Proceedings of the 16th Conference on

Formal Methods in Computer-Aided Design (FMCAD 2016)

Mountain View, California, USA, October 3 - 6, 2016

Edited by Ruzica Piskac and Muralidhar Talupur

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

Technical co-sponsorship of IEEE Council on
Electronic Design Automation

Proceedings of the 16th Conference on

Formal Methods in Computer-Aided Design

FMCAD 2016

October 3 - 6, 2016

Mountain View, California, USA

Edited by Ruzica Piskac and Muralidhar Talupur

ISBN: 978-0-9835678-6-8

Preface

The International Conference on Formal Methods in Computer-Aided Design (FMCAD) is a series of meetings presenting groundbreaking results on the theory and application of rigorous formal techniques for the automated design of systems. The FMCAD conference covers the entire spectrum of formal aspects of specification, verification, synthesis, and testing, and is a leading forum for researchers and practitioners in academia and industry alike. The sixteenth meeting in the series was held in Mountain View, California, USA, October 3-6, 2016.

FMCAD 2016 featured a high quality program comprised of five parts: a memorial session for Professor Helmut Veith, one of the original co-chairs for this event, a tutorial day with four tutorials, a series of invited talks by experts in areas adjoining formal methods, a student forum, and finally the main program consisting of the accepted papers.

Before his unexpected passing, Helmut Veith was one of the key persons in shaping this event. We have endeavored to stay true to his vision by creating a program that is informative, high quality, and enjoyable at the same time. We had a memorial session for Helmut with talks by his collaborators and friends. This was a followed by a keynote by Professor Christos Papadimitriou on algorithms and evolution. Helmut was excited about having this talk at FMCAD and we were glad that Professor Papadimitriou was able to give this talk.

This year FMCAD featured exciting keynote talks not just from formal methods, but also from adjoining areas:

- Formal Verification for Computer Security: Lessons Learned and Future Directions by Dawn Song (UC Berkeley)
- Understanding Evolution through Algorithms by Christos Papadimitriou (UC Berkeley)
- Network Verification - When Clarke Meets Cerf by George Varghese (UCLA)

All these talks were given by well known experts, possibly outside the traditional topics at FMCAD, but were crucial for expanding horizons of the formal methods community. The talk on evolution, in particular, was not connected to formal methods, but we honored Helmut's wish to have a fun invited talk on the latest advances in the broader field of Computer Science.

The tutorial day featured four tutorials, given below in the order of presentation. These gave the audience both a look into the latest developments in the field, and a retrospective on formal methods in the industry.

- Machine Learning and Systems for the Next Frontier in Formal Verification by Manish Pandey, Synopsys Inc.
- Verifying Hyperproperties of Hardware Systems by Bernd Finkbeiner (Saarland University) and Markus Rabe (UC Berkeley)
- A Paradigm Shift in Verification Methodology by Pranav Ashar (RealIntent)
- Program Synthesis for Networks by Pavol Černý (University of Colorado Boulder)

FMCAD also offered the fourth edition of the Student Forum, organized by Hossein Hojjat (Rochester Institute of Technology). The Forum is described in more detail later in these proceedings.

For the main program, we received 64 submissions, resulting in 23 high quality accepted papers. Each paper was reviewed by at least four reviewers. The authors were then given the opportunity to respond to the comments and correct any misunderstanding. This rebuttal phase lasted a week and was followed by discussions among the reviewers to converge on a final score for the paper.

The final set of accepted papers ranged from protocol verification, architectural specification capture, traditional hardware, software verification, SMT solvers, program synthesis, and verification of timed systems. During the conference each paper was presented by one of the authors followed by a brief Q&A. Each regular paper was allotted 30 minutes and the three short papers were allotted 15 minutes each.

A conference with such a diverse program and audience as FMCAD relies on a large number of people supporting the organization. The program committee members are too numerous to list individually; we thank each and every one of them for their time, dedication to the purpose of FMCAD, their willingness to help the authors improve their manuscripts, and their help with additional tasks such as selecting and reviewing papers for the Best Paper Award. Our sincere gratitude further goes out to the Publication Chair Florian Zuleger (Vienna University of Technology, Austria; in charge of these proceedings). Hossein Hojjat (Rochester Institute of Technology) took over the non-trivial task of serving as Student Forum Chair; his engagement and enthusiasm for the process ensured an encouragingly large number of Student Forum submissions. Special thanks goes to Jens Katelaan (Vienna University of Technology, Austria) and Keshav Kini (University of Texas at Austin) formally the Webchairs, creating and maintaining a snappy new FMCAD website. The conference would not be possible without the tireless efforts of our Local Chairs Sean Safarpour (Synopsys) and Divjyot Sethi (Cisco) who made all the arrangements for conference to take place. Sean in particular worked with Synopsys management to host FMCAD at its facilities and ultimately save us quite a bit of expenses. As always, the FMCAD Steering Committee was available with both guidance and encouragement whenever needed, and even when not. We thank Armin Biere (Johannes Kepler University in Linz, Austria), Alan Hu (University of British Columbia, Canada), Warren A. Hunt, Jr. (University of Texas at Austin), and Vigyan Singhal (Oski Technology).

We would like to express our gratitude to our industrial sponsors Amazon, Cadence Design Systems, Cisco Systems, FMCAD, Inc., IBM, Infosys, Mentor Graphics, OneSpin Solutions, Oski Technology, and Synopsys. for their continued financial support of the FMCAD community. The National Science Foundation and FMCAD Inc. provided generous funds in support of the Student Forum, without which this event would simply not be possible.

FMCAD 2016 once again received in-cooperation status with ACM under the Special Interest Groups on Programming Languages (SIGPLAN) and on Software Engineering (SIGSOFT). It also received technical sponsorship from the IEEE Council on Electronic Design Automation. The FMCAD 2016 Proceedings are available through the ACM Digital Library, the IEEE Xplore Digital Library, and are also available as a free download from the FMCAD Website.

At the heart of the conference are, of course, the accepted papers, the tutorials, and the keynotes; we thank all presenters for their efforts to devote a significant portion of their time to FMCAD. We are grateful to all authors of submissions, accepted or not, and all attendees of FMCAD 2016, for playing their part in making FMCAD a continued success story.

Finally, we would like to dedicate these proceedings to Helmut Veith, our cherished friend, and colleague. He will be dearly missed both as a collaborator and a friend.

Ruzica Piskac and Muralidhar Talupur
FMCAD 2016 Program Chairs
Mountain View, California, October 2016.

Organization Committee

Program Co-Chairs

Ruzica Piskac	Yale University
Muralidhar Talupur	FormalSim

Local Arrangement Chairs & Webmasters

Sean Safarpour	Synopsys
Divjyot Sethi	Cisco
Jens Katelaan	Technical University of Vienna
Keshav Kini	University of Texas Austin

Publication Chair

Florian Zuleger	Technical University of Vienna

Student Forum Chair

Hossein Hojjat	Rochester Institute of Technology

Steering Committee

Armin Biere	Johannes Kepler University in Linz, Austria
Alan J. Hu	University of British Columbia, Canada
Warren A. Hunt, Jr.	University of Texas at Austin, USA
Vigyan Singhal	Oski Tech

Program Committee

Pranav Ashar	Real Intent
Domagoj Babic	Google
Armin Biere	Johannes Kepler University
Roderick Bloem	Graz University of Technology
Ahmed Bouajjani	LIAFA, University Paris Diderot
Gianpiero Cabodi	Politecnico di Torino
Supratik Chakraborty	IIT Bombay
Leonardo de Moura	Microsoft Research
Michael Emmi	IMDEA Software Institute
Malay Ganai	Synopsys
Arie Gurfinkel	University of Waterloo
Ziyad Hanna	Cadence Design System
Fei He	School of Software, Tsinghua University
Keijo Heljanko	Aalto University
Warren Hunt	University of Texas
Himanshu Jain	Synopsys
Gerwin Klein	NICTA and UNSW
Daniel Kroening	Computer Science Department, University of Oxford
Shuvendu Lahiri	Microsoft Research
Rebekah Leslie-Hurd	Intel
Panagiotis Manolios	Northeastern University
Kenneth McMillan	Microsoft Research
John O'Leary	Intel Corporation
Lee Pike	Galois, Inc.
Ruzica Piskac	Yale University
Ahmed Rezine	Linköping University
Sean Safarpour	Synopsys
Divjyot Sethi	CISCO
Natasha Sharygina	University of Lugano
Sharon Shoham	The Academic College of Tel Aviv Yaffo
Muralidhar Talupur	FormalSim
Michael Tautschnig	Queen Mary University of London
Helmut Veith	Vienna University of Technology
Tomas Vojnar	Brno University of Technology
Chao Wang	Virginia Tech
Eran Yahav	Technion
Florian Zuleger	Technische Universiät Wien

Additional Reviewers

Alberti, Francesco
Alpernas, Kalev
Alt, Leonardo
Aminof, Benjamin
Andronick, June
Asarin, Eugene
Asathulla, Mudabir

Bannister, Callum
Bjesse, Per
Bortin, Maksym
Bozga, Marius
Brain, Martin
Braud-Santoni, Nicolas

Calderon Trilla, Jose Manuel
Cerny, Eduard
Ceska, Milan
Chakroborty, Souy
Chen, Ben
Chou, Cuong

Dang, Thao
David, Cristina

Ebrahimi, Masoud
Enea, Constantin

Fedyukovich, Grigory

Gopalakrishnan, Sivaram
Graf, Susanne
Guo, Shengjian

Hoenicke, Jochen
Holik, Lukas
Hyvärinen, Antti

Iabrudi, Andréa
Ivrii, Alexander

Katelaan, Jens
Khalimov, Ayrat
Kini, Keshav
Koenighofer, Bettina
Kumar, Ramana

Leslie-Hurd, Joe

Mador-Haim, Sela
Maranget, Luc
Marescotti, Matteo
Meshman, Yuri
Moondanos, John
Mukherjee, Rajdeep
Murray, Toby

Nadel, Alexander
Narayana, Srinivas
Ngo, Van Chan
Nonoshita, Hiroshi
Norrish, Michael

Padon, Oded
Palena, Marco
Pani, Thomas
Parizek, Pavel
Parthasarathy, Ganapathy
Partush, Nimrod
Pasini, Paolo
Peleg, Hila
Petri, Gustavo
Plassan, Guillaume
Ponce-De-Leon, Hernan

Radicek, Ivan
Rasin, Dan
Reynolds, Andrew
Roeck, Franz
Roy, Tonmoy

Saarikivi, Olli
Sangnier, Arnaud
Santolucito, Mark
Schrammel, Peter
Seidl, Martina
Selfridge, Ben
Sinn, Moritz
Sousa, Marcelo
Sproston, Jeremy
Subramanyan, Pramod
Sung, Chungha

Vendraminetto, Danilo
Vizel, Yakir

Westbrook, Edwin
Winwood, Simon
Wolfovitz, Guy
Wu, Meng

Zeljić, Aleksandar
Zhang, Naling
Zhou, Min

Table of Contents

Formal Verification for Computer Security: Lessons Learned and Future Directions

Dawn Song
UC Berkeley

ABSTRACT OF INVITED TALK

Formal verification techniques have been fruitful for a broad spectrum of different security applications and domains. However, many important questions and considerations influence the success of applying formal verification techniques to security applications and domains. In this talk, I will share lessons learned from experience of over a decade in applying formal verification techniques to security. I will also discuss new exciting application domains such as blockchain and smart contracts for formal verification. I will pose important, open challenges and discuss future directions for verifying next-generation systems such as learning systems.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Understanding Evolution through Algorithms

Christos Papadimitriou
UC Berkeley

Abstract of Invited Talk

Why is evolution so successful? What is the role of sex (recombination)? Why is there so much diversity in populations? How do novel traits arise? Are mutations random? And is evolution optimizing something? This talk will review recent work by the speaker and collaborators aiming at understanding the many persistent mysteries of evolution through computational ideas.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Network Verification - When Clarke Meets Cerf

George Varghese
UCLA

ABSTRACT OF INVITED TALK

Surveys reveal that network outages are prevalent, and that many outages take hours to resolve, resulting in significant lost revenue. Many bugs are caused by errors in configuration files which are programmed using arcane, low-level languages, akin to machine code. Taking our cue from program and hardware verification, we suggest fresh approaches. I will first describe a geometric model of network forwarding called Header Space. While header space analysis is similar to finite state machine verification, we exploit domain-specific structure to scale better than off-the shelf model checkers. Next, I show how to exploit physical symmetry to scale network verification for large data centers. While Emerson and Sistla showed how to exploit symmetry for model checking in 1996, they exploited symmetry on the logical Kripke structure. While header space models allow us to verify the forwarding tables in routers, there are also routing protocols such as BGP that build the forwarding tables. We show to go from header space verification to what we call control space verification to proactively catch latent bugs in BGP configurations. I will end with a vision for what we call Network Design Automation to build a suite of tools for networks inspired by the Electronic Design Automation Industry. (With collaborators at CMU, Edinburgh, MSR, Stanford, and UCLA.)

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Machine Learning and Systems for the Next Frontier in Formal Verification

Manish Pandey
Synopsys

Abstract of Tutorial Talk

This tutorial covers basics of machine learning, systems and infrastructure considerations for performing machine learning at scale, and applications of machine learning to improve formal verification performance and usability. It starts with blackbox classifier training with gradient descent, and proceeds on to deep network training and simple convolutional neural networks. Next, it discusses how machine learning can be performed at scale, overcoming the performance and throughput limitations of traditional compute and storage systems. Finally, the tutorial describes several ways in which machine learning can be applied for improving formal tools performance and enhancing debug capabilities.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Verifying Hyperproperties of Hardware Systems

Bernd Finkbeiner
Saarland University

Markus Rabe
UC Berkeley

ABSTRACT OF TUTORIAL TALK

This tutorial presents hardware verification techniques for hyperproperties. The most prominent application of hyperproperties is information flow security: information flow policies characterize the secrecy and integrity of a system by comparing two or more execution traces, for example by comparing the observations made by an external observer on execution traces that result from different values of a secret variable. Such a comparison cannot be represented as a set of traces and thus falls outside the standard notion of trace properties. A comparison between execution traces can, however, be represented as a set of sets of traces, which is called a hyperproperty. Hyperproperties occur naturally in many applications beyond their origins in security: examples include the symmetric access to critical resources in distributed protocols and Hamming distances between code words in coding theory.

The hardware verification approach of the tutorial is based on recently developed temporal logics for hyperproperties. Unlike classic temporal logics like LTL or CTL, *which refer to one computation path at a time, temporal logics for hyperproperties like HyperLTL and HyperCTL* can express properties that relate multiple traces by explicitly quantifying over multiple computation paths simultaneously. We will relate the logics to the linear-branching spectrum of process equivalences, and show that even though the satisfiability problem of the logics is undecidable in general, the model checking problem can be solved efficiently. We will show how the logics can be used to verify real hardware designs, including an I2C bus master, the symmetric access to a shared resource in a mutual exclusion protocol, and the functional correctness of encoders and decoders for error resistant codes.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

A Paradigm Shift in Verification Methodology

Pranav Ashar
Real Intent

Abstract of Tutorial Talk

Todays SoCs are driving unprecedented verification complexity. The combination of billions of gates, system-level functionality on a chip, complex design methodologies like asynchronous clock domains and an explosion of untimed paths on a chip, interacting dynamic power domains, aggressive reset schemes etcetera could have been the perfect storm to staunch productivity. Instead it has turned out to be the mother of all necessities that has driven significant innovation in verification and brought about a paradigm shift.

Static sign-off has proven to be a pillar in this new paradigm. This talk will discuss the template for what has made static techniques successful in verifying modern SoCs. The recent successes are, in no small part, due to the FMCAD community that has pursued formal methods doggedly for decades despite glacial practical adoption. Complementing the efforts of the research community has been the equally determined pursuit in the EDA community to bring structure and automation into the verification process. Through this partnership, we have been able to bring about an analysis framework within which a combination of semantic analysis and formal methods enables a systematic verification process that leads to sign-off level confidence for important failure modes. It will be gratifying for the FMCAD audience to realize that SAT, model checking, functional abstraction, QBF etcetera have become essential in being able to tape out some of the most complex chips in the world on time and within budget. The adoption of IC3/PDR into the verification process was almost immediate.

The recent successes represent a strong debut for static methods. What is the vision to extend the promise into bigger slices of the verification pie? System-level verification continues to be an art-form with very little of the automation, process and problem-framing that have proven successful in other domains. May be the FMCAD community should adopt that as its next major challenge.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Program Synthesis for Networks

Pavol Černý
University of Colorado Boulder

Abstract of Tutorial Talk

Software is eating the world. But how will we write all the programs to control everything from sensors to data centers? Program synthesis provides an answer. It increases the productivity of programmers by enabling them to capture their insights in a variety of forms, not just in standard code. In this tutorial, we focus on some challenges in programming networks, and we show how program synthesis algorithms can help.

Developing network programs is difficult, as networks are large distributed systems. In particular, implementing programs that update the configuration of a network in response to events is an intricate problem. First, even if initial and final configurations are correct, subtle bugs in update programs can lead to incorrect transient behaviors, including forwarding loops, black holes, and access control violations. Second, if the update program reacts to events occurring near simultaneously in different parts of the network, naive implementations can lead to causality violations and conflicts. We present scalable program synthesis algorithms that produce network programs that are both correct by construction and efficient.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

The FMCAD 2016 Graduate Student Forum

Hossein Hojjat
Computer Science Department
Rochester Institute of Technology

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 2016, the event took place in Mountain View, California, as integral part of the FMCAD conference. Ten 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 and synthesis, including automated reasoning, model checking of hardware, software, as well as hybrid systems, verification and synthesis of networks, and application of artificial intelligence techniques to circuit design.**

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

Graduate students were invited to submit short reports describing their ongoing research in the scope of the FMCAD conference. The submissions to the forum presented novel technical contributions and outlining future research planned by the authors. The presentations covered a broad range of topics in the field of verification and synthesis, including automated reasoning, model checking of hardware, software, as well as hybrid systems, verification and synthesis of networks, and application of artificial intelligence techniques to circuit design. Based on the reviews provided by members of the organizing committee as well as a number of external reviewers, 10 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 following contributions have been accepted:

- Elaheh Ghassabani, Michael W. Whalen, Andrew Gacek, Rockwell Collins: "Inductive Validity Cores for Formal Verification"
- Yu-Yun Dai, Robert Brayton: "Circuit Recognition with Convolutional Neural Networks"
- Bo-Yuan Hunag, Pramod Subramanyan, Sharad Malik, Sayak Ray, Hareesh Khattri, Jason Fung, Abhranil Maiti: "Instruction-Level Abstraction Based SoC Firmware Verification"
- William Hallahan, Ruzica Piskac, Ennan Zhai, Avi Silberschatz: "Automated Firewall Repair with Example Synthesis"
- Hongce Zhang, Sharad Malik: "Equivalence Checking Using the Intermediate Instruction-Level Abstraction"
- Baoluo Meng: "Solving Relational Constraints with Extensions to a Theory of Finite Set in SMT"
- Mark Santolucito, Ruzica Piskac: "Version Space Learning for Verification on Temporal Differentials"
- Jaideep Ramachandran: "Precise Arithmetic Reasoning using Approximate Solvers"
- Rohit Dureja, Kristin Rozier: "Comparative Safety Analysis of Wireless Communication Networks in Avionics"
- Andres Noetzli: "Proofs for Preprocessing in SMT Solvers"

The 2016 student forum is the second in the series to feature 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]

For the first time in the student forum series, we used a voting system to choose the best contribution. We allowed any participant (other than the authors and their supervisors) to vote for a single contribution. We created an electronic voting system and provided a link to the system in the registration packages of participants. Such a system offered several advantages. First of all, it inspired the poster presenters to do their best possible presentations for any participant, not only the referees who have a voice in determining the best contributions. Second, to increase the number of votes, the students were motivated to talk to more people and to try to attract their attentions to their posters. The overall experience was overwhelming in a more interactive poster session with more attendants.

The Student Forum would not have been possible without the excellent contributions of the student authors. The generous support of the National Science Foundation enabled us to subsidize the travel cost of the participating students. The help and advice of Georg Weissenbacher, who organized the previous student forums, as well as Sean Safarpour (Synopsys), who took care of local arrangements, was crucial to the success of the event. We are also indebted to Ruzica Piskac for her help with the reviews.

References

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

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

[3] 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/FMCAD16/

Soundness of the Quasi-Synchronous Abstraction

Guillaume Baudart*‡ Timothy Bourke‡* Marc Pouzet*†‡

* École normale supérieure, PSL Research University
† Sorbonne Universités, UPMC Univ. Paris 06
‡ Inria Paris

***Abstract*—Many critical real-time embedded systems are implemented as a set of processes that execute periodically with bounded jitter and communicate with bounded transmission delay. The *quasi-synchronous abstraction* was introduced by P. Caspi for model-checking the safety properties of applications running on such systems. The simplicity of the abstraction is appealing: the only events are process activations; logical steps account for transmission delays; and no process may be activated more than twice between two successive activations of any other.**

We formalize the relation between the real-time model and the quasi-synchronous abstraction by introducing the notion of a unitary discretization. Even though the abstraction has been applied several times in the literature, we show, surprisingly, that it is not sound for general systems of more than two processes. Our central result is to propose necessary and sufficient conditions on both communication topologies and timing parameters to recover soundness.

I. Introduction

The *Synchronous Real-Time Model* [2], [10] characterizes many distributed embedded systems: it applies whenever bounds exist on successive process executions and transmission delays. In particular, whenever computing units that execute periodically with jitter are connected together by network links. It is commonly employed in critical aerospace, power, and rail systems.

The *quasi-synchronous approach* [6], [8] formalizes a set of techniques for building distributed control systems that were observed by P. Caspi while consulting at Airbus on the distributed deployment of Lustre/SCADE[1] [17] designs. One of the key ideas is to model the computing units, network links, and shared memories themselves as a synchronous program [6, §3]. Such models can be verified using model-checking tools for discrete programs. This approach has, for instance, been applied to a Proximity Flight Safety (PFS) case-study from EADS Space Transportation [19] and to the analysis of systems specified in the Architecture Analysis and Design Language (AADL) [5], [20], [28].

An alternative way of developing real-time applications is to synchronize process executions. The Time-Triggered Architecture (TTA) [21], [22] thoroughly develops this approach and there are several clock synchronization protocols suitable for embedded systems. Once a clock synchronization scheme is adopted and assumed or verified correct, modeling and reasoning about applications is greatly simplified because non-determinism, in the form of possible interleavings, is either eliminated or reduced. The quasi-synchronous approach is nevertheless appropriate in certain applications either due to their simplicity, for example, microprocessors communicating directly over serial links, or the need for complete independence between subsystems, for example, as in redundant subnetworks connected only at voting units.

Figure 1 gives an overview of the quasi-synchronous approach. On the left is a real-time model comprising two processes, A and B, communicating through network links. Processes and links are annotated with timing bounds on executions ($T_{\min}$ and $T_{\max}$) and transmission delays ($\tau_{\min}$ and $\tau_{\max}$). Underneath is an example trace showing process activations and corresponding message transmissions. On the right is a discrete-time abstraction in which timing parameters are replaced by a discrete program called *Scheduler* that overapproximates their effect by controlling process activations, and, importantly, message transmissions are modeled by a single logical step. Underneath is a trace of the discrete-time model.

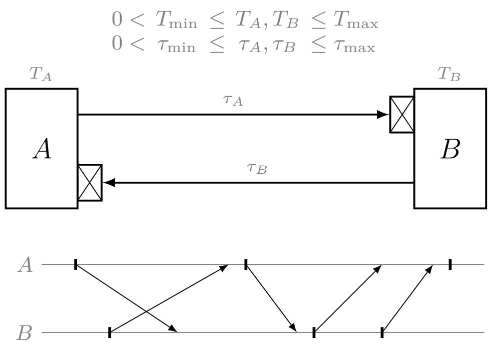
The ultimate aim is to verify properties of the real-time model in the simpler discrete-time model. The essential property is that every sequence of states that occurs in the real-time model can also occur in the discrete-time model.[2] Such an association guarantees soundness: all safety properties provable in the discrete-time model also hold of the real-time model. Since changes in state are directly related to received messages, we focus on traces without modeling process and network states explicitly. This means that a discrete model is a valid abstraction if every real-time trace has a discrete-time counterpart.

Contributions: We formalize the relation between real-time and discrete-time traces in the quasi-synchronous approach by introducing a *unitary discretization* based on the respective causality relations of the two models. With this tool we show that abstracting transmission delays as unit delays is not sound in general. We state and prove necessary and sufficient conditions on communication topologies and timing characteristics to recover soundness. We provide practical criteria for using the

[1]http://www.ansys.com/Products/Embedded-Software/ANSYS-SCADE-Suite

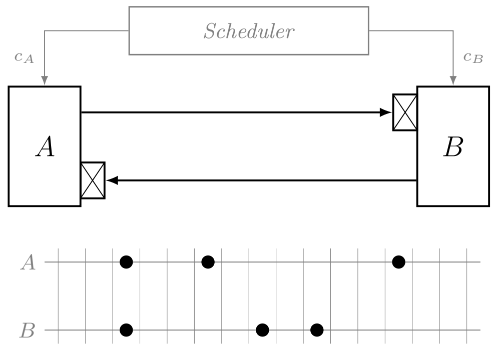
[2]Assuming the state of the processes does not reference real time.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.



$0 < T_{\min} \le T_A, T_B \le T_{\max}$
$0 < \tau_{\min} \le \tau_A, \tau_B \le \tau_{\max}$
T_A
T_B
τ_A
τ_B
A
B
A
B

(a) Real-time model (RT)



(b) Discrete-time model (DT)

Fig. 1: Soundness: A property φ that can be verified in the discrete-time model will also holds for the real-time model, $RT \models \varphi \impliedby DT \models \varphi$.

quasi-synchronous abstraction to formally verify real-time systems in discrete-time model-checking tools [16], [18].

A. The Real-time Model

We consider the classic synchronous real-time model [2], [10], noting that 'synchronous' does not mean 'lock step'.

Definition 1 (Synchronous Real-Time Model)**.** *A* synchronous real-time model *is a finite set of processes $\mathcal{P}$, where for every process, the delay T between two successive activations is bounded:*

$$0 \le T_{\min} \le T \le T_{\max}. \qquad \text{(RP)}$$

Values are transmitted between processes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{\min}$ and $\tau_{\max}$:

$$0 \le \tau_{\min} \le \tau \le \tau_{\max}. \qquad \text{(RT)}$$

Each message is buffered at receivers until a newer value is received. Execution time (τ_{exec}) can be modeled either as a part of the communication delay ($\tau = \tau_{\text{exec}} + \tau_{\text{trans}}$), or as part of the activation period ($\tau_{\text{exec}} < T_{\min}$) with the convention that application components communicate through logical delays: values computed in one reaction are sent at the beginning of the next one.

For readability, we assume global bounds on successive process activations, but our results are readily generalized to multirate systems.

B. The Discrete-time Model

The simplest discrete abstraction is the *asynchronous model* where time is ignored altogether and process activations may be interleaved arbitrarily. This is sound but far from complete: many properties that hold in the real-time model cannot be shown in the discrete one. Furthermore, the many possible interleavings complicate reasoning about or model-checking the discrete-time model.

A finer abstraction was proposed by Caspi for processes that execute 'almost periodically', that is, $T_{\min} \approx T_{\max}$. He realized that the interleavings of systems satisfying RP can be constrained [7, §3.2]:

> It is not the case that a component process executes more than twice between two successive executions of another process.

Furthermore, he observed that when transmission delays are 'significantly shorter than the periods of [process activations]' they can be modeled by unit delays in the discrete-time model, but that 'if longer transmission delays are needed, modeling should be more complex' [6, §3.2.1]. A unit delay models the fact that a message sent at one logical instant is received at the next one.

More complex modeling refers to the standard approach of placing buffer processes between communicating processes. Such buffers provide receive and send events and maintain internal state to track messages in transmission. The quasi-synchronous abstraction eschews explicit link models thereby simplifying scheduling logic and halving the number of variables needed to model communication.

These observations allow abstraction from the timing details of the real-time model in definition 1 to give a non-deterministic and discrete-time model of systems termed *quasi-synchronous.* In the discrete-time model, boolean variables called *clocks* are set to *true* to activate processes.

Definition 2 (Quasi-Synchronous Model)**.** *A* quasi-synchronous model *comprises a scheduler and finite set of processes $\mathcal{P}$. The scheduler is connected to each process by a discrete clock signal. It activates the processes non-deterministically but ensures that no pair of clock signals (c_A, c_B), for a pair of processes $A, B \in \mathcal{P}$, ever contains the subsequence*

$$\begin{bmatrix} t \\ \_ \end{bmatrix} \cdot \begin{bmatrix} f \\ f \end{bmatrix}^* \cdot \begin{bmatrix} t \\ f \end{bmatrix} \cdot \begin{bmatrix} f \\ f \end{bmatrix}^* \cdot \begin{bmatrix} t \\ \_ \end{bmatrix},$$

where t indicates an activation, f means no activation, and _ means either of the two. Processes communicate through unit delays activated at every scheduler tick.

This restriction on subsequences of pairs of clock signals [6, §3.2.2] expresses formally the constraint quoted beforehand. The forbidden subsequence involves at least three activations of one process (A) between two successive activations of another (B). A finite state scheduler that produces valid sequences is readily constructed from the given regular expression (using, for instance, the `reglo` tool [29]). The processes and unit delays can be modeled directly in Lustre [17], for instance, and verified by model-checking [5], [19], [20], [28].

The quasi-synchronous model aims to reduce the state-space of a model in two ways: 1) by limiting the interleavings of process activations and 2) by simplifying message transmission modeling. In this paper, we show how the constraints imposed by the latter choice limit the applicability of the abstraction.

C. Relating Real time and Discrete time

Given definitions 1 and 2, it is natural to query the exact relationship between them, namely: *what are necessary and sufficient conditions on the architecture to ensure the soundness of the abstraction?*

The first step is to formalize real-time traces and their causality (section II). The main contribution of this paper is then to characterize the link between this causality relation and the causalities expressible in the discrete model. Specifically, we define a 'unitary discretization' that relates real-time traces to discrete-time traces (section III). It is quite constraining due to the modeling of communications as unit delays, but it still allows for the treatment of practically-relevant systems of two processes [19], [20] and those with certain communication topologies. Based on these results, we define precisely when the quasi-synchronous model can be applied to a real-time system (section IV). We relate our work to classic distributed systems models, to the expression of causality in distributed systems, and to existing work on the quasi-synchronous abstraction (section V).

