. Example of an xMAS primitive

Our modelling language is basically a generalization of xMAS. Even though xMAS is executable, the primitives are too fine-grained for efficient execution. For example, a simple XY router takes about 17 primitives and modelling virtual channels is inelegant [8]. Moreover, modelling, e.g., cache protocols with xMAS is infeasible.

In our generalization, we model communication networks using generic building blocks. Blocks can be fine-grained primitives, such as arbiters or joins, but can also be abstract statefull blocks such as routers, credit-counters or a cache protocol. Each block is efficiently executable. A communication network is then modelled as a composition of executable building blocks.

We use Isabelle/HOL [9] to define the notion of "building block", and how a communication network is composed out of these blocks.

2.1. Definition of a generic building block

A communication network consists of blocks with an ID of type *′block* connected by a set of channels with IDs of type *′chan*[1]. Each channel has an initiator and a target, and three wires, *irdy*, *trdy* and *data*. An *irdy* wire indicates the initiator is ready to transmit data, a *trdy* wire indicates the target is ready to receive, and the *data* wire is used for data transmission. We use $c\cdot$w to denote wire w of channel c.

datatype ′chan wire = irdy ′chan | trdy ′chan | data ′chan

The wires can have either Boolean values (in case of irdy/trdy), a data value, or be undefined. We use the term *color* to refer to the data, in the same fashion as colored Petrinets. We assume the existence of type *′color* that represents the set of colors.

datatype ′color wire-value = B bool | C ′color | Undef

A block assigns wire values to certain wires: the irdy and data wires of its outgoing channels, and the trdy wires of its incoming channels. It does so, based on the wire values of its *environment*: the trdy wires of its outgoing channels, and the irdy and data wires of its incoming channels. A *valuation* is used to store wire values. It is a set of pairs of wires and wire values.

type-synonym (′chan, ′color) val =
 (′chan wire × ′color wire-value) set

A block is defined by two functions. The first function, *eval*, takes as input the current *internal state* of the block and a valuation. It computes new values for wires and adds these to the given valuation. For example, in case of a join (see Figure 1b), if the given valuation contains values for wires $a\cdot$irdy and $b\cdot$irdy, then the semantics of the join are able to compute a new value for wire $c\cdot$irdy, and that value is added to the valuation. The second function, *tick*, provides the set of possible next internal states, given the current internal state of the block and the current valuation. In case of a stateless block, this function can return the empty set.

record (′chan, ′color, ′istate) block =
 eval :: ′istate ⇒ (′chan, ′color) val ⇒ (′chan, ′color) val
 tick :: ′istate ⇒ (′chan, ′color) val ⇒ ′istate set

The complete *state* σ of the communication network is then simply a map of blocks to their internal state. Note that the wires are not part of the state: they are combinatorial.

type-synonym (′block, ′istate) state = ′block ⇒ ′istate

1. Types preceded by an apostrophe are polymorphic, e.g., we allow any set of block IDs and any set of channel IDs.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

2.2. Composition of building blocks

To build a network out of these blocks, one has to connect them using channels. A properly composed network should satisfy various properties, such as: two channels may not be connected to the same input of a block, there may be no combinatorial cycles, there may be no dangling channels, and for each wire it should be possible to derive a unique and properly typed wire value. Moreover, the algorithm for computing a valuation for the current state can be quite contrived: it should start with blocks that can provide wire values solely based on their current internal state (such as queues, sinks and sources) and then propagate these values to other blocks. That propagation is both forward (in case of irdy/data wires) and backwards (in case of trdy wire). Proving termination of this propagation is not possible in the generic case: for example, if there is a combinatorial cycle, then this propagation will not terminate.

We will show that all these problems can be dealt with at once, by formalizing the derivation of wire values as a least fixed point.

From now on, we assume the existence of a function *block* that provides the set of blocks (modelled as a map of block IDs to blocks). Moreover, we assume that for each block b, derivation of the wire values is monotonically increasing. This ensures that the valuation can only increase, i.e., if more wire values are known, then more new wire values can be computed. This assumption is formulated by requiring for each block b and for each internal state x of that block, monotonicity of its *eval* function. We use an Isabelle locale to introduce a context in which such a function *block* exists.

locale monotone-blocks =
  fixes block :: $'$block $\Rightarrow$ ($'$chan,$'$color,$'$istate) block
  assumes mono (eval (block b) x)

We now define a function *deriveWires* that computes, given a state σ, a valuation for all wires. This valuation is computed by the following least fixpoint:

$$deriveWires\ \sigma \equiv \mu Z \cdot eval\ (block\ b)\ (\sigma\ b)\ Z \subseteq Z$$

The valuation derived from a state σ is thus the smallest valuation Z such that Z contains all wire values computed by any block b given its current internal state (σ b).
We can now formally define the *step* function of a composition of building blocks. It is nondeterministic and returns, given the current state σ, a set of next states.

definition step ::
  ($'$block,$'$istate) state $\Rightarrow$ ($'$block,$'$istate) state set
  where step $\sigma \equiv$
  $\{\sigma'\ .\ \forall\ b\ .\ \sigma'\,b \in$ tick (block b) (σ b) (deriveWires σ)$\}$

For all blocks b, function *tick* is used to compute the next internal state of that block. Function *tick* is given the current internal state of that block (σ b) and the current valuation of wires (*deriveWires* σ). State σ' is a next state if and only if all blocks b have "ticked", i.e., moved to some next internal state.

An Executable Communication Interconnect Model (ECIM) is defined as a set of blocks, connected in such a way that in each state there is a unique valuation for all wires. This assumption at once takes care of all issues mentioned at the beginning of this section. For example, it eliminates combinatorial cycles, since such a cycle would prevent function *deriveWires* to assign a value to the wires participating in that cycle. We extend the existing locale by adding the assumption that for each wire w there should be a unique value v derived by function *deriveWires*.

locale ECIM = monotone-blocks +
  assumes $\exists!$ v . (w, v) $\in$ deriveWires σ

Within the ECIM context we can formulate an LTL logic and prove all kinds of sanity theorems, such as:

1) If each block is persistent (e.g., will maintain a high irdy signal once it is set, until a transfer occurs [10]) then the network as a whole is persistent.
2) If each block is correctly typed, e.g., does not assign colors to irdy/trdy wires, then this property holds always globally.
3) Each xMAS primitive is an ECIM building block. We provide a shallow embedding of a DSL that can be used to model xMAS-like primitives into ECIM.
4) Block- and idle equations [10] can be proven correct, e.g., the incoming channel of a queue is permanently blocked if and only if the queue is full and its outgoing channel is permanently blocked.

2.3. Some ECIM building blocks

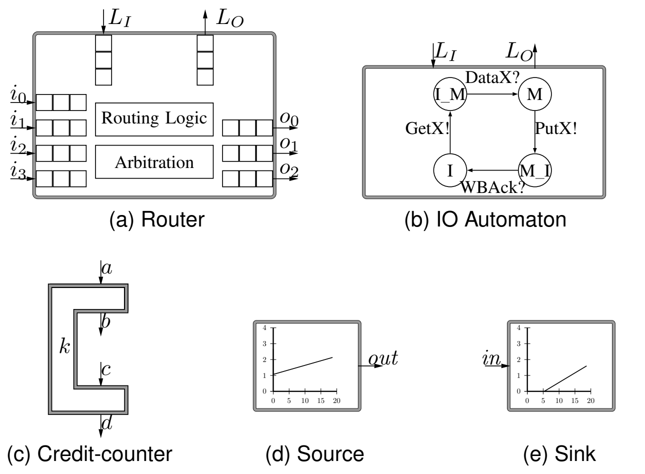


Figure 2. ECIM Building Blocks

Figure 2 shows some of the ECIM blocks that we use. First, we supply a *router* that has n inputs and m outputs. Based on a given routing logic, incoming messages will either be forwarded to some output, or – if they have arrived at their destination – be sent to the local out-port. Injection of messages can occur via the local in-port. The router is statefull: both the queues and the arbiter who resolves contention are part of the state.

Secondly, an *IO automaton* can be used to model protocols. Injection of a message (!) causes the L_O·irdy to be set to high and L_O·data to be set to, e.g., a GetX. Consumption of a message (?) is done by setting L_I·trdy high whenever L_I·data is, e.g., a DataX.

A *credit-counter* can be used to limit the amount of incoming packets "between" channels b and c. It is statefull, since it stores some integer less than k that indicates the number of counted messages.

Finally, *sources* and *sinks* can be used to generate traffic patterns. The next section discusses their implementation.

For all these blocks, we have implemented monotonic *eval* functions that determine when they consume and inject messages. Also, for each statefull block, we implement a *tick* function.

3. ECIM simulator

Assuming we have an ECIM built of efficiently executable blocks, a simulation algorithm is easily defined. We have implemented the pseudo code of Algorithm 3 in Java. The current state σ is an object with a method *step*. Each block is an object that implements a *tick* function that computes a next state. After each step, the clock is incremented.

```
function $\sigma$.STEP
    deriveWires
    for all block $b$ do
        b.tick
    end for
    t++
end function
```

Figure 3. ECIM simulation

In order to simulate a communication network, however, it is additionally required that traffic patterns can be efficiently simulated. For the case of *average* case latency analysis, one might use Poisson distributions to simulate random injections at sources and random consumptions at sinks. This has several major drawbacks when *worst-case* latency is considered. Firstly (regarding the sources) this does not accurately reflect the burstiness present in common traffic patterns, such as multimedia components [11]. Secondly, even with a low Poisson rate, the network might be flooded with traffic, even though the behavior of the source would not allow this. Thirdly (regarding the sinks) this does not provide a lower bound to the amount of consumptions: it theoretically allows for sinks to be dead for any period of time, thereby producing any worst-case latency as long as simulation is continued sufficiently long.

To this end, we use concepts of the network calculus to provide simulations for worst-case latency analysis [5], [6], [12]. We provide an efficient implementation to simulate *linear arrival curves*. Such curves are widely used to model traffic flows in a network and provide bursty traffic. For the sinks, we provide an efficient implementation of *linear service curves* to model the consumption behavior of sinks. Service curves (or LR servers) model an extensive class of network servers.

We only provide the definitions of the network calculus that are relevant to our simulator; for an introduction and more in-depth details, see [6], [12]. Network calculus concerns flows of traffic in the network. An *input flow* is characterized by function $R(t)$ which returns, given the current time t, the total cumulative amount of incoming traffic in the interval $[0, t]$. An *output flow* is characterized by cumulative function $R^*(t)$. We use $f_{\texttt{nc}}$ to denote the noncumulative version of a cumulative function f, i.e.,:

$$f(t) = \sum_{t' \leq t} f_{\texttt{nc}}(t')$$

3.1. Source simulation with linear arrival curves

An arrival curve can be defined by a function α that provides an upperbound to the amount of traffic.

Definition 1. *A source injects traffic adhering to arrival curve α, if and only if, for any time slot t:*

$$\forall 0 \leq s \leq t \cdot R(t) - R(s) \leq \alpha(t - s)$$

A linear arrival curve is defined by a natural number *burst* b and a real number *arrival rate* r: $\alpha(t) = rt + b$. Here b is a measure of burstiness, i.e., the amount of traffic that can be injected at once, and r is a maximally sustainable injection rate.

We provide an implementation that simulates linear arrival curves in time $\mathcal{O}(1)$ (see Figure 4). In this algorithm, we use the following variables:

Rt	the cumulative flow up to the current time slot (note that we only need to store the total flow up to the last time slot, and not per time slot).
$Rt_{\texttt{nc}}$	the non-cumulative flow in the current time slot, i.e., the number of injections
Mt	the current minimum value of $\alpha(t-s)+R(s)$ for all $s < t$

```
Require: $t > 0 \longrightarrow Mt = \min_{s<t} [\alpha(t-s) + R(s)]$
 1: function COMPUTEARRIVAL
 2:     if $t == 0$ then
 3:         $Rt_{\texttt{nc}} \leftarrow 0$
 4:     else
 5:         choose $Rt_{\texttt{nc}} \leq \lfloor Mt + r \rfloor - Rt$
 6:     end if
 7:     $Rt \mathrel{+}= Rt_{\texttt{nc}}$
 8:     $Mt \leftarrow t == 0\ ?\ b\ :\ min(Mt + r, b + Rt)$
 9:     t++
10:     return $Rt_{\texttt{nc}}$
11: end function
Ensure: $Mt = \min_{s<t} [\alpha(t-s) + R(s)]$
```

Figure 4. Simulation for linear arrival curves

After each call of function *computeArrival*, variable $Rt_{\mathtt{nc}}$ provides the number of packets that can be injected by the source in the current time slot. At Line 5, this value is computed by randomly choosing a value bounded by the current minimum value of $\alpha(t-s)+R(s)$, plus the rate, minus the current cumulative flow. Line 7 then updates variable Rt, so that it contains the current cumulative flow. Line 8 recomputes variable Mt, to preserve the invariant that variable Mt stores the minimum value of $\alpha(t-s)+R(s)$ for all $s<t$ when t is incremented at Line 9.

We prove correctness of this algorithm, by showing that Definition 1 holds invariably.

Theorem 1. *Let $R(0)=0$ and let $R(t+1)$ be the value of variable Rt after the tth call of function computeArrival. At any time slot t, we have:*

$$\forall 0 \le t' \le t \cdot R(t) - R(t') \le \alpha(t-t')$$

The proof is omitted, but an Isabelle formalization can be found online.

3.2. Sink simulation with linear service curves

Service curves model a server that provides a minimal amount of service. When incoming traffic arrives at a sink, the service curve may model a delay before packets are consumed, but eventually the sink will provide a service with some rate (as long as there is sufficient incoming traffic). A linear service curve β is defined by a natural number *delay* d and a real number *service rate* r:

$$\beta(t) = \begin{array}{ll} 0 & t \le d \\ r(t-d) & t > d \end{array}$$

The definition of a service curve is based on the notion of min-plus convolution.

Definition 2. *Let f and g be two weakly increasing functions. The min-plus convolution of f and g, notation $f \otimes g$, is defined as:*

$$(f \otimes g)(t) = \underset{0 \le s \le t}{inf} \; [f(s) + g(t-s)]$$

Definition 3. *A sink consumes traffic adhering to service curve β for the cumulative input flow R, if and only if, for any t:*

$$R^*(t) \ge (R \otimes \beta)(t)$$

Our algorithm is based on Propositions 1.3.1 and 1.3.2 from [6]. We here provide a corollary of these propositions:

Lemma 1. *Assume β is convex. There exists a weakly increasing function $\tau :: \mathbb{N} \mapsto \mathbb{N}$ such that for any time slot t:*

$$R^*(t) \ge R(\tau(t)) + \beta(t - \tau(t))$$

It is crucial that function τ is weakly increasing, and we leverage that fact to compute a lower bound for the service for the current time slot in amortized time $\mathcal{O}(1)$.

Remark. *For some service curves, the value of τ is computable: for a constant rate server without delay, and for strict service curves (modelling work-conserving sinks), the value of τ is the beginning of the last busy period. For linear service curves with delay that does not hold; the value of τ is unknown [6]. In the proof of Theorem 2 we will prove that in each time slot, only under a certain condition is it necessary to search for a new value for τ, and that the size of the range in which we have to search is constant.*

In the algorithm, we use the following variables:

τ	the current value such that $\tau < t$ and $R(\tau) + \beta(t-1-\tau)$ is minimal
Rshifted	a linked list storing values of function R, in such a way that $R(t) = \mathit{Rshifted}(t-\tau)$. The last value in this list always stores the total cumulative input flow $R(t)$, the first value stores $R(\tau)$. Lemma 1 shows that we can forget any value prior to τ.
Rt^*	the cumulative output flow up to the current time slot, i.e., the total number of consumptions
$Rt^*_{\mathtt{nc}}$	the non-cumulative output flow, i.e., the number of consumptions in the current time slot

Function *computeService* takes as input the current non-cumulative amount of incoming traffic in the current time slot t. It first stores that value in list *Rshifted*, by adding a new value to the end of that list (Lines 2 to 6). Then, we determine the value of the min-plus convolution by finding a value τ' for which $R(\tau')+\beta(t-\tau')$ is minimal. Currently, variable τ stores the value for which $R(\tau)+\beta(t-1-\tau)$ is minimal. Only if $t-\tau > d$ (Line 8) it is necessary to search for a new value for τ'. The search can be limited to a certain range (Lines 9 to 13). Otherwise, τ' remains the same as τ, since $R(\tau)+\beta(t-\tau)$ remains minimal.

If a new value for τ is found, we pop the elements in list *Rshifted* until we have reached that value (Lines 15 to 18). Line 19 then computes the minimum amount of service that is to be provided in the current time slot. The actual service is then some random value greater than that amount, but bounded by the amount of incoming traffic (Lines 20 and 21). Finally, the current cumulative amount of outgoing traffic (i.e., the total cumulative amount of consumptions) is updated, and the clock is increased.

We prove correctness of this algorithm, by showing that Definition 3 holds invariably.

Theorem 2. *Let $R^*_{\mathtt{nc}}(0)=0$ and let $R^*_{\mathtt{nc}}(t+1)$ be the value of variable $Rt^*_{\mathtt{nc}}$ after the tth call of function computeService. At any time slot t, we have:*

$$R(t) \ge R^*(t) \ge (R \otimes \beta)(t)$$

Proof: We first prove that for any n, the postcondition holds after the nth call of the algorithm:

$$R(\tau) + \beta(t-1-\tau) = \underset{s<t}{min} \; [R(s) + \beta(t-1-s)]$$

```
Require: $R(\tau)+\beta(t-1-\tau) = \min_{s<t} [R(s)+\beta(t-1-s)]$
 1: function COMPUTESERVICE(int $Rt_{\mathtt{nc}}$)
 2:     if $t == 0$ then
 3:         $Rshifted.add(Rt_{\mathtt{nc}})$
 4:     else
 5:         $Rshifted.add(Rt_{\mathtt{nc}} + Rshifted.last())$
 6:     end if
 7:     $\tau' \leftarrow \tau$
 8:     if $t - \tau > d$ then
 9:         for $s \leftarrow t - d$ to $t$ do
10:             if $f(s) \leq f(\tau')$ then
11:                 $\tau' \leftarrow s$
12:             end if
13:         end for
14:     end if
15:     while $\tau \neq \tau'$ do
16:         $Rshifted.removeFirst()$
17:         $\tau$++
18:     end while
19:     $min \leftarrow max(0, f(\tau) - Rt^*)$
20:     $max \leftarrow Rshifted.last() - Rt^*$
21:     choose $Rt^*_{\mathtt{nc}}$ st.: $min \leq Rt^*_{\mathtt{nc}} \leq max$
22:     $Rt^*$ += $Rt^*_{\mathtt{nc}}$
23:     $t$++
24:     return $Rt^*_{\mathtt{nc}}$
25: where
26:     $f(x) = Rshifted[x-\tau] + \beta(t-x)$
27: end function
Ensure: $R(\tau)+\beta(t-1-\tau) = \min_{s<t} [R(s)+\beta(t-1-s)]$
```

Figure 5. Simulation for linear service curves

The proof is by induction over n. For the base case, after the 0th call of the algorithm, we have $\tau = 0$ and $t = 1$ and the property holds trivially.

For the inductive case, the induction hypothesis is the precondition. We show that at Line 18, the algorithm has found a value for τ such that:

$$R(\tau) + \beta(t-\tau) = \min_{s \leq t} [R(s) + \beta(t-s)]$$

This implies that after incrementing time t (Line 23), the postcondition holds.