II. Traces and Causality

We define a formal model for reasoning about real-time models and their discretization. It has two components: (real-time) traces and their induced causality relations. In the following, we fix an arbitrary real-time model with processes $\mathcal{P}$ and parameters $T_{\min}$, $T_{\max}$, $\tau_{\min}$, and $\tau_{\max}$ that satisfy definition 1. We formalize pairs of sending and receiving processes using a *communicates-with* relation, written $\rightrightarrows$, between the processes of a real-time model. This relation is not necessarily symmetric, $A \rightrightarrows B$ need not imply $B \rightrightarrows A$, but it must be reflexive ($A \rightrightarrows A$).

Definition 3 (Trace)**.** *A (real-time)* trace $\mathcal{E}$ *is a set of* activation events $\{A_i \mid A \in \mathcal{P} \wedge i \in \mathbb{N}\}$ *and two functions:*

- *$t(A_i)$, the date of event A_i with respect to an ideal reference clock, and*
- *$\tau(A_i, B)$, the transmission delay of the message sent at A_i to a process B.*

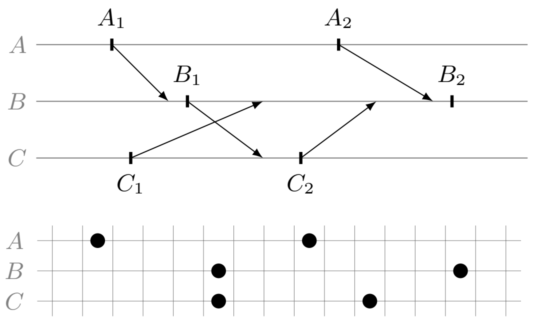


Fig. 2: A trace (above) and a possible unitary discretization.

Both $t(A_i)$ and $\tau(A_i, B)$ are non-negative reals satisfying the constraints of definition 1, namely if $A \rightrightarrows B$,

$$\begin{aligned} 0 \le T_{\min} &\le t(A_{i+1}) - t(A_i) \le T_{\max}, \text{ and} \\ 0 \le \tau_{\min} &\le \tau(A_i, B) \le \tau_{\max}. \end{aligned}$$

The causality relation between events within a given trace is essentially the *happened before* relation of Lamport [23]. Unlike Lamport, however, we do not explicitly model message reception. A message is received if the next execution of the receiver occurs after the corresponding transmission delay.

Definition 4 (Happened Before)**.** *For a trace $\mathcal{E}$, let $\rightarrow$ be the smallest relation on activation events that satisfies*

(local) *If $i < j$ then $A_i \rightarrow A_j$, and*
(recv) *If $A \rightrightarrows B$ and $t(A_i) + \tau(A_i, B) \le t(B_j)$ then $A_i \rightarrow B_j$.*

Activations at a single process are totally ordered (*local*); an activation at one process happens before an activation at another process when a message sent at the former is received before the latter (*recv*).

Compared to Lamport, we do not close the relation by transitivity. In this way, $A_i \rightarrow B_j$ means that B_j occurs strictly after the reception of the message sent by process A at A_i. The same technique is used elsewhere [31, definition 1].

III. Unitary Discretization

We now address the central question of relating the real-time and discrete-time models. The problem is essentially one of correctly discretizing real-time traces.

If process A sends messages to process B, the most general approach is to ensure that when an event A_i occurs before an event B_j in the discrete-time trace, A_i happens before B_j ($A_i \rightarrow B_j$) in the corresponding real-time trace and vice versa. Figure 2 shows an example trace for a three-process system and a possible unitary discretization.

Definition 5 (Unitary Discretization)**.**
A function $f : \mathcal{E} \rightarrow \mathbb{N}$ that assigns each event in a (real-time) trace to a logical instant of a corresponding discrete trace, is a unitary discretization *if for all $A_i, B_j \in \mathcal{E}$,*

$$A_i \rightarrow B_j \iff (f(A_i) < f(B_j) \text{ and } A \rightrightarrows B)\,. \qquad \text{(UD)}$$

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Discretizing a real-time model satisfying definition 1 to a model of the form given in definition 2 amounts to finding a unitary discretization for *each* of its (real-time) traces. The forward half of the equivalence comes from the fact that the $\rightarrow$ relation induces a partial order on events. Completing this relation to a total order gives a discretization that respects the causality of the real-time model [23].

A unitary discretization links the causality of events in the real-time model to the causality implicit in the discrete-time model. The backward direction of the equivalence imposes that if an event y occurs after an event x in the discrete-time model, that is, $f(x) < f(y)$, it is either because y is a later activation of the same process as x, or because y occurs strictly after the receipt of the message sent at x. It is the communication through unit delays on a common clock that tightly links the two causality relations.

In distributed systems terminology, condition UD is called *strong consistency* [30]. The problem of finding a unitary discretization is thus equivalent to the problem of finding a strongly consistent scalar clock. Raynal and Singhal report in their survey [30] that this is not possible in general, that is, there is no scalar clock function f that satisfies UD. This was already noted by Lamport in his original paper: 'We cannot expect the converse condition to hold as well [...]' [23, p.560].

The aim is to formulate sufficient conditions on the (static) $\rightrightarrows$ relation and on the timing characteristics of the real-time model to guarantee the existence of a unitary discretization. The following proposition will be useful.

Proposition 1. *If f is a unitary discretization for a trace, for a pair of processes where $A \rightrightarrows B$ we have that*

$$A_i \rightarrow B_j \Longrightarrow f(A_i) < f(B_j), \text{ and}$$
$$A_i \not\rightarrow B_j \Longrightarrow f(A_i) \geq f(B_j).$$

Proof. The first implication is a direct consequence of the definition of a unitary discretization. The second one follows by contraposition. If $f(A_i) < f(B_j)$, and since $A \rightrightarrows B$, we have $A_i \rightarrow B_j$ by the definition of f. □

An intermediate step to defining a static condition on communications is to characterize traces for which there is no unitary discretization. Our characterization will be based on a graph of the constraints of proposition 1.

Definition 6 (Trace Graph)**.** *Given a trace $\mathcal{E}$, its directed, weighted* trace graph *$\mathcal{G}$ has as vertices $\{A_i \mid A \in \mathcal{P} \wedge i \in \mathbb{N}\}$ and as edges the smallest relations that satisfy*

1) If $A_i \rightarrow B_j$ then $A_i \xrightarrow{1} B_j$, and
2) If $A \rightrightarrows B$ and $A_i \not\rightarrow B_j$ then $B_j \xrightarrow{0} A_i$.

An example trace graph is shown in figure 3. Edges labeled with one ($x \xrightarrow{1} y$) represent the constraints $f(x) < f(y)$. Each such edge indicates that the source activation must come before the destination activation in a unitary discretization, that is, the value of f, from source to destination, must increase by at least one. Edges labeled with zeros ($x \xrightarrow{0} y$) represent the constraints $f(x) \leq f(y)$. Each such edge indicates that the source activation cannot be placed before the destination activation in a unitary discretization, that is, the value of f, from source to destination, must be the same or larger. A path through several activations defines their relative ordering in all unitary discretizations.

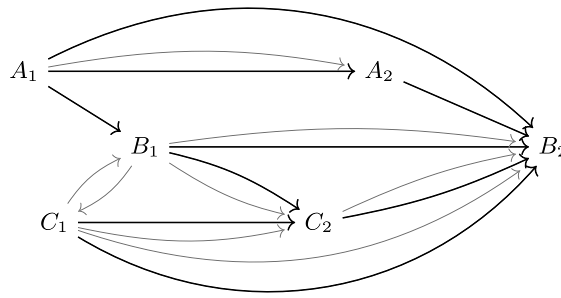


Fig. 3: The trace (sub-)graph of the trace in figure 2. Black thick arrows denote $x \xrightarrow{1} y / f(x) < f(y)$. Thin gray arrows denote $x \xrightarrow{0} y / f(x) \leq f(y)$.

The satisfaction of the required constraints, or the impossibility of satisfying them, can now be phrased in terms of cycles in the graph. A cycle comprising only $\xrightarrow{0}$'s is acceptable: its activations are all assigned the same discrete slot (for example, B_1 and C_1 in figure 3). Any cycle containing a $\xrightarrow{1}$ represents a set of unsatisfiable constraints: one of the events must be placed in two different slots.

Lemma 1 ($\exists$UD $\iff \overline{\exists\text{PC}}$)**.** *For a trace $\mathcal{E}$, there is a unitary discretization* ($\exists$UD) *if and only if there is no cycle of positive weight in the corresponding trace graph $\mathcal{G}$* ($\overline{\exists\text{PC}}$)*.*

Proof. Assume there is a cycle of positive weight. By the construction of $\mathcal{G}$ there is an event x such that, for any unitary discretization function, $f(x) < f(x)$, which is impossible.

Conversely, if there are no cycles of positive weight, we may define a function f that maps each event x to the weight of the longest path in $\mathcal{G}$ that leads to x. By construction, $A_i \rightarrow B_j \implies f(A_i) < f(B_j)$, which is the forward implication of UD (definition 5). The other direction of UD follows by contraposition. Assume $A_i \not\rightarrow B_j$. If $A \rightrightarrows B$, we have $B_j \xrightarrow{0} A_i$ and thus, by the definition of f, that $f(B_j) \leq f(A_i)$. This gives $\neg(f(A_i) < f(B_j))$ as required. The other case, $A \not\rightrightarrows B$, is trivial. □

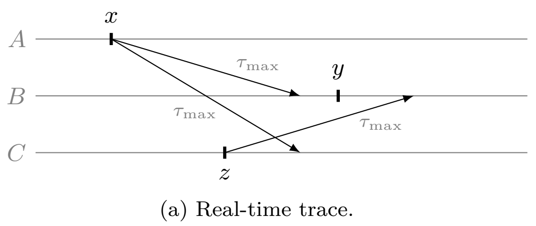
The unitary discretization described in the proof above is the most concise one and can be expressed as

$$f(x) = \max\ (\{f(y) + 1 \mid y \xrightarrow{1} x\} \cup \{f(z) \mid z \xrightarrow{0} x\} \cup \{0\})\,.$$

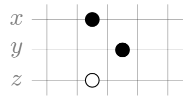
Other discretizations are constructed by adding 'extra' instants between process activations as in figure 2.

A. Discretizing general systems

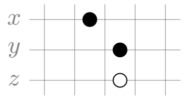
One might expect that real-time models are unitary discretizable if the transmission delays are 'significantly



(a) Real-time trace.



(b) $z \to y$.



(c) $x \to z$.

Fig. 4: A real-time trace that is not unitary discretizable ($x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} x$) and that may occur whenever $\tau_{\max} > 0$.

shorter' than the period of the process, that is $\tau_{\max} \ll T_{\min}$. Unfortunately this is not the case.

Theorem 1 (No General Unitary Discretization)**.** *General real-time models with three processes or more communicating non-instantaneously are not unitary discretizable.*

Proof. If $\tau_{\max} > 0$, figure 4a shows a trace with a cycle of positive weight, $x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} x$, for which there is no unitary discretization (lemma 1). □

Figure 4 shows the two possible discretizations of the counterexample. In figure 4b the message sent at z should have been received at y ($z \to y$); whereas in figure 4c the message sent at x should have been received at z ($x \to z$). Neither correctly abstracts the real-time trace of figure 4a.

B. Recovering Soundness

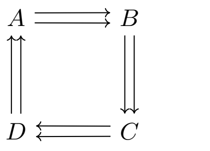
The counterexample of figure 4 shows that when three processes communicate such that $A \rightrightarrows B \leftleftarrows C \leftleftarrows A$, there is at least one trace that has no unitary discretization. Problematic cycles in traces can be prevented either by constraining the timing parameters of the model or by restricting communication graphs: forbidding $A \rightrightarrows B$ removes $A_i \xrightarrow{1} B_j$ and $B_j \xrightarrow{0} A_i$, for all i and j, in associated trace graphs (if $A \neq B$). We propose conditions that preclude cycles of positive weight in all possible traces and thus guarantee the existence of unitary discretizations.

Theorem 2. *Let L_c be the size of the longest elementary cycle in the communication graph. A real-time model satisfying definition 1 is unitary discretizable if and only if,*

1) *all u-cycles of the communication graph are cycles or balanced u-cycles, or $\tau_{\max} = 0$, and*
2) *there is no balanced u-cycle in the communication graph or $\tau_{\min} = \tau_{\max}$, and*
3) *there is no cycle in the communication graph or*

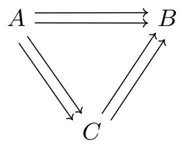
$$T_{\min} \geq L_c \tau_{\max}. \tag{CD}$$

A *u-cycle* is an elementary cycle in the undirected communication graph, that is, the graph obtained from the communication graph by forgetting the direction of the edges. A balanced u-cycle has the same number of edges in both directions. Figure 5 shows three examples of u-cycles, the rightmost one is also a balanced u-cycle. In the following $\mathcal{C}$, $u\mathcal{C}$, and $b\mathcal{C}$ denote the sets of cycles, u-cycles, and balanced u-cycles, respectively.



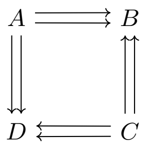
$$\begin{tikzcd} A \arrow[r, Rightarrow] & B \arrow[d, Rightarrow] \\ D \arrow[u, Rightarrow] & C \arrow[l, Rightarrow] \end{tikzcd}$$

(a) cycle



$$\begin{tikzcd} A \arrow[rr, Rightarrow] \arrow[dr, Rightarrow] & & B \\ & C \arrow[ur, Rightarrow] & \end{tikzcd}$$

(b) u-cycle



$$\begin{tikzcd} A \arrow[r, Rightarrow] \arrow[d, Rightarrow] & B \\ D & C \arrow[l, Rightarrow] \arrow[u, Rightarrow] \end{tikzcd}$$

(c) b-cycle

Fig. 5: Examples of communication topologies.

In simpler terms, theorem 2 states that communication topologies containing u-cycles are only permissible if communication is perfectly instantaneous. Cycles can be allowed by imposing the additional constraint CD and balanced u-cycles can be allowed by imposing $\tau_{\min} = \tau_{\max}$. The following proposition is needed in the proof.

Proposition 2. *If a trace graph has a cycle of positive weight, then it has a cycle of positive weight of the form:*[3]

$$A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \dots \xrightarrow{b_n} A^+$$

where processes $A, B, C, \dots$ are pairwise distinct.

We write A^+ to denote successive activations of process A.

Proof. From any cycle of positive weight one can build another cycle of positive weight with the correct form. The proof is given in the extended paper. □

We now present a proof sketch for theorem 2, the complete proof can be found in the extended paper.

Proof. The proof is by contraposition in both directions. Using lemma 1 we have $\overline{\exists \mathrm{UD}} \iff \exists \mathrm{PC}$. Therefore we prove the following statement, which is equivalent to theorem 2.

$$\exists \mathrm{PC} \iff \left| \begin{array}{ll} \exists c \in \mathcal{C} \text{ and } \overline{\mathrm{CD}}, \text{ or,} & (\mathrm{C}_1) \\ \exists c \in b\mathcal{C} \text{ and } \tau_{\min} < \tau_{\max}, \text{ or,} & (\mathrm{C}_2) \\ \exists c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C}) \text{ and } \tau_{\max} > 0 & (\mathrm{C}_3) \end{array} \right.$$

To prove that C_1 or C_2 or $\mathrm{C}_3 \implies \exists \mathrm{PC}$ we show that in each of the three possible cases one can build a trace with a cycle of positive weight. Figure 6 shows such a counterexample for a u-cycle of 5 processes.

To prove that $\exists \mathrm{PC} \implies \mathrm{C}_1$ or C_2 or C_3, suppose that there exists a trace with a cycle of positive weight. By proposition 2, there also exists a trace with a cycle of positive weight of the form:

$$A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \dots \xrightarrow{b_n} A^+, \tag{1}$$

where processes A, B, C, ... are pairwise distinct. By proposition 2, for two processes A and B, a transition

[3] $\xrightarrow{b_i}$ is used as a generic notation for either $\xrightarrow{1}$ or $\xrightarrow{0}$.

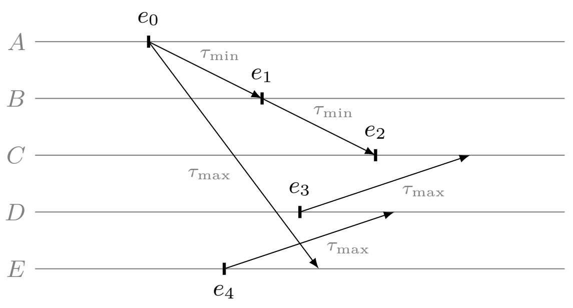


Fig. 6: Counterexample $e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_4 \xrightarrow{0} e_0$, based on the u-cycle $A \rightrightarrows B \rightrightarrows C \leftleftarrows D \leftleftarrows E \leftleftarrows A$.

$A_i \xrightarrow{0} B_j$ corresponds to a communication channel $A \leftleftarrows B$ with $A \neq B$ and a transition $A_i \xrightarrow{1} B_j$ corresponds to a communication channel in the opposite direction $A \rightrightarrows B$ with the possibility that $A = B$ for activations of the same process. Since the processes of (1) are pairwise distinct, the sequence of processes $c = A, B, C, \ldots, A$ forms a u-cycle of the communication graph. There are three cases:

1) $c \in \mathcal{C}$ imposes $T_{\min} < L_c\tau_{\max}$ ($\overline{\text{CD}}$), hence C_1 holds.
2) $c \in b\mathcal{C}$ imposes $\tau_{\min} < \tau_{\max}$ and C_2 holds.
3) $c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C})$ imposes $\tau_{\max} > 0$ and C_3 holds.

□

Theorem 1 is a particular case of theorem 2. Without assumptions on the communication graph there could be a u-cycle that is neither a cycle nor a balanced u-cycle.

Corollary 1 (2-process Unitary Discretization)**.** *A real-time model satisfying definition 1 with two processes can be unitarily discretized if and only if*

$$T_{\min} \geq 2\tau_{\max}. \tag{2D}$$

Proof. Direct consequence of theorem 2: for systems of two processes, $L_c = 2$ and CD becomes $T_{\min} \geq 2\tau_{\max}$. □

Two-process models were the focus of the original work on the quasi-synchronous approach [6] and they are relevant in practice [19], [20]. This result is coherent with Caspi's requirement that transmission delays be 'significantly shorter than the periods of [process activations]' [6, §3.2.1].

IV. The Quasi-Synchronous Abstraction

We now apply the preceding definitions and results on unitary discretizations to precisely describe when the quasi-synchronous model can be applied to a real-time system.

A discrete-time model is termed quasi-synchronous if 'it is not the case that a component process executes more than twice between two successive executions of another process' [7, §3.2]. Since any given process only detects the activations of another by receiving the corresponding messages, the quasi-synchronous condition corresponds to two constraints. For any process, 1) there are no more than two activations between two message receptions, and 2) there are no more than two message receptions between two activations. This definition can be formalized using unitary discretizations.

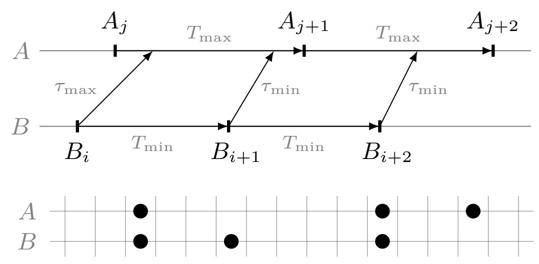


Fig. 7: A trace (above) and a possible discretization that violates definition 2.

Definition 7 (Quasi-Synchronous Model)**.** *A real-time model is quasi-synchronous if, for every trace t,*

1) *it has a unitary discretization f, and*
2) *for processes $A \leftleftarrows B$, there are no i and j such that*

$$\begin{aligned} f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned} \tag{QS}$$

This definition expresses the two central features of quasi-synchrony: 1) communications as 'logical' unit delays, and 2) constraints on interleavings of process activations.

Condition QS is less constraining than definition 2 from section I-B. That definition, proposed by Caspi, has the advantage of forbidding a single symmetric subsequence, but the link with process interleavings is obscured. In fact, the proposition below shows that it is violated in any real-time system with unidirectional communications ($A \leftleftarrows B$ but $A \not\rightrightarrows B$) that is not perfectly synchronous. So, while definition 7 does not directly translate definition 2, we argue that it more faithfully describes quasi-synchronous systems in terms of process interleavings.

Proposition 3. *A pair of (real-time) processes A and B where $A \leftleftarrows B$ but $A \not\rightrightarrows B$ cannot be quasi-synchronous in the sense of definition 2 if $T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$.*

Proof. If $T_{\min}+\tau_{\min} < T_{\max}+\tau_{\max}$, figure 7 shows an execution trace where $A_j \xrightarrow{0} B_i \xrightarrow{1} B_{i+1} \xrightarrow{1} A_{j+1} \xrightarrow{0} B_{i+2}$. A discretization f with $f(A_j) = f(B_i)$ and $f(A_{j+1}) = f(B_{i+2})$ is a valid unitary discretization that violates the condition of definition 2. □

While definition 7 conveys the essence of quasi-synchrony, its conditions are rather abstract. The following theorem incorporates the results of sections II and III to state concrete requirements on real-time parameters and communication topologies. The extended paper has a full proof.

Theorem 3. *A real-time model satisfying definition 1 is quasi-synchronous (condition QS) if and only if,*

1) *the conditions of theorem 2 hold, and*
2) *the following condition holds,*

$$2T_{\min} + \tau_{\min} \geq T_{\max} + \tau_{\max}. \tag{QT}$$

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

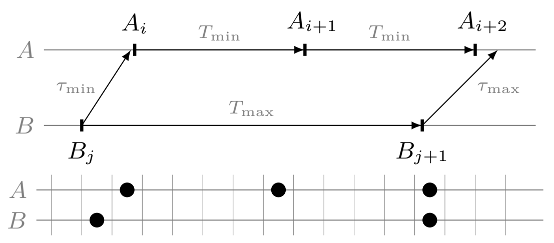


Fig. 8: Witness for QS $\implies$ QT.

Proof. The first condition ensures that the system is unitary discretizable. Now, if QS does not hold, there is a chain of events such that either

$$f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,}$$
$$f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}).$$

This gives $B_j \to A_i$, and $B_{j+1} \not\to A_{i+2}$ in the first case; and, $B_i \not\to A_j$ and $B_{i+2} \to A_{j+1}$ in the second; which implies $2T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$, that is, $\overline{\text{QT}}$.

Conversely, if QT does not hold, then figure 8 shows a trace where $B_j \xrightarrow{1} A_i \xrightarrow{1} A_{i+1} \xrightarrow{1} A_{i+2} \xrightarrow{0} B_{j+1}$. Then, by definition 5, the discretization f such that

$$f(B_j) < f(A_i) < f(A_{i+2}) = f(B_{j+1}).$$

is a valid unitary discretization that violates QS. □

Theorem 3 states precisely when the quasi-synchronous abstraction is sound. If a real-time system satisfies the given constraints on (logical) topology and timing, then the quasi-synchronous abstraction can be used to formally verify its properties. To give a few concrete examples, providing condition QT holds, it applies to: 1) topologies without feedback, for example, three filter sequences connected to a triple voter; 2) trees of communicating pairs if $T_{\min} \geq 2\tau_{\max}$, for example, a 'daisy chain' or a star of intercommunicating neighbours; and 3) any feedback loop of n nodes if $T_{\min} \geq n\tau_{\max}$, for example, unidirectional ring networks or filters with 'non-overlapping' feedback loops. It does not apply if condition QT is violated, or in topologies with certain cycles, notably those with more than one path between two processes. The extended paper includes some examples of allowed and forbidden topologies.

V. Related Work

a) Distributed systems: The spectrum of formal models for distributed systems runs from completely synchronous (definition 1) to completely asynchronous [26]. The completely synchronous model makes the strongest timing assumptions—though they are not unreasonable for embedded systems—and it is possible to simulate round-based applications and solve problems like consensus and leader election even in the presence of failures [2], [10].

The impossibility of consensus in the asynchronous model [14] and the desire to treat more general systems than the synchronous model motivates the study of *partially synchronous* models [26, Part III]. There are models with bounds on transmissions and the relative speeds of processes, and these bounds are not necessarily known or may only hold eventually [12]. In the *θ-model* [33] bounds are not given on transmissions but rather on the ratio of the longest and shortest end-to-end delays of messages simultaneously in transit. The *Finite Average Response* time model [13] only assumes a lower bound on activations and a finite average response time for transmissions. Timing assumptions may also be allowed to vary across different communication links [1]. The *Asynchronous Bounded-Cycle model* [31] avoids any reference to transmission delays or bounds on activations and instead constrains the causality chains induced by transmission.

We treat the standard synchronous distributed systems model and our treatment of causality and timing constraints has nothing to do with recovering possibility results or determining algorithmic complexity in a partially synchronous model. We study a different question: when is a very specific discrete abstraction sound for the synchronous real-time model? Our main problem comes from the unusual but potentially advantageous modeling of transmissions as unit delays. This gives rise to a unique form of causality—the unitary discretization—that is relevant to the model-checking problem we consider but not to the theory of distributed computing.

Unsurprisingly, our notion of causality follows Lamport's seminal work [23]. His causality relation was recently extended to a *syncausality* relation [3, Definition 2] by using upper bounds on transmission delays to complete causality chains. Our causality relation is similar but message reception is not modeled explicitly, the *(recv)* clause is based on actual transmission delays not an upper bound, and transitivity is not avoided. The syncausality relation is developed into 'centipede' and 'centibroom' abstractions to study coordination problems, whereas we develop the unitary discretization to verify the soundness of a discrete model. Our approach is closer to work on *execution graphs* [31]: we also use a non-transitive relation and count along causality chains. But our trace graphs incorporate two types of constraints ($\xrightarrow{0}$ and $\xrightarrow{1}$) due to the different nature of the problem we study. Furthermore, the work on execution graphs focuses on asynchronous systems and does not propose constraining real-time parameters and communication topologies to eliminate cycles.

b) Logical clocks: As already mentioned in section III, the existence of a unitary discretization is equivalent to the problem of finding a strongly consistent scalar clock. As this is not possible in general [23], [30], research has sought more powerful mechanisms, like vector clocks [27] and matrix clocks [15], for capturing the causalities of events. These mechanisms do not resolve the problem posed in this paper, since the modeling of transmissions as unit delays and the activations of processes on boolean streams require the total ordering given by a global scalar clock: a synchronous modeling of an asynchronous system.

c) Quasi-synchrony: Most existing work on the quasi-synchronous abstraction either assumes instantaneous communication [5], [28]—which may be valid in a shared memory model but not a message-passing one—or takes the discrete model as given and applies it directly to model and analyze systems [19], [20], [32]. We seek to clarify the original definitions [6] and to precisely define the relation between the real-time and discrete-time models. This leads to the understanding of discretization in terms of causality and the restrictions on process intercommunications and timing which are the central contributions of this paper.

Our work is complementary to the development of abstract domains to statically analyze synchronous real-time systems [4], and to the verification of properties like maximal lost messages, message inversions, and message latency, in an interactive theorem prover [24], [25].

In *n-synchrony*, unlike in *quasi-synchrony*, the difference of cumulative process activation counts is bounded [9]. The relation between a similar model and real-time has recently been studied [11]. Both n-synchrony and quasi-synchrony can be related to 'clock bounds' and 'drift bounds' [32].

VI. Conclusion

The quasi-synchronous abstraction provides a way to model and reason about a class of distributed embedded systems whose processes communicate by sampling with bounded jitter. Given a real-time model satisfying certain constraints on timing parameters and communication topologies, properties obtained of the corresponding quasi-synchronous model are also true of the original model. In other words, a precise class of practically-relevant distributed control systems can be verified without resorting to timed formalisms and tools, and by modeling message transmission as a unit delay, but not all of them.

Acknowledgments

We thank Cesare Tinelli for sharing his experiences on model-checking quasi-synchronous models and Thibault Rieutord and our anonymous reviewers for their suggestions and comments.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, pages 328–337, 2004.
[2] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *JACM*, 41(1):122–152, 1994.
[3] I. Ben-Zvi and Y. Moses. Beyond Lamport's happened-before: On time bounds and the ordering of events in distributed systems. In *DISC*, pages 421–436, 2010.
[4] J. Bertrane. *Static analysis of communicating imperfectly-clocked synchronous systems using continuous-time abstract domains.* PhD thesis, École Polytechnique, 2008.
[5] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli. Verification of quasi-synchronous systems with Uppaal. In *DASC*, pages 8A4–1–8A4–12, 2014.
[6] P. Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, Verimag, Crysis Project, 2000. "The Cooking Book".
[7] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *EMSOFT*, pages 80–96, 2001.
[8] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *SAFECOMP*, pages 215–226, 2001.
[9] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *POPL*, pages 180–193, 2006.
[10] F. Cristian. Synchronous and asynchronous group communication (long version). *CACM*, 1996.
[11] A. Desai, S. A. Seshia, S. Qadeer, D. Broman, and J. C. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. In *CAV*, pages 429–448, 2015.
[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.
[13] C. Fetzer, U. Schmid, and M. Süßkraut. On the possibility of consensus in asynchronous systems with finite average response times. In *ICDCS*, pages 271–280, 2005.
[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
[15] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *PODS*, pages 70–75, 1982.
[16] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. B. Jones, editors, *FMCAD*, pages 15:1–15:9, 2008.
[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79(9):1305–1320, 1991.
[18] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language LUSTRE. *IEEE Trans. Software Engineering*, 18(9):785–793, 1992.
[19] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *ACSD*, pages 3–14, 2006.
[20] E. Jahier, N. Halbwachs, and P. Raymond. Synchronous modeling and validation of schedulers dealing with shared resources. Technical Report 2008-10, Verimag, 2008.
[21] H. Kopetz. *Real-time systems: design principles for distributed embedded applications.* Springer-Verlag, 2011.
[22] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. IEEE*, 91(1):112–126, 2003.
[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.
[24] R. Larrieu and N. Shankar. A framework for high-assurance quasi-synchronous systems. In *MEMOCODE*, pages 72–83, 2014.
[25] W. Li, L. Gérard, and N. Shankar. Design and verification of multi-rate distributed systems. In *MEMOCODE*, pages 20–29, 2015.
[26] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
[27] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.
[28] S. Miller, S. Bhattacharyya, C. Tinelli, S. Smolka, C. Sticksel, B. Meng, and J. Yang. Formal verification of quasi-synchronous systems. Technical report, DTIC Document, 2015.
[29] P. Raymond. Recognizing regular expressions by means of dataflow networks. In *ICALP*, pages 336–347, 1996.
[30] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.
[31] P. Robinson and U. Schmid. The asynchronous bounded-cycle model. *TCS*, 412(1):5580–5601, 2011.
[32] G. Smeding and G. Goessler. A correlation preserving performance analysis for stream processing systems. In *MEMOCODE*, pages 11–20, July 2012.
[33] J. Widder and U. Schmid. The theta-model: achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

Synthesizing Adaptive Test Strategies from Temporal Logic Specifications

Roderick Bloem*, Robert Könighofer*, Ingo Pill†, and Franz Röck*
*Institute of Applied Information Processing and Communications, Graz University of Technology, Austria
†Institute of Software Technology, Graz University of Technology, Austria

***Abstract*—Constructing good test cases is difficult and time-consuming, especially if the system under test is still under development and its exact behavior is not yet fixed. We propose a new approach to compute test cases for reactive systems from a given temporal logic specification. The tests are guaranteed to reveal certain simple bugs (like occasional bit-flips) in *every* realization of the specification and for *every* behavior of the uncontrollable part of the system's environment. We aim at unveiling faults for the lowest of four fault occurrence frequencies possible (ranging from a single occurrence to persistence). Based on well-established hypotheses from fault-based testing, we argue that such tests are also sensitive for more complex bugs. Since the specification may not define the system behavior completely, we use reactive synthesis algorithms (with partial information) to compute *adaptive test strategies* that react to behavior at runtime. We work out the underlying theory and present first experiments demonstrating that our approach can be applied to industrial specifications and that the resulting strategies are capable of detecting bugs that are hard to detect with random testing.**

I. Introduction

Model checking [12], [42] cannot always be applied effectively to obtain confidence in the correctness of a system. Possible reasons include scalability issues, third-party IP components for which no code or detailed model is available, or a high effort for building system models that are sufficiently precise. Moreover, model checking cannot verify the final and "live" product but only an (abstracted) model.