Assume there exists some $\tau' \leq t$, such that (A):

$$R(\tau') + \beta(t-\tau') < R(\tau) + \beta(t-\tau)$$

We first assume the case where (B): $t - \tau > d$ (Line 8). Assume (C): $t - \tau' > d$. Then:

$$\begin{aligned}
R(\tau') + \beta(t-\tau') &< R(\tau) + \beta(t-\tau) && \text{(A)}\\
R(\tau') + \beta(t-\tau'-1) + r &< R(\tau) + \beta(t-\tau) && \text{(C)}\\
R(\tau') + \beta(t-\tau'-1) + r &< R(\tau) + \beta(t-\tau-1) + r && \text{(B)}\\
R(\tau') + \beta(t-\tau'-1) &< R(\tau) + \beta(t-\tau-1)
\end{aligned}$$

The induction hypothesis then implies that $\tau' = t$. This implies that $d < 0$, and thus assumption (C) is false. This implies that $t - \tau' \leq d$. Hence: $t - d \leq \tau' \leq t$. Thus, in

Case	Latency	Time (µs)	#cycles	#packets
A	14	1.5	10^9	$3.0 \cdot 10^8$
B	23	23	10^9	$1.8 \cdot 10^8$
C	∞	110	$< 10^3$	N/A
D	∞	200	$< 10^4$	N/A
E	91	187	10^7	$1.5 \cdot 10^6$

TABLE 1. SIMULATION RESULTS.

case (B), only this range of values has to be searched for candidates for a new value of τ (if any).

Now we consider the case where (B) is false. By Lemma 1, we have (D): $\tau' \geq \tau$. Then:

$$\begin{aligned}
R(\tau') + \beta(t-\tau') &< R(\tau) + \beta(t-\tau) && \text{(A)}\\
R(\tau') &< R(\tau) && \neg\text{(B)} \wedge \text{(D)}
\end{aligned}$$

However, since R is weakly increasing, this is a contradiction. Hence, if (B) is false, then (A) is false, meaning that there exists no τ' such that $R(\tau') + \beta(t-\tau') < R(\tau) + \beta(t-\tau)$. Hence $R(\tau) + \beta(t-\tau) = \min_{s \leq t} [R(s) + \beta(t-s)]$.

From this inductive proof, the theorem follows. It has been proven that at Line 18, the algorithm has found a value for τ such that $R(\tau) + \beta(t-\tau)$ is minimal $\forall \tau \leq t$. Thus, the number of consumptions in the current time slot $Rt^*_{\mathtt{nc}}$ is minimally that value minus the current cumulative incoming traffic, and maximally the current backlog. □

4. Experimental results

We present 5 case studies. All results (see Table 1) have been obtained on a 1,6 GHz Intel Core i5 (4 cores). For each case study we have twice run 4 simulations in parallel. Column "Latency" shows the *measured* maximum latency. Column "Time" shows the average time of simulating 4 clock cycles on 4 cores in microseconds. The last two columns show the number of clock cycles and the number of packets *per simulation*.

A) Source, queue, queue, sink. Figure 6 shows the first 80 clock cycles of in- and output flows for a simple example with one source, two consecutive queues q_0 and q_1 (resp. sizes 5 and 10) and a sink. In this example, the sink has a delay of 4 but is eventually sufficiently fast to consume the injected input flow (i.e., the service curve "overtakes" the arrival curve). We have modelled a sink that can consume from q_1 more than one packet per clock tick, so that there is no gap between the intended output flow and the actual output flow. The source injects at most 1 packet per clock tick and can thus lag behind the intended arrival flow. The maximum measured latency in the first 80 clock ticks is 9, measured between clock ticks 9 to 18.

We have run eight simulations of 10^9 clock ticks (taking about 25 minutes per four) and measured a maximum latency of 14 (see Table 1).

B) Two agents. Section 1 presents the "two agents" example of Intel. We have modelled the xMAS example using more abstract ECIM blocks such as counters and delays. We enabled the following heuristic: let the sources and sinks remain irregular but maximize the nondeterministic delay.

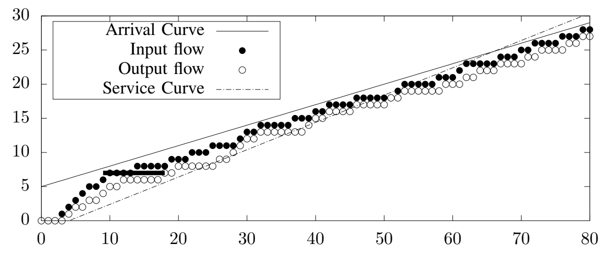


Figure 6. Source with $\alpha(5, 0.3)$, two queues, and a sink with $\beta(4, 0.4)$

All 8 simulations found a worst-case scenario that causes the 23 latency. On average, the worst-case scenario is found after about 300 million clock cycles. If the nondeterministic delay is increased to 10, simulation quickly finds a deadlock.

Spidergon case study. Figure 7 presents a Spidergon architecture with 8 nodes [13]. Packets consist of 2 bits that store a message type (GetX, DataX, PutX, or WBAck) and 3 bits that store the address of a node. The routing logic is *across first* meaning that if the shortest route to a packet requires an across channel, that channel is taken first. At each router, whenever two packets compete for the same output, a FIFO arbiter decides which packet can proceed and which not.

The protocol is a simple directory-based MI cache protocol. Caches 0 to 6 inject GetX messages to request exclusive access to a cache block. A GetX packet is accompanied by the address of the node that injects it. Its destination is always 7 and thus need not be stored in the packet. After injection of a GetX, a cache waits for data in state I_M. When a DataX is received, the cache moves to the M state where is has exclusive access to that block. To write back data, a PutX is injected, again accompanied with the address of the injecting node, and the cache moves to state M_I. Once a WBAck is received, the cache returns to state I.

Nodes 0 to 6 run this protocol; node 7 is a directory. It has two states I and M. In state I, upon receiving a GetX from node n, it injects a DataX with as destination n and moves to state M. In state M, upon receiving a PutX from node n, it injects a WBAck with destination n and returns to state I.

Finally, as shown in Figure 7, the cache protocol is connected to a source and a sink. Injection of GetX and PutX is done only when the source has a high irdy signal. We set the injection rate of the source to a Poisson distribution with $\lambda = 0.25$. This models that on average it takes 4 clock ticks for a cache to inject a packet. Dually, consumption of a packet is done only when the sink has a high trdy signal. We have set the sink to a linear service curve with $d = 4$ and $r = 0.5$, modelling that consumption of a packet can maximally be delayed 5 clock ticks, but after 6 clock ticks minimally one packet is consumed.

Note that this routing logic for the Spidergon architecture suffers from a routing deadlock. If simultaneously each node n injects packets destined for $n + 2 \mod 8$, then the clockwise circle becomes full and a circular wait occurs. The protocol prohibits that injection pattern and therefore – in this setup – *no routing deadlock* can occur. Moreover, there is *no protocol deadlock*. However, cross-layer deadlocks may occur.

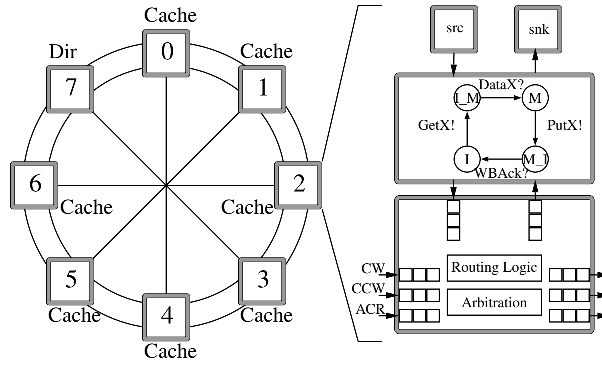


Figure 7. Spidergon Architecture with MI protocol

We measure the latency for a round trip of a GetX to return back to its source as a DataX. Note that this means that the measured latency includes cases where the GetX has to wait since another cache is currently owner. We modelled three cases C, D, and E described below. For the first two cases, runs are found in which the latency grows – seemingly – to infinity, i.e., for a long period of time the latency grows linearly with time. For the third case, the maximum measured latency is 91.

C) With deadlock. Consider the case where (at least) two caches inject a GetX, say caches n and m. The directory receives 7 packets and is only able to accept the first one, say from node n. It injects a DataX and moves to state M. The cache receives the data, and moves to state M as well. The only way progression is possible, is when cache n moves to state M_I by injecting PutX. However, the local queue from router 7 to the directory currently stores a GetX packet from cache m. Since queues are FIFO, the PutX cannot overtake the GetX and a cross-layer deadlock occurs.

D) Recycling packets: no deadlock, but starvation. The deadlock can be prevented by *recycling* packets: whenever a protocol cannot consume a packet it will be recycled to the end of the queue. This allows packets that can be consumed to overtake others. Indeed, this prevents the deadlock, if the size of the local queue from the router to the node is large enough. Worst case, that queue contains a PutX from the current sharer and 6 other GetX packets. Therefore the size of the queue has to be 8 to prevent the deadlock. Recycling, however, does introduce a starvation scenario. Since the packets in the local queue are no longer handled in FIFO order, it might be the case that a GetX is continuously overtaken by other GetX packets. This starvation scenario occured on average within 10.000 clock cycles.

E) Adding a VC: no deadlock or starvation. The starvation can be resolved by splitting the local queue of the directory into two virtual channels: one for GetX packets and one for the others. Moreover, GetX packets are not recycled, and thus handled in FIFO order. This resolves both the deadlock and the starvation. Note that to prevent both the deadlock and the starvation, only *one* virtual channel is required: the remainder of the network needs none. The size of the virtual channel should be sufficient to store six packets.

5. Related work

Simulation of NoCs is an extensive research area: various cycle-accurate tools exist that target heterogeneous and generic NoC architectures. BookSim [2] is a detailed, router-based simulator for NoCs, whose underlying network model has been validated to RTL implementations. OMNeT++ [1] is a framework where generic and high-level blocks can be simulated. It has been used to simulate among others the Spidergon architecture [14]. Generally, these tools generate synthetic traffic patterns. In contrast, our approach allows modelling of the cache coherence protocol deployed by the nodes, to more accurately model realistic traffic patterns. Also, we simulate a formal model, which has been defined in such a way that blocks can be derived from Verilog, or each block can be used to generate Verilog [4].

Zhao and Lu use network calculus to analyse xMAS models and use RTL simulation to verify tightness of their bounds [15]. In [12], Zhao uses the tool Simulink of MathWorks to simulate xMAS. Zhao derives Verilog from xMAS and uses that to find a worst-case latency bound for the two agents example. The presented results show simulation of about 10.000 packets. Since we simulate a high-level model instead of Verilog, we are able to simulate significantly more packets.

Salamat et al. use Noxim [3] to analyse both latency- and fault-tolerance of a 3D chip architecture [16]. Noxim takes more properties into account such as a power and thermal model. In contrast, our approach solely focusses on worst-case latency, which allows for more efficient execution.

6. Conclusion

This paper presents an approach for finding worst-case latency estimates based on simulation of formal models. The models – whose semantics have been formalized in Isabelle/HOL – consist of generic building blocks. Each block is high-level and therefore efficiently executable. The formal semantics ensure that the blocks can be implemented in Verilog, or that they can be extracted from Verilog.

We use novel implementations of simulating latency-rate servers to generate bursty traffic patterns. We implement some simple heuristics aimed at finding the worst case, such as maximizing delays whenever possible. We did, however, find that for some examples irregular traffic was necessary to get to the worst-case scenario, and that only after simulation of millions of packets the worst case was found.

In the near future, we aim to address the gap between the Isabelle model and the implementation, by using Isabelle's code generation to generate an implementation from the model. We expect this will impact performance, but it will enable a formally verified simulator for NoCs, where the model that is simulated can be used for theorem proving or model-based testing.

References

[1] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds., 2010, pp. 35–59.

[2] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 86–96.

[3] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 162–163.

[4] S. J. Joosten and J. Schmaltz, "Automatic extraction of micro-architectural models of communication fabrics from register transfer level designs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 1413–1418.

[5] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 5, pp. 611–624, 1998.

[6] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[7] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[8] F. Verbeek, P. M. Yaghini, A. Eghbal, and N. Bagherzadeh, "Advocat: Automated deadlock verification for on-chip cache coherence and interconnects," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1640–1645.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[10] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, *Verifying Deadlock-Freedom of Communication Fabrics*, 2011, pp. 214–231.

[11] A. E. Kiasari, Z. Lu, and A. Jantsch, "An analytical latency model for networks-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 113–123, 2013.

[12] X. Zhao, "Network on Chip: Performance Bound and Tightness," Ph.D. dissertation, KTH Royal Institute of Technology, 2015.

[13] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: a novel on-chip communication network," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*. IEEE, 2004, p. 15.

[14] L. Bononi and N. Concer, "Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh," in *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*. European Design and Automation Association, 2006, pp. 154–159.

[15] X. Zhao and Z. Lu, "Per-flow delay bound analysis based on a formalized microarchitectural model," in *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, April 2013, pp. 1–8.

[16] R. Salamat, M. Khayambashi, M. Ebrahimi, and N. Bagherzadeh, "A resilient routing algorithm with formal reliability analysis for partially connected 3D-NoCs," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3265–3279, Nov 2016.

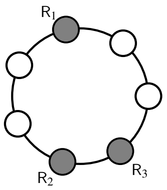
ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Parameterized Verification of Algorithms for Oblivious Robots on a Ring

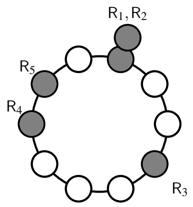
Arnaud Sangnier
IRIF, Univ Paris Diderot

Nathalie Sznajder,
Maria Potop-Butucaru
and Sébastien Tixeuil
Sorbonne Universités, UPMC Univ Paris 06, LIP6

***Abstract*—We study verification problems for autonomous swarms of mobile robots that self-organize and cooperate to solve global objectives. In particular, we focus in this paper on the model proposed by Suzuki and Yamashita of anonymous robots evolving in a discrete space with a finite number of locations (here, a ring). A large number of algorithms have been proposed working for rings whose size is not a priori fixed and can be hence considered as a parameter. Handmade correctness proofs of these algorithms have been shown to be error-prone, and recent attention had been given to the application of formal methods to automatically prove those. Our work is the first to study the verification problem of such algorithms in the parameterized case. We show that safety and reachability problems are undecidable for robots evolving asynchronously. On the positive side, we show that safety properties are decidable in the synchronous case, as well as in the asynchronous case for a particular class of algorithms. Several properties on the protocol can be decided as well. Decision procedures rely on an encoding in Presburger arithmetics formulae that can be verified by an SMT-solver. Feasibility of our approach is demonstrated by the encoding of several case studies.**



(a) A disoriented robot R$_1$



(b) A configuration with a tower

Fig. 1

I. Introduction

We consider sets of mobile oblivious robots evolving in a discrete space (modeled as a ring shaped graph). For our purpose, rings are seen as discrete graphs whose vertices represent the different positions available to host a robot, and edges model the possibility for a robot to move from one position to another. Robots follow the seminal model by Suzuki and Yamashita [23]: they do not remember their past actions, they cannot communicate explicitly, and are disoriented.

However, they can sense their environment and detect the positions of the other robots on the ring. If several robots share the same position on the ring (forming a *tower*, or multiplicity point), other robots may or may not detect the tower. If robots have *strong* multiplicity detection, as assumed in this paper, they are able to count the exact number of robots on a given position.

Robots are anonymous and execute the same deterministic algorithm to achieve together a given objective. Different objectives for ring shaped discrete spaces have been studied in the literature [17]: gathering – starting from any initial configuration, all the robots must gather on the same node, not known beforehand, and then stop [18], exploration with stop – starting from any initial configuration, the robots reach a configuration where they all are idle and, in the meanwhile, all the positions of the ring have been visited by a robot [16], exclusive perpetual exploration – starting from any tower-free configuration, each position of the ring is visited infinitely often and no multiplicity point ever appears [6], [11].

This work has been partly supported by the ANR research program PACS (ANR-14-CE28-0002)

Each robot behaves according to the following cycle: it takes a snapshot of its environment, then it computes its next move (either stay idle or move to an adjacent node in the ring), and at the end of the cycle, it moves according to its computation. Such a cycle is called a look-compute-move cycle.

Since robots cannot rely on a common sense of direction, directions that are computed in the compute phase are only *relative* to the robot. To tell apart its two sides, a robot relies on a description of the ring in both clockwise and counter-clockwise direction, which gives it two views of the configuration. There are two consequences to this fact. First, if its two views are identical, meaning that the robot is on an axis of symmetry, it cannot distinguish the two directions and thus either decides to stay idle, or to move. In the latter case, the robot moves becomes a non-deterministic choice between the two available directions. Second, when two robots have the same two views of the ring, the protocol commands them to move in the same relative direction, but this might result in moves in actual opposite directions for the two robots. Such a symmetrical situation is pictured in Figure 1.

Existing execution models consider different types of synchronization for the robots: in the fully synchronous model (FSYNC), all robots evolve simultaneously and complete a full look-compute-move cycle. The semi-synchronous model (SSYNC) consider runs that evolve in phases: at each phase, an arbitrary subset of the robots is scheduled for a full look-compute-move cycle, which is executed simultaneously by all robots of the subset. Finally, in the asynchronous model (ASYNC), robots evolve freely at their own pace: in particular, a robot can move according to a computation based on an obsolete observation of its environment, as others robots may have moved in between. Algorithms in the literature are typically parameterized by the number of robots and/or number of positions in the ring. In this work we focus on formally verifying algorithms parameterized by the number of ring positions only, assuming a a fixed number of robots.

A. *Related work*

Designing and proving mobile robot protocols is notoriously difficult. Formal methods encompass a long-lasting path of research that is meant to overcome errors of human origin. Unsurprisingly, this mechanized approach to protocol correctness was successively used in the context of mobile robots [7], [13], [5], [2], [20], [9], [4], [22], [3].

When robots are *not* constrained to evolve on a particular topology (but instead are allowed to move freely in a bidimensional Euclidian space), the Pactole (`http://pactole.lri.fr`) framework has been proven useful. Developed for the Coq proof assistant, Pactole enabled the use of high-order logic to certify impossibility results [2] for the problem of convergence: for any positive ε, robots are required to reach locations that are at most ε apart. Another classical impossibility result that was certified with Pactole is the impossibility of gathering starting from a bivalent configuration [9]. Recently, positive certified results for SSYNC gathering with multiplicity detection [10], and for FSYNC gathering without multiplicity detection [3] were provided. However, as of now, no Pactole library is dedicated to robots that evolve on discrete spaces.

In the discrete setting that we consider in this paper, model-checking proved useful to find bugs in existing literature [5], [14] and assess formally published algorithms [13], [5], [22]. Automatic program synthesis (for the problem of perpetual exclusive exploration in a ring-shaped discrete space) is due to Bonnet *et al.* [7], and can be used to obtain automatically algorithms that are "correct-by-design". The approach was refined by Millet *et al.* [20] for the problem of gathering in a discrete ring network. As all aforementioned approaches are designed for a bounded setting where both the number of locations and the number of robots are known, they cannot permit to establish results that are valid for any number of locations.

Recently, Aminof *et al.* [22] presented a general framework for verifying properties about mobile robots evolving on graphs, where the graphs are a parameter of the problem. While our model could be encoded in their framework, their undecidability proof relies on persistent memory used by the robots, hence is not applicable to the case of oblivious robots we consider here. Also, they obtain decidability in a subcase that is not relevant for robot protocols like those we consider. Moreover, their decision procedure relies on MSO satisfiability, which does not enjoy good complexity properties and cannot be implemented efficiently for the time being.