Testing is a natural alternative to complement formal methods, and automatic test case generation helps keeping the effort manageable. Black-box techniques, where tests are derived from a specification rather than the implementation, are particularly attractive: First, tests can be computed before the implementation work starts, and can thus guide the development. Second, the same tests can be reused across different realizations of the same specification. Third, a specification is usually much simpler than its implementation, which gives a scalability advantage. At the same time, the specification focuses on the critical aspects that require thorough testing. Fault-based techniques [26], where test cases are computed to reveal certain fault classes, are particularly appealing — after all, the foremost goal in testing is to detect bugs.

Methods to derive tests from declarative requirements (see, e.g., [23]) are sparse. One issue in this setting is controllability: the requirements leave plenty of implementation freedom, so they cannot be used to fully predict the system behavior for given inputs. Consequently, test cases have to be *adaptive*, i.e., able to react to observed behavior at runtime, rather than being fixed input sequences. This is particularly true for *reactive systems* that interact with their environment. Existing methods often work around this complication by requiring a deterministic system model as additional input [22]. Even a probabilistic model fixes the behavior in a way not necessarily required by the specification.

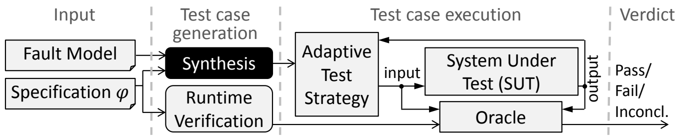


Fig. 1. Our testing setup. This paper focuses on test strategy synthesis.

We present a fault-based approach to compute adaptive test strategies for reactive systems such that certain coverage goals are achieved for *every* implementation of a given specification φ. The tests can thus be used across realizations of φ that differ not only in implementation details but also in their observable behavior. This is useful for standards and protocols that are implemented by multiple vendors, for systems under development, where the exact behavior is not yet fixed, etc.

Fig. 1 outlines our assumed testing setup, i.e., how our approach for synthesizing adaptive test strategies (illustrated in black) can be integrated in the testing chain. The user provides a specification φ, expressing requirements for the system under test (SUT) in Linear Temporal Logic (LTL) [40]. As coverage goal, the user also provides a fault model, which is an LTL formula that defines a class of faults for which the test shall cause a specification violation. We consider permanent and transient faults by distinguishing four fault occurrence frequencies[1] and compute tests to reveal faults for the lowest frequency possible. We compute test strategies using reactive synthesis [41] with partial information [29], providing strong guarantees about all uncertainties: If synthesis is successful and if the computed tests are executed long enough, they reveal all faults from the fault model for every realization of the specification and every behavior of the uncontrollable part of

This work was supported in part by the Austrian Science Fund (FWF) through the research network RiSE (S11406-N23) and by the European Commission through projects IMMORTAL (317753) and eDAS (608770).

[1]We consider faults that occur at least once, repeatedly, from some point on, or permanently.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

the system's environment[2]. Finally, existing techniques from runtime verification [6] can be used to build an oracle that checks the system behavior against the specification while tests are executed.[3]

In summary, this paper makes the following contributions:

- A novel approach to compute adaptive test strategies for reactive systems from temporal specifications that provide implementation freedom. The tests are guaranteed to reveal certain bugs for *every* specification realization.
- The underlying theory of our approach, i.e., we show that it is sound and complete for many interesting cases and provide a solution for the other cases.
- A proof of concept implementation and first experimental results demonstrating that our approach can be applied to industrial specifications and is capable of detecting bugs that are hard to detect with random testing.
- To the best of our knowledge, we are the first to combine fault-based and game-based testing.

Our approach is also an interesting application for synthesis. Synthesis suffers from scalability issues and the fact that writing complete specifications is hard. Our approach synthesizes test strategies from simple and incomplete specifications.

This paper is organized as follows. Section II discusses related work. Section III gives preliminaries and notation. Our test case generation approach is worked out in Section IV. Section V presents first experiments. Section VI concludes.

II. Background and Related Work

Fault-based testing. Fault-based test case generation methods like mutation testing [26] introduce simple faults into a system implementation or model and compute tests that uncover these faults. Based on hypotheses from fault-based testing [35], we argue that tests that reveal such faults are also sensitive to more complex issues. Two hypotheses extend the value of such tests. The Coupling Effect [16], [35] states that tests that detect simple faults are also sensitive to more complex faults. The Competent Programmer Hypothesis [16], [1] states that systems are mostly close to a correct version. Our approach follows the same philosophy, also relying on these hypotheses. However, most existing work focuses on permanent faults and deterministic system descriptions that define the behavior unambiguously. We also consider transient faults with different frequencies and uncover faults in *every* implementation of a given LTL [40] specification (and all behaviors of the uncontrollable part of the system's environment).

Adaptive tests. If the behavior of the system or the uncontrollable part of the environment is not fully specified, test cases may have to react to observed behavior at runtime in order to achieve their goals. Such adaptive test cases have been studied by Hierons [25] from a theoretical perspective, relying on fairness assumptions (every non-deterministic behavior is exhibited when trying often enough) or probabilities. Petrenko et al. compute adaptive tests for trace inclusion [37], [39], [38] or equivalence [36], [31], [38] from a specification given as non-deterministic finite state machine (FSM), also relying on fairness assumptions. Our work makes no such assumptions but considers the SUT to be fully antagonistic. Aichernig et al. [2] present a method to compute adaptive tests from (non-deterministic) UML state machines. Starting from an initial state, a trace to a goal state, the state that shall be covered by the resulting test case, is searched for every possible system behavior, issuing inconclusive verdicts only if the goal state is not reachable any more. Our approach uses reactive synthesis to enforce reaching the testing goal for all implementations if this is possible.

Testing as a game. Yannakakis [44] points out that testing reactive systems can be seen as a game between two players: the tester providing inputs and trying to reveal faults, and the SUT providing outputs and trying to hide faults. The tester can only observe outputs and has thus partial information about the SUT. The goal is to find a strategy for the tester that wins against every SUT. The underlying complexities are studied by Alur et al. [3]. Our work builds upon reactive synthesis [41] (with partial information [29]), which can also be seen as a game. However, we go far beyond the basic idea. We combine the game concept with user-defined fault models, work out the underlying theory, optimize the faults sensitivity in the temporal domain, and present a realization and experiments for LTL [40]. Nachmanson et al. [34] synthesize game strategies as tests for non-deterministic software models, but their approach is not fault-based and focuses on simple reachability goals. A variant considers the SUT to behave probabilistically with known probabilities [34]. This model is also used in [8]. Test strategies for reachability goals are also considered by David et al. [13] for timed automata.

Vacuity detection. [7], [30], [5] aim at finding cases where the specification is trivially satisfied (e.g., because the left side of an implication is false). Good tests avoid vacuities in order to challenge the SUT. The method by Beer et al. [7] can produce witnesses that satisfy the specification non-vacuously, which can serve as tests. Our approach avoid vacuities by requiring that certain faulty SUTs violate the specification.

Testing with a model checker. Various methods to compute tests from temporal specifications using a model checker have been proposed [23]. The method by Fraser and Ammann [20] ensures that properties are not vacuously satisfied and that faults propagate to observable property violations (using finite-trace semantics for LTL). Tan et al. [43] also define and apply a coverage metric based on vacuity for LTL. Ammann et al. [4] create tests from CTL [12] specifications using model mutations. All these methods assume that a deterministic system model is available in addition to the specification. Fraser and Wotawa [21] also consider non-deterministic models, but issue inconclusive verdicts if the system deviates from the behavior foreseen in the test case. In contrast, we search for

[2]Uncontrollable environment aspects can be seen as part of the system for the purpose of testing.

[3]While the semantics of LTL are defined over infinite execution traces, we can only run the tests for a finite amount of time. This can result in inconclusive verdicts [6]. We exclude this issue from the scope of this paper, relying on the user to judge when tests have been executed long enough, and on existing research on interpreting LTL over finite traces [24], [15], [14].

test strategies that achieve their goal for *every* realization of the specification. Boroday et al. [11] aim for a similar guarantee (calling it *strong test cases*) using a model checker, but do not consider adaptive test cases, and use an FSM as a specification.

III. Preliminaries and Notation

Traces. We want to test reactive systems that have a finite set $I = \{i_1, \dots, i_m\}$ of Boolean inputs and a finite set $O = \{o_1, \dots, o_n\}$ of Boolean outputs. The input alphabet is $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and $\Sigma = 2^{I \cup O}$. The set of infinite words over Σ is denoted by Σ^ω. We also refer to words as *(execution) traces*.

Linear Temporal Logic. We use *Linear Temporal Logic (LTL)* [40] as a specification language for reactive systems. The syntax is defined as follows: Every input or output $p \in I \cup O$ is an LTL formula; and if φ_1 and φ_2 are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\mathsf{X}\,\varphi_1$ and $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$. We write $\overline{\sigma} \models \varphi$ to denote that a trace $\overline{\sigma} = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ *satisfies* LTL formula φ. This is defined inductively as follows:

- $\sigma_0\sigma_1\sigma_2 \ldots \models p$ iff $p \in \sigma_0$,
- $\overline{\sigma} \models \neg\varphi$ iff $\overline{\sigma} \not\models \varphi$,
- $\overline{\sigma} \models \varphi_1 \vee \varphi_2$ iff $\overline{\sigma} \models \varphi_1$ or $\overline{\sigma} \models \varphi_2$,
- $\sigma_0\sigma_1\sigma_2 \ldots \models \mathsf{X}\,\varphi$ iff $\sigma_1\sigma_2 \ldots \models \varphi$, and
- $\sigma_0\sigma_1 \ldots \models \varphi_1 \mathbin{\mathsf{U}} \varphi_2$ iff $\exists j \geq 0 \,.\, \sigma_j\sigma_{j+1} \ldots \models \varphi_2 \wedge \forall 0 \leq k < j \,.\, \sigma_k\sigma_{k+1} \ldots \models \varphi_1$.

That is, $\mathsf{X}\,\varphi$ requires φ to hold in the *next* step, and $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$ means that φ_1 must hold *until* φ_2 holds (and φ_2 must hold eventually). We also use the usual abbreviations $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $\mathsf{F}\,\varphi = \mathsf{true} \mathbin{\mathsf{U}} \varphi$ (meaning that φ must hold *eventually*), and $\mathsf{G}\,\varphi = \neg\,\mathsf{F}\,\neg\varphi$ (φ must hold *always*). By $\varphi[x \leftarrow y]$ we denote the LTL formula φ where all occurrences of x have been textually replaced by y.

Mealy machines. We use Mealy machines to model the reactive system under test. A *Mealy machine* is a tuple $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \rightarrow Q$ is a total transition function, and $\lambda : Q \times \Sigma_I \rightarrow \Sigma_O$ is a total output function. Given the input trace $\overline{\sigma_I} = x_0x_1 \ldots \in \Sigma_I^\omega$, $\mathcal{S}$ produces the output trace $\overline{\sigma_O} = \mathcal{S}(\overline{\sigma_I}) = \lambda(q_0, x_0)\lambda(q_1, x_1) \ldots \in \Sigma_O^\omega$, where $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$. That is, in every time step i, the Mealy machine reads the input letter $x_i \in \Sigma_I$, responds with an output letter $\lambda(q_i, x_i) \in \Sigma_O$, and updates its state to $q_{i+1} = \delta(q_i, x_i)$. A Mealy machine can directly model synchronous hardware designs, but also other systems with inputs and outputs evolving in discrete time steps.

Moore machines. We use Moore machines to describe test strategies. A *Moore machine* is a special Mealy machine with $\forall q \in Q \,.\, \forall x, x' \in \Sigma_I \,.\, \lambda(q, x) = \lambda(q, x')$. That is, $\lambda(q, x)$ is insensitive to x, i.e., becomes a function $\lambda : Q \rightarrow \Sigma_O$. This means that the input x_i at step i can affect the next state q_{i+1} and thus the next output $\lambda(q_{i+1})$ but not the current output $\lambda(q_i)$. We write $\mathsf{Moore}(I, O)$ (resp. $\mathsf{Mealy}(I, O)$) for the set of all Moore (Mealy) machines with inputs I and outputs O.

Composition. When given two Mealy machines $\mathcal{S}_1 = (Q_1, q_{0,1}, 2^I, 2^{O_1}, \delta_1, \lambda_1) \in \mathsf{Mealy}(I, O_1)$ and $\mathcal{S}_2 = (Q_2, q_{0,2}, 2^{I \cup O_2}, 2^{O_2}, \delta_2, \lambda_2) \in \mathsf{Mealy}(I \cup O_1, O_2)$, we write $\mathcal{S} = \mathcal{S}_1 \circ \mathcal{S}_2$ for their sequential composition $\mathcal{S} = (Q_1 \times Q_2, (q_{0,1}, q_{0,2}), 2^I, 2^{I \cup O_1},\ \delta, \lambda) \in \mathsf{Mealy}(I, O_1 \cup O_2)$ with $\delta\big((q_1, q_2), x\big) = \big(\delta_1(q_1, x), \delta_2(q_2, x \cup \lambda_1(q_1, x))\big)$ and $\lambda\big((q_1, q_2), x\big) = \lambda_1(q_1, x) \cup \lambda_2\big(q_2, x \cup \lambda_1(q_1, x)\big)$.

Systems and test strategies. A *reactive system* $\mathcal{S}$ is a Mealy machine. An *(adaptive) test strategy* is a Moore machine $\mathcal{T} = (T, t_0, \Sigma_O, \Sigma_I, \Delta, \Lambda)$ with input and output alphabet swapped. That is, $\mathcal{T}$ produces values for input signals and reacts to values of output signals. A test strategy $\mathcal{T}$ can be *run* on a system $\mathcal{S}$ as follows. In every time step i (starting with $i = 0$), $\mathcal{T}$ first computes the next input $x_i = \lambda(t_i)$. Then, the system computes the output $y_i = \lambda(q_i, x_i)$. Finally, both machines compute their next state $t_{i+1} = \Delta(t_i, y_i)$ and $q_{i+1} = \delta(q_i, x_i)$. We write $\overline{\sigma}(\mathcal{T}, \mathcal{S}) = (x_0 \cup y_0)(x_1 \cup y_1) \ldots \Sigma^\omega$ for the resulting execution trace. If $\mathcal{T} = (T, t_0, 2^{O'}, \Sigma_I, \Delta, \Lambda) \in \mathsf{Moore}(O', I)$ can observe only a subset $O' \subseteq O$ of the outputs, we define $\overline{\sigma}(\mathcal{T}, \mathcal{S})$ with $t_{i+1} = \Delta(t_i, y_i \cap O')$. A *test suite* is a set $\mathrm{TS} \subseteq \mathsf{Moore}(O, I)$ of adaptive test strategies.

Realizability. A Mealy machine $\mathcal{S} \in \mathsf{Mealy}(I, O)$ *realizes* an LTL formula φ, written $\mathcal{S} \mathrel{|\!\!\models} \varphi$, if $\forall \mathcal{M} \in \mathsf{Moore}(O, I) \,.\, \overline{\sigma}(\mathcal{M}, \mathcal{S}) \models \varphi$. An LTL formula φ is *Mealy realizable* if there exists a Mealy machine that realizes it. Likewise, a Moore machine $\mathcal{M} \in \mathsf{Moore}(I, O)$ realizes φ, written $\mathcal{M} \mathrel{|\!\!\models} \varphi$, if $\forall \mathcal{S} \in \mathsf{Mealy}(O, I) \,.\, \overline{\sigma}(\mathcal{M}, \mathcal{S}) \models \varphi$.

Reactive synthesis. We use reactive synthesis to compute test strategies. A *reactive (Moore, LTL) synthesis procedure* takes as input a set I of Boolean inputs, a set O of Boolean outputs, and an LTL specification φ over these signals. It produces a Moore machine $\mathcal{M} \in \mathsf{Moore}(I, O)$ that realizes φ, or the message $\mathsf{unrealizable}$ if no such Moore machine exists. We denote this computation by $\mathcal{M} = \mathsf{synt}(I, O, \varphi)$. A *synthesis procedure with partial information* is defined similarly, but takes a subset $I' \subseteq I$ of the inputs as additional argument. It produces a Moore machine $\mathcal{M}' = \mathsf{synt}_p(I, O, \varphi, I')$ with $\mathcal{M}' \in \mathsf{Moore}(I', O)$ that realizes φ while only observing the inputs I', or the message $\mathsf{unrealizable}$ if no such Moore machine exists.

IV. Synthesis of Adaptive Test Strategies

This section presents our approach for synthesizing adaptive test strategies for reactive systems specified in LTL. First, we elaborate on the coverage objective we aim to achieve.Then we present our strategy synthesis algorithm. Finally, we discuss extensions and variants[4].

A. Coverage Objective for Test Strategy Computation

Many coverage metrics [33] have been proposed to assess the quality of a test suite. Since the goal in testing is to detect bugs, we follow a fault-centered approach: a test suite has high quality if it reveals certain kinds of faults in a system. As illustrated in Fig. 2, we assume that

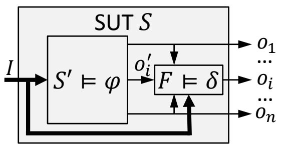


Fig. 2. Coverage goal illustration.

[4]All proofs can be found in the Appendix of the extended version available on arxiv.org

our SUT is "almost correct", i.e., that it is composed of a correct implementation $\mathcal{S}'$ of the specification φ, but with a fault F that affects one of the outputs. In order to make our approach flexible, we allow the user to define the considered faults as an LTL formula δ. Through δ, the user can define both permanent and transient faults of various types. For instance, $\delta = \mathsf{F}(o_i \leftrightarrow \neg o_i')$ describes a bit-flip that occurs at least once, $\mathsf{G}\,\mathsf{F}\,\neg o_i$ models a stuck-at-0 fault that occurs infinitely often, and $\mathsf{G}(\mathsf{X}(o_i) \leftrightarrow o_i')$ models a permanent shift by one time step. We strive for a test suite that reveals *every* fault that satisfies δ for *every* realization of φ. This renders the test suite independent of the implementation and the concrete fault manifestation. The following definition formalizes this intuition into a coverage objective.

Definition 1: A test suite $\mathsf{TS} \subseteq \mathsf{Moore}(O, I)$ for a system with inputs I, outputs O, and specification φ is *universally complete*[5] with respect to a given fault model δ iff

$$\begin{aligned}&\forall o_i \in O \,.\, \forall \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\} \setminus \{o_i\}) \,.\\ &\quad \forall F \in \mathsf{Mealy}(I \cup O \cup \{o_i'\} \setminus \{o_i\}, \{o_i\}) \,.\, \exists \mathcal{T} \in \mathsf{TS} \,.\\ &\Big(\big(\mathcal{S}' \models \varphi[o_i \leftarrow o_i'] \wedge F \models \delta\big) \rightarrow \big(\overline{\sigma}(\mathcal{T}, \mathcal{S}' \circ F) \not\models \varphi\big)\Big). \quad (1)\end{aligned}$$

That is, for every output o_i, system $\mathcal{S}' \models \varphi[o_i \leftarrow o_i']$, and fault $F \models \delta$, TS must contain a test strategy $\mathcal{T}$ that reveals the fault by causing a specification violation (cf. Fig. 2). Note that the test strategies $\mathcal{T} \in \mathsf{TS} \subseteq \mathsf{Moore}(O, I)$ cannot observe the signal o_i'. The reason is that this signal o_i' does not exist in the real system implementation(s) on which we run our tests — it was only introduced to define our coverage objective.

There can be an unbounded number of system realizations $\mathcal{S}' \models \varphi[o_i \leftarrow o_i']$ and faults $F \models \delta$. Computing a separate test strategy for each combination is thus not a viable option. We rather strive for computing only one test strategy per output variable.

Theorem 2: A universally complete test suite $\mathsf{TS} \subseteq \mathsf{Moore}(O, I)$ with respect to fault model δ exists for a system with inputs I, outputs O, and specification φ if

$$\begin{aligned}&\forall o_i \in O \,.\, \exists \mathcal{T} \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\\ &\qquad \overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \big((\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi\big). \quad (2)\end{aligned}$$

Intuitively, Theorem 2 holds because going from $\exists\mathcal{T}\forall\mathcal{S}$ to $\forall\mathcal{S}\exists\mathcal{T}$ and from $\overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \varphi[o_i \leftarrow o_i'] \wedge \delta$ to $\mathcal{S}' \circ F \models \varphi[o_i \leftarrow o_i'] \wedge \delta$ makes the formula weaker.

Theorem 2 states that Eq. 2 is a sufficient condition for a universally complete test suite to exist. If it were also a necessary condition, then computing one test strategy per output signal would be enough. Unfortunately, this is not the case in general.

Example 1. Consider a system with input $I = \{i\}$, output $O = \{o\}$, and specification $\varphi = \big(\mathsf{G}(i \rightarrow \mathsf{G}\,i) \wedge \mathsf{F}\,i\big) \rightarrow \big(\mathsf{G}(o \rightarrow \mathsf{G}\,o) \wedge \mathsf{F}\,o \wedge \mathsf{G}(i \vee \neg o)\big)$. The left side of the implication assumes that the input i is set to true at some point, after which i remains true. The right side requires the same for the output o. In addition, o must not be raised while i is still false. This specification is realizable (e.g., by always setting $o = i$). The test suite $\mathsf{TS} = \{\mathcal{T}_3\}$ with $\mathcal{T}_3$ shown in Fig. 3 is universally complete with respect to fault model $\delta = \mathsf{F}(o \leftrightarrow \neg o')$, which requires the output to flip at least once: As long as i is false, any correct system implementation $\mathcal{S}' \in \mathsf{Mealy}(\{i\}, \{o'\}) \models \varphi[o_i \leftarrow o_i']$ must keep the output $o' = \mathsf{false}$. Eventually, $F \models \delta$ must flip the output o to true. When this happens, i is set to true by $\mathcal{T}_3$ so that the resulting trace $\overline{\sigma}(\mathcal{T}, \mathcal{S}' \circ F)$ violates φ. Still, Eq. 2 is false[6]. Strategy $\mathcal{T}_3$ does not satisfy Eq. 2 because for the system $\mathcal{S} \in \mathsf{Mealy}(\{i\}, \{o, o'\})$ that sets $o' = \mathsf{true}$ and $o = \mathsf{false}$ in all steps, we have $\overline{\sigma}(\mathcal{T}_3, \mathcal{S}) \models \big(\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi\big)$. The reason is that i stays false, so $\varphi[o_i \leftarrow o_i']$ and φ are vacuously satisfied by $\overline{\sigma}(\mathcal{T}_3, \mathcal{S})$. The formula δ is satisfied because $o \leftrightarrow \neg o'$ holds in all time steps. Thus, $\mathcal{S}$ is a counterexample to $\mathcal{T}_3$ satisfying Eq. 2. Similar counterstrategies exist for all other test strategies.

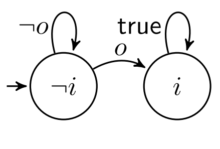

$$\begin{tikzcd}
\arrow[r] & \neg i \arrow[loop above, "\neg o"] \arrow[r, bend left, "o"] & i \arrow[loop above, "\mathsf{true}"]
\end{tikzcd}$$

Fig. 3. Strategy $\mathcal{T}_3$.

The fact that Eq. 2 is not a necessary condition for a universally complete test suite to exist is somewhat surprising, especially in the light of the following two lemmas. Based on these lemmas, the subsequent propositions will show that Eq. 2 is both sufficient and necessary (i.e., one test per output is enough) for many interesting cases.

Lemma 3: For every LTL specification ψ over some inputs I and outputs O, we have that $\exists \mathcal{T} \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \psi$ holds if and only if $\forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \exists \mathcal{T} \in \mathsf{Moore}(O, I) \,.\, \overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \psi$ holds.

Lemma 4: For all LTL specifications A, G over inputs I and outputs O, we have that

$$\begin{aligned}&\forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \exists \mathcal{T} \in \mathsf{Moore}(O, I) \,.\\ &\qquad (\mathcal{S} \models A) \rightarrow \big(\overline{\sigma}(\mathcal{T}, \mathcal{S}) \models G\big) \quad (3)\\ \text{iff}\quad &\forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \exists \mathcal{T} \in \mathsf{Moore}(O, I) \,.\\ &\qquad \overline{\sigma}(\mathcal{T}, \mathcal{S}) \models (A \rightarrow G). \quad (4)\end{aligned}$$

These two lemmas state that quantifiers can be swapped and that assuming $\overline{\sigma}(\mathcal{T}, \mathcal{S}) \models A$ is equivalent to assuming $(\mathcal{S} \models A)$ for the case where $\mathcal{T}$ has full information about the outputs of $\mathcal{S}$. Yet, in our setting, test strategies $\mathcal{T} \in \mathsf{Moore}(O, I)$ have incomplete information about the system $\mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\})$ because they cannot observe o_i'. Still, $\mathcal{T}$ must enforce $(\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi$, which refers to this hidden signal. Thus, Lemma 3 and 4 cannot be applied to Eq. 2 in general. However, in cases where there is (effectively) no hidden information, the lemmas can be used to prove that Eq. 2 is both a necessary and a sufficient condition for a universally complete test suite to exist. The following propositions show that this is the case for many cases of practical interest.

[5]The word "complete" indicates that every considered fault is revealed at every output. The word "universal" indicates that this is achieved for every (otherwise correct) system.

[6]This is (at least partially) confirmed by our test strategy synthesis tool: it reports that no test strategy with less than 12 states can satisfy Eq. 2.

Proposition 5: Given a fault model of the form $\delta = \mathsf{G}(o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O, a universally complete test suite $\mathrm{TS} \subseteq \mathsf{Moore}(O, I)$ with respect to δ, I, O, and φ exists if and only if Eq. 2 holds.

The intuitive reason is that $\varphi[o_i \leftarrow o_i']$ can be rewritten to $\varphi[o_i \leftarrow \psi]$ in Eq. 2, which eliminates the hidden signal such that Lemma 3 and 4 can be applied. Proposition 5 entails that computing one test strategy per output $o_i \in O$ is enough for fault models such as permanent bit flips (defined by $\delta = \mathsf{G}(o_i' \leftrightarrow \neg o_i)$).

Proposition 6: If the fault model δ does not reference o_i', a universally complete test suite $\mathrm{TS} \subseteq \mathsf{Moore}(O, I)$ with respect to δ, I, O, and φ exists iff Eq. 2 holds.

The assumption $\mathcal{S}' \mid\models \varphi[o_i \leftarrow o_i']$ can be dropped from Eq. 1 in this case. Correspondingly, $\overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \big((\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi\big)$ simplifies to $\overline{\sigma}(\mathcal{T}, \mathcal{S}) \models (\delta \rightarrow \neg\varphi)$ in Eq. 2. Since o_i' is now gone, Lemma 3 and 4 apply. In general, the assumption $\mathcal{S}' \mid\models \varphi[o_i \leftarrow o_i']$ is needed to prevent a faulty system $\mathcal{S}' \mid\not\models \varphi[o_i \leftarrow o_i']$ from compensating the fault $F \mid\models \delta$ such that $\mathcal{S}' \circ F \mid\models \varphi$. E.g., for $I = \emptyset$, $O = \{o\}$, $\varphi = \mathsf{G}\, o$ and $\delta = \mathsf{G}(o \leftrightarrow \neg o')$, Eq. 1 would be false without $\mathcal{S}' \mid\models \varphi[o_i \leftarrow o_i']$ because there exists an $\mathcal{S}'$ that always sets $o' = \mathsf{false}$, in which case $\mathcal{S}' \circ F$ has o correctly set to true. However, if δ does not reference o', such a fault compensation is not possible.

Proposition 6 applies to permanent or transient stuck-at-0 or stuck-at-1 faults (e.g., $\delta = \mathsf{F}\,\neg o_i$ or $\delta = \mathsf{G}\,\mathsf{F}\, o_i$), but also to faults where o_i keeps its previous value (e.g., $\delta = \mathsf{F}(o_i \leftrightarrow \mathsf{X}(o_i))$) or takes the value of a different input or output (e.g., $\delta = \mathsf{G}\,\mathsf{F}(o_i \leftarrow i_3)$). Together with Proposition 5, it shows that computing one test strategy per output is enough for many interesting fault models. Finally, even if neither Proposition 5 nor Proposition 6 applies, computing one test strategy per output may still suffice for the concrete φ and δ at hand. In the next section, we thus rely on Eq. 2 to compute one test strategy per output in order to obtain universally complete test suites.

B. Test Strategy Computation

Basic idea. Our test case generation approach builds upon Theorem 2: For every output $o_i \in O$, we want to find a test strategy $\mathcal{T}_i \in \mathsf{Moore}(O, I)$ such that $\forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \overline{\sigma}(\mathcal{T}, \mathcal{S}) \models \big((\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi\big)$ holds. Recall from Section III that a synthesis procedure $\mathcal{M} = \mathsf{synt}_p(I, O, \psi, I')$ with partial information computes a Moore machine $\mathcal{M} \in \mathsf{Moore}(I', O)$ with $I' \subseteq I$ such that a certain LTL objective ψ is enforced in all environments, i.e., $\forall \mathcal{S} \in \mathsf{Mealy}(O, I) \,.\, \overline{\sigma}(\mathcal{M}, \mathcal{S}) \models \psi$. If no such $\mathcal{M}$ exists, synt_p returns $\mathsf{unrealizable}$. Also recall that a test strategy is a Moore machine with input and output signals swapped. We can thus call $\mathcal{T}_i := \mathsf{synt}_p\big(O \cup \{o_i'\}, I, (\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi, O\big)$ for every output $o_i \in O$ in order to obtain a universally complete test suite with respect to fault model δ for a system with inputs I, outputs O, and specification φ. If synt_p succeeds (does not return $\mathsf{unrealizable}$) for all $o_i \in O$, the resulting test suite $\mathrm{TS} = \{\mathcal{T}_i \mid o_i \in O\}$ is guaranteed to be universally complete. However, since Theorem 2 only gives a sufficient but not a necessary condition, this procedure may fail to find a universally complete test suite, even if one exists, in general. In cases where Proposition 5 or Proposition 6 applies, it is both sound and complete, though.