B. *Contributions*

In this work, we tackle the more general problem of verifying protocols for swarms of robots for any number of locations.

We provide a formal definition of the problem, where the protocol can be described as a quantifier free Presburger formula. This logic, weak enough to be decidable, is however powerful enough to express existing algorithms in the literature. Objectives of the robots are also described by Presburger formulae and we consider two problems: when the objective of the robots is a safety objective – robots have to avoid the configurations described by the formula (SAFE), and when it is a reachability objective (REACH). We show that if REACH is undecidable in any semantics, SAFE is decidable in FSYNC and SSYNC. We also show that when the protocol is *uniquely-sequentializable*, safety properties become decidable even in the asynchronous case.

Finally, we show practical applicability of this approach by using an SMT-solver to verify safety properties for some algorithms from the literature.

Hence, we advocate that our formalism should be used when establishing such protocols, as a formal and non-ambiguous description, instead of the very informal and sometimes unclear definitions found in the literature. Moreover, if totally automated verification in the parameterized setting seems unfeasible, our method could be used as a "sanity check" of the protocol, and to automatically prove intermediate lemmas, that can then be used as formally proved building blocks of a handmade correction proof.

II. Model of Robots Evolving on a Ring

A. *Formal model*

In this section we present the formal language to describe mobile robots protocols as well as the way it is interpreted.

1) Preliminaries: For $a,b \in \mathbb{Z}$ such that $a \leq b$, we denote by $[a,b]$ the set $\{c \in \mathbb{Z} \mid a \leq c \leq b\}$. For $a \in \mathbb{Z}$ and $b \in \mathbb{N}$, we write $a \odot b$ the natural $d \in [0,(b-1)]$ such that there exists $j \in \mathbb{Z}$ and $a = b.j + d$ (for instance $-1 \odot 3 = 2$). Note that $\odot$ corresponds to the modulo operator, but for sake of clarity we recall its definition when a is negative.

We recall the definition of Existential Presburger (EP) formulae. Let Y be a countable set of variables. First we define the grammar for terms $\mathtt{t} ::= x \mid \mathtt{t}+\mathtt{t} \mid a \cdot \mathtt{t} \mid \mathtt{t} \bmod a$, where $a \in \mathbb{N}$ and $x \in Y$ and then the grammar for formulae is given by $\phi ::= \mathtt{t} \bowtie b \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists x.\phi$ where $\bowtie \in \{=,\leq,\geq,<,>\}$, $x \in Y$ and $b \in \mathbb{N}$. We sometimes write a formula ϕ as $\phi(x_1,\ldots,x_k)$ to underline that $x_1,\ldots,x_k$ are the free variables of ϕ. The set of Quantifier Free Presburger (QFP) formulae is obtained by the same grammar deleting the elements $\exists x.\phi$. Note that when dealing with QFP formulae, we allow as well negations of formulae.

We say that a vector $V = \langle d_1, \dots, d_k \rangle$ satisfies an EP formula $\phi(x_1, \dots, x_k)$, denoted by $V \models \phi$, if the formula obtained by replacing each x_i by d_i holds. Given a formula ϕ with free variables $x_1, \dots, x_k$, we write $\phi(d_1, \dots, d_k)$ the formula where each x_i is replaced by d_i. We let $[\![\phi(x_1, \dots, x_k)]\!] = \{\langle d_1, \dots, d_k \rangle \in \mathbb{N}^k \mid \phi(d_1, \dots, d_k) \models \phi\}$ be the set of models of the formula. In the sequel, we use Presburger formulae to describe configurations of the robots, as well as protocols.

2) Configurations and robot views: In this paper, we consider a fixed number $k > 0$ of robots and, except when stated otherwise, we assume the identities of the robots are $\mathcal{R} = \{R_1, \dots, R_k\}$. We may sometimes identify $\mathcal{R}$ with the set of indices $\{1, \dots, k\}$. On a ring of size $n \geq k$, a *(k,n)-configuration* of the robots (or simply a configuration if n and k are clear from the context) is given by a vector $\mathbf{p} \in [0, n-1]^k$ associating to each robot R_i its position $\mathbf{p}(i)$ on the ring. We assume w.l.o.g. that positions are numbered in the clockwise direction.

A *view* of a robot on this configuration gives the distances between the robots, starting from its neighbor, i.e. the robot positioned on the next occupied node (a distance equals to 0 meaning that two robots are on the same node). A *view* $\mathbf{V} = \langle d_1, \dots, d_k \rangle \in [0, n]^k$ is a k-tuple such that $\sum_{i=1}^{k} d_i = n$ and $d_1 \neq 0$. We let $\mathcal{V}_{n,k}$ be the set of possible views for k robots on a ring of size n. Notice that all the robots sharing the same position should have the same view. For instance, suppose that, on a ring of size 10, 2 robots R_1, and R_2 are on the same position of the ring (say position 1), R_3 is at position 4, R_4 is at position 8, and R_5 is at position 9 (see Figure 1b). Then, the view of R_1 and R_2 is $\langle 3, 4, 1, 2, 0 \rangle$. It is interpreted by the fact that there is a robot at a distance 3 (it is R_3), a robot a at distance $3+4$ (it is R_4) and so on. We point out that all the robots at the same position share the same view. We as well suppose that in a view, the first distance is not 0 (this is possible by putting 0 at the 'end' of the view instead). As a matter of fact in the example of Figure 1b, there is a robot at distance $3+4+1+2 = 10$ from R_1 (resp. R_2), which is R_2 (resp. R_1). The sum of the d_i corresponds always to the size of the ring and here the fact that in the view of R_1 we have as last element 0 signifies that there is a distance 0 between the last robot (here R_2) and R_1. When looking in the opposite direction, their view becomes: $\langle 2, 1, 4, 3, 0 \rangle$. Formally, for a view $\mathbf{V} = \langle d_1, \dots, d_k \rangle \in [0, n]^k$, we note $\overleftarrow{\mathbf{V}} = \langle d_j, \dots, d_1, d_k, \dots, d_{j+1} \rangle$ the corresponding view when looking at the ring in the opposite direction, where j is the greatest index such that $d_j \neq 0$.

Given a configuration $\mathbf{p} \in [0, n-1]^k$ and a robot $R_i \in \mathcal{R}$, the view of robot R_i when looking in the clockwise direction, is given by $\mathbf{V_p}[i \rightarrow] = \langle d_i(i_1), d_i(i_2) - d_i(i_1), \dots, n - d_i(i_{k-1}) \rangle$, where, for all $j \neq i$, $d_i(j) \in [1, n]$ is such that $(\mathbf{p}(i) + d_i(j)) \odot n = \mathbf{p}(j)$ and $i_1, \dots, i_k$ are indexes pairwise different such that $0 < d_i(i_1) \leq d_i(i_2) \leq \dots \leq d_i(i_{k-1})$. When robot R_i looks in the opposite direction, its view according to the configuration $\mathbf{p}$ is $\mathbf{V_p}[\leftarrow i] = \overleftarrow{\mathbf{V_p}[i \rightarrow]}$.

3) Protocols: In our context, a protocol for networks of k robots is given by a QFP formula respecting some specific constraints.

Definition 1 (Protocol): A protocol is a QFP formula $\phi(x_1, \dots, x_k)$ such that for all views $\mathbf{V}$ the following holds: if $\mathbf{V} \models \phi$ and $\mathbf{V} \neq \overleftarrow{\mathbf{V}}$ then $\overleftarrow{\mathbf{V}} \not\models \phi$

A robot uses the protocol to know in which direction it should move according to the following rules. As we have already stressed, all the robots that share the same position have the same view of the ring. Given a configuration $\mathbf{p}$ and a robot $R_i \in \mathcal{R}$, if $\mathbf{V_p}[i \rightarrow] \models \phi$, then the robot R_i moves in the clockwise direction, if $\mathbf{V_p}[\leftarrow i] \models \phi$ then it moves in the opposite direction, if none of $\mathbf{V_p}[i \rightarrow]$ and $\mathbf{V_p}[\leftarrow i]$ satisfies ϕ then the robot should not move. The conditions expressed in Definition 1 imposes hence a direction when $\mathbf{V_p}[i \rightarrow] \neq \mathbf{V_p}[\leftarrow i]$. In case $\mathbf{V_p}[i \rightarrow] = \mathbf{V_p}[\leftarrow i]$, the robot is disoriented and it can hence move in one direction or the other. For instance, consider the configuration $\mathbf{p}$ pictured on Figure 1a. Here, $\mathbf{V_p}[1 \rightarrow] = \langle 3, 1, 3 \rangle = \mathbf{V_p}[\leftarrow 1]$. Note that such a semantics enforces that the behavior of a robot is not influenced by its direction. In fact consider two symmetrical configurations $\mathbf{p}$ and $\mathbf{p}'$ such that $\mathbf{V_p}[i \rightarrow] = \overleftarrow{\mathbf{V_{p'}}[i \rightarrow]}$ for each robot R_i. If $\mathbf{V_p}[i \rightarrow] \models \phi$ (resp. $\mathbf{V_p}[\leftarrow i] \models \phi$), then necessarily $\mathbf{V_{p'}}[\leftarrow i] \models \phi$ (resp. $\mathbf{V_{p'}}[i \rightarrow] \models \phi$), and the robot in $\mathbf{p}'$ moves in the opposite direction than in $\mathbf{p}$ (and the symmetry of the two configurations is maintained).

We now formalize the way movement is decided. Given a protocol ϕ and a view $\mathbf{V}$, the moves of any robot whose clockwise direction view is $\mathbf{V}$ are given by:

$$move(\phi, V) = \begin{cases} \{+1\} & \text{if } \mathbf{V} \models \phi \text{ and } \mathbf{V} \neq \overleftarrow{\mathbf{V}} \\ \{-1\} & \text{if } \overleftarrow{\mathbf{V}} \models \phi \text{ and } \mathbf{V} \neq \overleftarrow{\mathbf{V}} \\ \{-1, +1\} & \text{if } \mathbf{V} \models \phi \text{ and } \mathbf{V} = \overleftarrow{\mathbf{V}} \\ \{0\} & \text{otherwise} \end{cases}$$

Here $+1$ (resp. -1) stands for a movement of the robot in the clockwise (resp. anticlockwise) direction.

B. Different possible semantics

We now describe different transition relations between configurations. Robots have a two-phase behavior : (1) look at the ring and (2) according to their view, compute and perform a movement. In this context, we consider three different modes. In the *semi-synchronous mode*, in one step, some of the robots look at the ring and move. In the *synchronous mode*, in one step, all the robots look at the ring and move. In the *asynchronous mode*, in one step a single robot can either choose to look at the ring, if the last thing it did was a movement, or to move, if the last thing it did was to look at the ring. As a consequence, its movement decision is a consequence of the view of the ring it has in its memory. In the remainder of the paper, we fix a protocol ϕ and we consider a set $\mathcal{R}$ of k robots.

1) Semi-synchronous mode: We begin by providing the semantics in the semi-synchronous case. For this matter we define the transition relation $\hookrightarrow_\phi \subseteq [0, n-1]^k \times [0, n-1]^k$ (simply noted $\hookrightarrow$ when ϕ is clear from the context) between configurations. We have $\mathbf{p} \hookrightarrow \mathbf{p}'$ if there exists a subset $I \subseteq \mathcal{R}$

of robots such that, for all $i \in I$, $\mathbf{p}'(i) = (\mathbf{p}(i)+m) \odot n$, where $m \in move(\phi, \mathbf{V_p}[i \rightarrow])$, and for all $i \in \mathcal{R} \setminus I$, $\mathbf{p}'(i) = \mathbf{p}(i)$.

2) Synchronous mode: The transition relation $\Rightarrow_\phi \subseteq [0,n-1]^k \times [0,n-1]^k$ (simply noted $\Rightarrow$ when ϕ is clear from the context) describing synchronous movements is very similar to the semi-synchronous case, except that all the robots have to move. Then $\mathbf{p} \Rightarrow \mathbf{p}'$ if $\mathbf{p}'(i) = (\mathbf{p}(i)+m) \odot n$ with $m \in move(\phi, \mathbf{V_p}[i \rightarrow])$ for all $i \in \mathcal{R}$.

3) Asynchronous mode: The definition of transition relation for the asynchronous mode is a bit more involved, for two reasons: first, the move of each robot does not depend on the current configuration, but on the last view of the robot. Second, in one step a robot either look or move. As a consequence, an *asynchronous configuration* is a tuple $(\mathbf{p},\mathbf{s},\mathbf{V})$ where $\mathbf{p} \in [0,n-1]^k$ gives the current configuration, $\mathbf{s} \in \{\mathbf{L},\mathbf{M}\}^k$ gives, for each robot, its internal state ($\mathbf{L}$ stands for ready to look and and $\mathbf{M}$ stands for compute and move) and $\mathbf{V} \in \mathcal{V}^k_{n,k}$ stores, for each robot, the view (in the clockwise direction) it had the last time it looked at the ring.

The transition relation for asynchronous mode is hence defined by a binary relation $\rightsquigarrow_\phi$ (or simply $\rightsquigarrow$) working on $[0,n-1]^k \times \{\mathbf{L},\mathbf{M}\}^k \times \mathcal{V}^k_{n,k}$ and defined as follows: $\langle \mathbf{p},\mathbf{s},\mathbf{V}\rangle \rightsquigarrow \langle \mathbf{p}',\mathbf{s}',\mathbf{V}'\rangle$ iff there exist $R_i \in \mathcal{R}$ such that the following conditions are satisfied:

- for all $R_j \in \mathcal{R}$ such that $j \neq i$, $\mathbf{p}'(j) = \mathbf{p}(j)$, $\mathbf{s}'(j) = \mathbf{s}(j)$ and $\mathbf{V}'(j) = \mathbf{V}(j)$,
- if $\mathbf{s}(i) = \mathbf{L}$ then $\mathbf{s}'(i) = \mathbf{M}$, $\mathbf{V}'(i) = \mathbf{V_p}[i \rightarrow]$ and $\mathbf{p}'(i) = \mathbf{p}(i)$, i.e. if the robot that has been scheduled was about to look, then the configuration of the robots won't change, and this robot updates its view of the ring according to the current configuration and change its internal state,
- if $\mathbf{s}(i) = \mathbf{M}$ then $\mathbf{s}'(i) = \mathbf{L}$, $\mathbf{V}'(i) = \mathbf{V}(i)$ and $\mathbf{p}'(i) = (\mathbf{p}(i) + m) \odot n$, with $m \in move(\phi, \mathbf{V}(i))$, i.e. if the robot was about to move, then it changes its internal state and moves according to the protocol, and *its last view of the ring*.

4) Runs: A semi-synchronous (resp. synchronous) ϕ-run (or a run according to a protocol ϕ) is a (finite or infinite) sequence of configurations $\rho = \mathbf{p}_0\mathbf{p}_1\ldots$ where, for all $0 \leq i < |\rho|$, $\mathbf{p}_i \hookrightarrow_\phi \mathbf{p}_{i+1}$ (resp. $\mathbf{p}_i \Rightarrow_\phi \mathbf{p}_{i+1}$). Moreover, if $\rho = \mathbf{p}_0 \cdots \mathbf{p}_n$ is finite, then there is no $\mathbf{p}$ such that $\mathbf{p}_n \hookrightarrow_\phi \mathbf{p}$ (respectively $\mathbf{p}_n \Rightarrow_\phi \mathbf{p}$). An asynchronous ϕ-run is a (finite or infinite) sequence of asynchronous configurations $\rho = \langle \mathbf{p}_0,\mathbf{s}_0,\mathbf{V}_0\rangle\langle \mathbf{p}_1,\mathbf{s}_1,\mathbf{V}_1\rangle \cdots$ where, for all $0 \leq i < |\rho|$, $\langle \mathbf{p}_i,\mathbf{s}_i,\mathbf{V}_i\rangle \rightsquigarrow_\phi \langle \mathbf{p}_{i+1},\mathbf{s}_{i+1},\mathbf{V}_{i+1}\rangle$ and such that $\mathbf{s}_0(i) = \mathbf{L}$ for all $i \in [1,k]$. Observe that the value of $\mathbf{V}_0$ has no influence on the actual asynchronous run, since any robot starts its computation by a look, hence changing the value of $\mathbf{V}_0$.

We let $\mathbf{Post}_{\mathrm{ss}}(\phi,\mathbf{p}) = \{\mathbf{p}' \mid \mathbf{p} \hookrightarrow_\phi \mathbf{p}'\}$, $\mathbf{Post}_{\mathrm{s}}(\phi,\mathbf{p}) = \{\mathbf{p}' \mid \mathbf{p} \Rightarrow_\phi \mathbf{p}'\}$ and $\mathbf{Post}_{\mathrm{as}}(\phi,\mathbf{p}) = \{\mathbf{p}' \mid$ there exist $\mathbf{V},\mathbf{s}',\mathbf{V}'$ s.t. $\langle \mathbf{p},\mathbf{s}_0,\mathbf{V}\rangle \rightsquigarrow_\phi \langle \mathbf{p}',\mathbf{s}',\mathbf{V}'\rangle\}$, with $\mathbf{s}_0(i) = \mathbf{L}$ for all $i \in [1,k]$. Note that in the asynchronous case we impose all the robots to be ready to look. We respectively write $\hookrightarrow^*_\phi$, $\Rightarrow^*_\phi$ and $\rightsquigarrow^*_\phi$ for the reflexive and transitive closure of the relations $\hookrightarrow_\phi$, $\Rightarrow_\phi$ and $\rightsquigarrow_\phi$ and we define $\mathbf{Post}^*_{\mathrm{ss}}(\phi,\mathbf{p})$, $\mathbf{Post}^*_{\mathrm{s}}(\phi,\mathbf{p})$ and $\mathbf{Post}^*_{\mathrm{as}}(\phi,\mathbf{p})$ by replacing in the definition $\mathbf{Post}_{\mathrm{ss}}(\phi,\mathbf{p})$,$\mathbf{Post}_{\mathrm{s}}(\phi,\mathbf{p})$ and $\mathbf{Post}_{\mathrm{as}}(\phi,\mathbf{p})$ the relations $\hookrightarrow_\phi$, $\Rightarrow_\phi$ and $\rightsquigarrow_\phi$ by their reflexive and transitive closure accordingly.

We now come to our first result that shows that when the protocols have a special shape, the three semantics are identical.

Definition 2: A protocol ϕ is said to be *uniquely-sequentializable* if, for all configuration $\mathbf{p}$, there is at most one robot $R_i \in \mathcal{R}$ such that $move(\phi, \mathbf{V_p}[i \rightarrow]) \neq \{0\}$.

When ϕ is uniquely-sequentializable at any moment at most one robot moves. Consequently, in that specific case, the three semantics are equivalent as stated by the following theorem.

Theorem 1: If a protocol ϕ is uniquely-sequentializable, then for all configuration $\mathbf{p}$, $\mathbf{Post}^*_{\mathrm{s}}(\phi,\mathbf{p}) = \mathbf{Post}^*_{\mathrm{ss}}(\phi,\mathbf{p}) = \mathbf{Post}^*_{\mathrm{as}}(\phi,\mathbf{p})$.