Algorithm 1 SYNTLTLTEST: Synthesizes adaptive test strategies from an LTL spec.

```
1: procedure SYNTLTLTEST($I, O, \varphi, \kappa$), returns: A set TS of test strategies $\mathcal{T}_i$
2:     TS := $\emptyset$
3:     for each $o_i \in O$ do
4:         for each frq from ($\mathsf{F}, \mathsf{G\,F}, \mathsf{F\,G}, \mathsf{G}$) in this order do
5:             $\mathcal{T}_i := \mathsf{synt}_p\big(O \cup \{o_i'\}, I, (\varphi[o_i \leftarrow o_i'] \wedge \mathsf{frq}(\kappa)) \rightarrow \neg\varphi, O\big)$
6:             if $\mathcal{T}_i \neq$ unrealizable then
7:                 TS := TS $\cup \{\mathcal{T}_i\}$; break
8:     return TS
```

Fault models. To simplify the user input, we split the fault model δ in our coverage objective from Definition 1 into two parts: the fault kind κ and the fault frequency frq. The fault kind κ is an LTL formula that is given by the user and defines *which* faults we consider. For instance, $\kappa = \neg o_i$ describes a stuck-at-0 fault, $\kappa = o_i \leftrightarrow \neg o_i'$ defines a bit-flip, and $\kappa = o_i' \leftrightarrow \mathsf{X}(o_i)$ describes a delay by one time step. The fault frequency frq describes *how often* a fault of the specified kind occurs, and is chosen by our algorithm. We distinguish 4 fault frequencies, which we describe using temporal LTL operators.

- Fault frequency G means that the fault is permanent.
- Frequency $\mathsf{F\,G}$ means that the fault occurs from some time step i on permanently. Yet, we do not make any assumptions about the precise value of i.
- Frequency $\mathsf{G\,F}$ states that the fault strikes infinitely often, but not when exactly.
- Frequency F means that the fault occurs at least once.

The fault model δ is then defined as $\delta = \mathsf{frq}(\kappa)$. Note that there is a natural order among our 4 fault frequencies: a fault of kind κ that occurs permanently (frequency G) is just a special case of the same fault κ occurring from some point onwards (frequency $\mathsf{F\,G}$), which is in turn a special case of κ occurring infinitely often (frequency $\mathsf{G\,F}$), which is a special case of κ occurring at least once. Thus, a test strategy that reveals a fault that occurs at least once (without knowing when) will also reveal a fault that occurs infinitely often, etc. In our approach, we thus compute test strategies to detect faults at the lowest frequency for which a test strategy can be found.

Algorithm. The procedure SYNTLTLTEST in Algorithm 1 formalizes our approach. The input consists of (1) the inputs I of the SUT, (2) the outputs O of the SUT, (3) an LTL specification φ of the SUT, and (4) a fault kind κ. The result of SYNTLTLTEST is a test suite TS. The algorithm iterates over all outputs $o_i \in O$ (Line 3) and over our 4 fault frequencies (Line 4), starting with the lowest one. Line 5 attempts to compute a strategy to reveal a fault that is compliant with the provided fault kind for the current frequency. If such a strategy exists, it is added to TS and the next output is processed. Otherwise, the algorithm tries the next higher fault frequency.

Sanity checks. Note that our coverage goal in Eq. 1 is vacuously satisfied by any test suite if φ or δ is unrealizable. The reason is that the test suite must reveal *every* fault F realizing δ for *every* system $\mathcal{S}'$ realizing φ. If there is no such fault or system, this is trivial. As a sanity check, we thus test the (Mealy) realizability of φ and $\mathsf{G}\,\kappa$ before starting Algorithm 1 (because if $\mathsf{G}\,\kappa$ is realizable, then so are $\mathsf{F}\,\mathsf{G}\,\kappa$, $\mathsf{G}\,\mathsf{F}\,\kappa$ and $\mathsf{F}\,\kappa$).

Handling unrealizability. If, for some output, Line 5 of Alg. 1 returns unrealizable for the highest fault frequency $\mathsf{frq} = \mathsf{G}$, we print a warning and suggest that the user examines these cases manually. There are two possible reasons for unrealizability. First, due to limited observability, we do not find a test strategy although one exists (see Example 1). Second, no test strategy exists because there is some $\mathcal{S}' \,|\!\!\models \varphi[o_i \leftarrow o_i']$ and $F \,|\!\!\models \delta$ such that the composition $\mathcal{S} = \mathcal{S}' \circ F$ (see Fig. 2) is correct, i.e., $\mathcal{S}' \circ F \,|\!\!\models \varphi$. In other words, for some realization, adding the fault may result in an equivalent mutant in the sense that the specification is still satisfied. For example, in case of a stuck-at-0 fault model, there may exist a realization of the specification that has the considered output $o_i \in O$ fixed to false. Such a high degree of underspecification is at least suspicious and may indicate unintended vacuities [7] in the specification φ, which should be investigated manually. If Proposition 5 or 6 applies, or if $\mathsf{synt}\big(O \cup \{o_i'\}, I, \big(\varphi[o_i \leftarrow o_i'] \wedge \mathsf{G}(\kappa)\big) \rightarrow \neg\varphi\big)$ returns unrealizable, we can be sure that the second case applies. Then, we can even compute additional diagnostic information in the form of two Mealy machines $\mathcal{S}' \,|\!\!\models \varphi[o_i \leftarrow o_i']$ and $F \,|\!\!\models \delta$ (by synthesizing some Mealy machine $S \,|\!\!\models (\varphi[o_i \leftarrow o_i'] \wedge \mathsf{G}(\kappa) \wedge \varphi)$ and splitting it into $\mathcal{S}'$ and F by stripping off different outputs). The user can then try to find inputs for $\mathcal{S}' \circ F$ such that the resulting trace violates the specification. Failing to do so, the user will understand why no test strategy exists (see also [28]). For cases where the specification is as intended but no test strategy exists, we can follow the approach by Faella [17], [18] to synthesize best-effort strategies that are not guaranteed to cause a specification violation but at least do not give up trying. But we leave this extension for future work.

Optimizations. As discussed at Proposition 6, Line 5 in Algorithm 1 can be simplified to $\mathcal{T}_i := \mathsf{synt}\big(O, I, \mathsf{frq}(\kappa) \rightarrow \neg\varphi\big)$ if κ does not reference o_i'. If frq=G and κ is of the form $\kappa = (o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O, Line 5 can be simplified to $\mathcal{T}_i := \mathsf{synt}\big(O, I, \varphi[o_i \leftarrow \psi] \rightarrow \neg\varphi\big)$. This is justified by the proof of Proposition 5. Note that in these cases, a synthesis procedure without partial information suffices.

Complexity. Both $\mathsf{synt}_p(O, I, \psi, O')$ and $\mathsf{synt}(O, I, \psi)$ are 2EXPTIME complete in $|\psi|$ [29], so the execution time of Alg. 1 is at most doubly exponential in $|\varphi| + |\kappa|$.

Theorem 7: For a system with inputs I, outputs O, and LTL specification φ over $I \cup O$, if the fault kind κ is of the form $\kappa = \psi$ or $\kappa = (o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O, $\textsc{SyntLtlTest}(I, O, \varphi, \kappa)$ will return a universally complete test suite with respect to the fault model $\delta = \mathsf{G}(\kappa)$ if such a test suite exists.

Theorem 7 states that SYNTLTLTEST is not only sound but also complete for many interesting fault models such as stuck-at faults or permanent bit-flips. For $\kappa = \psi$, Theorem 7 can even be strengthened to hold for all $\delta = \mathsf{frq}(\kappa)$ with $\mathsf{frq} \in \{\mathsf{F}, \mathsf{G}\,\mathsf{F}, \mathsf{F}\,\mathsf{G}, \mathsf{G}\}$.

C. Extensions and Variants

Faults at inputs. In this paper, we only consider faults at the outputs. However, considering SUTs that behave as if they would have read a faulty input is possible as well (by changing Line 3 in Algorithm 1 to "**for** each $o \in I \cup O$ **do**").

Multiple faults. Simultaneous faults at multiple (inputs or) outputs $\{o_1, \ldots, o_k\} \subseteq O$ can be considered by computing a test strategy
$\mathcal{T} := \mathsf{synt}_p\big(O \cup \{o_1', \ldots, o_k'\}, I, (\varphi[o_1 \leftarrow o_1', \ldots, o_k \leftarrow o_k'] \wedge \bigwedge_{i=1}^{k} \delta_i) \rightarrow \neg\varphi, O\big)$, where the fault model δ_i can be different for different outputs $o_i \in \{o_1, \ldots, o_k\}$.

Mutating the specification. We can also synthesize adaptive test strategies that would uncover bugs where the SUT implements a mutated (i.e., slightly modified) specification φ' instead of φ by calling $\mathcal{T} := \mathsf{synt}(O, I, \varphi' \rightarrow \neg\varphi)$. The implication requires the original specification φ to be violated under the assumption that the mutated specification φ' holds for the SUT. This variant does not require partial information synthesis.

Other specification formalisms. We worked out our approach for LTL, but it would work for other languages if (1) the language is closed under Boolean connectives ($\wedge, \neg$), (2) the desired fault models are expressible, and (3) a synthesis procedure (with partial information) is available. These prerequisites do not only apply to many temporal logics but also to various kinds of automata over infinite words.

V. Experimental Results

We evaluated our method on two different specifications: (1) an industrial specification, the AMBA Bus Arbiter for 2 masters [9], and (2) a specification of a door system that requires sophisticated strategies to satisfy the coverage objectives.

For the AMBA Bus Arbiter, we extended the LTL synthesis tool PARTY [27]. PARTY implements SMT-based bounded synthesis [19] for LTL, which sets a bound b on the number of states of the system to be synthesized. This bound is increased iteratively until a solution is found. For the second experiment, where we only use fault models that do not require partial information, we use the synthesis tool Acacia+ [10].

Test Setup. We generated mutants – stuck-at-0 and stuck-at-1 faults – for every line in the source code of the implementation and dropped those that do not violate the specification. Then we tested the mutated implementations for 20 time steps. All experiments are performed on an Intel Xeon E5430 CPU running at 2.66 GHz with a 64 bit Linux using one CPU core[7].

[7]Our implementation, input files and scripts are provided at https://seafile.iaik.tugraz.at/d/0d043a57d2/ with password `fmcad2016`.

TABLE I
RESULTS FOR THE AMBA BUS ARBITER. THE SUFFIX "K" MULTIPLIES BY 10^3.

		Decide Next				Start Access				Grant Bus				Full Spec			
Fault	o_i	frq	$\lvert\mathcal{T}\rvert$	sec	MB	frq	$\lvert\mathcal{T}\rvert$	sec	MB	frq	$\lvert\mathcal{T}\rvert$	sec	MB	frq	$\lvert\mathcal{T}\rvert$	sec	MB
Stuck at 0 ($\kappa = \neg o_i$)	hmaster0	FG	2	359	peak: 574 MB	-	-	147	peak: 138 MB	-	-	146	peak: 131 MB	GF	2	4,848	peak: 2,207 MB
	hgrant0	F	2	18						G	2	150		F	2	2,082	
	hgrant1	-	-	856						-	-	172		GF	2	4,991	
	hmastlock	-	-	803		-	-	133		-	-	133		GF	2	5,808	
	start					G	2	126		G	2	230		FG	2	9,367	
	locked	-	-	736						-	-	170		GF	2	5,236	
	decide	G	2	689										FG	2	9,934	
Stuck at 1 ($\kappa = o_i$)	hmaster0	FG	2	1,237	peak: 783 MB	G	2	133	peak: 130 MB	G	2	153	peak: 131 MB	F	2	2,388	peak: 1,917 MB
	hgrant0	-	-	6,775						-	-	171		GF	2	5,681	
	hgrant1	F	2	19						G	2	151		F	2	1,970	
	hmastlock	G	2	9,64		G	2	115		G	2	186		F	2	1,473	
	start					GF	3	53		-	-	129		GF	2	5,934	
	locked	GF	2	800						-	-	202		GF	2	5,423	
	decide	-	-	1,011										GF	2	4,169	
Flip ($\kappa = o_i \leftrightarrow \neg o_i'$)	hmaster0	G	2	22k	peak: 6,176 MB	G	2	54k	peak: 472 MB	GF	2	1,828	peak: 1,476 MB	Timeout (> 6 days for first output)			
	hgrant0	F	2	29						F	2	10					
	hgrant1	F	2	38						F	2	10					
	hmastlock	G	2	3,385		G	2	53k		GF	2	1,057					
	start					FG	2	43k		G	2	163					
	locked	GF	2	1,525						GF	2	86					
	decide	F	3	61													

A. *AMBA Bus Arbiter Case Study*

The ARM AMBA bus arbiter specification is an industrial specification. Deriving test cases for it illustrates that our approach can successfully handle real world examples.

The specification has 7 inputs, and 7 outputs. The properties can be clustered into 3 (interdependent) parts [9]: deciding about the next access, starting an access, and granting the bus. In order to improve scalability and demonstrate that our approach can operate on incomplete specifications, we synthesize test strategies for these 3 parts separately. Each part is combined with all assumptions to ensure that the synthesized test strategies can be run on the full system.

Results. Table I summarizes the results. The groups in the rows contain results for different fault models. Sub-rows distinguish the output signals $o_i \in O$. The column-blocks contain results for the 3 specification parts and the full specification. The sub-columns list (1) the lowest fault frequency for which Alg. 1 found a solution ("-" indicates that no strategy with $\leq b$ states exists, even with frq = G), (2) the number of states in the resulting test strategy, (3) the execution time, and (4) the peak memory consumption over all outputs. An empty sub-row indicates that the output does not occur in that specification parts.

In many cases, the synthesized test strategies cannot only reveal permanent faults but also transient faults with low frequencies. For stuck-at-0 and stuck-at-1, we can consider the entire spec, and we get such strategies for 12 cases. If we use the fault model that flips the output, we have to restrict ourselves to a subset of the spec. Nevertheless, we can derive another 6 strategies of the required quality.

Applying the strategies derived from the full specification on mutated implementations, the tests are able to detect between 25% and 48% of our faulty implementations with transient mutants. Random sequences are able to detect the same number.

Discussion. Deriving strategies for parts of the specification that reveal (transient) flips succeeds more often than deriving strategies revealing (transient) stuck-at faults because, with the latter fault model, an output may be (temporarily) stuck at the correct value. On the other hand, synthesizing flip-tests consumes more resources because the optimization discussed in Proposition 6 cannot be applied. This is also the reason for the timeout with flips on the full specification. Although test stragies are found for most outputs when processing the 3 specification parts separately, processing the full specification yields better strategies but takes longer.

While the specification is large, it is still rather simple which is confirmed by the resulting strategies which all contain at most three states. Therefore, it is not unexpected that random testing is as successful as our strategies. The achieved score on killed mutants illustrates a quite successful application of our approach on a real world example as we generate strategies without any knowledge of the particular implementation and the strategies, therefore, only focus on detecting faulty output signals.

B. *Door with PIN*

Although the AMBA specification is industrial, the strategies are rather straightforward. To evaluate our approach on a specification that is hard to test with random testing, we apply it on an example that specifies the opening mechanism of a door that is secured by a PIN when the door is locked and requires inputs that are are not very likely to be hit by random sequences.

The specification of the example has 7 inputs, and 5 outputs. The first output signal, `doorclosed`, displays whether the door is closed or open, the second, `doorlocked`, whether it is locked or unlocked. The other three binary output signals provide an arbitrary PIN value in the range of 0 . . . 7. The 4 input signals control the `open`, `close`, `lock` and `unlock` mechanism of the door, and the door is only opened if it is not locked. Only one of these signals can be high at a time. The other 3 inputs are binary PIN inputs. Whenever `unlock` is high, the next two time steps the input PIN signals have to copy the displayed PIN to satisfy the security policy, a requirement that is unlikely to be met with random testing. If the PIN is wrong the first time, the door is locked even if it was unlocked before.

TABLE II
RESULTS FOR THE DOOR SPECIFICATION.

Fault	o_i	frq	$\lvert\mathcal{T}\rvert$	sec	MB
stuck-at-0					
	doorclosed	GF	25	22,341	347
	doorlocked	FG	29	2,425	285
stuck-at-1					
	doorclosed	GF	45	23,290	1,000
	doorlocked	FG	52	3.100	148

Results. Table II summarizes the results using Acacia+. Our approach is able to derive G F strategies for the output `doorclosed`. For the output `doorlocked` our approach derived strategies detecting F G faults. As the number of states indicates, the strategies are larger, because they are independent of hardcoded PINs.

When testing a door implementation containing permanent faults, our strategies are able to detect 83% of our stuck-at-0 mutants and 95% of our stuck-at-1 mutants while random tests only detect 29% of the former and 10% of the latter.

Discussion. While `doorlocked` is not specified for every step in time, our approach is still able to derive strategies. Although our strategies originally only focus on stuck-at faults of the output values they are able to detect most of the mutants. The low success-rate of random tests is due to the fact that they fail entering the correct PIN, whereas our strategies are able to come across such difficult prefixes.

VI. Conclusion

We presented a new approach to compute adaptive test strategies from temporal logic specifications using reactive synthesis with partial information. The computed test strategies reveal all instances of a user-defined fault class for every realization of a given specification. Thus, they do not rely on implementation details, which is important for products that are still under development or for standards that will be implemented by multiple vendors. Our approach is sound but incomplete in general, i.e., may fail to find test strategies even if they exist. However, for many interesting cases, we showed that it is both sound and complete. The worst-case complexity is doubly exponential in the specification size, but in our setting, the specifications are typically small. This also makes our approach an interesting application for reactive synthesis. Our experiments demonstrate that our approach can compute meaningful tests for industrially sized specifications and that generated strategies are capable of detecting faults hidden in paths that are unlikely taken by random sequences.

Current directions for future work include improving scalability, success-rate, and usability of our approach. To this end, we are investigating using random testing for inputs in the strategies that are not fixed to single values, and best-effort strategies [17], [18] for the case that there are no test strategies that can guarantee triggering the fault.

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, 1979.

[2] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 25(8):716–748, 2015.

[3] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *STOC'95*. ACM, 1995.

[4] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS'01*, pages 212–221. IEEE, 2001.

[5] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *CAV'03*, LNCS 2725, pages 368–380. Springer, 2003.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[7] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulaas. In *CAV'97*, pages 279–290. Springer, 1997.

[8] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. In *FATES'05*, LNCS 3997, pages 32–46. Springer, 2005.

[9] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *DATE'07*, pages 1188–1193, 2007.

[10] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for ltl synthesis. In *Computer Aided Verification*, pages 652–657. Springer, 2012.

[11] S. Boroday, A. Petrenko, and R. Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007.

[12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer, 1981.

[13] A. David, K. G. Larsen, S. Li, and B. Nielsen. A game-theoretic approach to real-time system testing. In *DATE'08*. ACM, 2008.

[14] G. De Giacomo, R. De Masellis, and M. Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *AAAI'14*, 2014.

[15] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13*. IJCAI/AAAI, 2013.

[16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[17] M. Faella. Games you cannot win. In *Workshop on Games and Automata for Synthesis and Validation*, Lausanne, Switzerland, September 2007.

[18] M. Faella. Admissible strategies in infinite games over graphs. In *MFCS'09*, LNCS 5734, pages 307–318. Springer, 2009.

[19] B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[20] G. Fraser and P. Ammann. Reachability and propagation for LTL requirements testing. In *QSIC'08*, pages 189–198. IEEE, 2008.

[21] G. Fraser and F. Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *ICSEA'07*, 2007.

[22] G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403–1418, 2009.

[23] G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.*, 19(3):215–261, 2009.

[24] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *ASE'01*, pages 135–143. IEEE, 2001.

[25] R. M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Information Processing Letters*, 98(2):56–60, 2006.

[26] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[27] A. Khalimov, S. Jacobs, and R. Bloem. PARTY: Parameterized synthesis of token rings. In *CAV'13*, LNCS 8044, pages 928–933. Springer, 2013.

[28] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD'09*. IEEE, 2009.

[29] O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *ICTL'97*, pages 91–106, 1997.

[30] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

[31] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994.

[32] D. A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.

[33] A. P. Mathur. *Foundations of Software Testing*. Addison-Wesley, 2008.

[34] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, pages 55–64. ACM, 2004.

[35] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[36] A. Petrenko, A. da Silva Simão, and N. Yevtushenko. Generating checking sequences for nondeterministic finite state machines. In *ICST'12*, pages 310–319. IEEE, 2012.

[37] A. Petrenko and A. Simão. Generalizing the DS-methods for testing non-deterministic FSMs. *Computer Journal*, 58(7):1656–1672, 2015.

[38] A. Petrenko and N. Yevtushenko. Conformance tests as checking experiments for partial nondeterministic FSM. In *FATES'05*, LNCS 3997, pages 118–133. Springer, 2005.

[39] A. Petrenko and N. Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *HASE'14*, pages 224–228. IEEE, 2014.

[40] A. Pnueli. The temporal logic of programs. In *FOCS'77*. IEEE, 1977.

[41] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL'89*, pages 179–190. ACM, 1989.

[42] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, LNCS 137, pages 337–351. Springer, 1982.

[43] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'04*, pages 493–498. IEEE, 2004.

[44] M. Yannakakis. Testing, optimizaton, and games. In *LICS'04*, 2004.

Reducing Interpolant Circuit Size by Ad-Hoc Logic Synthesis and SAT-Based Weakening

G. Cabodi, P. E. Camurati, M. Palena, P. Pasini, D. Vendraminetto
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Turin, Italy
Email: {gianpiero.cabodi, paolo.camurati, marco.palena, paolo.pasini, danilo.vendraminetto}@polito.it

***Abstract*—We address the problem of reducing the size of Craig interpolants used in SAT-based Model Checking. Craig interpolants are AND-OR circuits, generated by post-processing refutation proofs of SAT solvers. Whereas it is well known that interpolants are highly redundant, their compaction is typically tackled by reducing the proof graph and/or by exploiting standard logic synthesis techniques. Furthermore, strengthening and weakening have been studied as an option to control interpolant quality.**

In this paper we propose two interpolant compaction techniques: (1) A set of ad-hoc logic synthesis functions that, revisiting known logic synthesis approaches, specifically address speed and scalability. Though general and not restricted to interpolants, these techniques target the main sources of redundancy in interpolant circuits. (2) An interpolant weakening technique, where the UNSAT core extracted from an additional SAT query is used to obtain a gate-level abstraction of the interpolant. The abstraction introduces fresh new variables at gate cuts that must be quantified out in order to obtain a valid interpolant. We show how to efficiently quantify them out, by working on an NNF representation of the circuit.

The paper includes an experimental evaluation, showing the benefits of the proposed techniques, on a set of benchmark interpolants arising from both hardware and software model checking problems.

I. Introduction

Craig interpolants (ITPs) [1], introduced by McMillan [2] in the Unbounded Model Checking (UMC) field, have shown to be effective on difficult verification instances.

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

From the perspective of Software Model Checking, instead, interpolants are used to strengthen the results of predicate abstraction [3]. In case the inductive invariant representing a program is insufficient to prove a given property, interpolants can be used as predicates to refine such an abstraction [4].

The most interesting features of Craig interpolants are their completeness and the fact can be used as an automated abstraction mechanism, whereas one of their major drawbacks is the inherent redundancy of interpolant circuits, as well as the need for fast and scalable techniques to compact them. Improvements over the base method [2] were proposed in [5], [6], [7], [8] and [9], in order to push forward applicability and scalability of the technique.

Craig interpolants can be computed as AND-OR circuits, generated by post-processing refutation proofs of SAT solvers. Modern SAT solvers are capable, without incurring into large additional cost, to generate a resolution proof from unsatisfiable runs [10]. Due to the nature of the algorithms employed by SAT solvers, a resolution proof may contain redundant parts and a strictly smaller resolution proof can be obtained.

Although a Craig interpolant is linear in the proof size, the proof itself may be large and highly redundant. SAT solvers are not usually targeted to produce proofs of minimal size, therefore they may be deemed ultimately responsible for Craig interpolant size and redundancy. This is the main reason why most efforts on interpolant size reduction have been addressed as SAT solver improvement and/or proof reduction.

A. Contributions

In this paper we propose a fast and scalable logic synthesis approach, as well as a novel interpolant weakening (and strengthening) technique that also addresses circuit compaction. The main contributions are thus two interpolant compaction techniques:

- A set of ad-hoc logic synthesis functions specifically addressing speed and scalability. Though general and not limited to interpolants, they target the main sources of redundancy int interpolant circuits;
- An interpolant weakening technique, where an additional SAT query is performed in order to obtain a gate-level abstraction of the interpolant. Although fresh new variables are introduced at gate cuts, clearly outside the set of shared symbols, we show how to quantify them out by working on an NNF encoding of the circuit.

B. Related works

Interpolant compaction has been addressed in [11] and [12]. With respect to [11], we present additional techniques addressing scalability and interpolant compaction by weakening/strengthening. Interpolant weakening/strengthening is the subject of many papers, with little relation with our work. Among them, we consider [13] for an interesting discussion on the relationship between interpolant strength and quality.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

The notion of dominance between nodes of a directed graph is central in this work. Dominators have been used in the context of logic synthesis before, such as [14], [15].

C. Outline

Section II introduces background notions and notation about Boolean circuits, Craig interpolants, gate-level abstraction and circuit compaction techniques. Section III describes the proposed ad-hoc logic synthesis functions, whereas our interpolant weakening technique is illustrated in Section IV. Section V presents and discusses the experiments we performed. Finally, Section VI concludes with some summarizing remarks.

II. Background

A. Combinational Boolean Circuits

Definition 1. *A* Boolean circuit *(or* network*) is a directed acyclic graph $\mathcal{G} = (V, E)$, where a node $v \in V$ represents either a* logic gate*, a* primary input *(PI) or a* primary output *(PO) of the circuit and each directed edge $(u, v) \in E$ represents a signal in the circuit connecting the output of node u to an input of node v. The fanin (fanout) of a node is the set of incoming (outgoing) edges of that node. Primary inputs are nodes with no fanin, whereas primary outputs are nodes with no fanout. Every logic gate $v \in V$ is associated with a Boolean function $f_v : \mathbb{B}^n \to \mathbb{B}$, where n is its number of inputs.*

The *fanin* (*fanout*) sets are typically represented by *lists*. With abuse of notation we use the terms fanin and fanout to identify both edges and the related sets of adjacent nodes. Given a gate node v, $type(v)$ is used to indicate the type of logic function associated with v (AND, OR, NOT, etc.).

Definition 2. *Given a circuit $\mathcal{G} = (V, E)$, a node u* dominates[1] *a node v iff every path from v to any of the primary outputs of $\mathcal{G}$ contains u. A node u that dominates a node v is called a* dominator *of v.*

Definition 3. *Given a circuit $\mathcal{G} = (V, E)$ and a node r, a* cone *$\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ rooted in r is a sub-graph of $\mathcal{G}$ consisting of r and some of its non–primary input predecessors such that any node in $\mathcal{C}$ has a path to r that lies entirely in $\mathcal{C}$. The fanin (fanout) of a cone is the number of nodes u not in $\mathcal{C}$ that are inputs (outputs) of a node t in $\mathcal{C}$.*

Node r is called *root* of the cone $\mathcal{C}$, and denoted by $root(\mathcal{C})$, non-root nodes of the cone are called *internal nodes*, whereas nodes in the fanin of the cone are called *cut nodes* of $\mathcal{C}$ and denoted by $cut(\mathcal{C})$. Nodes of $\mathcal{C}$ that have at least one cut node v in their fanin are called *entry points* in $\mathcal{C}$ for v. The Boolean function f_v associated with the cone root is called *cone function*. With abuse of notation we sometimes use $v \in \mathcal{C}$ to mean that $v \in V_{\mathcal{C}}$.

[1]Note that the notion of dominance as defined here corresponds to the dual notion of post-dominance from graph theory. For the sake of conciseness, we herein use the term dominance, with the definition provided above, to refer to the actual notion of post-dominance.

Definition 4. *A* cluster *is a cone $\mathcal{C}$ rooted in r such that, for each node v in $\mathcal{C}$, v has unit fanout and is dominated by r in $\mathcal{G}$.*

Note that cut nodes of a cluster $\mathcal{C}$ are either a PI or fanout branches, and the root r of $\mathcal{C}$ is either a PO or a fanout stem. Note also that the sub-graph of the circuit that defines a cluster $\mathcal{C}$ is a tree. Given a node $v \in \mathcal{C}$, every successor u of v in $\mathcal{C}$ is a dominator of v in $\mathcal{G}$.

Definition 5. *A* macrogate *is a cluster $\mathcal{M}$ such that every node v in $\mathcal{M}$ represents the same associative Boolean function. An* OR-macrogate *(*AND-macrogate*) is a macrogate composed of logical disjunction (conjunction) nodes.*

The definitions provided for cones are naturally extended to clusters and macrogates. An example of clusters and macrogates appears in Figure 1, where one cluster includes one OR- and two AND-macrogates.

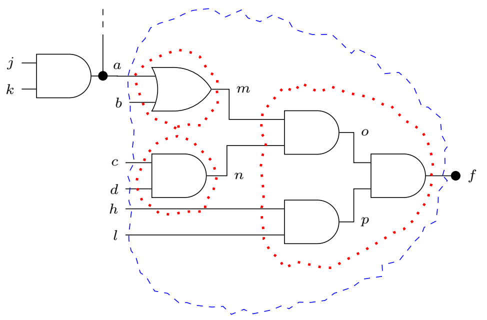


Fig. 1: A subcircuit partitioned in clusters (enclosed by a blue dashed line) and macrogates (enclosed by a dotted red line).

Definition 6. *Given a cone $\mathcal{C}$ rooted in r and a variable $a \in cut(\mathcal{C})$, variable a is not observable on f_r iff $f_r(X, \bot) \equiv f_r(X, \top)$, with $X = cut(\mathcal{C}) \setminus a$.*

A *literal* is either a Boolean variable or its negation. A *clause* is a disjunction of literals. A Boolean formula F is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses. Given a Boolean formula F, we denote with $supp(F)$ the set of Boolean variables over which F is defined.

A Boolean formula F is in *Negation Normal Form* (*NNF*) if the negation operator ($\neg$) is only applied to its variables, and the only other operators allowed are conjunction ($\wedge$) and disjunction ($\vee$). Any formula can be transformed to NNF in linear time through direct application of De Morgan's laws and the elimination of double negations. In the worst case, the size of the circuit implementing a formula F might double when F is transformed into NNF.

B. Craig Interpolants

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

We use ITP to denote the interpolation operation. An interpolant $I = \text{ITP}(A, B)$ can be derived, as an AND-OR circuit, from the refutation proof of $A \wedge B$. Most modern SAT solvers

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

are capable of producing resolution proofs. A resolution proof provides evidence of unsatisfiability for a CNF formula F as a series of applications of the *binary resolution* inference rule. Given two clauses $C_1 = (l \vee l_1 \vee ... \vee l_n)$ and $C_2 = (\neg l \vee l'_1 \vee ... \vee l'_m)$, a resolvent C is computed using a resolution operator, defined as: $C = Res(C_1, C_2) = (l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_m)$. Starting from the clauses of F, such a rule is applied until the empty clause is derived.

Craig interpolants are generated from resolution proofs as described in [2]. The resulting ITP circuit is isomorphic to the proof: where original clauses are translated as either OR gates or constants and resolutions steps are translated as either AND or OR gates. Interpolants in the range between A and $\neg B$ depend on SAT solver decisions, thus their resulting strength/weakness is not under user control. This motivated research on ex-post interpolant strengthening/weakening.

C. Combinational Circuit Compaction

This subsection briefly overviews, without any claim of completeness and generality, some combinational synthesis techniques our circuit compaction approach is based upon.

Redundancies affecting non canonical combinational circuits are removed by structural hashing, cut-based [16], BDD-based [17] and SAT-based [18] sweeping. The above methods basically rely on finding and merging classes of functionally equivalent circuit nodes. Other reduction efforts exploit various decomposition, rewriting and balancing strategies. In [19] a mix of locally canonical transformations and DAG-aware rewritings on technologically independent circuits have been first proposed. [14] introduces a technique for preprocessing combinational logic before technology mapping. We follow [14] in its use of And-Inverter Graphs (AIGs), composed of two-input ANDs and inverters[2]. Scalability is achieved by making all operations local, and moving to a global scope by iterated application of local reductions. The result is that the cumulative effect of several rewriting steps is often superior to traditional synthesis in terms of quality.

Redundancy removal under Observability Don't Cares (ODCs) is a powerful variant of redundancy removal, where node equivalences are established taking into account their observability at circuit outputs. All ODC-based approaches rely on a computation of don't care conditions for nodes involved in redundancy checks. As exact computation is prohibitively expensive, approximate techniques have been proposed. BDD-based Compatible Observability Don't Care (CODC) sets were computed in SIS [21]. Approximated ODCs (by "windowing") were introduced in [22], where scalability was achieved by restricting the sub-circuit environment to a locality. SAT-based quantifier elimination [23], augmented with random sampling, is a further attempt to exploit the power of SAT solvers.

D. Gate-Level Abstraction

Abstraction techniques are a well known area of research in Model Checking. Our paper is related to a form of localization abstraction [24] called Gate-Level Abstraction [25]. Abstraction by localization is based on removing circuit components (i.e. cutting wires) not necessary for a proof. Detection of unnecessary parts has been proposed following two main schemes:

- Counterexample-Based Abstraction-refinement (CBA) [26], where an initially weak abstraction is iteratively refined (strengthened) based on spurious counterexample analysis;
- Proof Based Abstraction (PBA), exploiting the ability of modern SAT solvers to generate proofs of unsatisfiability, is a more recently followed variant, investigated in standalone mode or combined with CBA, as in [27].

In most model checkers, localization is done at register boundaries. Gate-Level Abstraction [25] is a particular abstraction scheme (compatible in principle with both CBA and PBA strategies), where localization is done at gate nodes.

III. Interpolants Compaction by Ad-Hoc Logic Synthesis

In this section we present a set of procedures to reduce the size of Boolean circuits, based on local simplification techniques arising from logic synthesis. Although applicable to any Boolean circuit, our approach specifically targets the main sources of redundancy of interpolant circuits: gates that can be replaced by a constant value, or sub-circuits that can be merged being functionally equivalent (though topologically distinct). We consider an interpolant as a single-output circuit $\mathcal{G}$. Starting, from an AIG representation of the circuit, we:

- Identify AND and OR gates;
- Partition $\mathcal{G}$ into a set of maximal clusters;
- Group trees of AND (resp. OR) gates in macrogates.

Our target is to address gate redundancies by fast operations, where circuit transformations are performed within clusters. The reason for limiting our scope to clusters is related to the fact that fanout stems propagate *shared* subformulas through different paths within the circuit graph. Simplifications affecting multiple fanout paths are both complex and of limited impact.

The circuit $\mathcal{G}$ is partitioned into a maximal set of clusters, each of which is in turn partitioned into a set of macrogates. This is done by means of a depth-first visit of $\mathcal{G}$ starting from its root node r. Each node v is associated with two pieces of information: its cluster dominator, $domC(v)$, and its macrogate dominator, $domG(v)$. As long as the visited nodes have unit fanout, cluster dominator information in propagated. As long as the visited nodes have unit fanout and are of the same type, macrogate dominator information in propagated. Performing such an operation requires $O(|E|)$ time.

We thus propose a procedure based on two kinds of local simplifications:

- Redundancy removal (gates equivalent to a constant) based on ODC-like implications within clusters.
- Enforcement of sub-formula sharing (equivalent gates merging) through macrogate refactoring.

[2]Another motivation for our choice is the fact that AIGER is the netlist interchange format chosen for Hardware Model Checking Competitions [20].

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

A. ODC Implications Removal

The first simplification technique we propose aims at finding local ODC implications that can be exploited to replace a gate with a constant. Such a technique relies on the following two identities:

$$\begin{aligned} f(X,a) &= a \wedge g(X,a) \equiv a \wedge g(X,\top) \\ f(X,a) &= a \vee g(X,a) \equiv a \vee g(X,\bot) \end{aligned}$$

Let us consider a Boolean function $f(X,a)$ expressed as the conjunction (resp. disjunction) of a variable a and a function g of a. Then a can be replaced by the $\top$ (resp. $\bot$) constant in g. Note that the instance of variable a in the support of g is not observable on f. From a circuit graph perspective, given $\mathcal{G}$ implementing f, a is an input variable and g is a subcircuit of $\mathcal{G}$ with a in its fanin. There are at least two re-convergent paths from node a to the output node of f.

We call such cases *ODC implications* for f, as the implications $f \rightarrow a$ and $\neg a \rightarrow \neg f$ (resp. $\neg f \rightarrow \neg a$ and $a \rightarrow f$) dually hold in each of the two respective cases.

We exploit the notion of ODC implications to perform local simplification of functions in the Boolean circuit. This is done by detecting cones $\mathcal{C}$ in the circuit whose function can be expressed as either $a \wedge g(X,a)$ or $a \vee g(X,a)$. In these cases, $\mathcal{C}$ can be simplified by disconnecting the redundant edge from a to its entry point in $\mathcal{C}$ and injecting a constant. Detection of ODC implications is restricted at macrogate and/or cluster boundaries in order to avoid problems arising from shared elements.

We consider both *direct ODC implications* and *transitive ODC implications*. Direct ODC implications arise when the input of a function f is directly implied by f. Figure 2 exemplifies a direct ODC implication. Input b is a direct ODC implication for f_t since $f_t(a,b,c) = b \wedge g(a,b,c)$ with $g(a,b,c) = c \wedge (a \vee b)$, and therefore $f_t \rightarrow b$. Transitive ODC implications occur when the input of a function f is transitively implied by f through another of its inputs. Figure 3 provides an example of transitive ODC implication. Input b is a transitive ODC implication for f_t, in fact, d is a direct ODC implication for f_t and b is a direct ODC implication of f_d, therefore, $f_t \rightarrow d \rightarrow b$.

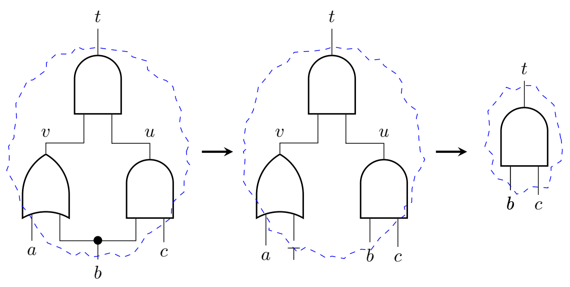


Fig. 2: Example of direct ODC implication.

The DirectOdcSimplify procedure (Algorithm 1) tries to identify cluster inputs that are made redundant by direct ODC implications. Given a cluster $\mathcal{C}$ rooted in r and one of its inputs v, the algorithm tries to find a node d in $\mathcal{C}$ such that

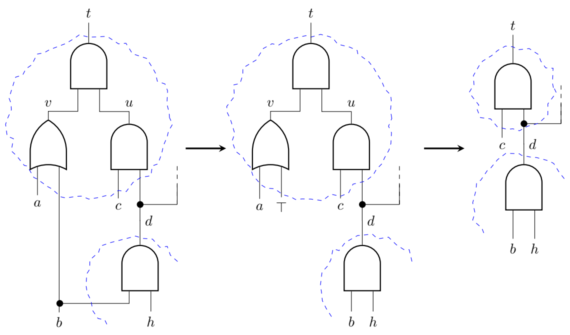


Fig. 3: Example of transitive ODC implication.

v is a direct ODC implication for f_d. Considering the cluster as a tree of macrogates, this corresponds to finding a common successor d for two of the entry points of v in $\mathcal{C}$, called u and t, so that d is a direct successor of either u or t. Since we are considering a tree of macrogates, d being a direct successor of t means that t is connected to d through either a chain of only AND or OR gates. For each cluster $\mathcal{C}_i$, the algorithm scans each of its cut nodes. For each $v \in cut(\mathcal{C}_i)$, every pair u, t of distinct entry points of v in $\mathcal{C}_i$ is considered. In order to find a common successor for u and t, first each macrogate dominator of u is marked by the procedure MarkDominators. Then, the algorithm checks if the macrogate dominator of t is marked. If that is the case, being $d = domG(t)$, we have either $f_d(X,v) = v \wedge g(X,v)$ or $f_d(X,v) = v \vee g(X,v)$ for some g. Therefore, v in g is not observable on f_d and the circuit can be simplified by calling function Simplify. Such a function takes a couple of nodes and a gate type as arguments, removes the edge (v,u) from the circuit and injects an appropriate constant value in the newly created free input. The injected constant is $\top$ if the gate type passed as argument is AND, $\bot$ if is OR. After injecting the constant, the circuit is simplified accordingly. Otherwise, if $domG(t)$ is not marked, the algorithm proceeds with the next pair of entry points. Time complexity of DirectOdcSimplify is $O(|V| \max_{\mathcal{C}_i \in \mathcal{G}}\{|cut(\mathcal{C}_i)|\})$.

```
DirectOdcSimplify($\mathcal{G}$)
1: for all clusters $C_i \in \mathcal{G}$ do
2:   for all nodes $v$ in $cut(C_i)$ do
3:     for all pair $(u,t)$ in $fanout(v) \cap C_i$ with $u \neq t$ do
4:       MarkDominators($u$)
5:       if $domG(t)$ is marked then
6:         Simplify($v, u, type(t)$)
7:       UnmarkDominators($u$)
```

Algorithm 1. DirectOdcSimplify*($\mathcal{G}$)*

The TransitiveOdcSimplify procedure (Algorithm 2) tries to identify cluster inputs that are made redundant by transitive ODC implications. Two lists are maintained for each cluster: a direct implication list and a transitive implication list. Given a cluster $\mathcal{C}$ rooted in r, its direct implication list, denoted as $Impl(\mathcal{C})$, contains all cluster inputs v for which at least one of the entry points of v in $\mathcal{C}$ has r as macrogate dominator. Therefore, for each $v \in Impl(\mathcal{C})$ either $f_r \rightarrow v$, if $type(r)$ is

AND, or $\neg f_r \rightarrow \neg v$, if $type(r)$ is OR. Direct implication lists are provided as an argument to TRANSITIVEODCSIMPLIFY. Transitive implication lists, denoted as $Trans(\mathcal{C})$, are used to collect those nodes v for which there exists a sequence of clusters $\mathcal{C}_0, \ldots, \mathcal{C}_n$ such that the following conditions hold:

- $\mathcal{C}_0 = \mathcal{C}$;
- $\mathcal{C}_{i+1} \in Impl(\mathcal{C}_i)$ for each $0 \leq i < n$;
- $type(\mathcal{C}_{i+1}) = type(\mathcal{C}_i)$ for each $0 \leq i < n$;
- $v \notin Impl(\mathcal{C}_i)$ for $0 \leq i < n$;
- $v \in Impl(\mathcal{C}_n)$.

Transitive implication lists are computed while TRANSITIVEODCSIMPLIFY runs and used to detect transitive ODC implications w.r.t. the root of each cluster.

In TRANSITIVEODCSIMPLIFY clusters are scanned in topological order. For each cluster $\mathcal{C}_i$, its transitive implication list is first computed. This is done by conjoining the current $Trans(\mathcal{C}_i)$ with every node that is either in the transitive or direct implication list of the clusters that are in $Impl(\mathcal{C}_i)$ and are of the same type of $\mathcal{C}_i$. Once the transitive implication list for $\mathcal{C}_i$ has been computed, the procedure scans each node $v \in cut(\mathcal{C}_i)$ that is in $Trans(\mathcal{C}_i)$. These nodes are inputs of $\mathcal{C}_i$ for which a transitive ODC implication exists (through some of the other inputs of $\mathcal{C}_i$). Therefore, each entry point u of these nodes can be simplified by calling SIMPLIFY. Time complexity of Algorithm 2 depends on the size of the transitive lists: $O(|V| \max_{\mathcal{C}_i \in \mathcal{G}}\{|Trans(\mathcal{C}_i)|\})$. Although the sizes of such lists, in the worst case, could be quadratic in the number of nodes, experimentally it is possible to notice that in our context of application the size of these lists stays within $O(|V|)$.

```
TRANSITIVEODCSIMPLIFY($\mathcal{G}$, $Impl$)
 1: for all clusters $C_i \in \mathcal{G}$ in topological order do
 2:   $Trans(C_i) \leftarrow \emptyset$
 3:   for all clusters $C_k$ in $Impl(C_i)$ do
 4:     for all $v$ in $Trans(C_k) \cup Impl(C_k)$ do
 5:       if $type(C_k) = type(C_i)$ then
 6:         $Trans(C_i) \leftarrow Trans(C_i) \cup \{v\}$
 7:   for all nodes $v$ in $cut(C_i)$ do
 8:     if $v$ in $Trans(C_i)$ then
 9:       for all node $u$ in $fanout(v) \cap C_i$ do
10:         SIMPLIFY($v, u, type(C_i)$)
```

Algorithm 2. TRANSITIVEODCSIMPLIFY*($\mathcal{G}$, Impl)*

B. Macrogate Refactoring

The second simplification approach we propose tries to refactor portions of the circuit implementing the same type of Boolean function in order to explicit sub-functions implemented by nodes already present in the circuit. If successful, sharing can be enforced to reduce the overall size of the circuit. This technique is applied to macrogates in order to guarantee that each node removed by means of refactorization has unit fanout and thus the size of the circuit actually decreases. As an example, consider an AND-macrogate in Figure 4, implementing the function $f_t(a, b, c, d) = (a \wedge b) \wedge (c \wedge d)$. The idea is to identify a couple of inputs (i, j), such that the node realizing $i \wedge j$ does not appear in the macrogate but it exists in a different point of the circuit. Suppose a node m implementing $f_m = c \wedge b$ exists, the macrogate function f_t can be refactored as $f_t(a, b, c, d) = m \wedge (a \wedge d)$ so that the gate m can be shared. The final result of such a step of refactoring is a reparenthesization of the original macrogate function, for which the number of nodes decreases by one, one being now shared. A similar reasoning applies to OR-macrogates as well.

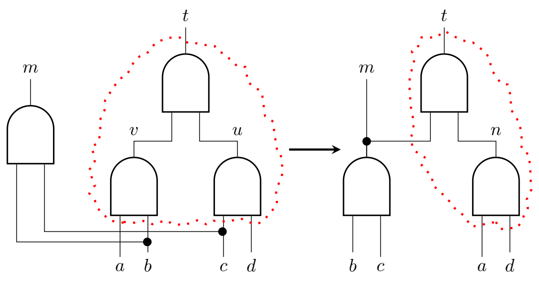


Fig. 4: Example of macrogate refactoring.

Note that refactoring a macrogate may change the current circuit partitioning as a previously non-shared node becomes shared.

The MACROGATEREFACTOR procedure (Algorithm 3) tries to refactor macrogates of the circuit in order to enforce better sharing. For each macrogate $\mathcal{M}_i$, first its cut nodes are marked. Then, for each input node of $\mathcal{M}_i$, the procedure scans all the nodes in its fanout list that do not appear in $\mathcal{M}_i$ but are of the same type. Those nodes u are gates of the same type of $\mathcal{M}_i$ that share an input with $\mathcal{M}_i$. For each of those nodes, the algorithm checks whether its other input node is shared with $\mathcal{M}_i$, by testing if such a node is marked. In such a case, $\mathcal{M}_i$ can be refactored to enforce sharing with u. Function REFACTOR handles macrogate refactoring. It also updates any other macrogate that could have been affected by the refactoring. Time complexity of MACROGATEREFACTOR is $O(|V| \max_{v \in V}\{|fanout(v)|\})$.

```
MACROGATEREFACTOR($\mathcal{G}$)
 1: for all macrogate $\mathcal{M}_i \in \mathcal{G}$ do
 2:   Mark nodes in $cut(\mathcal{M}_i)$
 3:   for all $v$ in $cut(\mathcal{M}_i)$ do
 4:     for all $u$ in $fanout(v)$ do
 5:       if $domG(v) \neq domG(u)$ and $type(v) = type(u)$
          then
 6:         if $left(u) \neq v$ and $left(u)$ is marked then
 7:           REFACTOR($\mathcal{M}_i, u, left(u)$)
 8:         else if $right(u) \neq v$ and $right(u)$ is marked then
 9:           REFACTOR($\mathcal{M}_i, u, right(u)$)
10:   Unmark nodes in $cut(\mathcal{M}_i)$
```

Algorithm 3. MACROGATEREFACTOR*($\mathcal{G}$)*

IV. SAT-BASED WEAKENING

Previously described reductions follow the trend of fast circuit-based optimizations. We now present a novel approach combining the ideas of interpolant compaction and weakening.

Given an interpolant $I = \text{ITP}(A, B)$, a weaker (resp. stronger) interpolant I_w (resp. I_s) is another interpolant, such that $I \to I_w$ ($I_s \to I$). Interpolant weakness and strength are dual concepts. Considering an interpolant I for A, B, its complement $\neg I$ is an interpolant for B, A. A weaker interpolant for A, B corresponds to a stronger interpolant for B, A. As mentioned in section I, interpolant strength and/or weakness can be related to the quality of the interpolant itself [13]. State-of-the-art approaches to interpolant strengthening/weakening are based on SAT proof transformations [28]. Interpolant re-computation is another straightforward and practical way to compact an interpolant and change its strength. Given $I = \text{ITP}(A, B)$, we can generate a weaker interpolant $I_w = \text{ITP}(I, B)$ or a stronger one $I_s = \text{ITP}(A, \neg I)$. Empirically, we spend extra time, performing an additional interpolant computation, in order to obtain a better interpolant, where *better* could mean weaker/stronger and possibly more compact. Unfortunately, compaction is not guaranteed, as the size of the final interpolant depends on a SAT solver run. Experimentally, we have observed both increases and decreases in terms of interpolant size.

Our strategy is to spend extra time by re-running a SAT solver query (either $A \wedge \neg I$ or $I \wedge B$), while computing the new interpolant in a different way, that guarantees compaction. In the following, we outline the main steps of our weakening approach (strengthening is dual):

- I is encoded as NNF, producing I_{NNF}
- A Gate-Level Abstraction of I_{NNF} is performed, using a PBA approach:
  - SAT query $I_{NNF} \wedge B$, guaranteed UNSAT, is solved and used to generate the UNSAT core $C(I_{NNF} \wedge B)$, the full proof is not necessary
  - Using the UNSAT core, a proof-based abstraction of I_{NNF} is computed: $I^{pba} = PBA(I_{NNF}, C)$
- As a result of PBA, fresh new variables Δ at all cut (abstraction) points are introduced. So, $supp(I^{pba}) = \Gamma \cup \Delta$, with $\Gamma = supp(A) \cap supp(B)$. The presence of these extra variables prevents I^{pba} from being a correct interpolant. Efficient existential quantification of Δ variables can be performed exploiting NNF encoding. In particular, $\exists \Delta\, I^{pba}$ is performed by replacing all variables in Δ with a $\top$ constant: $I_{w,NNF} = I^{pba}|_{\Delta = \{\top, \top, \ldots \top\}}$.
- The compacted interpolant $I_{w,NNF}$ is converted back to the (non NNF) AIG encoding.

Encoding a circuit as NNF implies a certain cost in terms of size. However, we experimentally observed (see section V) that this cost is negligible for interpolants, since they originate as pure AND-OR circuits with negations limited at input boundaries. Conversely, we have the advantage of *quantification by substitution*. Given a Boolean function $f(X, \Delta)$ in NNF form, with Δ appearing only in non-negated form, Δ can be existentially (resp. universally) quantified by substitution:

$$\begin{aligned} \exists \delta\, f(X, \Delta) &= f(X, \top) \\ \forall \delta\, f(X, \Delta) &= f(X, \bot) \end{aligned}$$

The top-level procedure is described in Algorithm 4. Given a node v, the function $CNF(v)$ is used to retrieve the CNF representation of f_v.

```
ItpWeaken($I, B$)
1: $I_{NNF}$ ←Aig2Nnf($I$)
2: $C$ ← SatWithUnsatCore($I_{NNF} \wedge B$)
3: for all nodes $v$ in $I_{NNF}$ do
4:     if $CNF(v) \notin C$ then
5:         Replace($v, \top$)
6: $I_{w,NNF}$ ← RecomputeCircuit($I_{NNF}$)
7: $I_w$ ←Nnf2Aig($I_{w,NNF}$)
Return $I_w$
```

Algorithm 4. ITPWEAKEN*(I, B)*

The algorithm shows weakening of I w.r.t. B, being strengthening with A dual. Furthermore, we use PBA-based abstraction, whereas a CBA-based approach is possible as well. The proposed code unifies GLA (Gate-Level Abstraction) with existential quantification, as, given the UNSAT core (C), circuit nodes with a corresponding CNF variable not in C are immediately abstracted and replaced with the $\top$ constant.

V. Experimental Results

We implemented a prototype version of our interpolant compaction procedures on top of the PdTRAV tool [29], a state-of-the-art verification framework. Experimental data in this section provide an evaluation of the techniques proposed. Experiments were run on an Intel Core $i7-3770$, with 8 CPUs running at 3.40 GHz, 16 GBytes of main memory DDR III 1333, and hosting a Ubuntu 12.04 LTS Linux distribution. We set memory limits to 900 seconds (3600 for the weakening experiments) and 8 GB, respectively.

We performed an extensive experimentation on a selected subset of interpolants used in [11]. These interpolants are extracted from publicly available benchmarks from the past HWMCC [20] suites and are represented as AIGs. We took into account also interpolants derived from software verification problems [12]. The former set is composed of 2472 instances, ranging from 1.1×10^5 to 8.5×10^6 nodes. The latter set is composed of 1872 instances, ranging from 4×10^2 to 6×10^4 nodes[3].

We gathered initial data from the first set of interpolants in order to purge *easy* instances. We considered easy those instances with less than 1.5×10^4 nodes and for which our logic synthesis procedure was able to reach a fix-point within 150 seconds. The purged set of benchmarks, comprising 87 instances ranging from 4×10^5 to 8.5×10^6 nodes, was used to conduct a more in-depth experimentation.

Figures 5 and 6 show the results obtained for compaction with logic synthesis (section III) and GLA-based weakening (section IV), respectively. Compaction techniques are applied incrementally, i.e., we always apply simplifications described in [11][4], followed by the techniques described in this paper.

[3]The interpolant circuits are available at http://fmgroup.polito.it/index.php/download.

[4]With the exception of the most time-consuming, and less scalable, ITE-based decomposition.

A. Compaction by Logic Synthesis

In our experiments, we evaluated techniques of section III by applying them as follows. First the circuit is partitioned into clusters and macrogates. A trivial simplification is performed by removing each duplicated input from macrogates. Then DirectOdcSimplify, MacrogateRefactor and TransitiveOdcSimplify are iterated in this order, recomputing the circuit partition between each call, until two consecutive iterations reduce the circuit size for less than 1%.

For each benchmark, we first apply the AIG balancing procedure of ABC prior to applying any of the aforementioned techniques. We consider the size of interpolants after balancing as baseline for the following experimentation. In order to test individual contributions of the proposed techniques we performed an initial run with all simplifications enabled, we call this run ItpSimplify, followed by a set of runs in which we selectively disabled them one at a time: NoDirectOdcSimplify, NoMacrogateRefactor and NoTransitiveOdcSimplify respectively. As a last test, we disabled our techniques altogether and performed ITP compaction using only standard logic synthesis (rewriting/refactoring, using the state-of-the-art ABC [30] tool).

Figures 5a and 5b illustrate the cumulative size and execution time, respectively, over all the benchmarks. In both cases, the closer a line is to the x axis, the better the result.

The two figures easily illustrate the compromise between execution time and potential size reduction obtained. On the one hand the purely ABC-based simplification is the best performing one, but it requires a significant amount of time. Different compaction rates are achievable with less computational effort adopting less aggressive approaches. We excluded timeouts from the visual representation.

As mentioned in section II-D, the size of implication lists could be a limit to the scalability of the proposed methods, as well. Although such lists could theoretically grow quadratically in the number of nodes, experimentally we noticed at worst a multiplicative factor of 20.

B. Compaction by Weakening

In order to characterize the rate of ITP compaction achievable through SAT-based weakening/strengthening, we raised the time limits to 3600 seconds. Such an approach is conceived to be used when ITP size reduction is crucial, and/or weakening/strengthening are actually the target, which motivates a bigger effort in terms of total execution time.

A preliminary step for all the proposed techniques requires to convert a given interpolant into NNF form. This step could lead to an increase in circuit size up to a factor of 2, in the general case. Given the nature and structure of interpolants themselves the increase in size is almost negligible. Taking into account all the experiments conducted, the biggest experienced increase was below 0.5%, confirming the intuitive arguments in section IV.

We conducted a set of experiments taking into account the same subset of 87 interpolants, iterating sequences of weakening (labelled B) and/or strengthening (labelled A) steps in different patterns. We propose an experimental evaluation for six different sequences: A, B, AB, BA, $ABAB$ and $BABA$. We run our logic synthesis compaction procedure before any weakening/strengthening attempt (baseline). Figures 6a and 6b illustrate the cumulative size and execution time, respectively, over all the benchmarks. It is fairly noticeable the impact on the choice of the first kind of chosen compaction: starting with B tends to produce better results, related to the fact that most of the interpolants proposed have more room for weakening than strengthening.

Overall, it is fairly clear that SAT-based abstraction leads to dramatic compaction, though paid in terms of time.

VI. Conclusions

We addressed the problem of optimizing interpolants size for SAT-based UMC. Our main contribution is to provide an integrated approach, that targets interpolation compaction, providing different tradeoffs between time and memory according the proper context of application. We work both at the level of logic synthesis and at SAT level, proposing different techniques aimed at interpolant size reduction. Overall, our main target is to increase the scalability of existing UMC approaches, taking into account resource limitations and compromising between optimal results and applicability of the proposed methods. We experimentally observed that the proposed optimizations can be beneficial to existing reachability schemes, based on interpolation.

VII. Acknowledgements

We thank prof. Natasha Sharygina, dr. Antti E. J. Hyvärinen and Leonardo Alt from Università della Svizzera Italiana (USI), Switzerland, for the benchmarks generated from software verification problems.

References

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

[2] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV*, ser. LNCS, vol. 2725, Boulder, USA, 2003, pp. 1–13.

[3] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *Proc. of CAV*, London, UK, UK, 1997, pp. 72–83.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *SIGPLAN Not.*, vol. 39, pp. 232–244, Jan. 2004.

[5] J. Marques-Silva, "Improvements to the implementation of Interpolant–Based Model Checking," in *Proc. of CHARME*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 367–370.

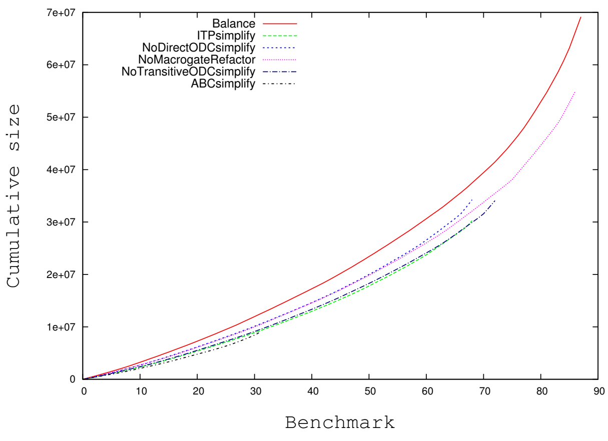
[6] V. D'Silva, M. Purandare, and D. Kroening, "Approximation Refinement for Interpolation-Based Model Checking," in *Verification, Model Checking and Abstract Interpretation*, vol. 4905, 2008, pp. 68–82.

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

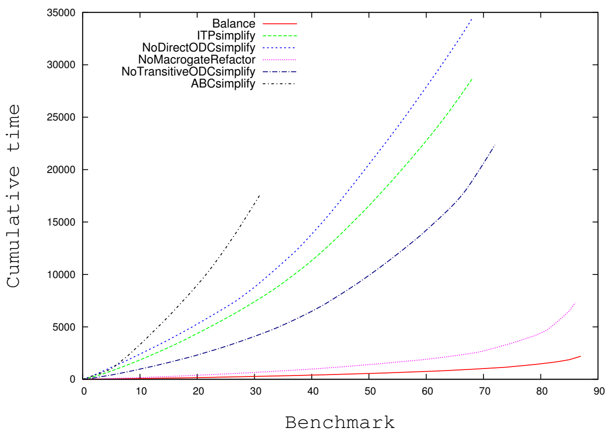
[8] G. Cabodi, P. Camurati, and M. Murciano, "Automated Abstraction by Incremental Refinement in Interpolant-based Model Checking," in *Proc. of ICCAD*. San Jose, California: ACM Press, Nov. 2008, pp. 129–136.