C. Problems under study

In this work, we aim at verifying properties on protocols where we assume that the number of robots is fixed (equals to $k > 0$) but the size of the rings is parameterized and satisfies a given property. Note that when executing a protocol the size of the ring never changes. For our problems, we consider a ring property that is given by a QFP formula $\mathtt{Ring}(y)$, a set of bad configurations given by a QFP formula $\mathtt{Bad}(x_1,\ldots,x_k)$ and a set of good configurations given by a QFP formula $\mathtt{Goal}(x_1,\ldots,x_k)$. We then define two general problems to address the verification of such algorithms: the SAFE_m problem, and the REACH_m problem, with $m \in \{\mathrm{ss},\mathrm{s},\mathrm{as}\}$.

The SAFE_m problem is to decide, given a protocol ϕ and two formulae $\mathtt{Ring}$ and $\mathtt{Bad}$ whether there exists a size $n \in \mathbb{N}$ with $n \in [\![\mathtt{Ring}]\!]$, and a (k,n)-configuration $\mathbf{p}$ with $\mathbf{p} \notin [\![\mathtt{Bad}]\!]$, such that $\mathbf{Post}^*_m(\phi,\mathbf{p}) \cap [\![\mathtt{Bad}]\!] \neq \emptyset$.

The REACH_m problem is to decide given a protocol ϕ and two formulae $\mathtt{Ring}$ and $\mathtt{Goal}$ whether there exists a size $n \in \mathbb{N}$ with $n \in [\![\mathtt{Ring}]\!]$ and a (k,n)-configuration $\mathbf{p}$, such that $\mathbf{Post}^*_m(\phi,\mathbf{p}) \cap [\![\mathtt{Goal}]\!] = \emptyset$. Note that the two problems are not dual due to the quantifiers.

As an example, we can state in our context the SAFE_m problem that consists in checking that a protocol ϕ working with three robots never leads to collision (i.e. to a configuration where two robots are on the same position on the ring) for rings of size strictly bigger than 6. In that case we have $\mathtt{Ring} := y > 6$ and $\mathtt{Bad} := x_1 = x_2 \ \vee\ x_2 = x_3 \ \vee\ x_1 = x_3$.

III. Undecidability results

In this section, we present undecidability results for the two aforementioned problems. The proofs rely on the encoding of a deterministic k-counter machine run. A deterministic k-counter machine consists of k integer-valued registers (or counters) called c_1, ..., c_k, and a finite list of labelled instructions L. Each instruction is either of the form $\ell : c_i = c_i + 1; \mathtt{goto}\ \ell'$, or $\ell : \mathtt{if}\ c_i > 0\ \mathtt{then}\ c_i = c_i - 1; \mathtt{goto}\ \ell'; \mathtt{else\ goto}\ \ell''$, where $i \in [1,k]$. We also assume the existence of a special instruction $\ell_h : \mathtt{halt}$. Configurations of a k-counter machine are elements of $L \times \mathbb{N}^k$, giving the current instruction and the current values of the registers. The initial configuration is $(\ell_0, 0, \ldots, 0)$, and the set of halting configurations is $\mathsf{HALT} = \{\ell_h\} \times \mathbb{N}^k$.

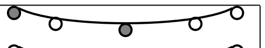
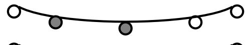
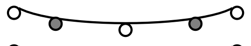
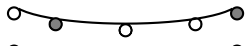
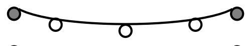
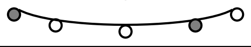
Given a configuration $(\ell, n_1, \ldots, n_k)$, the successor configuration $(\ell', n'_1, \ldots, n'_k)$ is defined in the usual way and we note $(\ell, n_1, \ldots, n_k) \vdash (\ell', n'_1, \ldots, n'_k)$. A run of a k-counter machine is a (finite or infinite) sequence of configurations $(\ell_0, n_1^0, \ldots, n_k^0), (\ell_1, n_1^1, \ldots, n_k^1) \cdots$, where $(\ell_0, n_1^0, \ldots, n_k^0)$ is the initial configuration, and, for all $i \geq 0$, $(\ell_i, n_1^i, \ldots, n_k^i) \vdash (\ell_{i+1}, n_1^{i+1}, \ldots, n_k^{i+1})$. The run is finite if and only if it ends in a halting configuration, i.e. in a configuration in HALT.

Theorem 2: $\mathsf{SAFE}_{\text{as}}$ is undecidable.

Sketch of proof. The proof relies on a reduction from the halting problem of a deterministic two-counter machine M to $\mathsf{SAFE}_{\text{as}}$ with $k = 42$ robots. It is likely that an encoding using less robots might be used for the proof, but for the sake of clarity, we do not seek the smallest possible amount of robots. The halting problem is to decide whether the run of a given deterministic two-counter machine is finite; this problem is undecidable [21]. The idea is to simulate the run of M in a way that ensures that a collision occurs if and only if M halts. Positions of robots on the ring are used to encode values of counters and the current instruction of the machine. The k-protocol makes sure that movements of the robots simulate correctly the run of M. Moreover, one special robot moves only when the initial configuration is encoded, and another only when the final configuration is encoded. The collision is ensured in the following sequence of actions of the robots: when the initial configuration is encoded, the first robot computes its action but does not move immediately. When the halting configuration is reached, the second robot computes its action and moves, then the first robot finally completes its move, entailing the collision. Note that if the ring is not big enough to simulate the counter values then the halting configuration is never reached and there is no collision.

Instead of describing configurations of the robots by applications giving positions of the robots on the ring, we use a sequence of letters F or R, representing respectively a free node and a node occupied by a robot. When a letter $\mathsf{A} \in \{\mathsf{F}, \mathsf{R}\}$ is repeated i times, we use the notation A^i, when it is repeated an arbitrary number or times (including 0), we use A^*. To distinguish between the two representations of the configurations, we use respectively the terms configurations or word-configurations. The correspondence between a configuration and a word-configuration is obvious. A *machine-like* (word-)configuration is a configuration of the form $\mathsf{B}_3\mathsf{F}^*\mathsf{R}\mathsf{F}^*\mathsf{B}_4\mathsf{F}^*\mathsf{R}\mathsf{F}^*\mathsf{B}_5\mathsf{F}^*\mathsf{R}\mathsf{F}^*\mathsf{B}_6\mathsf{F}^*\mathsf{R}\mathsf{F}^*\mathsf{B}_7\mathsf{F}^*\mathsf{R}\mathsf{F}^*\mathsf{B}_8 P_1P_2P_3P_4P_5$ RFR, where B_i is a shorthand for $\mathsf{F}\mathsf{R}^i\mathsf{F}$, and $P_1P_2 \in \{\mathsf{RF}, \mathsf{FR}\}$ and exactly one $P_i = \mathsf{R}$ for $i \in \{3,4,5\}$, $P_j = \mathsf{F}$ for $j \in \{3,4,5\} \setminus \{i\}$ (see Table I for a graphic representation of the section $P_1P_2P_3P_4P_5$ of the ring). Observe that the different blocks B_i yield for every robot in the ring a distinct view, since it allows the robots to locate their position on the ring. Hence, in the rest of the proof we abuse notations and describe the protocol using different names for the different robots, according to their position in the ring, even if they are formally anonymous. We let $\mathcal{R}$ be the set of robots involved. A machine-like (word-)configuration $\mathsf{B}_3\mathsf{F}^{n_1}\mathsf{R}_{c_1}\mathsf{F}^*\mathsf{B}_4\mathsf{F}^{n_2}\mathsf{R}_{c_2}\mathsf{F}^*\mathsf{B}_5\mathsf{F}^m\mathsf{R}_c\mathsf{F}^n\mathsf{B}_6\mathsf{F}^i\mathsf{R}_\ell\mathsf{F}^{i'}\mathsf{B}_7\mathsf{F}^p\mathsf{R}_{\ell'}\mathsf{F}^r\mathsf{B}_8$ $\mathsf{R}_{tt}\mathsf{F}\mathsf{R}_t\mathsf{F}\mathsf{F}\mathsf{R}_g\mathsf{F}\mathsf{R}_d$ is said to be *stable* because of the positions of robots R_t and R_{tt} (see Table I). Moreover, it encodes the configuration (ℓ_i, n_1, n_2) of M (due to the relative positions of robots R_{c_1}, R_{c_2} and R_ℓ respectively to B_3, B_4 and B_6). We say that a configuration $\mathbf{p}$ is machine-like, stable, etc. if its corresponding word-configuration is machine-like, stable, etc. In the following, we distinguish configurations of the 2-counter machine, and configurations of the robots, by calling them respectively M-configurations and ϕ-(word)-configurations. For a stable and machine-like ϕ-configuration $\mathbf{p}$, we let $M(\mathbf{p})$ be the M-configuration encoded by $\mathbf{p}$. We first present the part of the algorithm simulating the behavior of M. We call this algorithm ϕ'. Since the machine is deterministic, only one instruction is labelled by ℓ_i, known by every robot. The simulation follows different steps, according to the positions of the robots R_t and R_{tt}, as pictured in Table I.

TABLE I: Different types of configurations

stable configuration	$\mathsf{R}_{tt}\mathsf{F}\mathsf{R}_t\mathsf{F}\mathsf{F}$	
moving1 configuration	$\mathsf{F}\mathsf{R}_{tt}\mathsf{R}_t\mathsf{F}\mathsf{F}$	
moving2 configuration	$\mathsf{F}\mathsf{R}_{tt}\mathsf{F}\mathsf{R}_t\mathsf{F}$	
moving3 configuration	$\mathsf{F}\mathsf{R}_{tt}\mathsf{F}\mathsf{F}\mathsf{R}_t$	
stabilizing1 configuration	$\mathsf{R}_{tt}\mathsf{F}\mathsf{F}\mathsf{F}\mathsf{R}_t$	
stabilizing2 configuration	$\mathsf{R}_{tt}\mathsf{F}\mathsf{F}\mathsf{R}_t\mathsf{F}$	

We explain the algorithm ϕ' on the configuration (ℓ_i, n_1, n_2) with the transition $\ell_i : \texttt{if } c_1 > 0 \texttt{ then } c_1 = c_1 - 1\texttt{;goto } \ell_j\texttt{;else goto } \ell_{j'}$.

- When in a stable configuration, robot R_{tt} first moves to obtain a $\mathsf{moving1}$ configuration.
- In a $\mathsf{moving1}$ configuration, robot R_c moves until it memorizes the current value of c_1. More precisely, in a $\mathsf{moving1}$ configuration where $n_1 \neq m$, robot R_c moves : if $n_1 > m$, and $n \neq 0$, R_c moves towards B_6, if $n_1 < m$, it moves towards B_5, if $n_1 > m$ and $n = 0$, it does not move.
- In a $\mathsf{moving1}$ configuration where $n_1 = m$, R_t moves to obtain a $\mathsf{moving2}$ configuration.
- In a $\mathsf{moving2}$ configuration, if $n_1 = m \neq 0$, then R_{c_1} moves towards B_3, hence encoding the decrementation of c_1.
- In a $\mathsf{moving2}$ configuration, if $n_1 = m = 0$ or if $n_1 \neq m$, (then the modification of c_1 is either impossible, or already done), robot $\mathsf{R}_{\ell'}$ moves until it memorizes the position of robot R_ℓ: if $p < i$, and $r \neq 0$, $\mathsf{R}_{\ell'}$ moves towards B_8; if $p > i$, $\mathsf{R}_{\ell'}$ moves towards B_7.
- In a $\mathsf{moving2}$ configuration, if $p = i$, then R_t moves to obtain a $\mathsf{moving3}$ configuration.
- In a $\mathsf{moving3}$ configuration, if $n_1 = m = 0$, and robot $\mathsf{R}_{\ell'}$ encodes ℓ_i (i.e. $p = i$), then $c_1 = 0$ and robot R_ℓ has to move until it encodes $\ell_{j'}$. If on the other hand $n_1 < m$, then robot R_ℓ moves until it encodes ℓ_j. More precisely, if $n_1 = m = 0$, and the position encoded by R_ℓ is smaller than j' ($i < j'$), and if $i' \neq 0$, then R_ℓ moves towards B_7. If $n_1 = m = 0$, and the position encoded by R_ℓ is greater than j', R_ℓ moves towards B_6. If $n_1 < m$, then robot R_ℓ

moves in order to reach a position where it encodes ℓ_j (towards B_6 if $i > j$, towards B_7 if $i < j$ and $i' \neq 0$).

- In a moving3 configuration, if the position encoded by $\mathsf{R}_{\ell'}$ is ℓ_i, if $n_1 = m = 0$ and the position encoded by R_ℓ is $\ell_{j'}$, or if $n_1 \neq m$, and the position encoded by R_ℓ is ℓ_j, then the transition has been completely simulated : the counters have been updated and the next transition is stored. The robots then return to a stable configuration: robot R_{tt} moves to obtain a stabilizing1 configuration.
- In a stabilizing1 configuration, robot R_t moves to obtain a stabilizing2 configuration.
- In a stabilizing2 configuration, robot R_t moves to obtain a stable configuration.

For other types of transitions, the robots move similarly. When in a stable configuration encoding a configuration in HALT, no robot moves. We describe now the algorithm ϕ that simply adds to ϕ' the two following rules. Robot R_g (respectively R_d) moves in the direction of R_d (respectively in the direction of R_g) if and only if the robots are in a stable machine-like configuration, and the encoded configuration of the machine is $(\ell_0, 0, 0)$ (respectively is in HALT), (since the configuration is machine-like, the distance between R_g and R_d is 2). Observe that if the sub-algorithm ϕ' is uniquely-sequentializable, ϕ is not.

On all configurations that are not machine-like, the algorithm makes sure that no robot move. This implies that once R_g or R_d has moved, no robot with a view up-to-date ever moves. One can easily be convinced that the algorithm can be expressed by a QFP formula ϕ.

Let the formulae $\texttt{Bad}(\mathbf{p}_1, \ldots, \mathbf{p}_{42}) = \bigvee_{\substack{i,j \in [1,42] \\ i \neq j}} (\mathbf{p}_i = \mathbf{p}_j)$ that is satisfied by all the configurations where two robots share the same position and $\texttt{Ring}(y) = y \geq 0$. We can show that M halts if and only if there exits a size $n \in [\![\texttt{Ring}]\!]$, a $(42, n)$-configuration $\mathbf{p}$ with $\mathbf{p} \notin [\![\texttt{Bad}]\!]$, such that R_g and R_d eventually collide, i.e., $\mathbf{Post}^*_{\mathrm{as}}(\phi, \mathbf{p}) \cap [\![\texttt{Bad}]\!] \neq \emptyset$. Note that R_g and R_d can collide only in an asynchronous run.

□

Theorem 3: REACH_m is undecidable, for $m \in \{\mathrm{ss}, \mathrm{s}, \mathrm{as}\}$.

Sketch of proof. The proof relies on a reduction from the repeated reachability problem of a deterministic three-counter zero-initializing bounded-strongly-cyclic machine M, which is undecidable [19]. A counter machine is zero-initializing if from the initial instruction ℓ_0 it first sets all the counters to 0. Moreover, an infinite run is said to be *space-bounded* if there is a value $K \in \mathbb{N}$ such that all the values of all the counters stay below K during the run. A counter machine M is bounded-strongly-cyclic if every space-bounded infinite run starting from any configuration visits ℓ_0 infinitely often. The repeated reachability problem we consider is expressed as follows: given a 3-counter zero-initializing bounded-strongly-cyclic machine M, does there exist an infinite space-bounded run of M? A configuration of M is encoded in the same fashion than in the proof of Theorem 2, with 3 robots encoding the values of the counters. A transition of M is simulated by the algorithm in the same way than above *except that if a counter is to be increased, the corresponding robot moves accordingly even if there is no room to do it, yielding a collision.* Since the machine is bounded-strongly-cyclic and zero-initializing, any infinite run will eventually visit $(\ell_0, 0, 0, 0)$, so any infinite execution of the robots encode an infinite space-bounded run of M starting in $(\ell_0, 0, 0, 0)$.

□

IV. Decidability results and case study

In this section, we show that even if $\mathsf{SAFE}_{\mathrm{as}}$, $\mathsf{REACH}_{\mathrm{as}}$, $\mathsf{REACH}_{\mathrm{ss}}$ and $\mathsf{REACH}_{\mathrm{s}}$ are undecidable, the other cases $\mathsf{SAFE}_{\mathrm{s}}$ and $\mathsf{SAFE}_{\mathrm{ss}}$ can be reduced to the satisfiability problem for EP formulae, which is decidable and NP-complete [8].

A. Reducing safety to successor checking

The first step towards decidability is to remark that to solve $\mathsf{SAFE}_{\mathrm{s}}$ and $\mathsf{SAFE}_{\mathrm{ss}}$ it is enough to look at the one-step successor. Let ϕ be a protocol over k robots and $\texttt{Ring}$ and $\texttt{Bad}$ be respectively a ring property and a set of bad configurations. We have then the following lemma.

Lemma 1: Let $n \in \mathbb{N}$ such that $n \in [\![\texttt{Ring}]\!]$ and $m \in \{\mathrm{s}, \mathrm{ss}\}$. There exists a (k, n)-configuration $\mathbf{p}$ with $\mathbf{p} \notin [\![\texttt{Bad}]\!]$, such that $\mathbf{Post}^*_m(\phi, \mathbf{p}) \cap [\![\texttt{Bad}]\!] \neq \emptyset$ iff there exists a (k, n)-configuration $\mathbf{p}'$ with $\mathbf{p}' \notin [\![\texttt{Bad}]\!]$, such that $\mathbf{Post}_m(\phi, \mathbf{p}') \cap [\![\texttt{Bad}]\!] \neq \emptyset$.

This last result may seems strange at a first sight but it can easily be explained by the fact that robots protocols are most of the time designed to work without any assumption on the initial configuration, except that it is not a bad configuration.

B. Encoding successor computation in Presburger

We now describe various EP formulae to be used to express the computation of the successor configuration in synchronous and semi-synchronous mode.

First we show how to express the view of some robot R_i in a configuration $\mathbf{p}$, with the following formula:

$$\begin{gathered} \texttt{ConfigView}_i(y, p_1, \ldots, p_k, d_1, \ldots, d_k) := \\ \exists d'_1, \ldots, d'_{k-1} . i_1, \ldots, i_{k-1} \cdot \bigwedge_{j=1}^{k-2} d'_j \leq d'_{j+1} \wedge \\ \bigwedge_{\ell=1}^{k-1} (\bigvee_{j=1, j \neq i}^{k} p_j = (p_i + d'_\ell \mod y) \wedge i_\ell = j) \wedge \\ 0 < d'_1 \wedge \bigwedge_{j=1}^{k-1} d'_j \leq y \wedge \bigwedge_{\ell \neq j} i_\ell \neq i_j \wedge \\ d_1 = d'_1 \wedge \bigwedge_{j=2}^{k-1} d_j = d'_j - d'_{j-1} \wedge d_k = y - d'_{k-1}, \end{gathered}$$

Note that this formula only expresses in the syntax of Presburger arithmetic the definition of $\mathbf{V}_{\mathbf{p}}[i \rightarrow]$ where the variable y is used to store the length of the ring, $p_1, \ldots, p_k$ represent $\mathbf{p}$ and the variables $d_1, \ldots, d_k$ represent the view.