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

[10] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proc. of DATE*, Washington, DC, USA, 2003.

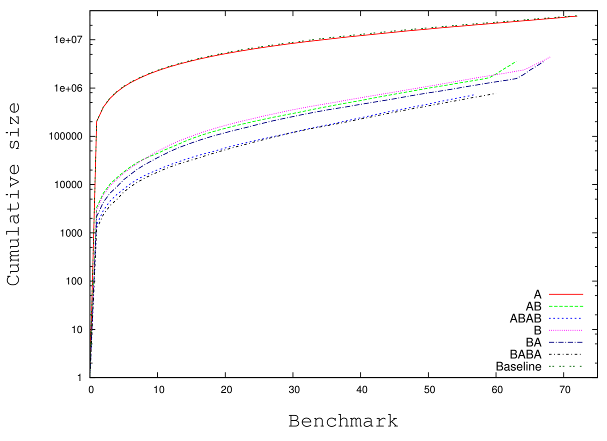


(a)

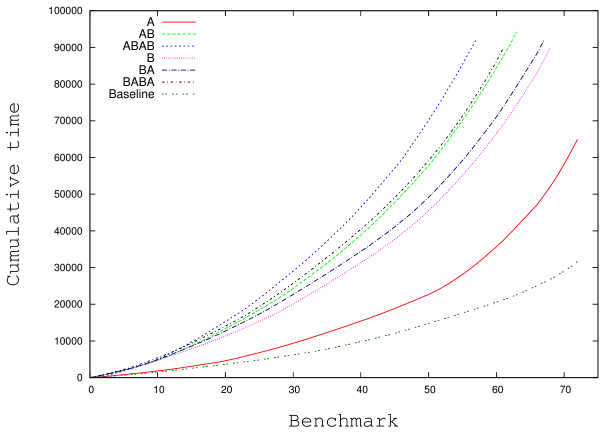


(b)

Fig. 5: Cumulative results of ITP compaction based on logic synthesis, in terms of size and execution time.



(a)



(b)

Fig. 6: Cumulative results of ITP compaction based on SAT, in terms of size and execution time. Sizes are plotted on a log scale given the higher ratio of compaction achieved.

[11] G. Cabodi, C. Loiacono, and D. Vendraminetto, "Optimization techniques for craig interpolant compaction in unbounded model checking," *Formal Methods in System Design*, vol. 46, no. 2, pp. 135–162, 2015.

[12] L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "A proof-sensitive approach for small propositional interpolants," in *Verified Software: Theories, Tools, and Experiments - Revised Selected Papers*, San Francisco, CA, USA, Jul. 2015, pp. 1–18.

[13] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *Proc. of VMCAI*, vol. 5944, January 2010, pp. 129–145.

[14] R. K. Brayton and S. Chatterjee and A. Mishchenko, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. of DAC*, 2006, pp. 532–536.

[15] D. B. neres, J. Cortadella, and M. Kishinevsky, "Dominator-based partitioning for delay optimization," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '06. New York, NY, USA: ACM, 2006, pp. 67–72.

[16] N. Eén, "Cut Sweeping," Cadence Research Labs, Berkeley, USA, Tech. Rep., May 2007.

[17] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. of DAC*, Anaheim, California, Jun. 1997, pp. 263–268.

[18] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Property Checking," in *Proc. of ICCAD*, San Jose, California, Nov. 2004, pp. 50–57.

[19] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression For Formal Verification," in *Proc. of ICCAD*, San Jose, California, Nov. 2004.

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

[21] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization." in *Proc. of ICCAD*, 1991, pp. 514–517.

[22] A. Mishchenko and R. K. Brayton, "Sat-based complete don't-care computation for network optimization," *CoRR*, vol. abs/0710.4695, 2007.

[23] K. L. McMillan, "Applying sat methods in unbounded symbolic model checking." in *Proc. of CAV*, vol. 2404, 2002, pp. 250–264.

[24] R. P. Kurshan, "Computer Aided Verification of Coordinating Processes," in *Princeton University Press*, Princeton, NJ, 1994.

[25] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "Gla: Gate-level abstraction revisited," in *Proc. of DATE*, ser. DATE '13, San Jose, CA, USA, 2013, pp. 1399–1404.

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

[27] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental sat formulation of proof- and counterexample-based abstraction," in *Proc. of FMCAD*, Oct 2010, pp. 181–188.

[28] K. L. McMillan and R. Jhala, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV*, ser. LNCS, vol. 3725, Edinburgh, Scotland, UK, 2005, pp. 39–51.

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

[30] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

Extracting Behaviour from an Executable Instruction Set Model

Brian Campbell and Ian Stark
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh, UK
Email: Brian.Campbell@ed.ac.uk, Ian.Stark@ed.ac.uk

Abstract—**Presenting large formal instruction set models as executable functions makes them accessible to engineers and useful for less formal purposes such as simulation. However, it is more difficult to extract information about the behaviour of individual instructions for reasoning. We present a method which combines symbolic evaluation and symbolic execution techniques to provide a rule-based view of instruction behaviour, with particular application to automatic test generation for large MIPS-like models.**

I. Introduction

It is a common practice to construct large formal models of instruction set architectures in the form of executable functions. Recent examples include an x86 model with system calls in ACL2 by Goel et al. [1] and the models constructed in Fox's L3 domain specific language [2], which can be translated into several systems. We have been using the latter in this work in the HOL4 theorem proving system.

There are several appealing aspects to such models. They can be used as simulators, existing tests suites can be run through them for validation, and they are in a form familiar to engineers. Indeed, L3 models are usually written in a form close to the pseudocode found in architecture reference manuals. However, they do not expose the structure normally found in a rule-based operational semantics, such as stating the conditions required for an instruction to have well-defined behaviour as explicit hypotheses.

Our goal is to extract this type of structure from the executable model for individual instructions, and use it to extend our previous automated test generation work [3] to new models. The core of that work used an existing verification support library [4] to provide a rule-based view of instructions and so obtain the constraints required to execute a randomly chosen sequence successfully, then express them in terms of the initial state and use an SMT solver to find such a state.

These *step* libraries have been constructed for several architectures and we have successfully used them for test generation with a model for the ARM Cortex-M0 microcontroller and a simple MIPS model. However, each library requires a considerable amount of effort, typically over 1000 lines of code per target, and ongoing maintenance when the model is altered.

We wanted to extend our MIPS testing to a much more complete model of the experimental CHERI processor [5]. CHERI features a hybrid capability system which can improve security by limiting access to resources while maintaining compatibility with existing code. The model includes a large number of new instructions for the additional security features, more complex representations of state and memory, and full memory management. It is over twice as large as the plain MIPS model and no *step* library has been written for it[1]. Moreover, we also wished to have the option of testing processor exception handling, which these libraries do not currently support.

We have constructed a new library to extract rules for individual instructions similar to those from the *step* libraries, but with much greater automation. To achieve this, and to deal with such a large model, we combine standard symbolic evaluation with a form of symbolic execution. The symbolic evaluation provides general computation using rewriting from the normal HOL4 libraries. The symbolic execution explores the different possible paths of execution, recording in the hypotheses the *path condition* which describes when each can be reached, and it treats the large state record carefully for reasonable performance.

Our contributions are to present our new library for extracting instruction behaviour from these executable models while maintaining a close, formal, connection to the model; to discuss its application to automatic test case generation; to demonstrate that a theorem proving system such as HOL is a practical setting for symbolic execution; and to show that these perform well enough for practical use on a realistic processor model, producing high test coverage. The close connection between the formal model and the generated tests is particularly important for CHERI, where some colleagues are now proving security properties about the model. Our code is available online[2].

In Section II we outline the form of the models, the desired form for expressing the extracted behaviour, and the testing process. Section III presents our combination of symbolic evaluation and symbolic execution, followed by a discussion of how sound and complete the process is in Section IV. Then Section V describes the application of the process to

[1]A basic *step* library for a simplified version of the model was produced after this paper was written, but it only covers a fraction of the behaviour that we are interested in testing.

[2]https://bitbucket.org/bacam/m0-validation

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

```
dfn'ADDI (rs,rt,immediate) =
(λstate.
   (let s =
      if NotWordValue (FST (GPR rs state)) then
        SND
          (raise'exception
            (UNPREDICTABLE "ADDI: NotWordValue")
             state)
      else state
    in
    let v = (32 >< 0) (FST (GPR rs s))
              + sw2sw immediate
    in
      if word_bit 32 v ≠ word_bit 31 v
      then SignalException Ov s
      else write'GPR (sw2sw ((31 >< 0) v),rt) s))
```

Fig. 1. HOL4 version of 32-bit signed immediate add MIPS instruction

```
[s.CP0.Config.BE, ¬s.CP0.Status.RE,
 ¬s.exceptionSignalled,
 ¬if word_bit 31 (s.gpr 2w)
  then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
  else (63 >< 32) (s.gpr 2w) ≠ 0w,
 s.MEM s.PC = 36w, s.MEM (s.PC + 1w) = 65w,
 s.MEM (s.PC + 3w) = 3w, s.MEM (s.PC + 2w) = 0w,
 (1 >< 0) s.PC = 0w, s.exception = NoException,
 s.BranchDelay = NONE, s.BranchTo = NONE]
⊢ NextStateMIPS s =
   SOME (s with
   <|BranchDelay := NONE; BranchTo := NONE;
    CP0 := s.CP0 with Count := s.CP0.Count + 1w;
    PC := s.PC + 4w;
    exceptionSignalled := F;
    gpr := (1w =+ sw2sw ((31 >< 0) (s.gpr 2w) + 3w))
           s.gpr|>)
```

Fig. 2. *Step* theorem for MIPS 32-bit unsigned immediate addition

testing, and in particular with the MIPS and CHERI models. We discuss related work in Section VI and possible further work in Section VII.

II. Background

Fox's L3 domain specific language [2] provides a natural environment for writing instruction set architecture specifications in the form of a function to compute successive states. The language features support for working with data at the bit level, including pattern matching for decoding and bit-field records for registers, exceptions to indicate undefined behaviour, incrementally defined global processor state and an instruction abstract syntax datatype automatically derived from the instruction definition functions. The L3 tool translates the language into the logics of several proof tools and also the SML programming language for simulation. L3 models have been constructed for a number of architectures, including ARMv7-A, ARMv6-M, MIPS, CHERI and partial models of ARMv8-A and x86-64, and there are several external users of these models.

The translation to HOL4 must transform several of L3's language features into the logic. Figure 1 shows the HOL4 translation of the MIPS 32-bit signed immediate addition instruction, where the global state has been implemented by threading a state record throughout the definition, in the `state` and `s` variables. The undefined behaviour when the value of the `rs` register cannot be represented in 32 bits is modelled by the `raise'exception` function which records the failure in the state record. Failures are then detected at the end of the instruction execution rather than using an exception monad to simplify certain types of reasoning (see [4, §2] for discussion about this design). The `SignalException` function is a normal function to set up a processor exception, rather than an L3 exception to indicate undefined behaviour. In this case it is for an arithmetic overflow, which is detected by performing a 33-bit addition (where the `(32 >< 0)` operator extracts the bottom 33 bits of the 64-bit register). The use of the `sw2sw` function in the last line extends the result to 64 bits, ready to be written to the destination register.

Each model has a main function which computes the next state, combining the parts of the model that perform instruction fetch, decoding and execution (such as the function in Figure 1). The return value of the main function is an option: either the next state is returned, or nothing is returned if the model is undefined on the input state.

A. L3 Support Libraries

In our previous work we used pre-existing libraries by Fox that are part of a system for verifying machine code with respect to an L3 model [4]. The main interface for this system is a program logic in the style of separation logic, but we are interested in the intermediate *step* library which provides a more equational view. This library presents the behaviour of instructions as theorems providing the result of the main next state function as a series of state updates when a number of hypotheses hold, roughly:

$$\frac{\begin{array}{c}\text{flags set correctly in state } s\\ s \text{ contains instruction in memory}\\ s \text{ contains data to be read}\end{array}}{\mathsf{Next}\ s = \mathsf{SOME}\ (s \text{ with sequence of state updates})}$$

These theorems cut across the model, including decoding, execution and memory accesses. A MIPS example is shown in Figure 2, which is a 32-bit immediate addition, but unlike Figure 1 it is unsigned to avoid showing the complexity of a potential processor exception. Some of the less relevant details are shown in light grey. The hypotheses include the processor flags, the unfolded `NotWordValue` test to avoid undefined behaviour, and the presence of the instruction in memory. The conclusion updates the state, and in particular the `(1w =+ ...) s.gpr` updates register 1 with the result.

In general, there may be multiple theorems for an instruction when there are several branch choices. This MIPS instruction actually has three variants due to the use of branch delay slots in the MIPS architecture.

The *step* libraries are primarily based around symbolic evaluation under sets of assumptions for each case of a class of instructions. The main evaluation procedure, which includes setting up appropriate rewrites for the model's datatypes and definitions, and some specialised conversions, is in a common library used for all architectures. However, the per-model parts must still contain a substantial amount of information about the different classes of instruction present in the model, the different cases of each, and how to combine results about fetching, decoding and running instructions into a single result.

B. Test generation process

Our automatic test generation system [3] starts with random sequences of instructions, typically chosen to be long enough to exercise the pipeline. It finds an initial state which will run the sequence, avoiding undefined behaviour and (when undesired or unsupported) processor exceptions, by solving constraints derived from the *step* library information. It then compares the model's predicted final state with the result of an actual execution from the initial state. The process can be summarised as:

1) Generate instruction sequence
2) **Extract instruction behaviour from model**
3) Calculate sequence's constraints and behaviour in terms of the initial state
4) Solve constraints to build test with an SMT solver
5) Add test harness

The hypotheses from the extracted behaviour are the source of the constraints, while the conclusions are used to rewrite them in terms of the initial state of the whole sequence. We do not need there to be precisely one rule per branch choice to do this, so we can accept any reasonable partitioning of instruction behaviour into rules so long as the conclusion is in the form of a sequence of state updates.

We solve the constraints using an off-the-shelf SMT solver through an existing translation of a subset of HOL4 terms [6]. To adapt the testing to a new architecture the instruction generation, behaviour extraction and harness code must be adapted. This is routine for targets with a suitable *step* library, but for new targets such as CHERI we need a replacement for the behaviour extraction phase.

The *step* libraries have a few features that the testing does not require: there is support for partial instructions (where some operands are left as variables), it is compatible with the separation logic library which we do not use, and there is support for caching the resulting theorems.

III. Extracting Behaviour

One of our goals is to reduce the amount of user effort needed to extract instruction behaviour on a new target, so the new library must require much less model-specific information. Thus our replacement library discovers the different cases for each instruction rather than being provided with them, using the structure present in the model's definitions. This also removes the need to know about the different instruction classes. To increase automation we process the entire next state function at once, rather than building up a result from separate lemmas about the model's functions for fetching, decoding and executing instructions.

The threading of the global state record through the definitions by the L3 translation tool provides the structure used to discover the different cases of each instruction. To get results like Figure 2 which conclude with a sequence of state updates we need to break up any conditionals or pattern matches encountered in this threaded computation, producing a separate theorem for each path.

Symbolic execution techniques fit this view of the model; they follow the imperative structure of a program (in our case, the threaded state) and consider each path in the program independently, producing a separate result for each one. The parts of the computation which do not directly modify the state, such as the calculation of `v` in Figure 1, are left to symbolic evaluation, by which we mean rewriting the term under a set of assumptions producing a single equivalent term rather than a set of possible terms.

We avoid undesirable interactions between the evaluation and the execution by restricting the evaluation of **`let`** terms. There are beneficial interactions where conditionals and pattern matches can be simplified. For example, if we choose the always-zero register for `rs` in Figure 1 then evaluation will dispose of the `NotWordValue` test before execution even considers it. The implementation interleaves the symbolic evaluation and execution in a single recursive function.

A. Symbolic evaluation

For most of the symbolic evaluation of the model we use the `computeLib` call-by-value evaluation library included in HOL4 [7], appealing to rules for evaluating terms on bitvectors, arithmetic, pairs and other definitions from standard HOL4 theories. It also deals with operations on the model's datatypes and certain functions from the model, reusing some of the utility functions from Fox's libraries that can generate rewrites for any model. This is combined with some limited use of HOL4's simplifier for more complex rewrites, for example those which require higher-order matching.

In addition to avoiding evaluation of the **`let`** terms that the symbolic execution will explore, the evaluation must avoid expensive expansion of terms before there is sufficient information to reduce them properly. For example, the L3 translator uses a `FOR` combinator for loops, but if the number of iterations is not yet known (because some symbolic execution of the state is required first) then the standard evaluation rule will never terminate. We solved this by replacing the `FOR` rule by a restricted conversion that requires a concrete number for the loop bound. Similarly, most of the model definitions are only unfolded by the symbolic execution because they cannot be usefully evaluated before the state at the point of application is known.

Our symbolic evaluation also uses the current set of hypotheses as rewrite rules, including general user-specified assumptions about the particular target, such as the processor's endianness. The user can provide more specialised rewrite rules which introduce extra assumptions during evaluation. For example, we use this to restrict memory accesses to the small region of the address space that is used by our tests.

B. Symbolic execution

Traditional symbolic execution [8] requires a symbolic set of values, symbolic evaluation of expressions, a symbolic store, and a *path condition* to record the circumstances which lead to the part of the program currently being executed so that incompatible paths later in the execution can be avoided. The values and evaluation we get 'for free' by working in HOL with the evaluation procedure outlined above. Our initial treatment of the state was to substitute the entire symbolic value for the state record every time it was updated. However, we discovered that the performance was unacceptably poor for models with large state records such as CHERI. Instead, we maintain a rewrite for each field of the state which expresses its current value in terms of the initial state, and add these rewrites to the symbolic evaluation. In principle we could go further by using a separate rewrite for each entry of subrecords and maps in the state, for example, having one rewrite per register rather than the entire register map, but this has not been necessary in practice.

To maintain the path condition we add the appropriate assumption for each branch taken at a case split to the list of hypotheses. The symbolic evaluation will then use these assumptions to automatically eliminate incompatible branches later in the execution, and they may also be used for other simplification. For example, if a conditional takes one branch when a variable is zero, then in the execution which takes that branch the variable will be rewritten to zero throughout.

The symbolic execution procedure is summarised in Figure 3, where judgements of the form

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

mean that under the set of hypotheses H and the per-field state rewrites S, the execution of term t results in a set of terms t' paired with hypotheses H', which may extend H with path conditions and assumptions from special rewrite rules (such as limiting the range of memory addresses used). We also write u and v for terms, x for variables, and c for the names of constants in the rules.

$$\frac{H, S \vdash u \rightsquigarrow \overline{(H', u')}}{H, S \vdash (t, u) \rightsquigarrow \overline{(H', (t, u'))}}\ \textsc{Pair}$$

$$\frac{H, S \vdash t \rightsquigarrow \overline{(H', t')}}{H, S \vdash \texttt{SND}\ t \rightsquigarrow \overline{(H', \texttt{SND}\ t')}}\ \textsc{Snd}$$

$$\frac{\begin{array}{c} H, S \vdash t \rightsquigarrow \overline{(H', (t', s'))} \\ \forall i.\ H'_i, S \lhd s'_i \vdash u[t'_i/x] \rightsquigarrow \overline{(H''_i, u'_i)} \end{array}}{H, S \vdash \texttt{let}\ (x, s) = t\ \texttt{in}\ u \rightsquigarrow \bigcup_i \overline{(H''_i, u'_i)}}\ \textsc{Let}$$

$$\frac{\begin{array}{c} (H, t), S \vdash u \rightsquigarrow \overline{(H', u')} \\ (H, \neg t), S \vdash v \rightsquigarrow \overline{(H'', v')} \end{array}}{H, S \vdash \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ v \rightsquigarrow \overline{(H', u')} \cup \overline{(H'', v')}}\ \textsc{Cond}$$

$$\frac{\forall i.\ (H, pt_i = t), S \vdash u_i \rightsquigarrow \overline{(H'_i, u'_i)}}{H, S \vdash \texttt{case}\ t\ \texttt{of}\ pt_1 \Rightarrow u_1 | \ldots \rightsquigarrow \bigcup_i \overline{(H'_i, u'_i)}}\ \textsc{Case}$$

$$\frac{}{H, S \vdash \texttt{raise'exception}\ t\ u \rightsquigarrow \emptyset}\ \textsc{Undef}$$

$$\frac{\begin{array}{c} c\, x_1 \ldots x_{n+1} := t \qquad H, S \vdash v \rightsquigarrow \overline{(H', v')} \\ \forall i.\ H'_i, S \vdash t[u_1/x_1, \ldots, u_n/x_n, v'_i/x_{n+1}] \rightsquigarrow \overline{(H''_i, t'_i)} \end{array}}{H, S \vdash c\ u_1 \ldots u_n\, v \rightsquigarrow \bigcup_i \overline{(H''_i, t'_i)}}\ \textsc{App}$$

Fig. 3. Rules used in symbolic execution

The L3 translator always places the state record in the rightmost position of a tuple, so the PAIR and SND rules merely follow the state, then reconstruct the surrounding context. The LET rule propogates state updates: for each state s'_i found by executing t, we form a new set of rewrites, denoted $S \lhd s'_i$, which updates the rewrites in S with the changes in s'_i. This new set is then used for the symbolic execution of u.

Case splits are handled by the COND and CASE rules. In each branch we add a new hypothesis corresponding to the guard or pattern match, and then proceed with that branch in isolation. The actual implementation also replaces the variables which are bound in patterns with fresh ones to prevent clashes. In principle, case splitting at every conditional or pattern match would lead to an explosion in the number of cases to consider. In practice, many of the cases are eliminated by the symbolic evaluation due to existing assumptions or the path condition, and from the remainder most lead to some form of undesirable behaviour which is discarded. The UNDEF rule does this for the L3 exceptions which indicate undefined behaviour. Extra rules can be added for any other function in the model; for example, when we are not interested in the handling of processor exceptions we discard paths where we reach the `SignalException` function.

Other functions involving the state are handled by the APP rule. The state is always passed in the final argument, so we process it first then unfold the function's definition. Functions which do not involve the state are unfolded by the symbolic evaluation.

Any term that does not fit one of the rules is only run through the symbolic evaluation.

C. Example

To illustrate the procedure we consider the main definition for a single instruction on fixed operands,

```
dfn'ADDI (2w,1w,3w) s
```

which is the 32-bit signed addition of 3 to the contents of register 2, placing the result in register 1. The APP rule unfolds the definition, which we saw in Figure 1. The first part of the **let**,

```
if NotWordValue (FST (GPR 2w state))
  then SND (raise'exception
             (UNPREDICTABLE "ADDI: NotWordValue")
             state)
  else state
```

is processed recursively, and COND examines each of the branches separately. The first is discarded by SND and UNDEF because the processor's behaviour on a value that cannot be represented in 32 bits is undefined. The second case is trivial, except that we now have an extra hypothesis,

```
¬if word_bit 31 (s.gpr 2w)
 then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
 else (63 >< 32) (s.gpr 2w) ≠ 0w
```

which is the result of evaluating the guard, `NotWordValue ( FST (GPR 2w state))`. The same hypothesis is present in the unsigned case in Figure 2.

In the second part of the **let** the computation of `v` cannot change the state, so it is evaluated, leaving us with the final conditional:

```
if word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
   word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
then SignalException Ov s else
  write'GPR
    (sw2sw
      ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)),
     1w) s
```

Again, COND considers each branch. For the sake of brevity, we will not consider the overflow processor exception. In the second branch, the `write'GPR` definition is unfolded and it continues to the result

```
dfn'ADDI (2w,1w,3w) s =
   ((),
    s with c_gpr :=
      (1w =+ sw2sw
         ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)))
       s.c_gpr))]
```

which updates register 1 with the sum, under the two hypotheses,

```
¬if word_bit 31 (s.gpr 2w)
 then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
 else (63 >< 32) (s.gpr 2w) ≠ 0w

¬word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
 word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
```

which ensure the argument can be represented in 32 bits and that there is no overflow, respectively.

To illustrate how the state updates $S \lhd s$ are calculated, suppose that we start out with an unchanged initial state, `s0`, and want to update it with the result above. The initial set of field rewrites S will be:

```
s.c_gpr   = s0.c_gpr
s.c_state = s0.c_state
...
```

The new set of rewrites reflects the updates to each field. In our case only the register field `c_gpr` is affected:

```
s.c_gpr   = (1w =+ sw2sw (...)) s0.c_gpr
s.c_state = s0.c_state
...
```

Note that the previous rewrite is applied, replacing `s.c_gpr` with `s0.c_gpr`, so that it is expressed in terms of the initial state.

IV. SOUNDNESS, INCOMPLETENESS AND COMPLETENESS

The correctness of the extracted behaviour with respect to the model is ensured by construction because every stage of the process produces a theorem witnessing it. In particular, for each rule of the symbolic execution

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

the system generates a theorem for each result:

$$H'_i \vdash t = t'_i.$$

However, these theorems are generated dynamically, so any bug in the implementation will only be detected during symbolic execution.

This shows that the results will be sound, but we also wish to know whether they will be complete, that is whether the procedure finds all of the relevant behaviour. It must be incomplete in the sense that some behaviour is intentionally excluded; undefined behaviour is not useful for testing, the tests must respect restrictions on endianness and memory layout to run on the test system, and we only wish to explore processor exceptions in a controlled manner.

In principle we could codify all of the undesirable behaviour and construct an additional overall theorem stating that under only basic assumptions either one of the conclusions from the extracted behaviour will be reached, or one of the undesirable situations will. We believe it would be feasible to construct such a theorem because the intermediate results required would be of a similar size and number to the actual results we already compute. However, it would require considerable effort to add the code to compute this theorem, and, as before, it would only detect failure at runtime.

Less formally, we can compare our method with the *step* libraries. These are primarily intended for verification, and some of them deliberately restrict the supported behaviours even further. For example, the ARMv7-A step library requires all word loads to be aligned, even though the architecture permits unaligned loads in some circumstances. This is added by an explicit assumption in the model-specific part of the step library. While such restrictions can be useful for simplifying verification they also illustrate how easily behaviour can be accidentally excluded. In contrast, our library systematically explores the model with fewer user-provided assumptions, so accidentally missing behaviour is less likely. Thus we believe the procedure is complete for a model specialised to any particular processor, fixing details such as endianness and available memory.

V. APPLICATION TO TESTING

To use the library with an instruction set model we still need to provide some model-specific information, such as identifying the functions which raise L3 exceptions and processor exceptions, the standard set of assumptions, specialised rewrites, and the definitions which should not be unfolded (typically to enable a rewrite). For example, one use of model-specific rewrites is to avoid accessing hardware resources that are present in the model for simulation, such as serial ports.

There is a bootstrapping problem for the more complex models. In simple models such as the M0 microcontroller that we previously worked with, instructions can be injected into memory by adding hypotheses for each byte of each instruction. More complex models feature address translation and different memory representations. For example, in the CHERI model memory is represented in chunks that contain either a capability record or a raw capability-sized bitvector. Rather than writing a function to generate suitable assumptions for each model by hand, we use the procedure above to generate a theorem about the behaviour of the model's `Fetch` function, then build a rewrite from it which will ensure that the desired instruction is loaded. The behaviour of the instruction can then be extracted from the `Next` function if we add the rewrite to the set used by the symbolic evaluation. This approach has the advantage that it is relatively robust to changes in the model.

The first model the procedure was applied to was the plain MIPS model. This has a *step* library which we could use for comparison during development and for the performance comparisons below. We then moved on to our main target of interest, the large CHERI model. We had already tested the plain MIPS model against the CHERI hardware design, but CHERI has a considerable number of new instructions without a corresponding *step* library. Moreover, the instructions have a large number of security checks that result in processor exceptions, so in addition to checking fault-free instruction sequences we also generated tests where one of the instructions in the middle of the sequence exercised one of its exceptional behaviours, extracted in the same way.

Testing with the plain MIPS model had already detected bugs in the model and the hardware design. Extending testing to CHERI using our library not only produced tests that would detect those bugs, but also found problems in areas that were not supported by the *step* library; in particular, in the model store conditional instructions did not check enough of the supplied address, and several instructions wrote back results incorrectly when a processor exception was signalled.

We have been able to track changes to the model with very few adjustments. For example, a new instruction will need to be added to the instruction generator, but not the behaviour extraction library. Moreover, when we first targeted the test system at CHERI the most labour intensive model-specific work was adapting the test harness construction code which initialises the test state, because the model-specific part of our behaviour extraction library is so small.

A. Performance

Our main goal is the batch production of tests, so we could accept a large increase in the test generation time. We have not yet explored opportunities to accelerate the process, such as replicating the feature in the *step* libraries to extract the general behaviour of an instruction and cache it for future use with a range of different operands. Nonetheless, we generated 500 8-instruction tests for the plain MIPS model using both the *step* library and our library to compare the libraries and to determine whether behaviour extraction is a bottleneck in test generation.

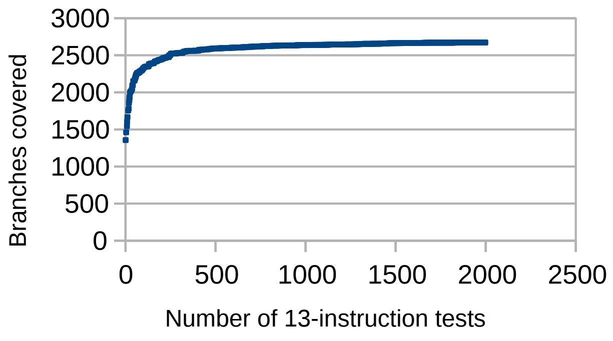


Fig. 4. Cumulative coverage graph

While the median behaviour extraction time increased considerably from 0.23 seconds to 3.16 seconds, it still represents a small fraction of the overall time for test generation, whose median increased from 16.98 seconds to 19.15 seconds. Thus improving our library would not make a huge difference to the rate of test generation unless substantial improvements were made elsewhere. Moreover, the model-specific code for plain MIPS for our library is an order of magnitude smaller than that for the original *step* library.

This is still true with the larger CHERI model. While the median behaviour extraction times are much greater, around 30 seconds for 8-instruction tests, they are still only a third of the total test generation time, and sufficient for batch test generation.

B. Coverage

Following the production of a large batch of CHERI tests for use against the hardware design, we wished to check how much of the model's instruction behaviour was actually exercised and whether any bugs in the behaviour extraction were causing cases to be missed. The testing system was set to produce 13 instruction tests, where the middle instruction raises a processor exception and all other instructions do not.

We measured the branch coverage by building the SML simulator version of the L3 model and using the MLton compiler's coverage support. To see if we had generated enough tests to demonstrate the available coverage we produced a graph showing the cumulative branch coverage of the model, shown in Figure 4. The flattening of the curve suggests that additional random tests would add little to the overall coverage.