We also use the formula $\texttt{ViewSym}(d_1, \ldots, d_k, d'_1, \ldots, d'_k)$ that is useful to compute the symmetric of a view.

$$\begin{gathered} \texttt{ViewSym}(d_1, \ldots, d_k, d'_1, \ldots, d'_k) := \\ \bigvee_{j=1}^{k} (\bigwedge_{\ell=j+1}^{k} (d_\ell = 0 \wedge d'_\ell = 0) \wedge \bigwedge_{\ell=1}^{j} d'_\ell = d_{j-\ell+1} \wedge d_j \neq 0) \end{gathered}$$

We are now ready to introduce the formula $\texttt{Move}^{\phi}_i(y, p_1, \ldots, p_k, d_1, \ldots, d_k, p')$, which is true if and only if, on a ring of size n (represented by the variable y), the move of robot R_i according to the protocol $\phi(d_1, \ldots, d_k)$ from the configuration $\mathbf{p}$ yields to the new position p'. Here the variables $p_1, \ldots, p_k$ characterizes $\mathbf{p}$ and $d_1, \ldots, d_k$ the view of process i. Note that in the asynchronous semantics, the view of i might completely differ from the current configuration.

$$\begin{gathered}\mathtt{Move}^{\phi}_i(y, p_1, \dots, p_k, d_1, \dots, d_k, p') := \\ \exists d'_1, \cdots, d'_k \cdot \mathtt{ViewSym}(d_1, \dots, d_k, d'_1, \dots, d'_k) \wedge \\ \Big(\phi(d_1, \dots, d_k) \wedge \big((p_i < y-1 \wedge p' = p_i + 1) \\ \vee (p_i = y-1 \wedge p' = 0)\big)\Big) \vee \\ \Big(\phi(d'_1, \dots, d'_k) \wedge \big((p_i > 0 \wedge p' = p_i - 1) \vee (p_i = 0 \wedge p' = y-1)\big)\Big) \\ \vee \Big(\neg\phi(d_1, \dots, d_k) \wedge \neg\phi(d'_1, \dots, d'_k) \wedge (p' = p_i)\Big)\end{gathered}$$

Now, given two (k,n)-configurations $\mathbf{p}$ and $\mathbf{p}'$, and a k-protocol ϕ, it is easy to express the fact that $\mathbf{p}'$ is a successor configuration of $\mathbf{p}$ according to ϕ in a semi-synchronous run (resp. synchronous run); for this we define the two formulae $\mathtt{SemiSyncPost}_\phi(y, p_1, \dots, p_k, p'_1, \dots, p'_k)$ and $\mathtt{SyncPost}_\phi(y, p_1 \dots, p_k, p'_1, \dots, p'_k))$ as follows:

$$\begin{gathered}\mathtt{SemiSyncPost}_\phi(y, p_1, \dots, p_k, p'_1, \dots, p'_k) := \exists d^1_1, \dots, d^1_k, \dots \\ d^k_1, \dots, d^k_k \cdot \textstyle\bigwedge_{j=1}^{k} \mathtt{ConfigView}_j(y, p_1, \dots, p_k, d^j_1, \dots, d^j_k) \wedge \\ \textstyle\bigvee_{i=1}^{k} \big(\mathtt{Move}^{\phi}_i(y, p_1, \dots, p_k, d^i_1, \dots, d^i_k, p'_i) \wedge \\ \textstyle\bigwedge_{j=1, j\neq i}^{k} ((p'_j = p_j) \vee \mathtt{Move}^{\phi}_j(y, p_1, \dots, p_k, d_1, \dots, d_k, p'_j))\big)\end{gathered}$$

$$\begin{gathered}\mathtt{SyncPost}_\phi(y, p_1, \dots, p_k, p'_1, \dots, p'_k) := \exists d^1_1, \dots, d^1_k, \dots, \\ d^k_1, \dots, d^k_k \cdot \textstyle\bigwedge_{i=1}^{k} \big(\mathtt{ConfigView}_i(y, p_1, \dots, p_k, d^i_1, \dots, d^i_k) \wedge \\ \mathtt{Move}^{\phi}_i(y, p_1, \dots, p_k, d^i_1, \dots, d^i_k, p'_i)\big)\end{gathered}$$

Lemma 2: For all $n \in \mathbb{N}$ and all (k,n)-configurations $\mathbf{p}$ and $\mathbf{p}'$, we have:

1) $\mathbf{p} \hookrightarrow \mathbf{p}'$ if and only if $n, \mathbf{p}, \mathbf{p}' \models \mathtt{SemiSyncPost}_\phi$,
2) $\mathbf{p} \Rightarrow \mathbf{p}'$ if and only if $n, \mathbf{p}, \mathbf{p}' \models \mathtt{SyncPost}_\phi$.

C. Results

Now since to solve $\mathsf{SAFE}_{\mathrm{ss}}$ and $\mathsf{SAFE}_{\mathrm{s}}$, we only need to look at the successor in one step, as stated by Lemma 1, and thanks to the formulae $\mathtt{SemiSyncPost}_\phi$ and $\mathtt{SyncPost}_\phi$ and their properties expressed by Lemma 2, we deduce that these two problems can be expressed in Presburger arithmetic.

Theorem 4: $\mathsf{SAFE}_{\mathrm{s}}$ and $\mathsf{SAFE}_{\mathrm{ss}}$ are decidable and in NP.
Proof: We consider a ring property $\mathtt{Ring}(y)$, a protocol ϕ for k robots (which is a QFP formula) and a set of bad configurations given by a QFP formula $\mathtt{Bad}(x_1, \dots, x_k)$. We know that there exists a size $n \in \mathbb{N}$ with $n \in [\![\mathtt{Ring}]\!]$, and a (k,n)-configuration $\mathbf{p}$ with $\mathbf{p} \notin [\![\mathtt{Bad}]\!]$, such that $\mathbf{Post}^*_{\mathrm{s}}(\phi, \mathbf{p}) \cap [\![\mathtt{Bad}]\!] \neq \emptyset$ if and only if there exists a (k,n)-configuration $\mathbf{p}'$ with $\mathbf{p}' \notin [\![\mathtt{Bad}]\!]$, such that $\mathbf{Post}_m(\phi, \mathbf{p}') \cap [\![\mathtt{Bad}]\!] \neq \emptyset$. By Lemma 2, this latter property is true if and only if the following formula is satisfiable:

$$\begin{gathered}\mathtt{SyncPost}_\phi(y, p_1, \dots, p_k, p'_1, \dots, p'_k) \wedge \\ \mathtt{Ring}(y) \wedge \neg\mathtt{Bad}(p_1, \dots, p_k) \wedge \\ \mathtt{Bad}(p'_1, \dots, p'_k)\end{gathered}$$

For the semi-synchronous case, we replace the formula $\mathtt{SyncPost}_\phi$ by $\mathtt{SemiSyncPost}_\phi$. The NP upper bound is obtained by the fact that the built formula is an EP formula. □

When the protocol ϕ is uniquely-sequentializable, i.e. when in each configuration at most one robot make the decision to move then Theorem 1 leads us to the following result.

Corollary 1: When the protocol ϕ is uniquely-sequentializable, $\mathsf{SAFE}_{\mathrm{as}}$ is decidable.

D. Expressing other interesting properties

Not only the method consisting in expressing the successor computation in Presburger arithmetic allows us to obtain the decidability for $\mathsf{SAFE}_{\mathrm{s}}$ and $\mathsf{SAFE}_{\mathrm{ss}}$, but they also allow us to express other interesting properties. For instance, we can compute the successor configuration in asynchronous mode for a protocol ϕ working over k robots thanks to the formula $\mathtt{AsyncPost}_\phi(y, p_1, \dots, p_k, s_1, \dots, s_k, v_1 \dots, v_k, p'_1, \dots, p'_k, s'_1, \dots, s'_k, v'_1, \dots, v'_k)$, which is given by:

$$\begin{gathered}\mathtt{AsyncPost}_\phi := \exists d_1, \dots, d_k \cdot \\ \textstyle\bigvee_{i=1}^{k} \Big(\bigwedge_{j\neq i}(p'_j = p_j \wedge s'_j = s_j \wedge v'_j = v_j) \wedge \\ s'_i = 1 - s_i \wedge \big((s_i = 0 \wedge v'_i = \langle d_1, \dots, d_k\rangle \wedge \\ \mathtt{ConfigView}_i(y, p_1, \dots, p_k, d_1, \dots, d_k) \wedge p'_i = p_i) \vee \\ (s_i = 1 \wedge v'_i = v_i \wedge \mathtt{Move}^{\phi}_i(y, p_1, \dots, p_k, d_1, \dots, d_k, p'_i))\big)\Big)\end{gathered}$$

To prove the correctness of this formula for an asynchronous configuration $(\mathbf{p}, \mathbf{s}, \mathbf{V})$ with k robots we make the analogy between the flags $\mathbf{L}$ and $\mathbf{M}$ and the naturals 0 and 1, which means that in the definition of the vector $\mathbf{s} \in \{\mathbf{L}, \mathbf{M}\}^k$, we encode $\mathbf{L}$ by 0 and $\mathbf{M}$ by 1 and we then apply the definition of $\rightarrow_{as}$.

One can also express the fact that one configuration is a predecessor of the other in a straightforward way.

It is as well possible to check whether a protocol ϕ over k robots fits into the hypothesis of Corollary 1, i.e. whether it is uniquely-sequentializable. We define the formula $\mathtt{UniqSeq}_\phi$ that is satisfiable if and only if ϕ is uniquely-sequentializable.

$$\begin{gathered}\mathtt{UniqSeq}_\phi := \neg\exists y . p_1, \dots, p_k, p'_1, \dots, p'_k \cdot \\ \textstyle\bigvee_{i\neq j, 1\leq i,j\leq k} (\mathtt{Move}^{\phi}_i(n, p_1, \dots, p_k, p'_i) \wedge \mathtt{Move}^{\phi}_j(n, p_1, \dots, p_k, p'_j) \\ \wedge p'_i \neq p_i \wedge p'_j \neq p_j .\end{gathered}$$

Hence we deduce the following statement.

Theorem 5: Checking whether a protocol ϕ is uniquely-sequentializable is decidable.

E. Applications

We have considered the *exclusive perpetual exploration* algorithms proposed by Blin *et al.* [6], and generated the formulae to check that no collision are encountered for different cases. We have used the SMT solver Z3 [12] to verify whether the generated formulae were satisfiable or not. We have been able to prove that, in the synchronous case, the algorithm using a minimum of 3 robots was safe for any ring of size greater than 10 and changing a rule of the algorithm has allowed us to prove that we could effectively detect bugs in the Algorithm. In fact, in this buggy case, the SMT solver provides a configuration leading to a collision after one step. We have then looked for absence of collision for the algorithms using a maximum number of robots, always in the synchronous case. Here, the verification was not parametric as the size of the ring is fixed and depends on the number of robots (it is exactly 5 plus the number of robots). The algorithm in [6] was designed to work for any number of robots k odd and co-prime with $k+5$. However, we found bugs for $k = 7, 9$. Note that for 11 robots, the SMT solver Z3 was taking more than 10 minutes and we did not let him finish its computation. We observe that when there is a bug, the SMT solver goes quite fast to generate a bad configuration but it takes much more time when the algorithm is correct. The files containing the SMT formulae are all available on the webpage [15] in the SMTLIB format.

V. Conclusion

We have addressed two main problems concerning formal verification of protocols of mobile robots, and answered the open questions regarding decidability of the verification of such protocols, when the size of the ring is given as a parameter of the problem. Note that in such algorithms, robots can start in any position on the ring. Simple modifications of the proofs in this paper allow to obtain undecidability of both the reachability and the safety problem in any semantics, when the starting configuration of the robots is given. Hence we give a precise view of what can be achieved in the automated verification of protocols for robots in the parameterized setting, and provide a means of partially verifying them. Of course, to fully demonstrate the correctness of a tentative protocol, more properties are required (like, all nodes are visited infinitely often) that are not handled with our approach. Nevertheless, as intermediate lemmas (for arbitrary n) are verified, the whole process of proof writing is both eased and strengthened.

An application of Corollary 1 and Theorem 5 deals with robot program synthesis as depicted in the approach of Bonnet *et al.* [7]. To simplify computations and save memory when synthesizing mobile robot protocols, their algorithm only generates uniquely-sequentializable protocols (for a given k and n). Now, given a protocol description for a given n, it becomes possible to check whether this protocol remains uniquely-sequentializable for any n. Afterwards, regular safety properties can be devised for this tentative protocol, for all models of computation (that is, FSYNC, SSYNC, and ASYNC). Protocol design is then driven by the availability of a uniquely-serializable solution, a serious asset for writing handwritten proofs (for the properties that cannot be automated).

Last, we would like to mention possible applications of our approach for problems whose core properties seem related to reachability only. One such problem is exploration with stop [5]: robots have to explore and visit every node in a network, then stop moving forever, assuming that all robots initial positions are distinct. All of the approaches published for this problem make use of *towers*, that is, locations that are occupied by at least two robots, in order to distinguish the various phases of the exploration process (initially, as all occupied nodes are distinct, there are no towers). Our approach still makes it possible to check if the number of created towers remains acceptable (that is below some constant, typically 2 per block of robots that are equally spaced) from any given configuration in the algorithm, for any ring size n. As before, such automatically obtained lemmas are very useful when writing the full correctness proof.

References

[1] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[2] C. Auger, Z. Bouzid, P. Courtieu, S. Tixeuil, and X. Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In *Proc. of SSS'13*, volume 8255 of *LNCS*, pages 178–186. Springer, 2013.

[3] T. Balabonski, A. Delga, L. Rieg, S. Tixeuil, and X. Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. In *Proc. of SSS'16*, volume 10083 of *LNCS*, pages 7–19. Springer, 2016.

[4] B. Bérard, P. Courtieu, L. Millet, M. Potop-Butucaru, L. Rieg, N. Sznajder, S. Tixeuil, and X. Urbain. Formal Methods for Mobile Robots: Current Results and Open Problems. *Int. J. Inform. Soc.*, 7(3):101–114, 2015. Invited Paper.

[5] B. Bérard, P. Lafourcade, L. Millet, M. Potop-Butucaru, Y. Thierry-Mieg, and S. Tixeuil. Formal verification of mobile robot protocols. *Distr. Comp.*, 2016.

[6] L. Blin, A. Milani, M. Potop-Butucaru, and S. Tixeuil. Exclusive perpetual ring exploration without chirality. In *Proc. of DISC'10*, volume 6343 of *LNCS*, pages 312–327. Springer, 2010.

[7] F. Bonnet, X. Défago, F. Petit, M. Potop-Butucaru, and S. Tixeuil. Discovering and assessing fine-grained metrics in robot networks protocols. In *Proc. of SRDS'14*, pages 50–59. IEEE Press., 2014.

[8] I. Borosh and L. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *Amer. Math. Soc.*, 55:299–304, 1976.

[9] P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Impossibility of Gathering, a Certification. *Inf. Process. Lett.*, 115:447–452, 2015.

[10] P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Certified universal gathering in \mathbb R ^2 for oblivious mobile robots. In *Proc. of DISC'16*, volume 9888 of *LNCS*, pages 187–200. Springer, 2016.

[11] G. D'Angelo, G. D. Stefano, A. Navarra, N. Nisse, and K. Suchan. A unified approach for different tasks on rings in robot-based computing systems. In *Proc. of IPDPSW'13*, pages 667–676. IEEE Press., 2013.

[12] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[13] S. Devismes, A. Lamani, F. Petit, P. Raymond, and S. Tixeuil. Optimal Grid Exploration by Asynchronous Oblivious Robots. In *Proc. of SSS'12*, volume 7596 of *LNCS*, pages 64–76. Springer, 2012.

[14] H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of a mobile robots perpetual exploration algorithm. In *Proc. of SOFL+MSVL, Revised Selected Papers*, volume 10189 of *LNCS*, pages 201–219, 2016.

[15] https://www.irif.fr/~sangnier/robots.html.

[16] P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

[17] P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synt. Lect. Distr. Comp. Th. Morgan & Claypool Publishers, 2012.

[18] E. Kranakis, D. Krizanc, and E. Markou. *The Mobile Agent Rendezvous Problem in the Ring*. Synt. Lect. Distr. Comp. Th. Morgan & Claypool Publishers, 2010.

[19] R. Mayr. Undecidable problems in unreliable computations. *Theoret. Comput. Sci.*, 297(1-3):337–354, 2003.

[20] L. Millet, M. Potop-Butucaru, N. Sznajder, and S. Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *Proc. of SSS'14*, volume 8756 of *LNCS*, pages 237–251. Springer, 2014.

[21] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[22] S. Rubin, F. Zuleger, A. Murano, and B. Aminof. Verification of asynchronous mobile-robots in partially-known environments. In *Proc. of PRIMA'15*, volume 9387 of *LNCS*, pages 185–200. Springer, 2015.

[23] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999.

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

Automated Repair By Example for Firewalls

William T. Hallahan, Ennan Zhai, Ruzica Piskac
Yale University
{william.hallahan, ennan.zhai, ruzica.piskac}@yale.edu

***Abstract*—Firewalls are widely deployed to manage enterprise networks. Because enterprise-scale firewalls contain hundreds or thousands of rules, ensuring the correctness of firewalls – that the rules in the firewalls meet the specifications of their administrators – is an important but challenging problem. Although existing firewall diagnosis and verification techniques can identify potentially faulty rules, they offer administrators little or no help with automatically fixing faulty rules. This paper presents FireMason, the first effort that offers automated repair by example for firewalls. Once an administrator observes undesired behavior in a firewall, she may provide input/output examples that comply with the intended behaviors. Based on the examples, FireMason automatically synthesizes new firewall rules for the existing firewall. This new firewall correctly handles packets specified by the examples, while maintaining the rest of the behaviors of the original firewall. Through a conversion of the firewalls to SMT formulas, we offer formal guarantees that the change is correct. Our evaluation results from real-world case studies show that FireMason can efficiently find repairs.**

I. Introduction

Firewalls play an important role in today's individual and enterprise-scale networks. A typical firewall is responsible for managing all incoming and outgoing traffic between an internal network and the rest of the Internet by accepting, forwarding, or dropping packets based on a set of rules specified by its administrators. Because of the central role firewalls play in networks, small changes can propagate unintended consequences throughout the networks. This is especially true in increasingly large and complex enterprise networks.

A single line in a firewall could, for example, allow anyone to access production services, and therefore it is critical to ensure the *correctness* of firewall rules. Broadly speaking, a firewall is correct if the rules of that firewall meet the specification of its administrator. There have been many efforts that aim to check the correctness of firewall rules through techniques such as firewall analysis [23], [30], verification [22], and root-cause troubleshooting [26], [31], [34]. For instance, systems like Margrave [26] and Fang [23] build an event tree recording states of an observed error, and backtrack through it to find the root causes.

While existing tools can identify the cause of an error, administrators still have to manually find an effective repair to the firewall so that it meets the specification. We propose the first framework, called FireMason, that not only detects errors in firewall behaviors, but also automatically repairs the firewall. Specifically, a user provides a list of examples of packet routing (*e.g.*, all packets with a certain source IP address should be dropped) to describe what the firewall should do. The current firewall might or might not route the packets as specified in the examples. Given the complexity of enterprise-scale networks, finding such a repair requires considerable expertise on the part of the administrator. To the best of our knowledge, there is no existing effort that automates firewall repair.

The main challenge of firewall repair is to show that a generated firewall is indeed repaired and that new rules do not change the routing of packets which are not described by the given examples. We employ an SMT solver for this task. In a nutshell, FireMason translates a given firewall into a sequence of first-order logic formulas falling into the EUF+LIA logic [25], thus allowing us to use an SMT solver for reasoning about the firewalls. By using SMT solvers, FireMason provides formal guarantees that the repaired firewalls satisfy two important properties:

- Those packets described in the examples will be routed in the repaired firewall, as specified.
- All other packets will be routed by the repaired firewall exactly as they were in the original firewall.

Taken together, these two properties allow administrators confidence that the repairs had the intended effect.