To get a more qualitative idea of how good the coverage is we manually examined the branches in the instruction definitions that were not covered after 2000 tests[3]. The branches that were not covered fell into two groups: those which are impossible due to undefined behaviour or an unnecessary default case in a complete pattern match; and those which

[3]Examining the model as a whole is not appropriate due to the amount of code that is outside of the scope of the tests, such as serial ports, interrupts, full address translation, instruction encoding, and disassembly.

testing did not reach by pure chance, which were almost all due to the very large number of security checks in the new CHERI instructions. We know that the latter group were due to chance because the same checks in other instructions were tested; if we made the random test generation more targeted it should be possible to cover these cases without greatly increasing the number of tests. There was one other case amongst the branches that were not covered: a trap instruction that had accidentally been omitted in the instruction generation phase, which was easily corrected.

VI. Related Work

Automated conversions between functional semantics and more structured operational semantics has been studied in many forms. Note that the output of our system is not structured to the same extent as a Plotkin-style structured operational semantics [9]; while we do produce rules with stylised conclusions, separate hypotheses, and where we have one rule for each behaviour, we do not break the execution down into a set of intuitive judgements and build up behaviour in derivation trees. Instead, we provide monolithic rules where everything is described in basic terms, slicing across the entire model. For our automated testing this is quite reasonable because we will eventually present constraints derived from the rules to an SMT solver which has no direct knowledge of the model.

General purpose tools for converting or reasoning about functional semantics are more structured. The Function mechanism for Coq [10] generates an inductive relation for the graph of a function, together with induction and inversion principles. The relation does split out the different behaviours of the function, but there is a relation for every function, which would be difficult to use on a model with hundreds of definitions. Similarly, Owens et al. [11] recently advocated using functional definitions for programming language semantics when suitable induction principles are generated. In their case their principles came from HOL4's mechanism for defining recursive functions.

There are other functional semantics for instruction sets where rules have been manually specified for each instruction, and proved as lemmas. Srivas and Miller [12] did this when verifying the microcode for the AAMP5 processor, staying close to the pseudocode initially, then deciding to derive rules, saying:

> They were more readable, simpler to validate, and were closer to what a user wanted to know in the first place. They also made it possible to specify a small portion of the `next_macro_state` function, i.e., to specify one instruction or part of an instruction at a time.

However, this is exactly the type of work we wish to automate. Similarly, Jensen et al. [13] formalised a subset of the x86 architecture in Coq for verifying machine code programs where they proved manually specified separation logic rules.

Fox's libraries [4], which we described in Section II, fit in between these completely manual rules and our almost automatic system. The results differ from ours in several ways: they must correspond closely to the accompanying program logic library, work with partially specified operands, cache results, and produce one rule per branching choice, rather than partitioning the behaviour according to the structure of the definition. It may be possible to adapt our library to cover some of these points, but we leave that to future work.

There is also a body of work on translating from rule-based relational semantics to functions, such as Isabelle's predicate compiler [14] and a similar feature for Coq by Tollitte et al. [15]. This is an attractive way to animate semantics which have been presented relationally, as is common for many programming languages. Indeed, Lochbihler and Bulwahn have done this for a Java-like language [16]. However, if we were to rewrite our model like this we would lose the close correspondence to the designers' pseudocode. Transformations in this direction do have the advantage that they can produce several functions, depending on which parameters of the relation are chosen as inputs and outputs of the generated function.

Turning our attention to the techniques involved, symbolic evaluation is widely used with executable models. Fox's libraries provide one example. Moore [17] advocates the use of symbolic evaluation of a machine model for exploring the behaviour of assembly programs, which he calls *symbolic simulation*. Having demonstrated that an example program can be simulated in ACL2 despite only partially specifying the input, Moore suggests that this is reasonably accessible to engineers and that a special purpose user interface would aid adoption.

Symbolic execution has a long history of use in testing, early work by King [8] used it for interactive testing, while Boyer et al. [18] primarily generated test cases. It is now commonly used for automatic test case generation on large programs using *concolic* testing [19], where the symbolic execution follows the same path as the concrete execution of a test case, and part of the resulting path condition is negated to force the solver to find a test case which explores a new branch without searching through the full space of paths. Our library is closer to Boyer et al. because we explore all of the well-defined paths for a single instruction during symbolic execution, but leave the test case generation to a later phase of the testing process that uses the entire instruction sequence. In contrast, concolic methods have been used for single instruction simulator testing by Wagstaff et al. [20] for high coverage, and similar methods by Martignoni et al. [21] for cross-testing.

VII. Further work

The most straightforward area of possible work is to use the library in test generation for other L3 models; partly to test these models, and partly to identify any remaining aspects of the library that are too reliant on the particular models above. This would still require some manual effort for each model to construct the instruction generator and the production of harness code.

One danger with other L3 models is that they may make greater use of looping constructs. The plain MIPS and CHERI models use no recursion, have `FOR` loops with statically known iteration counts, and do not have more than one well-defined path inside a loop (which prevents the number of cases exploding). While other models may be similar, it is possible that difficult loops may appear occasionally. In fact, there is one in the CHERI model's address translation but it does not affect our testing because it is not used for the parts of memory that our tests run in. The loop tests each TLB entry, resulting in an exponential number of paths, although the surrounding code restricts the well-defined paths to one per entry. To tackle these issues we could manually prove that the loop can be replaced with a simpler form, or attempt a general solution, perhaps by analysing the loop body separately and adding more structure to the output.

We could also investigate adding more of the features from Fox's library, as described above. The exact requirements for interfacing with his program logic libraries are unclear, and it may be that our results are not sufficiently idiomatic to be compatible. Support for partially specified instructions and caching seem more feasible, and would improve performance.

A more ambitious task would be to apply the same approach to an instruction set modelling language which supports some weak memory model, both to test the sequential behaviour of such models, and to investigate automatic test generation for multicore architectures.

VIII. Conclusion

We have constructed a library which uses symbolic execution in a theorem prover to extract rule-based descriptions of processor behaviour from L3 executable instruction set models with minimal user-provided information about the model. The structure of the results can be used to drive an automatic test generation system, and the soundness of the procedure is ensured by the HOL4 theorem proving system. The resulting system has been successfully used with a large model of the CHERI experimental MIPS-like processor.

Acknowledgment

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/K008528/1], Rigorous Engineering for Mainstream Systems (REMS).

References

[1] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh, "Simulation and formal verification of x86 machine-code programs that make system calls," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14. Austin, TX: FMCAD Inc, 2014, pp. 18:91–18:98.

[2] A. Fox, "Directions in ISA specification," in *Interactive Theorem Proving (ITP 2012)*, ser. LNCS, L. Beringer and A. Felty, Eds. Springer, 2012, vol. 7406, pp. 338–344.

[3] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, March 2016.

[4] A. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, 2015, pp. 187–202.

[5] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 2014, pp. 457–468.

[6] T. Weber, "SMT solvers: New oracles for the HOL theorem prover," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 419–429, 2011.

[7] B. Barras, "Programming and computing in HOL," in *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, 2000, pp. 17–37.

[8] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] G. D. Plotkin, "A structural approach to operational semantics," Computer Science Department, Aarhus University, Tech. Rep. DAIMI FN-19, 1981.

[10] G. Barthe, J. Forest, D. Pichardie, and V. Rusu, "Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant," in *Functional and Logic Programming (FLOPS 2006)*, ser. Lecture Notes in Computer Science, M. Hagiya and P. Wadler, Eds., vol. 3945, 2006, pp. 114–129.

[11] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, "Functional big-step semantics," in *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, pp. 589–615.

[12] M. K. Srivas and S. P. Miller, "Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods," *Formal Methods in System Design*, vol. 8, no. 2, pp. 153–188, 1996.

[13] J. B. Jensen, N. Benton, and A. Kennedy, "High-level separation logic for low-level code," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, 2013, pp. 301–314.

[14] S. Berghofer, L. Bulwahn, and F. Haftmann, "Turning inductive into equational specifications," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5674, pp. 131–146.

[15] P. Tollitte, D. Delahaye, and C. Dubois, "Producing certified functional code from inductive specifications," in *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, 2012, pp. 76–91.

[16] A. Lochbihler and L. Bulwahn, "Animating the formalised semantics of a Java-like language," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds. Springer Berlin / Heidelberg, 2011, vol. 6898, pp. 216–232.

[17] J. Strother Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and P. Windley, Eds. Springer Berlin / Heidelberg, 1998, vol. 1522, pp. 530–530.

[18] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 234–245.

[19] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272.

[20] H. Wagstaff, T. Spink, and B. Franke, "Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '14, K. S. Chatha, R. Ernst, A. Raghunathan, and R. Iyer, Eds. ACM, 2014, pp. 15:1–15:10.

[21] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: hi-fi tests for lo-fi emulators," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, 2012, pp. 337–348.

Categorical Semantics of Digital Circuits

Dan R. Ghica
University of Birmingham, UK

Achim Jung
University of Birmingham, UK

Abstract—**This paper proposes a categorical theory of digital circuits based on monoidal categories and graph rewriting. The main goal of this paper is conceptual: to fill a foundational gap in reasoning about digital circuits, which is currently almost exclusively semantic (simulations). The level of abstraction we target is circuits with discrete signal levels, discrete time, and explicit delays, which is appropriate for modelling a range of components such as boolean gates or transistors working in saturation mode.**

We start with an algebraic signature consisting of the basic electronic components of a given class of circuits and extend it gradually (and in a free way) with further algebraic structure (representing circuit combinations, delays, and feedback), while quotienting it with a notion of equivalence corresponding to input-output observability. Using well-known results about the correspondence between free monoidal categories and graph-like structures we can develop, in a principled way, a graph rewriting system which is shown to be useful in reasoning about such circuits. We illustrate the power of our system by reasoning equationally about a challenging class of circuits: combinational circuits with feedback.

Index Terms—**Digital circuits, circuit topology, feedback circuits, multivalued logic, category theory, monoidal categories**

I. Introduction

A. Syntactic vs. semantic modelling

Theories of *programming languages* can be either *syntactic*, formulated equationally, or *semantic*, formulated via translation into a mathematical domain. The former is commonly known as *operational semantics* [1] and the latter as *denotational semantics* [2]. Even though denotational models are attractive conceptually, the difficulties of producing precise ("*fully abstract*") models of realistic programming languages led to a prevalence of operational reasoning methods in practice, for example, when proving compilers correct [3].

In contrast, formal models of digital circuits are virtually exclusively of a denotational nature, where circuits are translated into an executable model (e.g. automata [4]) so that their behaviour can be *simulated*. As far as denotational models go, such translations tend to obscure the structure of the circuit and are more akin to *compilation* than genuine modelling. This is not to say such models are not effective in reasoning about circuits, in fact they are the foundation of a thriving industry which places a premium on correctness. However, the contrast between hardware and software in terms of reasoning techniques is striking.

Our primary motivation is to address a surprising methodological and conceptual gap, the paucity of syntactic models for digital circuits. Our methodology is heavily influenced by developments, in the last decade, in diagrammatic reasoning for a variety of computational models such as quantum computing [5], signal flow [6] or asynchronous circuits [7]. The basic insight of this approach is that the same algebraic structures ("*monoidal categories*") which are very helpful in describing the structure of systems in general [8] also occur in certain diagrams [9]. This approach formalises the intuitive connection between systems and diagrams, also reflected in the rich diversity of graphical environments for circuits and systems (e.g. structural HDLs, Simulink).

B. Methodology and contribution

In terms of level of abstraction we will simply consider digital circuits as broadly construed, i.e. processing a finite number of discrete input and output levels, with explicit and discrete delays down to a smallest observable delay (δ). This level of abstraction applies to Boolean circuits but also to transistor-level modelling, if the transistor operates in non-linear mode.

We will start from a basic algebraic signature meant to represent abstractly the basic components (e.g. transistors) used in the construction of a circuit. Then we will follow a sequence of free categorical constructions and quotientings which will systematically lead to models of digital circuits with delays and feedback. The free constructions represent step-by-step expansions of the algebraic language with new features, whereas the quotientings represent *cutting down* the model by imposing a notion of *observability*. The free constructions have well-known diagrammatic equivalents which will facilitate reasoning and calculations via diagrammatic graph rewriting.

Our most important guiding principle is the use of "*small*" axioms, i.e. axioms describing the *local* interactions of components, rather than "*coherences*" describing the interplay of complex sub-circuits. We believe this approach to be more physically realistic and more promising for a potential efficient implementation. Coherences, wherever needed, will need to be derived as theorems. The main technical result of the paper is that diagrams of circuits with feedback (*trace*, categorically)

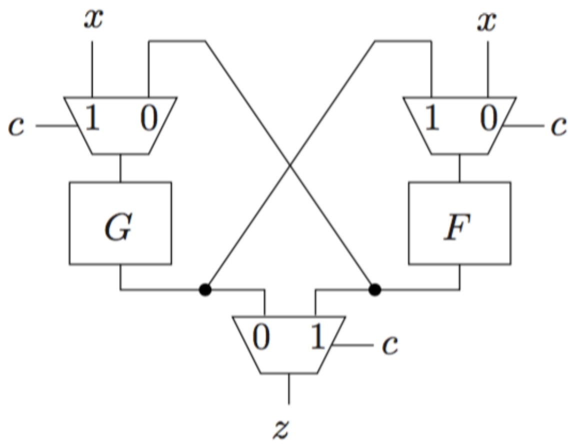


Fig. 1. Combinational circuit with (false) feedback ([10, Fig. 1])

have products and therefore feedback can be treated as iteration.

This work was inspired by Berry et. al.'s semantic treatment of cyclic combinational circuits [10], a class of circuits which straddles the combinational-sequential boundary. We aim to provide an equational counterpart to their approach.

C. A motivating class of examples

A particularly intriguing class of circuits are combinational circuits with feedback, as identified by Malik [11]. The presence of feedback suggests that these circuits are *not* combinational yet their behaviour does not lead to any latching effects. Moreover, the feedback loop in purely combinational circuits is banned by the usual synchronous design methodology and cannot be handled by conventional tools. A *semantic* characterisation of such circuits is given in [10]. Our aim is provide an equational method to complement the designer's toolkit of reasoning methodologies.

A simple such circuit is presented in Fig. 1, where a (false) feedback loop is used to create shared data-path resources. The circuit computes $z = \texttt{if}\ c\ \texttt{then}\ F(G(x))\ \texttt{else}\ G(F(x))$, which could also be implemented in a cycle-free fashion by duplicating the sub-circuits F and G. Being able to handle such circuits with zero-delay cycles directly has many benefits, as explained in *loc. cit.*, but we target them not particularly for their benefit but for the technical challenge they pose to conventional modelling frameworks.

II. Combinational digital circuits

Our formal language of circuits is based on *category theory*, and in general requires two sorts of variables: *object variables* for labelling (collections of) wires and *morphism variables* for labelling boxes (e.g., gates and circuits). Since we use only one kind of wire we can do away with wire labels and, instead, use just natural numbers to indicate the width of the wire collection (bus). We obtain what is usually called a category of PROducts and Permutations, or PROP for short [12].

Definition 1. *Let* $\mathbf{Circ}$ *be a categorical signature with objects the natural numbers* $\mathbb{N}$ *and a (typically finite) set of morphisms which may be grouped into the following three classes:*

- levels $v : 0 \to 1$*;*
- gates $k : m \to 1$*; and*
- *the special morphisms* join $\mathsf{j} : 2 \to 1$, fork $\mathsf{f} : 1 \to 2$*, and* stub $\mathsf{w} : 1 \to 0$.

We further assume that there are only finitely many levels and that they form a lattice $(\mathbf{V}, \sqsubseteq)$*. Instead of "level" we will also use "value".*

All circuit signatures include combinators for joining two outputs (*join*) and duplicating an input (*fork*), as well as the ability to discard an output (*stub*). What varies from signature to signature is the number of signal levels we consider, as well as the sets of gates we want to model. Since they form a lattice, the levels must always include a smallest element ($\bot$), corresponding to a disconnected input, and a top element ($\top$) corresponding to an illegal output ("short circuit"). In the simplest and most common instance, the set of level has two other elements, *high* and *low*, but it can go beyond that. For example, in the case of metal-oxide-semiconductor field-effect transistors (MOSFET) it makes sense, in certain designs, to model the diode properties of the transistor by taking into account four levels (strong and weak high and low voltage).

Circuits, in the sense of this paper, are the morphisms of a free categorical construction over their signature. Beginning with *combinational circuits*, the free construction is as follows:

Definition 2. *Let* $\mathbf{CCirc}$ *be the free symmetric monoidal category over* $\mathbf{Circ}$*, subject to the following additional equations:*

Input-output characterisation of gates *(*extensional completeness*): For any gate* $k\colon m \to 1$*, for any values* $v_i, 1 \leq i \leq m$*, there exists a unique value* v' *such that* $k \circ \bigotimes_{i=1,m} v_i = v'$.

Monotonicity*: For any gate* $k : m \to 1$*, for any values* $v_i, v_i', 1 \leq i \leq m$*, if* $v_i \sqsupseteq v_i'$ *then* $k \circ \bigotimes_{i=1,m} v_i \sqsupseteq k \circ \bigotimes_{i=1,m} v_i'$.

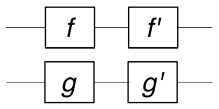
We also require the following equations:

Fork: $\mathsf{f} \circ v = v \otimes v$.
Join*:* $\mathsf{j} \circ (v \otimes v') = v \sqcup v'$.
Stub*:* $\mathsf{w} \circ v = 0$.

The last three encapsulate the understanding that a fork duplicates a value, a join coalesces two values, and a stub discards anything it receives (0 being the identity on the object 0, representing an absence of a wire).

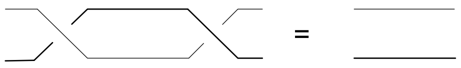
It is known that, in a formal sense, the equality of morphisms in a free symmetric monoidal category (SMC) corresponds to graph isomorphisms in the diagrammatic language [13], where diagrams are created by the operations of sequential composition ($\circ$), parallel composition ($\otimes$) and symmetry ($\mathsf{x}_{m,n}$, the swapping of two buses with m and n wires, respectively), governed by coherence equations. We will usually write composition in diagrammatical order $f \cdot g = g \circ f$. We write the identity (bus of width m) $\mathrm{id}_m : m \to m$ as simply m. For simplicity we also write $\bigotimes_{i=1,m} f = f^m$, $\bigotimes_{i=1,m} f_i = \mathbf{f}$ and $\bigotimes_{i=1,m} v_i = \mathbf{v}$.

Without enumerating them, the equations of SMCs reflect either the fact that the same diagram can be described in two ways or the fact that two diagrams are graph isomorphic. An example of the first case is the *functoriality* of $\otimes$, which gives two equal ways of describing the diagram below.



$$(f \cdot f') \otimes (g \cdot g') = (f \otimes g) \cdot (f' \otimes g').$$

An example of the second kind is the fact that bus-swapping is *involutive*:



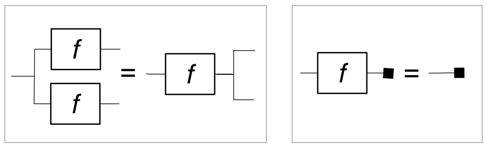
$$\mathsf{x}_{m,m} \cdot \mathsf{x}_{m,m} = 2m$$

The *extensional completeness* principle states that in a circuit the behaviour of a gate must be determined by its input values only. This means that we will deliberately ignore global interactions between components such as electromagnetic interference or quantum tunnelling. *Monotonicity* says that gates must behave monotonically with respect to the ordering of the levels.[1]

We shall see later the importance of these equations. Finally, the last equation represents our commitment to an input-output based notion of equivalence: no matter what value we plug into an unobserved output (*stub*) we get equivalent circuits, so that all circuits with no inputs and no outputs end up being equal.

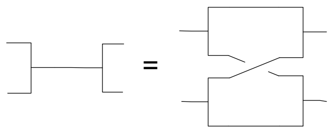
By simple inductive arguments on the structure of morphisms we can establish that all circuits are extensionally complete, i.e. for any circuit (not just gates) $f : m \to n$, for any values $v_i, 1 \leq i \leq m$, there exists unique values $v'_j, 1 \leq j \leq n$ such that $f \circ \bigotimes_{i=1,m} v_i = \bigotimes_{i=1,n} v'_i$. We can further say that two circuits with the same input-output behaviour are *extensionally equivalent*, and we can easily prove, also by structural induction, that this is a *congruence*, i.e. it is an equivalence preserved by sequential and parallel composition. Therefore it makes sense to *quotient* our category **CCirc** and create a new category **ECCirc** in which equivalent circuits are made equal.

ECCirc has interesting additional categorical properties which aid reasoning, but two are of particular importance. The first one is that **ECCirc** is *Cartesian*, i.e. it has a notion of *product*. The *diagonal* circuit is defined by $\Delta_0 = 0$ and $\Delta_{n+1} = (\Delta_n \otimes \mathsf{f}) \cdot (n \otimes \mathsf{x}_{(1,n)} \otimes 1)$ and it represents the forking of a bus of width n. The diagonal has two important *coherences* represented by the following diagram equalities valid for any diagram $f : n \to m$.



$$\langle f, f \rangle = \Delta_n \cdot (f \otimes f) = f \cdot \Delta_m \qquad f \cdot \mathsf{w}^m = \mathsf{w}^m.$$

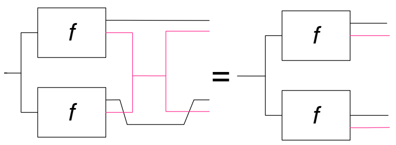
These coherences are immediate by structural induction over f using extensionality. Also using extensionality we can easily show that $(\mathsf{f}, \mathsf{j}, \mathsf{w}, \perp)$ forms what is known as a *Frobenius monoid*, i.e. an algebraic structure in which $(\mathsf{j}, \perp)$ is a commutative monoid, (f, w) is a co-commutative co-monoid, interacting subject to the following law:



$$\mathsf{j} \cdot \mathsf{f} = \mathsf{f}^2 \cdot (1 \otimes \mathsf{x}_{1,1} \otimes 1) \cdot \mathsf{j}^2.$$

In a special context, which will prove to be useful, join and fork behave as if they are inverses.

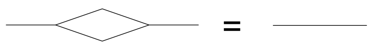
Proposition 1. *For any* $f : m \to n + 1$.



$$\Delta_m \cdot f^2 \cdot (n \otimes (\mathsf{j} \cdot \mathsf{f}) \otimes n) = \Delta_m \cdot f^2 \cdot (n \otimes 1)^2.$$

Of course, composed the other way round, fork and join are always inverses of each other:

[1]Regarding the earlier example of the four values $\{\perp, \text{low}, \text{high}, \top\}$, note that we would not order *low* below *high*; both are valid levels on an equal footing.



$\mathsf{f} \cdot \mathsf{j} = 1.$

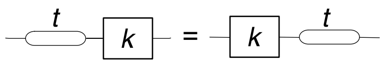
Finally, it will be useful to use a *co-diagonal* which is the joining of two buses of width n, defined as $\nabla_0 = 0$ and $\nabla_{n+1} = (n \otimes \mathsf{x}_{1,n} \otimes 1) \cdot (\nabla_n \otimes \mathsf{j})$. Note that $\nabla_1 = \mathsf{j}$.

III. Circuits with Discrete Delays

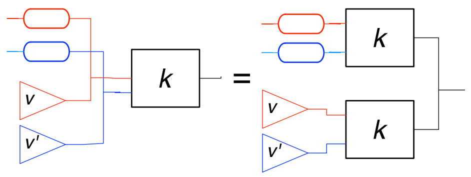
The next step is the free introduction of *delays* which we represent diagrammatically as an elongated oval.

Definition 3. *Let* $\mathbf{CCirc}_\delta$ *be the category obtained by freely extending* $\mathbf{ECCirc}$ *with a* $\mathbb{Z}$*-indexed family of morphisms* $\delta_t : 1 \to 1$ *such that* $\delta_0 = 1$*,* $\delta_{t+t'} = \delta_t \cdot \delta_{t'}$ *and:*

Timelessness*: For any gate or structural morphism* $k : m \to n$ *and delay* $t \in \mathbb{Z}$*,* $\delta_t \cdot k = k \cdot \delta_t$*:*



Streaming*: For any levels* $\mathbf{v} = v \otimes v'$ *and gate* k*,* $(\delta^2 \otimes \mathbf{v}) \cdot \nabla_2 \cdot k = ((\delta^2 \cdot k) \otimes (\mathbf{v} \cdot k)) \cdot \nabla_1$.



Disconnect*:* $\bot \cdot \delta = \bot$.
Unobservable delay*:* $\delta \cdot \mathsf{w} = \mathsf{w}$.

Timelessness means that for any idealised, instantaneous gate or structural morphism (i.e. wire, fork, join, swap, etc.) delaying the inputs by some value t can be compensated by "anti-delaying" the output, by $-t$ [14]. An immediate consequence is that delays can be propagated through combinational circuits, akin to *retiming* [15]. For simplicity we write $\delta_1 = \delta$. Note that any circuit with negative delays, which are not realistic, can be retimed into a realistic circuit by delaying the output:

Theorem 2. *For any circuit* $f : m \to n$*, there exists* $t \in \mathbb{Z}$ *such that* $f \cdot \delta_t^n$ *has no negative delays.*

The proof is immediate by induction on the structure of f and using retiming. We will call the sub-category of circuits without negative delays $\mathbf{CCirc}_+$

Streaming is used to handle waveforms equationally. A waveform of length n is a sequence of n levels $v_n :: v_{n-1} :: \cdots :: v_1$ created using delays and joins, so they always have the form $s_1 = v_1$, $s_{n+1} = (s_n \cdot \delta \otimes v_n) \cdot \mathsf{j}$. The streaming axiom states that to process a waveform we can create two separate instances of a gate, to process the "head" and the "tail" separately, then join the outputs. As far as we know this is a new axiom.

The final two axioms describe the (trivial) interaction between delays and dangling inputs or outputs.

Circuits with delays can also be described extensionally in terms of their input-output behaviour. Because of the presence of delays, now we must use *finite waveforms*, described above and ranged over by s. As before, we write $\bigotimes_{i=i,m} s = s^m$ and $\bigotimes_{i=i,m} s_i = \mathsf{s}$.

Theorem 3 (Extensionality of waveforms)**.** *For any morphism* f *in* $\mathbf{CCirc}_+$ *we have that for any input waveform* s *there exists a unique output waveform* s' *such that* $\mathsf{s} \cdot f = \mathsf{s}'$.

The proof is by induction on the structure of f and uses routine calculations and the following lemma.

Lemma 4. *For any waveform* s *of size* n *and for any* $m \geq n$ *there exists a waveform* s' *of size* m *such that* $\mathsf{s} = \mathsf{s}'$.

The larger waveform s' is constructed by adding delayed $\bot$ values wherever required. $s_{n+1} = (s_n \otimes (\bot \cdot \delta_n)) \cdot \mathsf{j} = s_n$ from monoid axioms and the *disconnect* axiom.

As in the case of circuits without delays, we can show that extensionality is a congruence and we can quotient by it, creating an *extensional* category of circuits with delays, $\mathbf{ECCirc}_+$.

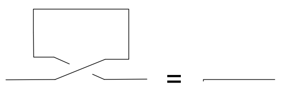
It is a routine exercise to show $\mathbf{ECCirc}_+$ is Cartesian, with the diagonal and terminal object defined the same as in $\mathbf{ECCirc}$, imitating the proof from the previous section.

IV. Circuits with Feedback

We can now introduce feedback.

Definition 4. Let $\mathbf{CCirc}_\delta^*$ (and $\mathbf{CCirc}_+^*$, respectively) be the category obtained from $\mathbf{ECCirc}_\delta$ ($\mathbf{ECCirc}_+$, respectively) by freely adding a trace operator.

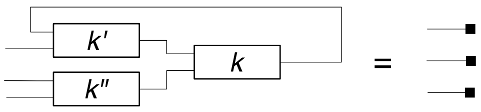
Diagrammatically, the trace operator applied to a diagram $f : m + k \to n + k$ corresponds to a feedback loop of width k, written $\mathrm{Tr}^k(f) : m \to n$. Symmetric traced monoidal categories (STMC) satisfy a number of equations (coherences) which we will not enumerate for lack of space [16]. As before, their interpretation coincides with equality of diagrams (with feedbacks) up to graph isomorphism. For example, of particular interest is the axiom "*yanking*" a loop into a straight line:



$\mathrm{Tr}^m(\mathsf{x}_{m,m}) = m.$

This is indeed an axiom that indicates that, conceptually, we are on the right track. The swapping of two wires is a trivial combinational circuit, and applying a trace creates a (false) feedback loop which can be simply eliminated.

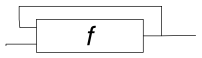
As before, we are committed to an extensional view of circuits where the only observable is the input-output behaviour. In combinational circuits, with or without delays, the only way we can create a circuit with 0 outputs is by explicitly composing a circuit $f : m \to n$ with w^n. However, 0-output circuits can arise in more complicated ways in the presence of feedback, whenever all the outputs are fed back. For example, the diagram on the left can be reduced to just three unobserved inputs:



Such equalities cannot be proved out of local interactions, so we will simply impose the equivalence of all 0-output circuits, an equivalence which is trivially a congruence. The new quotient category is called $\mathbf{OCirc}^*_\delta$. In this category all diagrams of shape $f : m \to 0$ are therefore equal which, categorically speaking, makes 0 a "*terminal object*".

We are now approaching the main result of our paper: reasoning equationally about circuits with feedback. In general, in programs feedback corresponds to recursion and iteration, and syntactic models (operational semantics) of such programs involve creating two copies of the code recursed over. For example, the operational semantics of the Y-combinator as applied to some G is $YG = G(YG)$.

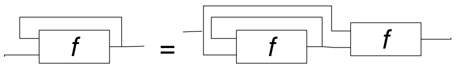
A similar rule does not exist in general for SMTCs unless the category is also Cartesian. Such categories, also called *control-flow categories* [17], admit an *iterator* defined for any $f : m + n \to n$:



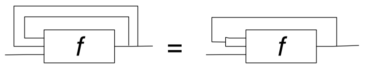
$$\mathrm{iter}^n(f) = \mathrm{Tr}^n(f \cdot (\Delta_n \otimes n)) : m \to n$$

which satisfies the following equations:

Iteration: $\mathrm{iter}(f) = \langle m, \mathrm{iter}(f)\rangle \cdot f$



Diagonal: $\mathrm{iter}^n(\mathrm{iter}^n(f)) = \mathrm{iter}^n((\langle n, n\rangle \otimes m) \cdot f)$.