Previous work has modeled firewalls using less expressive logics. For example, Zhang *et al.* [34] use SAT and QBF formulas, while Margrave [26], uses first-order relational logic (specifically, through the use of KodKod [30]). By using our formalism we are able to check some important and widely used, but previously out-of-scope, properties. In particular, the ability to reason about linear integer arithmetic with an SMT solver is invaluable in handling *rate limits*. Rate limits, which are frequently used in all modern firewalls, put a restriction on the number of packets matched in a given amount of time. Using SMT solvers we are able to efficiently reason about limiting rules. Due to the complexity of modeling limits, no previous work has considered firewalls with such rules. Such rules say, for example, that we can only accept 6 packets per minute from a certain IP address. As before, the user provides a list of examples, but with relative times. This requires reasoning about the priorities and permissions of each firewall entry, as well as the temporal patterns of the incoming packets.

Furthermore, FireMason is also a stand-alone verification tool, that can either prove that a given specification holds, or produce counterexamples.

We evaluated our tool using real-world firewall issues, and observed that FireMason is able to efficiently generate correct firewalls meeting administrators' examples, without introducing any new problems. In addition, our evaluations show that FireMason scales well to enterprise-scale networks.

In summary, this paper makes the following contributions. We developed a formalism to model firewalls and their behavior. This formalism allows us to use SMT solvers. By using them we can easily prove formal guarantees for verification and repair,

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

thus making FireMason the first work capable of automatically repairing firewalls based on easily specified examples. This allows administrators to conveniently specify their desired behaviors, and automate the repair process. Additionally, by using SMT solvers we can efficiently reason about limit rules, which were not considered by any of existing tools. Finally, we built a workable system that scales well with real-world examples and larger-scale datasets.

II. Preliminaries

Repair by Example. In this paper, we introduce the *repair by example* paradigm, which repairs faulty code so that it satisfies the given examples. In some ways, this resembles the programming by example paradigm [14], [21]. However, in programming by examples, the output is code which generalizes the given input examples. On the other hand, in the repairing by example paradigm the input is both an existing program and a set of examples. The goal is to adjust the input program to satisfy the examples, but otherwise to have a minimal effect on the programs behavior. This allows a user to easily specify instances of faulty behavior, but have confidence that the program will continue to function as it did before. With repair by example, it is important that the effect of the changes is constrained, whereas in programming by example there is no such restriction.

ACL-Based Firewalls. We focus on one of the most representative types of firewalls, Access Control List-based firewalls (or ACL-based firewalls). ACL-based firewalls, such as iptables [4], Juniper [18], and Cisco firewalls [13], are widely used in practice. A typical ACL-based firewall contains an ordered list of *rules*, each of which has *criteria* and an *action*. A criterion describes which preconditions need to hold for the action to take place (*e.g.*, dropping or accepting a packet) [28]. When a network packet is received by an ACL-based firewall, the packet is evaluated against all the rules according to the order in which they appear. After the firewall finds the first rule whose criteria is satisfied by the packet, it performs the corresponding action. The criteria in a rule may refer to properties of the packet that is currently being processed, or to information tracked by the firewall. For instance

```
iptables -A INPUT -p 16 -s 123.23.12.1 -j DROP
```

has criteria denoting packets with a protocol of 16 and a source IP address of `123.23.12.1`, and an action specifying those packets should be dropped.

Actions are either *terminating* or *non-terminating*. Terminating actions end the packet's traversal (for example, once a packet is accepted or dropped, it no longer needs to check more rules in the ACL). Non-terminating actions (such as printing to a log file) allow a packet to continue traversing the ACL rules and finding a match for more rules. An action might also refer to another ACL, which then needs to be used to evaluate the packet. We refer to this as a *jump* to a different ACL.

The ACL jumps can not form a loop. That is, if ACL $\mathcal{A}_1$ contains a jump to ACL $\mathcal{A}_2$, there can be no jumps from $\mathcal{A}_2$ back to $\mathcal{A}_1$. However, suppose a packet is evaluated against all rules in an ACL $\mathcal{A}_2$ and does not match any rule with a terminating action. The packet will then continue being evaluated at the next rule in $\mathcal{A}_1$. If the packet started in $\mathcal{A}_1$, and the packet does not match any rule in $\mathcal{A}_1$ with a terminating action, the packet will be routed according to the *policy* of $\mathcal{A}_1$. The policy is the default action on packets that start in a given ACL, and must be to either accept or drop the packet [9].

Rate Limiting Rules. When an administrator wants to restrict the amount of packets matching a certain rule, a rate limit can be specified for that rule. We call a firewall with such rules a *rate limiting firewall*. In many firewalls, including iptables [9], Juniper [18], and Cisco [13] firewalls, a *limit* is a criterion that specifies how frequently a rule can be matched. A limit is implemented as a counter l, and allows a match of a rule only if $l > 0$. A limit has two parameters: an average rate of packets per some time unit, *ra*, and a burst limit, b. Whereas other criteria are based solely on evaluating a single packet, a limit requires the firewall to maintain its counter, and hence warrants special consideration.

Firewalls use the token bucket algorithm [29] to determine if a packet should be dropped or further processed. The counter l decrements when a packet matches the rule, and increments every $1/ra$ time units. The counter can never exceed the burst limit b. The next example shows how limits work in practice.

Example. Suppose that we set a limit on incoming packets, with $ra = 1$ packet / second and $b = 5$ packets. The firewall is initialized with $l = b = 5$. If we do not exceed the limit, we will accept incoming packets. If we do exceed it, we will drop them. Suppose in the first, third, and fourth seconds after initialization, we receive 1 packet, 3 packets, and 4 packets, respectively. We receive no packets during the second second.

At the end of the first second, $l = 5 - 1 = 4$, since 1 packet arrives. At the beginning of the second second, l will be incremented back to 5. The counter l will not be incremented at the beginning of the third second, since if it was incremented it would exceed the burst limit. However, since 3 packets arrive during the third second, by the end of the third second $l = 2$. At the beginning of the fourth second, l is incremented again to 3. During the fourth second, 4 packets arrive. The first 3 will be accepted, but will result in the counter being decremented, to $l = 3 - 3 = 0$. Since $l = 0$ when the fourth packet arrives, that packet can not match the limit. Therefore, it is dropped.

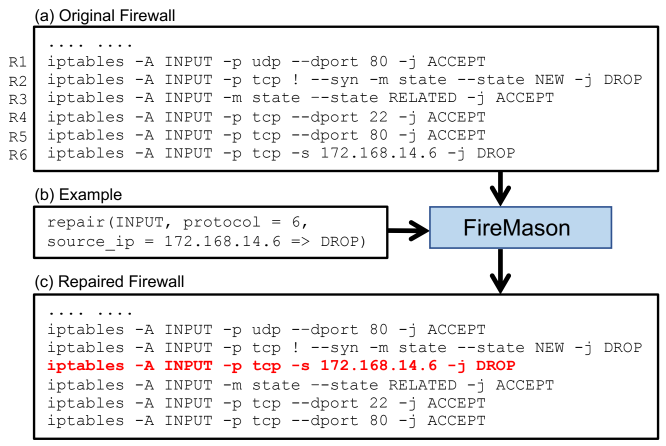
III. Motivating Examples

Stateless Example. We start to demonstrate the functionality of FireMason with a problem inspired by a StackExchange post [1], shown in Figure 1. An administrator is maintaining firewall rules written in iptables [4], one of the most representative firewall script languages. The firewall initially contained rules labeled R1 to R5.

If the administrator wants to block TCP requests coming from the IP address `172.168.14.6`, she may try expressing that as a rule and putting it at the end of the current firewall, cf. rule R6 in Figure 1. Such an action is very common in enterprise-scale firewall management, because administrators prefer appending a new rule to the existing rules [20].

FireMason can be used as a standard firewall analysis tool. To test her changes, the administrator can execute the query:

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.



(a) Original Firewall

```
   .... ....
R1 iptables -A INPUT -p udp --dport 80 -j ACCEPT
R2 iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
R3 iptables -A INPUT -m state --state RELATED -j ACCEPT
R4 iptables -A INPUT -p tcp --dport 22 -j ACCEPT
R5 iptables -A INPUT -p tcp --dport 80 -j ACCEPT
R6 iptables -A INPUT -p tcp -s 172.168.14.6 -j DROP
```

(b) Example

```
repair(INPUT, protocol = 6,
source_ip = 172.168.14.6 => DROP)
```

(c) Repaired Firewall

```
.... ....
iptables -A INPUT -p udp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
iptables -A INPUT -p tcp -s 172.168.14.6 -j DROP
iptables -A INPUT -m state --state RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

Figure 1: An example of a firewall repair problem.

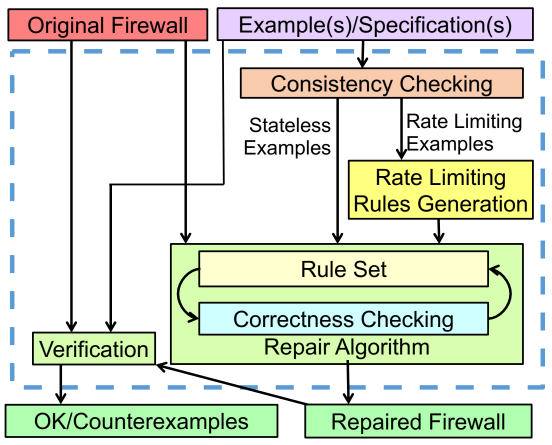


Figure 2: The workflow overview of FireMason.

```
verify(INPUT, protocol = tcp,
source_ip = 172.168.14.6 => DROP)
```

FireMason reports to the administrator that the specification is violated, and gives an example of a packet that will be incorrectly routed (For example, a TCP packet with the SYN flag set, a source ip address of 172.168.14.6, and a destination port of 22. Such a packet would be accepted by R3 or R4).

Knowing that her repair does not work as intended, the administrator can also use FireMason as a repair tool. She provides an example of what should be changed in the firewall and invokes FireMason as shown in Figure 1 (b).

FireMason returns a repaired firewall, Figure 1 (c), to the administrator. The new rule is positioned close to similar rules, namely, those rules related to the TCP protocol. This positioning is very important. While one may argue that directly appending a rule to the top of firewall can also make the firewall behave correctly (in terms of functionality), such a way would, unfortunately, destroy the structure and organization of the firewall. Much like traditional code, keeping the firewall rules organized is important to facilitate later understanding and maintaining. Most importantly though, the rule is positioned so any packet matching the example is guaranteed to be dropped. R1 specifies a protocol other than TCP, and so never matches the example. The example could match R2, but R2 drops any matching packet anyway.

Rate Limiting Example. We next show how an administrator can use FireMason to add/repair rate limiting rules. To the best of our knowledge no existing firewall analysis tools can address this problem. Suppose an administrator wants to allow TCP connections with the SYN flag set once every 10 seconds (a task inspired by a forum post on StackExchange [6].) To do this, the administrator may provide a sequence of example packets and relative times, in seconds:

```
repair(INPUT, SYN, time = 0 => ACCEPT;
       INPUT, SYN, time = 5 => DROP;
       INPUT, SYN, time = 10 => ACCEPT)
```

As a result FireMason creates and inserts two new rules:

```
iptables -A INPUT -m limit --limit 6/minute \
--limit-burst 1 -p tcp --tcp-flags SYN SYN -j ACCEPT
iptables -A INPUT -p tcp --tcp-flags SYN SYN -j DROP
```

This limit satisfies the administrator's requirement. Only one TCP SYN packet can be received every 10 seconds.

IV. System Design

Figure 2 shows the overview of FireMason's workflow. FireMason takes as input a firewall and a user command, which can be either a verification command or a repair command and contains a list of examples.

FireMason first translates the firewall and examples into a set of formulas belonging to a fragment of first-order logic. The translation (described in Sec. IV-A) produces two sets of EUF+LIA formulas [25], which means we can use an SMT solver to reason about firewalls.

The verification process (described in Sec. IV-B) checks if the rules specified in the examples are violated by the new firewall. If there are such rules, FireMason reports counterexamples to the user.

The repair process first checks consistency of the input examples and reports to the user if they are contradictory (Sec. IV-D). This also allows us to detect sets of examples that can be used to generate rate limiting rules. FireMason creates any needed rate limiting rules to handle provided examples. (Sec. IV-F). FireMason next runs the repair algorithm (Sec. IV-G). Finally, FireMason adds the rules to the firewall (Appendix, Sec. IV-C), checks if there are redundant rules in the newly generated firewall (Sec. IV-H), and outputs a correct firewall.

A. *Encoding Firewalls and Examples as FOL Formulas*

Translating Examples. FireMason starts with a list of examples provided by the user, either for a verification or a repair process. Those examples are expressed using the grammar:

$$\begin{aligned}
comm &:= \mathsf{verify}(\{(acl, rule)\}^+) \mid \mathsf{repair}(\{(acl, rule)\}^+)\\
rule &:= precon^+ \Rightarrow action\\
precon &:= \mathsf{protocol} = \textsc{Int} \mid \mathsf{source\_ip} = \textsc{IP\_Address}\\
&\quad \mid \mathsf{destination\_port} = \textsc{Int} \mid \ldots \mid \mathsf{not}\ \ precon\\
action &:= \textsc{Accept} \mid \textsc{Drop}\\
acl &:= \textsc{String} \qquad\qquad \backslash\backslash \text{ ACL Name}
\end{aligned}$$

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

We represent every example by a tuple (a, r, t), where a is the name of the ACL to which the rule r applies, and t is the time given in the example. If no time was given, we set $t = \emptyset$. This tuple is then used in FireMason's algorithms. For instance, the example $\mathsf{repair}(\mathsf{protocol} = 16, \mathsf{time} = 5 \Rightarrow \textsc{Accept})$ is translated to $(\text{INPUT}, \mathsf{protocol} = 16 \Rightarrow \textsc{Accept}, 5)$.

Translating Firewall scripts. Broadly speaking, FireMason describes a firewalls behavior with a sequence of first-order logic formulas. The translation results in formulas that are amenable for reasoning with a SMT solver. Such encoding has two benefits: the computational burden of checking consistencies or finding redundant rules is done by a solver. In addition, we can easily formalize that the repaired firewall is indeed repaired and that only packets described by the examples will be treated differently and according to the specification.

While the majority of the rules could be easily translated to first-order formulas, one obstacle is when the firewall contains jumps. This becomes an issue especially when the ACL also uses limits. Consider, for example, an ACL that has at least two jumps to an ACL A_1. Let us assume that the ACL A_1 has some limit rules. If a packet has to go through both the jumps, then when it reaches the limit in A_1 the second time, the limit in A_1 will have counted the packet twice.

We introduce a data structure, called a *FirewallMap*, which simplifies modeling of jumps and limits. A FirewallMap $\mathcal{M}$ maps unique IDs (we use natural numbers) to tuples of ACL names and lists of the ACLs rules. A rule is modeled as an implication, where a set of criteria implies an action. Possible actions are ACCEPT, DROP, and $\mathsf{GO}(a)$. GO is parameterized by a natural number a, and represents a jump to the ACL with ID a. In the FirewallMap $\mathcal{M}$ there is at most one GO referring to a particular ACL ID. Every rule in $\mathcal{M}$ is assigned a tuple (a, r), where a is an ID of the ACL where the rule appears and r is an ID of the rule in that ACL. This way there exists a single unique path through the FirewallMap to reach any individual rule. Without this property, it would be significantly more difficult to correctly model the order in which rules must be checked. Any ACL jumped to from more than one place in the original firewall is duplicated and assigned multiple IDs. The ACL mapped to by each of these IDs is identical, except any GOs in them must also have different IDs. We refer to these duplicated ACLs as *equivalent* to each other.

Language for Encoding Firewall Behavior into Formulas. We now describe a first-order language that we use to model firewalls and packets. Most of these predicates take a FirewallMap $\mathcal{M}$ as an argument. One can think of $\mathcal{M}$ as a firewall script.

Table I lists a selection of those predicates, functions, and their meanings. FireMason uses these functions and predicates to encode the firewall. For example, if rule r in ACL a in a FirewallMap $\mathcal{M}$ had criteria specifying that it matched a packet p with protocol 17 and destination port 8, then FireMason translates that as follows:

$$\begin{aligned}&\mathsf{matches\_criteria}(\mathcal{M}, p, a, r)\\ &\qquad\Leftrightarrow (\mathsf{protocol}(p) = 17 \land \mathsf{destination\_port}(p) = 8)\end{aligned}$$

Table II shows some axioms describing general relationships between the predicates and functions, and encoding actual firewall behavior. All formulas in the table are implicitly universally quantified, with additional guards $0 \le p < \mathsf{max\_packets}$ and $\mathsf{valid\_rule}(\mathcal{M}, a, r)$. Since the sets of values for $\mathcal{M}, p, a,$ and r are finite, these formulas (as well as the definitions of $\mathsf{reaches\_end}$, $\mathsf{reaches\_return}$, $\mathsf{reaches\_exit}$, and $\mathsf{matches\_rule}$ from Table I) can be finitely instantiated. Thus, no universal quantifiers are needed, and we encode the firewalls in the decidable EUF+LIA logic [25].

Modeling Limits. Limits have two attributes: an average rate *ra* in packets per time unit, and a burst limit of b packets. Each limit also uses a counter to decide if a packet can match the rule. Intuitively, it may seem one could easily model the behavior of a limit using linear integer arithmetic. However, *ra* might not be an integer when the units are converted to seconds. For example, 31 packets per minute is $.51\overline{6}$ packets per second. Therefore, we introduce a new *sub* variable, which represents the time unit used by the limit, converted to seconds. For example, a limit with an average rate of 31 packets per minute and a burst of 10 will be assigned $ra = 31$, $sub = 60$, and $b = 600$ in the formula. Essentially, this corresponds to multiplying the whole formula by sub, to reduce the problem to integers. *ra* is now 31 *tokens* per second, we have a maximum of 600 tokens, and we require 60 tokens to send a single packet.

To have a correct counter of the number of packets, in our model we assign to each limit from the firewall two integer IDs, a main ID i and a secondary ID j. Limits for the same rule in equivalent ACLs all have the same main ID. The secondary IDs start from 0, and they increase every time a packet could meet that limit. We define two functions, $\mathsf{counter\_pre}(\mathcal{M}, i, j, p)$ and $\mathsf{counter\_post}(\mathcal{M}, i, j, p)$, parameterized by the limit's main and secondary IDs, and the packet ids. They are used to track the value of the counter at any given point in time. $\mathsf{counter\_pre}(\mathcal{M}, i, j, p)$ is the value of counter (i, j) immediately before packet p reaches the rule containing that limit. $\mathsf{counter\_post}(\mathcal{M}, i, j, p)$ is the value of that counter immediately after. To check if a limit will allow a packet to match, we check if $\mathsf{counter\_pre}(\mathcal{M}, i, j, p) \ge sub$.

The SMT formulas related to computation of limits are given in Table III Note that, since we multiply ra and $\Delta t(p)$, we must know one of their values for this formula to be in LIA. Fortunately, when reading a limit from an existing firewall script we know *ra*. In Sec. IV-F we explain how $\Delta t(p)$ is known in advance from the examples, so we can obtain ra from the SMT solver.

B. Firewall Verification

Since firewalls are not annotated with standard specifications, systems for verifying firewalls, such as Margrave [26], verify firewalls against user provided queries. When performing the verification process, FireMason also checks if the given examples violate the firewall rules.

We first explain the verification process for examples without time (limit) constraints. Given an example, $e = (n, c \Rightarrow act, \emptyset)$, and a firewall $\mathcal{M}$, we verify e against $\mathcal{M}$ by showing that the following formula $\mathcal{F}$ is valid:

Table I: Partial list of predicates and functions used to model firewalls.

Predicate	**Meaning of the predicate**
$\mathsf{valid\_acl}(\mathcal{M}, a)$	There exists an ACL with ID a in FirewallMap $\mathcal{M}$
$\mathsf{valid\_rule}(\mathcal{M}, a, r)$	$\mathsf{valid\_acl}(\mathcal{M}, a)$ and there exists a rule with ID r in a
$\mathsf{matches\_criteria}(\mathcal{M}, p, a, r)$	Packet p satisfies the criteria of rule r in ACL a in FirewallMap $\mathcal{M}$
$\mathsf{reaches}(\mathcal{M}, p, a, r)$	Packet p reaches rule r in ACL a in FirewallMap $\mathcal{M}$
$\mathsf{starting\_acl}(\mathcal{M}, a)$	Returns true if ACL a is not jumped to from some other ACL
$\mathsf{is\_go}(act)$	Returns whether the action *act* is $\mathsf{GO}(a)$ for some arbitrary a
$\mathsf{reaches\_end}(\mathcal{M}, p, a)$	$\mathsf{reaches}(\mathcal{M}, p, a, \mathsf{acl\_length}(\mathcal{M}, a))$
$\mathsf{reaches\_return}(\mathcal{M}, p, a)$	$\mathsf{reaches}(\mathcal{M}, p, a, r) \wedge \mathsf{rule\_action}(\mathcal{M}, a, r) == RETURN$
$\mathsf{reaches\_exit}(\mathcal{M}, p, a)$	$\mathsf{reaches\_end}(\mathcal{M}, p, a) \vee \mathsf{reaches\_return}(\mathcal{M}, p, a)$
$\mathsf{matches\_rule}(\mathcal{M}, p, a, r)$	$\mathsf{matches\_criteria}(\mathcal{M}, p, a, r) \wedge \mathsf{reaches}(\mathcal{M}, p, a, r)$
$\mathsf{matches\_example}(p, e)$	Packet p matches the criteria of an example e
$\mathsf{protocol}(p)$	The protocol of packet p
$\mathsf{acl\_length}(\mathcal{M}, a)$	Returns the number of rules in ACL a
$\mathsf{max\_packets}$	Returns the maximum number of packets to be considered
$\mathsf{terminates\_with}(\mathcal{M}, p)$	Returns if the FirewallMap $\mathcal{M}$ would ACCEPT or DROP packet p
$\mathsf{rule\_action}(\mathcal{M}, a, r)$	Returns the action of rule r in ACL a in FirewallMap $\mathcal{M}$
$\mathsf{insert\_rule}(\mathcal{M}, R, a, r)$	Returns FirewallMap $\mathcal{M}$, but with rule R inserted in ACL a as rule r
$\mathsf{equivalent}(\mathcal{M}, n)$	Returns the set of IDs in FirewallMap $\mathcal{M}$ for the ACL named n
$\mathsf{go\_acl}(act)$	For $act = \mathsf{GO}(a)$ returns a, otherwise -1

Table II: Formulas to model a firewall, and packets that firewall is processing.

$a_1 \neq a_2 \wedge \mathsf{reaches}(\mathcal{M}, p, a_1, 0) \wedge \mathsf{starting\_acl}(\mathcal{M}, a_1) \wedge \mathsf{starting\_acl}(\mathcal{M}, a_2) \implies \neg\mathsf{reaches}(\mathcal{M}, p, a_2, 0)$
$\mathsf{reaches}(\mathcal{M}, p, a, r) \wedge \neg\mathsf{matches\_criteria}(\mathcal{M}, p, a, r) \implies \mathsf{reaches}(\mathcal{M}, p, a, r+1)$
$\mathsf{reaches}(\mathcal{M}, p, a, r+1) \implies \mathsf{reaches}(\mathcal{M}, p, a, r)$
$\mathsf{matches\_rule}(\mathcal{M}, p, a, r) \wedge \mathsf{is\_go}(\mathsf{rule\_action}(\mathcal{M}, a, r)) \equiv \mathsf{reaches}(\mathcal{M}, p, \mathsf{go\_acl}(\mathsf{rule\_action}(\mathcal{M}, a, r)), 0)$
$\mathsf{matches\_rule}(\mathcal{M}, p, a, r) \wedge \mathsf{is\_go}(\mathsf{rule\_action}(\mathcal{M}, a, r)) \implies \mathsf{reaches\_exit}(\mathcal{M}, p, \mathsf{go\_acl}(\mathsf{rule\_action}(\mathcal{M}, a, r))) = \mathsf{reaches}(\mathcal{M}, p, a, r+1)$
$\mathsf{reaches}(\mathcal{M}, p, a, r) \wedge \neg\mathsf{is\_go}(\mathsf{rule\_action}(\mathcal{M}, a, r)) \wedge \mathsf{rule\_action}(\mathcal{M}, a, r) \neq \mathsf{RETURN} \wedge \neg\mathsf{terminating}(\mathcal{M}, a, r) \implies \mathsf{reaches}(\mathcal{M}, p, a, r+1)$
$\mathsf{reaches\_return}(\mathcal{M}, p, a) \implies \neg\mathsf{reaches}(\mathcal{M}, p, a, r+1)$
$\mathsf{matches\_rule}(\mathcal{M}, p, a, r) \wedge \mathsf{terminating}(\mathsf{rule\_action}(\mathcal{M}, p, a, r)) \implies \neg\mathsf{reaches}(\mathcal{M}, p, a, r+1)$
$\mathsf{matches\_rule}(\mathcal{M}, p, a, r) \wedge \mathsf{terminating}(\mathsf{rule\_action}(\mathcal{M}, p, a, r)) \implies \mathsf{terminates\_with}(\mathcal{M}, p) = \mathsf{rule\_action}(\mathcal{M}, p, a, r)$
$\mathsf{reaches\_end}(\mathcal{M}, p, a, r) \wedge \mathsf{starting\_acl}(\mathcal{M}, a) \implies \mathsf{terminates\_with}(\mathcal{M}, p) = \mathsf{policy}(\mathcal{M}, a)$

Table III: Logical formulas related to limits, all variables are implicitly universally quantified with additional constraints that rule r in ACL a has a limit with main ID i and secondary ID j, and $0 \leq p <$ max_packets. We use j_max(i) to denote the maximum secondary ID for the limit with main ID i.

$\forall p. p \geq 1 \implies \mathsf{arrival\_time}(p) \geq \mathsf{arrival\_time}(p-1)$
$\Delta t(p) = \begin{cases} \mathsf{arrival\_time}(p) - \mathsf{arrival\_time}(p-1) & \text{if } 1 \leq p < \mathsf{max\_packets} \\ 0 & \text{otherwise} \end{cases}$
$\mathsf{counter\_pre}(\mathcal{M}, i, j, p) = \begin{cases} \mathsf{counter\_post}(\mathcal{M}, i, j-1, p) & \text{if } j \geq 1 \\ \min(\mathsf{counter\_post}(\mathcal{M}, i, \mathsf{j\_max}(i), p-1) + ra * \Delta t(p), b) & \text{if } p \geq 1 \text{ and } j = 0 \\ b & \text{otherwise} \end{cases}$
$\mathsf{counter\_post}(\mathcal{M}, i, j, p) = \begin{cases} \mathsf{counter\_pre}(\mathcal{M}, i, j, p) - sub & \text{if } \mathsf{counter\_pre}(i, j, p) \geq sub \wedge \mathsf{matches\_rule}(\mathcal{M}, p, a, r) \\ \mathsf{counter\_pre}(\mathcal{M}, i, j, p) & \text{otherwise} \end{cases}$

$$\forall p, a.\ a \in \mathsf{equivalent}(\mathcal{M}, n) \wedge \mathsf{reaches}(\mathcal{M}, p, a, 0)$$
$$\wedge\ \mathsf{matches\_example}(p, e) \Rightarrow \mathsf{terminates\_with}(\mathcal{M}, p) = act$$

Formula $\mathcal{F}$ states that every packet arriving to ACL n and satisfying criteria c terminates with action a. Note that when negated, the formula is only existentially quantified.

To verify a list of examples with times, $e_k = (n_k, c_k \Rightarrow act_k, t_k)$, for $0 \leq k \leq N$ we apply a similar procedure. After setting up all packets with appropriate times, the verification condition states that at least one packet does not terminate as desired (expressed already in the negated form):

$$\forall k \exists a. 0 \leq k \leq N \wedge a \in \mathsf{equivalent}(\mathcal{M}, n_k)$$
$$\wedge\ \mathsf{reaches}(\mathcal{M}, p_k, a_k, 0) \wedge \mathsf{matches\_example}(p_k, e_k)$$
$$\wedge\ (\bigvee_{0 \leq j \leq N} \mathsf{terminates\_with}(\mathcal{M}, p_j) \neq act_j)$$

C. Adding Rules

Here we outline how to create rules from the provided examples. We describe in more detail how to create rate limiting rules in Sec. IV-F. We first focus on stateless rules. We use the repair algorithm, Algorithm 2 in Sec. IV-G, to assign each rule a position. After positions are assigned, it is straightforward to add the rules to the firewall. We simply copy the old rules from the original firewall, convert the new rules from our internal language to the iptables language, and insert them at the appropriate positions. For example, the tuple (`INPUT`, `protocol = 6, source_ip = 1.2.3.4 => ACCEPT`, $\emptyset$) would become `iptables -A INPUT -p 6 -s 1.2.3.4 -j ACCEPT`.

D. Consistency Checking

The purpose of consistency checking is both to let the administrator know whether the provided examples contradict each other, and to detect when to invoke the algorithm for addressing limits. Consider the two examples below:

```
repair(INPUT, protocol = 17 => ACCEPT),
repair(INPUT, source_ip = 1.1.0.0/16 => DROP)
```

If a packet with `protocol = 17` and a source IP address of `1.1.1.1` enters the INPUT ACL, it is not clear whether such a packet should be accepted or dropped. We consider these examples *rule inconsistent*.

Formally, we say two examples, $(n_1, c_1 \Rightarrow act_1, t_1)$ and $(n_2, c_2 \Rightarrow act_2, t_2)$ are rule inconsistent if $n_1 = n_2$, $c_1 \wedge c_2$ is satisfiable by a single packet, and $act_1 \neq act_2$. We find the contradictory examples by using an SMT solver and we inform the administrator about ambiguities. Note that this definition makes no reference to time, and handling of rule inconsistent examples with different times will be covered in Sec IV-F.

E. Formal Guarantees for Repaired Firewalls

FireMason offers two guarantees on the behavior of repaired firewalls. The first guarantee is the packets or sequences of packets described by the examples are correctly routed in the repaired firewall. The second guarantee is that the changes have a minimal effect; that is, that the routing of every packet not described by the examples is the same as it was in the original firewall. Together, these guarantees allow an administrator to be confident that the repairs had the intended effect, and only the intended effect.

Here we give formulas which can be used by an SMT solver to check if the formal guarantees hold.

For given examples of the form $e_k = (n_k, crit_k \Rightarrow act_k, \emptyset)$, for $0 \leq k < N$, the first guarantee can be written with Formula (1),

$$\begin{aligned} &\forall k, a. 0 \leq k < N \wedge a \in \mathsf{equivalent}(\mathcal{M}, n_k) \wedge \\ &\mathsf{matches\_example}(k, e_k) \wedge \mathsf{reaches}(\mathcal{M}', k, a, 0) \\ &\quad \implies \mathsf{terminates\_with}(\mathcal{M}', k) = act_k \end{aligned} \tag{1}$$

Now suppose we have examples with relative times, $e_k = (n_k, crit_k \Rightarrow act_k, t_k)$. Without loss of generality, assume that for $k_1 < k_2$, we have $t_{k_1} < t_{k_2}$. In this case we ensure that packets arriving at the appropriate times, with the appropriate criteria, are correctly routed, given that no other packets matching the examples criteria are processed before their arrival. Formally, we write:

$$\begin{aligned} &\forall k, a. 0 \leq k < N \wedge a \in \mathsf{equivalent}(\mathcal{M}, n_k) \\ &\bigwedge_{0 \leq m \leq k} \Big(\mathsf{arrival\_time}(m) = t_m \wedge \mathsf{matches\_example}(m, e_m) \\ &\wedge \mathsf{reaches}(\mathcal{M}, m, a, 0)\Big) \bigwedge_{m' > k} \mathsf{nonexample}(\mathcal{M}, m', k) \\ &\implies \mathsf{terminates\_with}(\mathcal{M}, k) = act_k \end{aligned} \tag{2}$$

where we use the predicate nonexample to determine if the packet p either does not correspond to or arrives after the last relevant example.

$$\begin{aligned} &\mathsf{nonexample}(\mathcal{M}, p, e) = \\ &\quad \forall k, a. 0 \leq k < e \wedge a \in \mathsf{equivalent}(\mathcal{M}, n_k) \implies \\ &\quad t_e < \mathsf{arrival\_time}(p) \vee \neg\mathsf{reaches}(\mathcal{M}, p, a, 0) \\ &\qquad \vee \Big(\bigwedge_{0 \leq m \leq k} \neg\mathsf{matches\_example}(p, e_m)\Big) \end{aligned}$$

The second guarantee, that the changes we make are minimal, is stated as Formula (3):

$$\begin{aligned} &\forall p. \mathsf{terminates\_with}(\mathcal{M}, p) = \mathsf{terminates\_with}(\mathcal{M}', p) \\ &\vee \Big(\exists k, a. 0 \leq k < N \wedge a \in \mathsf{equivalent}(\mathcal{M}, n_k) \\ &\quad \wedge \mathsf{matches\_example}(p, e_k) \wedge \mathsf{reaches}(\mathcal{M}, k, a, 0)\Big) \end{aligned} \tag{3}$$

F. Rate Limiting Rules Generation

Algorithm 1: Limit Generating Algorithm

```
input : $E$, the list of examples, all with relative times,
        optional parameters $minRulesAndLimits$ and
        $minTotalSub$ (both default to $\emptyset$)
output: $r$ a list of rules
1  $E' \leftarrow []$;
2  foreach $(n, r, t) \in E$ do
3      $r_2 \leftarrow r$, with a limit template, consisting of symbolic
       values for $ra$, $b$, $sub$, and $useLimit$, and a Boolean
       $enableRule$ added to the criteria
4      $E'$.append($(n, r_2, t)$);
5  sortByNameByTime($E'$);
6  if $minRulesAndLimits \neq \emptyset$ and $minTotalSub \neq \emptyset$ then
7      Assert $rulesAndLimits < minRulesAndLimits$
       $\vee(rulesAndLimits = minRulesAndLimits \wedge totalSub <$
       $minTotalSub)$
8  Convert $E'$ to SMT formulas, create formulas defining
   score and totalSub, run SMT Solver;
9  $sat \leftarrow$ getSat;
10 if $sat$ = $UNSAT$ then
11     $r \leftarrow$ getRulesFromModel(model);
12     return $r$;
13 else
14     model $\leftarrow$ getModel;
15     $(rulesAndLimits, totalSub) \leftarrow$ getScore(model);
16     call this recursively, to lexicographically minimize
       $(rulesAndLimits, totalSub)$;
```

After the consistency checking, some examples may have to be resolved via rate limiting. Specifically, this is required for rules that are rule inconsistent, but have relative times. Algorithm 1 generates rate limiting rules satisfying these examples. Our algorithm takes a list of rule inconsistent examples, E, each with a time. It returns an ordered list of satisfying rules, which are later inserted into the firewall using Algorithm 2.

Recall that we may express an example as consisting of an ACL name, a rule, and a time. We create E' from E, by adding two criteria to each examples rule. The first is a limit template, which uses variables in place of actual integers for *ra*, *b*, and *sub*. It also has a Boolean variable $useLimit$, which enables and disables the limit. The second criterion is a Boolean, *enableRule*. Packets can match the rule if and only if *enableRule* is true. We will use this template with an SMT solver to search for the solution that requires the fewest limits and rules.

We sort E' into distinct groups according to which ACL the rules are meant to be added to, and then sort each group by ascending time, at line 5. We extract the rules from E' into lists (ACLs) to form a templated FirewallMap $\mathcal{M}$. This allows us to convert to an SMT formula, using exactly the same formulas and logic as in Sec. IV-A.

For each original example, $e_p = (n_p, c_p \Rightarrow act_p, t_p)$, we pick $a \in \mathsf{equivalent}(\mathcal{M}, n_p)$and assert that the packet with ID p matches the requirements of that example:

$$\begin{aligned} &\mathsf{arrival\_time}(p) \wedge \mathsf{matches\_example}(p, e_p) \\ &\wedge \mathsf{reaches}(\mathcal{M}, p, a, 0) \wedge \mathsf{terminates\_with}(\mathcal{M}, p) = act_p \end{aligned} \tag{4}$$

For all the pairs $0 \leq r, q < length(E'), r \neq q$, we check if $c_r \wedge \neg c_q$ is satisfiable by a single packet. For each pair which is, we assert:

$$\neg \mathsf{matches\_example}(r, e_q) \tag{5}$$

The SMT solver can then find values for each *ra*, b, *sub*, u, and *enableRule* that guide the packets as required by the examples. Formula (4) ensures that the found solution satisfies the requirements of the examples sequence. Formula (5) ensures that the SMT solver does not make assumptions about packets criteria that the user likely does not intend. For example, if the administrator provided the examples:

```
repair(
  INPUT, protocol = 17, time = 0 => ACCEPT;
  INPUT, protocol = 17, time = 5 => DROP;
  INPUT, source_ip = 1.1.0.0/16, time = 10 => ACCEPT;
  INPUT, source_ip = 1.1.0.0/16, time = 15 => DROP)
```

Formula (5) would prevent the SMT solver finding a solution that required any of the packets satisfying `protocol = 17 AND source_ip = 1.1.0.0/16`.

Such a model is always possible to find. One valid solution is to set all the *enableRule* to true, all the bursts to 1, and all the rates and subs such that the limit recharging even once takes longer than the total time between the first and last packet arriving. Then, each packet will be sorted according to the rule that came from its modified example.

To make our solution capable of handling more general cases, we assign a lexicographic score to our formula. The first value is calculated by adding the number of limits and the number of non-ignored rules, which we call *rulesAndLimits*. The second value is the sum of the limit's sub values, which we call *totalSub*. We aim to make this score as small as possible. This can be done by repeatedly asserting there exists a formula with a better score. If *(minRulesAndLimits, minTotalSub)* is the current best score, we assert:

Algorithm 2: Rule Adding Repair Algorithm

```
input  : $E$, the list of examples; $\mathcal{M}$, a FirewallMap
output : a FirewallMap with a rule for each $e \in E$ added
1  foreach $(n, newR, t) \in E$ do
2      $a' \leftarrow$ ACL id of an arbitrary representation of the ACL $n$ in $\mathcal{M}$;
3      $res \leftarrow$ SAT ;
4      $maxR \leftarrow \mathsf{acl\_length}(a') - 1$;
5      while $res$ = $SAT$ do
6          Pick $r' \leq maxR$ , using a similarity measure to $newR$;
7          $\mathcal{M}$' $\leftarrow$ insertRule($\mathcal{M}$, $newR$, $a', r'$) ;
8          $res \leftarrow$ SMTCheckCorrectness($\mathcal{M}$, $\mathcal{M}$', $e$);
9          if $res$ = $SAT$ then
10             $maxR \leftarrow r' - 1$;
11     $\mathcal{M} \leftarrow \mathcal{M}$'
```

$$\begin{aligned} &\mathit{rulesAndLimits} < \mathit{minRulesAndLimits} \vee (\mathit{rulesAndLimits} = \\ &\quad \mathit{minRulesAndLimits} \wedge \mathit{totalSub} < \mathit{minTotalSub}) \end{aligned}$$

When the SMT solver returns UNSAT, we can guarantee we found the solution which minimizes the number of rules plus the number of limits used.

There are two small potential problems with this approach, and luckily, both have straightforward solutions. First, recall from Sec. IV-A that the model involves the value of $ra * \Delta t(p)$, but to stay in the theory of LIA, we must avoid multiplying two variables. In that section, there was an assumption that the value of *ra* was known, whereas here it clearly is not. Fortunately, while we do not know the value of *ra*, we can precompute, and fix as a constant, the time difference between neighboring packets, $\Delta t(p)$.

Second, some firewalls languages constrain the value of sub to a fixed list of possible values $s_1, \ldots s_v$. This can be handled through one additional assertion per *sub* value, $\vee_{u=1}^{v} sub = s_u$. This occasionally leads to cases where there is no valid way to generate the limits, but such cases can be detected when the first call to the SMT solver is UNSAT.

G. Repair Algorithms

Given the formulas representing the target firewall and examples, we need to run a repair algorithm to generate a correct firewall based on the examples. We will first consider rule insertion for non-rule inconsistent examples. Then, we will explain how this same algorithm can be used to insert the rate limiting rules found by Algorithm 1. Suppose we have N non-rule inconsistent examples, $e_1 = (n_1, r_1 = (c_1 \Rightarrow act_1), t_1), ..., e_N = (n_N, r_N, t_N)$. Given a firewall represented by a FirewallMap $\mathcal{M}$, our goal is to to find a new FirewallMap $\mathcal{M}'$ which ensures all the examples are satisfied, but that guarantees all non-described packets maintain the same behavior. We also want $\mathcal{M}'$ to be well organized, meaning that "similar rules" all appear together. Our procedure (omitted due to space restrictions) to decide the similarity assigns a score based on the number and kinds of criteria used in the rules, but could be replaced by any desired scoring algorithm.

Consider the k^{th} example, $1 \leq k \leq N$. We express the desired condition with respect to example e_k by instantiating k in Formulas 1 and 3. We then show that Algorithm 2 outputs a firewall which satisfies this condition. For each example $e_i = (n_i, r_i, t_i)$, we take some $a' \in \mathsf{equivalent}(\mathcal{M}, n_i)$ and find the ID r' of the existing rule most similar to r_i in ACL a'. Next we set $\mathcal{M}' = \mathcal{M}$, and run $\mathsf{insert\_rule}(\mathcal{M}', r_i, a', r')$ to insert r_i in all ACLs equivalent to a' at position r' in $\mathcal{M}'$.

We convert both $\mathcal{M}$ and $\mathcal{M}'$ to SMT formulas, and use an SMT solver to check that Formulas 1 and 3 are valid. To do this, we must eliminate the two universal quantifiers that remain after instantiating k. There are only a finite number of values that a may attain - namely, it can only be the values in $\mathsf{equivalent\_to\_name}(\mathcal{M}, a')$. Using this observation, we can easily eliminate the universal quantifier using finite instantiation. Once the formula is only universally quantified by p, we negate it, and try to show that its negation is unsatisfiable.

If the SMT solver does find the formula to be unsatisfiable, we know that the original formula was valid, i.e. the firewall satisfies the considered example. However, if the formula is satisfiable, we search for a different place to insert the rule, that comes before rule r' in ACL a'. We do not consider any rule after this rule, as any route along which $\mathcal{M}$ and $\mathcal{M}'$ could incorrectly diverge would also exist if the new rule was inserted after a'. Also note that the condition is guaranteed to hold if the new rule is inserted as rule 0 in ACL a'; and although this placement is often not ideal for the structure of the firewall, it does guarantee termination.

When rules are from consistent examples, we can insert them in any order. By definition, two consistent examples cannot describe any of the same packets, so it does not matter which corresponding rule comes before the other in the firewall. However, the rules found by Algorithm 1 are rule inconsistent. In this case, insertion of the rules must be done in reverse order of the corresponding example's times. This ensures that the inconsistent rules have the same relative order in E' (from Sec. IV-F) as in $\mathcal{M}'$, and thus we can expect the same behavior from the examples in both E' and $\mathcal{M}'$.

H. Redundant Rule Detection

The final step in repairing the firewall is removing *redundant* rules – that is, rules which cannot be matched by any packet. Thanks to the SMT model, this is straightforward.

As before, the firewall is converted to an SMT formula. Then, for each ACL name and rule ID, n and r, respectively, check that there exists a packet that matches the rule, or some equivalent rule by asserting

$$\exists a'.a' \in \mathsf{equivalent}(\mathcal{M}, n) \land \mathsf{matches\_rule}(\mathcal{M}, p, a', r)$$

If this call returns SAT, then clearly there exists some packet that matches the rule, and the rule is therefore not redundant. If it returns UNSAT, then there was no packet that matched the rule, and it is therefore redundant. In this case, it can be commented out. This does involve a large number of calls to the SMT solver, but these calls tend to be fast.

V. Implementation and Evaluation

FireMason is developed in Haskell and fully implements the design described in Sec. IV. The default firewall language that we support is the iptables language [4], but the framework can be easily extended to other firewall languages, such as Juniper [18] and Cisco firewalls [13]. The syntax of these languages varies, but the semantics are largely the same. Therefore, only the translation step (essentially a parser) needs to be rewritten for a particular language, which means that FireMason can easily be adapted to repair firewalls written in other languages. As an SMT solver we used Microsoft's Z3 [24]. The source code for our implementation is available at https://github.com/BillHallahan/FireMason.

The evaluation was conducted with an Intel Xeon Quad Core HT 3.7 GHz.

Scalability evaluation. We evaluated the scalability of FireMason with regard to real-world network sizes by using three examples as specification, and varying the number of rules in the target firewall between 100 and 500. These firewalls were randomly generated. As shown in Figure 3, FireMason scales well to large-scale firewalls.

One might expect the rate limiting rules insertion to be slower than the non rate limiting rules insertion, due to the additional runtime of Algorithm 1. However, Algorithm 1's runtime depends only on the number of examples, and not on the number of rules in the original firewall, its runtime is constant across the rate limiting tests. In the rate limiting case our three examples result in only two rules to insert, whereas in the non rate limiting case, we insert three rules. Thus, the additional runtime is due to Algorithm 2.

We also evaluated the performance of FireMason for different numbers of provided examples, as shown in Table V. In the stateless case this scales linearly. In the rate limiting case, the time required increases rather sharply as the number of examples generating a single limit increases. However, this is not a major concern, as we have found that a small number of examples is typically sufficient to find an appropriate limit.

Case study: Repairing real-world firewalls. To demonstrate that FireMason can repair real-world firewalls, we found firewall repair problems on Server Fault [7] and Stack Overflow [8]. We recreated each scenario, and generated corrected firewalls using FireMason.

Table IV presents five such problems. We list the examples which an administrator may provide to clarify how the firewall should be repaired and present the resulting repairs to the firewall. We also include the running time, the number of calls to the SMT solver, and the number of rules in the original iptables script.

We manually checked the correctness of each result and compared them to the repairs suggested on the forums. We found that the output returned by FireMason not only fixed the problems, but also avoided any side effects. Furthermore, we manually confirmed the "minimality" of the repairs, in terms of the impact on the firewalls overall behavior. In some cases,

Table IV: Case study: Sampled firewall repair problems and our solutions.

Case Study 1 [1]	An administrator appended a rule `iptables -A INPUT -s 73.143.129.38 -j DROP`, but can still receive packets from `73.143.129.38`.
Input example	`repair(INPUT, source_ip = 73.143.129.38 => DROP)`
Results	1. Remove the appended rule, and insert a new rule `iptables -A INPUT -s 73.143.129.38/32 -j DROP` in front of an original rule `iptables -A INPUT -i lo -j ACCEPT`.
Original Rule Count	11
Repair Time	.109 s
SMT Solver calls	26
Case Study 2 [2]	An administrator wants to allow SSH access from the IP address `71.82.93.101`, but does not know how.
Input examples	1. `repair(INPUT, protocol = 22, source_ip = 71.82.93.101 => ACCEPT),` 2. `repair(INPUT, protocol = 22, not source_ip = 71.82.93.101 => DROP)`
Results	Insert new rules `iptables -I INPUT 0 -p 22 -s 71.82.93.101/32 -j ACCEPT` and `iptables -I INPUT 0 -p 22 ! -s 71.82.93.101/32 -j DROP` in front of an original rule `iptables -I INPUT -p icmp --icmp-type time-exceeded -j ACCEPT`.
Original Rule Count	11
Repair Time	.088 s
SMT Solver calls	23
Case Study 3 [3]	An administrator is trying to limit the number of inbound SSH packets, but it just seems to lock her out.
Input examples	1. `repair(INPUT, protocol = 22, time = 0 => ACCEPT),` 2. `repair(INPUT, protocol = 22, time = 20 => ACCEPT),` 3. `repair(INPUT, protocol = 22, time = 30 => ACCEPT),` 4. ... ... (In total, this repair uses 8 examples, we cannot list all the examples due to limited space)
Results	Insert new rules `iptables -A INPUT -m limit --limit 2/minute --limit-burst 4 -p 22 -j ACCEPT` and `iptables -A INPUT -p 22 -j DROP` at the beginning of the original firewall.
Original Rule Count	9
Repair Time	21.10 s
SMT Solver calls	44
Case Study 4 [6]	A server is attacked by TCP SYN flooding, so the administrator wants a limit on SYN packets per second.
Input examples	1. `repair(INPUT : source_ip = 192.132.209.0/24, SYN, time = 10 => ACCEPT),` 2. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 11 => ACCEPT),` 3. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 12 => ACCEPT),` 4. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 13 => DROP),` 5. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 19 => DROP),` 6. `repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 21 => ACCEPT)`
Results	Append two new rules, `iptables -I INPUT 0 -s 192.132.209.0/24 -p 6 --tcp-flags SYN -j DROP` and `iptables -I INPUT 0 -m limit --limit 6/minute --limit-burst 3 -s 192.132.209.0/24 -p 6 --tcp-flags SYN SYN -j ACCEPT`, to the original firewall.
Original Rule Count	11
Repair Time	6.046 s
SMT Solver calls	42
Case Study 5 [5]	An administrator has the IP address `192.168.1.99`, and wants to SSH to the IP address 192.168.1.15. She appended a rule `iptables -A INPUT -p tcp -i eth0 --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT` but still cannot SSH 192.168.1.15.
Input example	1. `repair(OUTPUT, protocol = , destination_ip = 192.168.1.15 => ACCEPT),` 2. `repair(INPUT, source_ip = 192.168.1.15 => ACCEPT)`
Results	Insert two new rules `iptables -A INPUT -s 192.168.1.15/32 -j ACCEPT` and `iptables -A OUTPUT -d 192.168.1.15/32` in front of the fourth and fifth rules in the original firewall, respectively.
Original Rule Count	4
Repair Time	.054 s
SMT Solver calls	14

Table V: Scalability for number of examples (when inserting into a firewall with 100 rules).

Number of examples	Stateless Time (s)	Rate Limiting Time
3	3.567	2.177
6	4.545	2.004
9	5.804	36.37

FireMason outputs a different solution from the posted solution. After manual comparison, we found that both solutions work correctly, but FireMason's output required adding fewer new rules.

Interestingly, the case studies involving rate limits took significantly longer than those only involving stateless examples. This is not at odds with the results of the scalability evaluation. As shown in table V, for a small number of examples, rate limit rule generation is generally faster, whereas for a larger number of examples, stateless rule generation is faster.

VI. Related Work

This section presents existing efforts on firewall analysis, verification and generation, and discusses why these efforts are not helpful to our target.

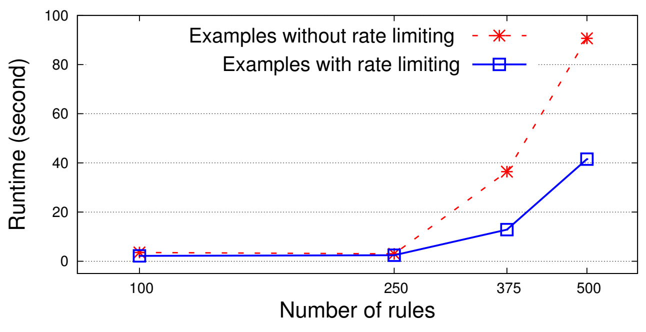


Figure 3: Scalability for number of rules.

Firewall synthesis. Zhang *et al.* [34] proposed a symbolic firewall synthesis approach such that the synthesized firewall has the same behavior as a given firewall, but with the smallest possible number of rules. As this approach focuses on automatically simplifying redundant rules, rather than repairing an observed error, it is not applicable to our goal.

As software defined networks (SDN) have become increas-

ISBN: 978-0-9835678-7-5. Copyright owned jointly by the authors and FMCAD, Inc.

ingly popular, automatic programming approaches for SDN have been proposed [27], [33]. Yuan *et al.* [33] proposed an automatic SDN policy generation approach, named NetEgg, based on a scenario-based programming technique. NetEgg can only generate a new policy, it can not account for the effect of a new policy on existing policies in the network. Furthermore, NetEgg can not synthesize rate limiting rules.

Firewall analysis and verification. Mayer *et al.* [23] developed the first systematic firewall analysis engine, Fang, to analyze diverse properties of firewalls. Fang and its sequel Lumeta [31] allow checking the correctness of firewall configurations by sending their analysis engines queries. Other efforts [10], [16] propose packet-filter based schemes to detect conflicting or violated rules. Frantzen *et al.* [17] and Kamara *et al.* [19] proposed different data-flow based approaches to analyze vulnerability risks in firewalls. Wool [32] conducted a case study on understanding and classifying the configuration errors of firewalls.

The Margrave firewall verification tool [26] encodes firewall rules and queries into first-order logic. It uses KodKod [30] to search for finite state models. Compared with another firewall verification tool, NoD [22], Margrave cannot produce all differences between policies in a compact way, and does not scale for large firewall rule sets.

Firewall testing. El-Atawy *et al.* [15] proposed targeting test packets for better fault coverage. Al-Shaer *et al.* [11] developed a system-wide framework to generate targeted packets and obtain good coverage during firewall testing. Brucker *et al.* [12] proposed a formal firewall conformance testing approach, which uses Isabelle/HOL to generate test-cases from constraint satisfaction problems.

VII. Conclusion

In this paper, we have presented FireMason, an automatic tool for formally verifying and repairing firewalls. To this end, we use a first-order intermediary language to model firewalls, which allows us use of an SMT solver to obtain formal guarantees on the correctness of verification and repair. We showed that FireMason not only generates correctly repairs real-world firewall scripts, but also is able to scale to large-scale firewalls. We hope that this work will inspire more reasoning about firewalls in the formal methods community.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also thank Avi Silberschatz and Arvind Swaminathan for their valuable feedback on the earlier version of this work. This research was supported in part by CCF-1553168.

References

[1] Cannot Block IP Address, http://stackoverflow.com/questions/16142446/why-cant-i-block-an-ip-address-with-iptables-on-debian-6
[2] Good iptables starting rules for a webserver, http://serverfault.com/questions/118669/good-iptables-starting-rules-for-a-webserver
[3] How can I rate limit SSH connections with iptables, http://serverfault.com/questions/298954/how-can-i-rate-limit-ssh-connections%20-with-iptables
[4] iptables, http://ipset.netfilter.org/iptables.man.html
[5] iptables issue cannot SSH remote machines, http://askubuntu.com/questions/476626/iptables-issue-cant-ssh-remote-machines
[6] Is there a rule for iptables to limit the amount of SYN packets, http://askubuntu.com/questions/240360/is-there-a-rule-for-iptables-to-limit-%20the-amount-of-syn-packets-a-24-range-of-i
[7] Server Fault, http://serverfault.com/
[8] Stack Overflow, http://stackoverflow.com/
[9] iptables Linux User's Manual (June 2015)
[10] Adiseshu, H., Suri, S., Parulkar, G.M.: Detecting and resolving packet filter conflicts. In: 19th IEEE International Conference on Computer Communications (INFOCOM) (Mar 2000)
[11] Al-Shaer, E., El-Atawy, A., Samak, T.: Automated pseudo-live testing of firewall configuration enforcement. IEEE Journal on Selected Areas in Communications 27(3), 302–314 (2009)
[12] Brucker, A.D., Brügger, L., Wolff, B.: Hol-TestGen/fw - An environment for specification-based firewall conformance testing. In: 10th Theoretical Aspects of Computing (ICTAC) (Sep 2013)
[13] Cisco: Policing and Shaping Overview, http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfpolsh.html
[14] Cypher, A., Halbert, D.: Watch what I Do: Programming by Demonstration. MIT Press (1993)
[15] El-Atawy, A., Ibrahim, K., Hamed, H.H.: Policy segmentation for intelligent firewall testing. In: IEEE Workshop on Secure Network Protocols (NPSec) (Nov 2005)
[16] Eppstein, D., Muthukrishnan, S.: Internet packet filter management and rectangle geometry. In: 12th Symposium on Discrete Algorithms (SODA) (Jan 2001)
[17] Frantzen, M., Kerschbaum, F., Schultz, E.E., Fahmy, S.: A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals. Computers & Security 20(3), 263–270 (2001)
[18] Juniper: Traffic Policier Overview, http://www.juniper.net/documentation/en_US/junos12.3/topics/concept/policer-overview.html
[19] Kamara, S., Fahmy, S., Schultz, E.E., Kerschbaum, F., Frantzen, M.: Analysis of vulnerabilities in Internet firewalls. Computers & Security 22(3), 214–232 (2003)
[20] Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-Miner: A tool for finding copy-paste and related bugs in operating system code. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
[21] Lieberman, H.: Your Wish Is My Command: Programming by Example. Morgan Kaufmann (2001)
[22] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked System Design and Implementation (NSDI) (May 2015)
[23] Mayer, A.J., Wool, A., Ziskind, E.: Fang: A firewall analysis engine. In: IEEE Symposium on Security and Privacy (IEEE S&P) (May 2000)
[24] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Mar 2008)
[25] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (Oct 1979), http://doi.acm.org/10.1145/357073.357079
[26] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: 24th Large Installation System Administration Conference (LISA) (Nov 2010)
[27] Plotkin, G.D., Bjørner, N., Lopes, N.P., Rybalchenko, A., Varghese, G.: Scaling network verification using symmetry and surgery. In: 43rd ACM Symposium on Principles of Programming Languages (POPL) (Jan 2016)
[28] Qian, J., Hinrichsa, S., Nahrstedt, K.: ACLA: A Framework for Access Control List (ACL) Analysis and Optimization, pp. 197–211. Springer Science+Business Media New York (2001)
[29] Tanenbaum, A.: Computer Networks. Prentice Hall, 4th edn. (2002)
[30] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Mar 2007)
[31] Wool, A.: Architecting the Lumeta firewall analyzer. In: USENIX Security Symposium (Aug 2001)
[32] Wool, A.: A quantitative study of firewall configuration errors. IEEE Computer 37(6), 62–67 (2004)
[33] Yuan, Y., Lin, D., Alur, R., Loo, B.T.: Scenario-based programming for SDN policies. In: ACM CoNEXT (CoNEXT) (Dec 2015)
[34] Zhang, S., Mahmoud, A., Malik, S., Narain, S.: Verification and synthesis of firewalls using SAT and QBF. In: 20th IEEE International Conference on Network Protocols (ICNP) (Oct 2012)