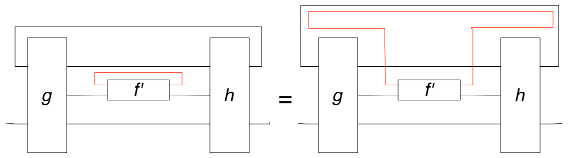


Fig. 2. Global trace form

To use this essential axiom we need to first establish that the SMTC of circuits with feedback is Cartesian. This will be the main technical result of this paper. Before we do that we will establish the following result which holds in general about SMTCs and can be proved by diagrammatic reasoning. Each diagram with feedbacks can be constructed from a feedback-free diagram and one single global trace.

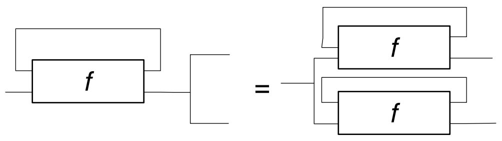
Lemma 5 (Global trace)**.** *For any morphism f in a SMTC PROP there exists a trace-free morphisms $\hat{f}$ such that $f = \mathrm{Tr}^A(\hat{f})$ for some object A.*

In the case of a PROP the object $A = m \in \mathbb{N}$. The diagram $\hat{f}$ is constructing by "*pulling out*" any internal feedback loop and applying to the whole diagram in a process similar to lambda-lifting. Pictorially, this construction looks is illustrated in Fig. 2.

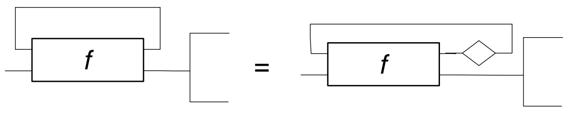
Theorem 6. *The category $\mathbf{OCirc}^*_+$ is Cartesian with diagonal Δ_n.*

Proof. We need to prove the *naturality* of the diagonal, i.e. for all $f : m \to n$, $f \cdot \Delta_n = \Delta_m \cdot (f \otimes f)$. We use induction on the structure of the diagram f. For all gates and structural morphisms the equation holds because it holds in the category of combinational circuits with delays $\mathbf{ECCirc}_\delta$. Tensor and composition are immediate by induction and simple algebraic calculations.

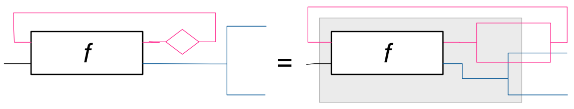
The most interesting case is that of the trace, where we need to show that assuming $f \cdot \Delta_m = \Delta_n \cdot (f \otimes f)$ we have that $\mathrm{Tr}(f) \cdot \Delta_m = \Delta_n \cdot (\mathrm{Tr}(f) \otimes \mathrm{Tr}(f))$:



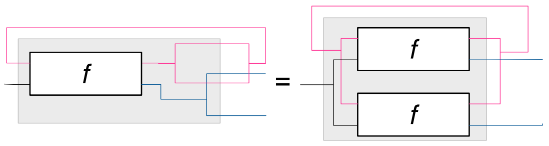
Using the Global Trace Lemma, we can assume that f is trace-free otherwise we can simply incorporate all the internal traces into the global trace. Since $\mathsf{f} \cdot \mathsf{j} = 1$ we have the following equality of diagrams:



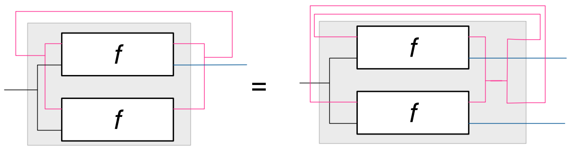
Using coherences (graph isomorphisms) of the SMTC we can rearrange the diagram as follows:



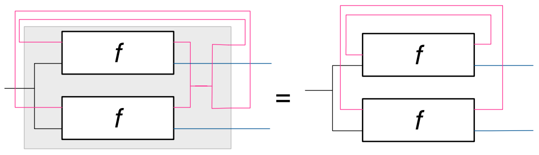
Note that now in the greyed diagram we have the trace-free morphism $f \cdot \Delta_n = \Delta_m \cdot (f \otimes f)$, so by induction hypothesis:



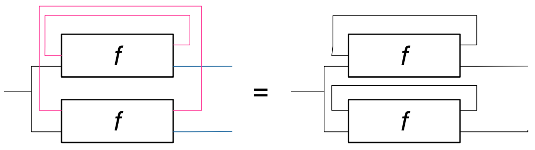
We note the following circuits are graph-isomorphic so they represent equal diagrams in the SMTC:



Noting that the grey sub-diagram is still trace-free we can apply Prop. 1 and obtain the following equal diagram:



The final step is again purely diagrammatic, involving two graph-isomorphic circuits:



□

V. Equational reasoning

We have established a comprehensive equational theory which allows us to reason purely syntactically about digital circuits with feedback and discrete delays. We will now apply it to reason about circuits such as the one in Fig. 1. Since that circuit is built using only multiplexers as constants (*gates*, as broadly construed) we are going to consider a categorical signature consisting of one gate (m) and two levels, high (h) and low (l) in addition to the low-impedance ($\perp$) and illegal ($\top$) values. This is for the sake of simplicity, as we could go down to standard boolean gates or even transistors. The equations describing the multiplexer are the standard ones.

In our categorical notation, the circuit diagram is represented by the following term: $M = \mathrm{Tr}^1\big((\mathsf{x}_{1,1} \otimes \mathsf{f}) \cdot (\mathsf{f} \otimes \mathsf{x}_{1,1} \otimes 1) \cdot (\mathsf{f} \otimes (\mathsf{m} \cdot G \cdot \mathsf{f}) \otimes 1) \cdot (3 \otimes \mathsf{x}_{1,1}) \cdot (1 \otimes (\mathsf{m} \cdot F) \otimes i) \cdot (\mathsf{x}_{1,1} \otimes 1)\big) \cdot (\mathsf{x}_{1,1} \otimes 1) \cdot \mathsf{m}$. We will also consider that each wire segment has some delay δ_t. By applying retiming we can reduce the number of required delays and obtain the diagram in Fig. 3 (the delays may all be different).

We will prove that if we apply high or low to the input that connects to the control port of the MUX, the resulting circuit is combinational. Given a value v let us define the constant waveform $v^\omega = \mathrm{Tr}((1 \otimes \mathsf{f}) \cdot (\delta \otimes 1))$.

Example 1. If $s \in \{\mathsf{h}^\omega, \mathsf{l}^\omega\}$, and F, G are combinational circuits then $(v \otimes 1) \cdot M$ is combinational.

Proof. The two cases to consider are all similar. We only show $v = \mathsf{l}$, which is more interesting. We will not show the detailed algebraic calculations but emphasise the diagrammatic reasoning, which is mathematically equivalent and more intuitive. Diagrammatically, we write the constant low (l^ω) as a small diamond.

First we use the axioms for *fork* and *swap* several times, so the diagram reduces to that in Fig. 4.

At this stage we would like to reason extensionally about the MUXs which receive a low (l^ω) waveform on the control port, but the presence of the delay stops us. We need to use the timelessness of the MUX, and reason as follows:

$$\begin{aligned}
&(\mathsf{l}^\omega \otimes 2) \cdot (\delta_t \otimes \delta_{t'} \otimes \delta_{t''}) \cdot \mathsf{m} \\
=&(\mathsf{l}^\omega \otimes 2) \cdot (\delta_{t-t} \otimes \delta_{t'-t} \otimes \delta_{t''-t}) \cdot \mathsf{m} \cdot \delta_t \\
=&(1 \otimes \delta_{t'-t} \otimes \delta_{t''-t}) \cdot (\mathsf{l}^\omega \otimes 2) \cdot \mathsf{m} \cdot \delta_t
\end{aligned}$$

The second step is by functoriality. $(\mathsf{l}^\omega \otimes 2) \cdot \mathsf{m} = \mathsf{w}^2 \otimes 1$ by extensional reasoning about trace-free circuits ($\mathbf{ECCirc}_+$). So the above is further equal to

$$\begin{aligned}
=&(1 \otimes \delta_{t'-t} \otimes \delta_{t''-t}) \cdot (\mathsf{w}^2 \otimes 1) \cdot \delta_t \\
=&\delta_{t''-t} \cdot \delta_t = \delta_{t''}.
\end{aligned}$$

Note that above we may have strayed temporarily outside the safe confines of circuits without negative delays as $t' - t$ and $t'' - t$ may be negative! This is not a problem so long as we carefully avoid using properties which only apply to circuits with no negative delays.

Applying this equation for all the MUXs, the diagram becomes as in Fig. 5.

From here on, using the unobservability of delays on blocked outputs, the fact that (f, w) is a co-monoid, and combining delays we get the diagram on the left which can be "yanked" using the STMC coherences is shown in Fig. 6.

This is a combinational circuit. The fact that the feedback loop was false is confirmed by the fact that we yanked rather than

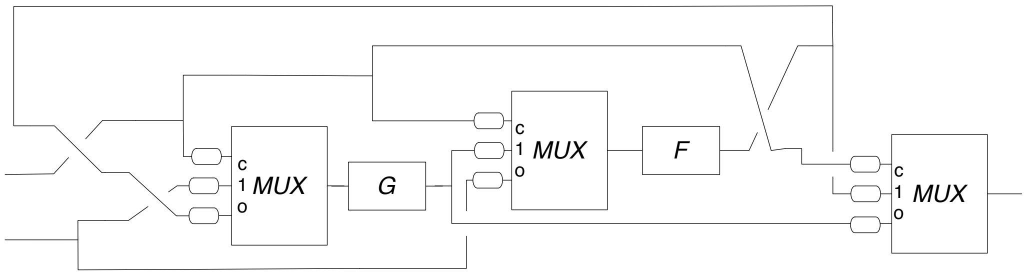


Fig. 3. Diagrammatic representation of the circuit in Fig. 1

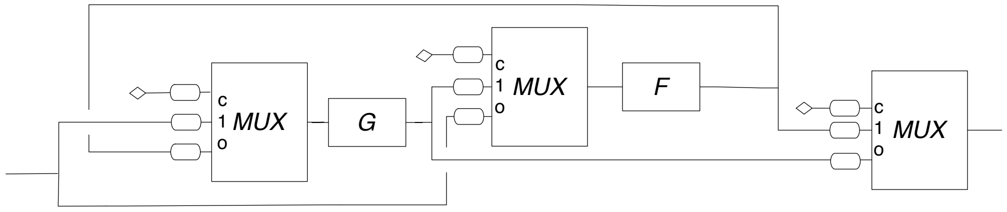


Fig. 4. Reduced representation of the circuit in Fig. 1

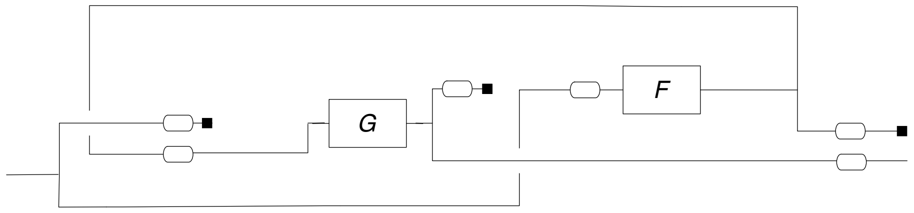


Fig. 5. Further reduced representation of the circuit in Fig. 1

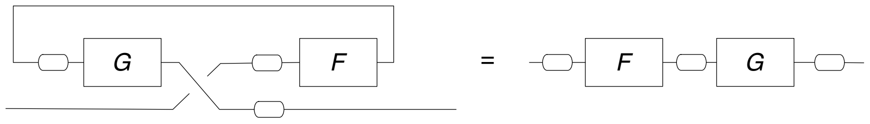


Fig. 6. Final representation of the circuit in Fig. 1

iterate the trace.

A stronger version of this result is possible, where the input waveforms are arbitrary, but the proof is more complex.

VI. Related and further work

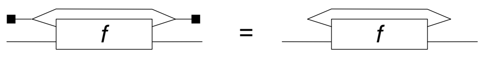
The structured style of presenting digital circuits and some of the equations (e.g. re-timing) have been prefigured by the pioneering work of Sheeran [18], further developed by Luk [19]. Our work represents a categorical systematisation of their approach. Otherwise, there is a dearth of syntactic reasoning methods for digital circuits, but semantic or simulation-based reasoning is extremely broadly studied and very useful.

One interesting point of contrast between our approach and more standard approaches is that we introduce the joining of two wires $\mathsf{j} : 2 \to 1$ as an explicit combinator, which is unusual in Boolean designs but technically essential for us in proving the fact that circuits with feedback have a product and therefore satisfy the iteration equation. Joins are also used in formalising waveforms. The natural interpretation of the wire-join is the value-join in the lattice of logical levels. Thus, for interesting circuits we require at least four values: $\perp$ (disconnected), h (logical high), l (logical low), $\top$ (illegal). Combining high and low produces an illegal value: $(\mathsf{h} \otimes \mathsf{l}) \cdot \mathsf{j} = \top$. Models of digital circuits requiring ternary logic (unknown, true, false) have been used for a long time [20], but we need the full lattice of values. Having a *join* combinator offers the additional benefit that it allows the representation of sub-logical circuits such as pass-through gates, or even transistors operating in saturation mode. This will be developed in forthcoming papers.

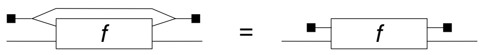
The ternary logic approach is also used by Mendler et. al. [10]. It would be interesting to study whether their semantic model is in fact an alternative (syntax-independent) concrete category of digital circuits, satisfying the same axiom and coherences as ours.

From a mathematical point of view our work is inspired by the deep connections between monoidal categories and diagrams [9] which have been also used in the modelling of quantum protocols [5] and signal-flow graphs [6]. Some contrasts are quite interesting. Unlike in quantum protocols, all digital circuits with no inputs and no outputs are equal whereas in quantum computing they correspond to *scalars*, which allow quantitative aspects to be expressed. Should we have taken a similar direction we could have included quantitative aspects such as power consumption in our formalism, but we would have lost the diagonal property. Obviously, two copies of a circuit will at least sometimes consume more power than one copy!

The signal-flow graph model in [6] is essentially linear and reversible, which is not the case for sequential circuits. Without elaborating the mathematics too much, a key difference between their model and ours can be illustrated by the following equality, involving the interaction between fork, join, and disconnected wires, as a trace can be created out of a fork and a join:



Of course, by comparison, in our setting the directionality of the wires never changes, so the correct equality is:



An even broader connection exists between this work and model-checking languages such as NuSMV[2]. One would expect that the waveforms produced by digital, finite-state circuits can be made to correspond to the languages of finite-state transducers. However, proving this connection is not as easy as it might seem because possible lack of productivity in circuits such as $true \cdot \mathrm{iter}(and)$, which can be unfolded any number of time without creating any sub-terms that could be reduced. Such a circuit should output just $\perp$, but this cannot be established without induction principles we do not provide yet. Once this connection is fully established it would indeed be reasonable to view our paper not only as a way to reason syntactically (or axiomatically) about circuits, but about finite state transducers in general. This remains as future work.

A more immediate development is moving from the algebraic reasoning which we expose here towards a fully automated rewriting system that can serve as an "operational semantics" for digital circuits. We can do this by exploiting connections between monoidal categories and graph-like structures, which we adapt to deal with traces and delays in an efficient fashion [21].

[2]http://nusmv.fbk.eu/

Acknowledgements: This work was motivated by initial discussions with Michael Mendler. Alex Smith and George Constantinides provided useful comments on preliminary work. Ross Duncan and Peter Selinger helped with technical advice on monoidal categories.

References

[1] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.
[2] M. Hennessy, *The Semantics of Programming Languages*. Wiley, 1990.
[3] R. Krebbers, X. Leroy, and F. Wiedijk, "Formal C semantics: Compcert and the C standard," in *5th Int. Conf. Thm. Prov.*, 2014, pp. 543–548.
[4] R. P. Kurshan and K. L. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 10, no. 11, pp. 1356–1371, 1991.
[5] S. Abramsky and B. Coecke, "A categorical semantics of quantum protocols," in *19th IEEE Symp. Logic in Comp. Sci.*, 2004, pp. 415–425.
[6] F. Bonchi, P. Sobocinski, and F. Zanasi, "Full abstraction for signal flow graphs," in *42nd Ann. ACM Symp. on Princ. of Prog. Lang.*, 2015, pp. 515–526.
[7] D. R. Ghica, "Diagrammatic reasoning for delay-insensitive asynchronous circuits," in *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, 2013, pp. 52–68.
[8] J. Baez and M. Stay, *Physics, topology, logic and computation: a Rosetta Stone*. Springer, 2010.
[9] P. Selinger, "A survey of graphical languages for monoidal categories," in *New structures for physics*. Springer, 2010, pp. 289–355.
[10] M. Mendler, T. R. Shiple, and G. Berry, "Constructive boolean circuits and the exactness of timed ternary simulation," *Form. Meth. Syst. Des.*, vol. 40, no. 3, pp. 283–329, 2012.
[11] S. Malik, "Analysis of cyclic combinational circuits," in *Proc. IEEE/ACM Int. Conf. on Comp. Aided Design*, 1993, pp. 618–625.
[12] S. Lack, "Composing PROPs," *Theory and App. of Categories*, vol. 13, no. 9, pp. 147–163, 2004.
[13] A. Joyal and R. Street, "The geometry of tensor calculus, i," *Adv. in Math.*, vol. 88, no. 1, pp. 55–112, 1991.
[14] G. Jones and M. Sheeran, "Timeless truths about sequential circuits," in *Concurrent Computations*. Springer, 1988, pp. 245–259.
[15] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1-6, pp. 5–35, 1991.
[16] A. Joyal, R. Street, and D. Verity, "Traced monoidal categories," in *Math. Proc. of the Cambridge Phil. Soc.*, vol. 119, no. 03. Cambridge Univ. Press, 1996, pp. 447–468.
[17] V. E. Căzănescu and G. Ştefănescu, "Towards a new algebraic foundation of flowchart scheme theory," *Fund. Inf.*, vol. 13, no. 2, pp. 171–210, 1990.
[18] M. Sheeran, "muFP, A language for VLSI design," in *LISP and Func. Prog.*, 1984, pp. 104–112.
[19] W. Luk, "Pipelining and transposing heterogeneous array designs," *J. of VLSI Sig. Proc. Sys.*, vol. 5, no. 1, pp. 7–20, 1993.
[20] M. A. Breuer, "A note on three-valued logic simulation," *IEEE Trans. Comp.*, vol. 21, no. 4, pp. 399–402, 1972.
[21] D. R. Ghica, A. Jung, and A. Lopez, "Diagrammatic operational semantics for digital circuits," submitted.

Equivalence Checking By Logic Relaxation

Eugene Goldberg
email: eu.goldberg@gmail.com

***Abstract*—We introduce a new framework for Equivalence Checking (EC) of Boolean circuits based on a general technique called Logic Relaxation (LoR). LoR is meant for checking if a propositional formula G has only "good" satisfying assignments specified by a design property. The essence of LoR is to relax G into a formula G^{rlx} and compute a set S that contains all assignments that satisfy G^{rlx} but do not satisfy G. If all bad satisfying assignments are in S, formula G can have only good ones and the design property in question holds. Set S is built by a procedure called partial quantifier elimination.**

The appeal of EC by LoR is twofold. First, it facilitates generation of powerful *inductive proofs*. Second, proving inequivalence comes down to checking the existence of some assignments satisfying G^{rlx} i.e. a *simpler* version of the original formula. We give experimental evidence that supports our approach.

1. Introduction

A. Motivation

Our motivation for this work is threefold. First, **Equivalence Checking (EC)** is a crucial part of hardware verification. Second, more efficient EC enables more powerful logic synthesis transformations and so strongly impacts design quality. Third, intuitively, there should exist robust and efficient EC methods meant for combinational circuits computing values in a "similar manner". Once discovered, these methods can be extended to EC of sequential circuits and even software.

B. Proving equivalence by induction

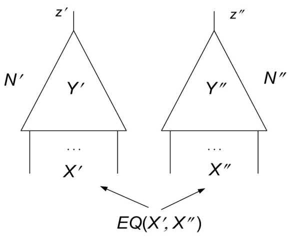


Fig. 1. Equivalence checking of N' and N''

Let $N'(X',Y',z')$ and $N''(X'',Y'',z'')$ be single-output circuits to be checked for equivalence. Here X' and Y' specify the sets of input and internal variables of N' respectively and z' specifies the output variable of N'. The same applies to X'',Y'',z'' of circuit N''. A traditional way to verify the equivalence of N' and N'' is to form a two-output circuit shown in Fig. 1 and check if $z' \neq z''$ for some input assignment $(\boldsymbol{x'},\boldsymbol{x''})$ where $\boldsymbol{x'}$=$\boldsymbol{x''}$. Here $\boldsymbol{x'}$ and $\boldsymbol{x''}$ are assignments to variables of X' and X'' respectively. (By saying that $\boldsymbol{p}$ is an assignment to a set of variables V, we will assume that $\boldsymbol{p}$ is a *complete* assignment unless otherwise stated. That is every variable of V is assigned a value in $\boldsymbol{p}$.)

Formula $EQ(X',X'')$ relating inputs of N' and N'' in Fig. 1 evaluates to 1 for assignments $\boldsymbol{x'}$ and $\boldsymbol{x''}$ to X' and X'' iff $\boldsymbol{x'}$=$\boldsymbol{x''}$. Usually, N' and N'' are just assumed to share the same set of input variables. In this paper, for the sake of convenience, we separate input variables of N' and N'' but assume that N' and N'' must be equivalent only for input assignments satisfying $EQ(X',X'')$.

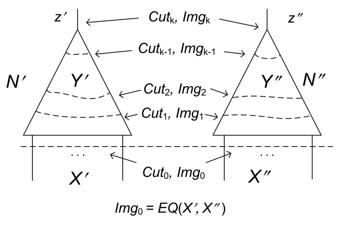


Fig. 2. An inductive proof of equivalence

A natural way to prove equivalence of N' and N'' is to build a sequence of cuts as shown in Fig. 2 and compute relations between cut points of N' and N'' [3], [11], [17]. A straightforward method of computing relations between cut points is to build formulas Img_i specifying **cut images**. The image of i-th cut is the set of all assignments to Cut_i that can be produced in N',N'' by input assignments satisfying $EQ(X',X'')$. Here $Cut_0 = X' \cup X''$, $Img_0 = EQ(X',X'')$ and $Cut_k = \{z',z''\}$. Circuits N' and N'' are equivalent iff $Img_k(z',z'') \rightarrow (z' \equiv z'')$. Formula Img_{i+1} can be derived from formula Img_i and formula specifying the gates located between i-th and $(i+1)$-th cuts. For that reason we will refer to the proofs employing a sequence of cuts as **proofs by induction**.

EC based on computing cut images is inefficient because the size of formulas Img_i is, in general, prohibitively large. In EC by logic relaxation, cut image formulas are replaced with formulas that, for structurally similar circuits, are dramatically simpler than the former.

C. EC by logic relaxation

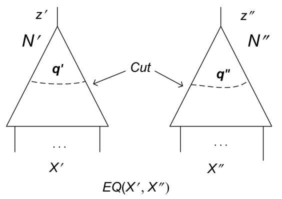


Fig. 3. A cut in N' and N''

Let Img_{cut} be a cut image formula built for the cut shown in Fig. 3. A cut assignment can be represented as $(\boldsymbol{q'},\boldsymbol{q''})$ where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are assignments to cut variables of N' and N'' respectively. We will say that formula Img_{cut} **excludes** cut assignment $(\boldsymbol{q'},\boldsymbol{q''})$ if the latter falsifies the former. The set of all cut assignments excluded by Img_{cut} can be represented as a union of sets $S^{cut}_{N'}$, $S^{cut}_{N''}$ and S_{rlx}. Assignment $(\boldsymbol{q'},\boldsymbol{q''})$ is in

- set $S^{cut}_{N'}$ if no input $\boldsymbol{x'}$ of N' can produce $\boldsymbol{q'}$
- set $S^{cut}_{N''}$ if no input $\boldsymbol{x''}$ of N'' can produce $\boldsymbol{q''}$
- set S_{rlx} if there is an input $(\boldsymbol{x'},\boldsymbol{x''})$, $\boldsymbol{x'} \neq \boldsymbol{x''}$ for which $(\boldsymbol{q'},\boldsymbol{q''})$ is produced but the latter cannot be produced if inputs are constrained by $EQ(X',X'')$.

Informally, set S_{rlx} specifies the cut assignments that can be produced only when inputs are *relaxed* i.e. are not constrained

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

by formula $EQ(X', X'')$.

The essence of EC by Logic Relaxation (LoR) is to replace computation of cut image formulas with that of so-called boundary formulas. A **boundary formula** H_{cut} is implied by Img_{cut} and excludes only a small subset of cut assignments excluded by Img_{cut}. Namely, only exclusion of assignments of S_{rlx} is mandatory for H_{cut}. If $(\boldsymbol{q'}, \boldsymbol{q''}) \in S^{cut}_{N'} \cup S^{cut}_{N''}$, the value of H_{cut} can be arbitrary. This means that H_{cut} depends on the *relation* between N' and N'' (specified by S_{rlx}) rather than their *individual functionality* (specified by $S^{cut}_{N'}$ and $S^{cut}_{N''}$.) We call formula H_{cut} boundary because it describes the difference i.e. a "boundary" between original and relaxed EC problems.

Computing a boundary formula H_{cut} for the cut $\{z', z''\}$ either immediately solves EC of N' and N'' or requires a few simple SAT-checks to finish it. Suppose $H_{cut}(z', z'')$ evaluates to 0 for assignment $\boldsymbol{z_1} = (z' = 0, z'' = 1)$ and assignment $\boldsymbol{z_2} = (z' = 1, z'' = 0)$. Then $\boldsymbol{z_1}$ and $\boldsymbol{z_2}$ cannot be produced when inputs are constrained by $EQ(X', X'')$ because $Img_{cut} \rightarrow H_{cut}$ entails $\overline{H}_{cut} \rightarrow \overline{Img}_{cut}$. So N' and N'' are *equivalent*. If $H_{cut}(z', z'')$ evaluates to 1, say, for $\boldsymbol{z_1}$ above, one needs to check if $(z' = 0, z'' = 1)$ can be produced when inputs are relaxed. This comes down to checking that N' is not constant 1 and N'' is not constant 0. If this is the case, i.e. N' and N'' can produce outputs 0 and 1 respectively, then assignment $\boldsymbol{z_1}$ can also be produced when inputs are constrained by $EQ(X', X'')$. (Otherwise, H_{cut} would evaluate to 0 under $\boldsymbol{z_1}$.) So N' and N'' are *inequivalent*.

D. The appeal of EC by LoR

The appeal of EC by LoR is threefold. First, boundary formulas are much smaller and easier to compute than cut image formulas. Generation of Img_{cut} requires solving the quantifier elimination problem whereas boundary formula H_{cut} can be found by partial quantifier elimination (PQE). In PQE, only a part of the formula is taken out of the scope of quantifiers. So PQE can be much more efficient than complete quantifier elimination.

Second, similarly to cut image formulas, boundary formulas can be computed by induction for a sequence of cuts starting with cut $X' \cup X''$ and ending with cut $\{z', z''\}$. So EC by LoR facilitates generation of inductive proofs. These proofs do not require the existence of particular relations like equivalence between internal points of N' and N''. So they are much more robust than inductive proofs generated in existing approaches (see, for example, [11], [12], [17]).

Third, the machinery of boundary formulas facilitates proving inequivalence. Let $F_{N'}(X', Y', z')$ and $F_{N''}(X'', Y'', z'')$ be formulas specifying N' and N'' respectively. We will say that a Boolean formula F_N specifies circuit N if every assignment satisfying F_N is a consistent assignment to variables of N and vice versa. (An assignment to variables of N is called *consistent* if, for every gate g of N, the value assigned to the output of g is implied by the values assigned to its input variables.) We will assume that all formulas mentioned in this paper are Boolean formulas in Conjunctive Normal Form (CNF) unless otherwise stated.

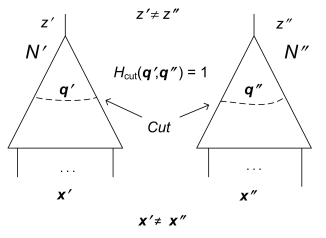


Fig. 4. Using a boundary formula for bug hunting

Circuits N' and N'' are inequivalent iff formula $EQ(X', X'') \wedge F_{N'} \wedge F_{N''} \wedge (z' \not\equiv z'')$ is satisfiable. Denote this formula as α. As we show in this paper, α is equisatisfiable with formula β equal to $H_{cut} \wedge F_{N'} \wedge F_{N''} \wedge (z' \not\equiv z'')$. Here H_{cut} is a boundary formula computed with respect to a cut (see Fig. 4.) In general, formula β is easier to satisfy than α for the following reason. Let $\boldsymbol{p}$ be an assignment satisfying formula β . Let $\boldsymbol{x'}$ and $\boldsymbol{x''}$ be the assignments to variables of X' and X'' respectively specified by $\boldsymbol{p}$. Since variables of X' and X'' are not constrained by $EQ(X', X'')$ in formula β, in general, $\boldsymbol{x'} \neq \boldsymbol{x''}$ and so $\boldsymbol{p}$ does not satisfy α. Hence, neither $\boldsymbol{x'}$ nor $\boldsymbol{x''}$ are a counterexample. They are just inputs producing cut assignments $\boldsymbol{q'}$ and $\boldsymbol{q''}$ (see Fig. 4) such that a) $H_{cut}(\boldsymbol{q'}, \boldsymbol{q''}) = 1$ and b) N' and N'' produce different outputs under cut assignment $(\boldsymbol{q'},\boldsymbol{q''})$. To turn $\boldsymbol{p}$ into an assignment satisfying α one has to do *extra work*. Namely, one has to find assignments $\boldsymbol{x'}$ and $\boldsymbol{x''}$ to X' and X'' that are *equal to each other* and under which N' and N'' produce cut assignments $\boldsymbol{q'}$ and $\boldsymbol{q''}$ above. Then $\boldsymbol{x'}$ and $\boldsymbol{x''}$ specify a counterexample. So the equisatisfiability of α and β allows one to prove N' and N'' inequivalent (by showing that β is satisfiable) without providing a counterexample.

E. Contributions and structure of the paper

Our contributions are as follows. First, we present a generic method of EC based on LoR. This method is formulated in terms of a new technique called PQE that is a "light" version of quantifier elimination. Showing the potential of PQE for building new verification algorithms is our second contribution. Third, we provide a theoretical proof that boundary formulas computed in EC by LoR are small for a broad class of structurally similar circuits. Fourth, we give experimental evidence in support of EC by LoR.

The structure of this paper is as follows. In Section 2, we show the correctness of EC by LoR and relate the latter to PQE. Boundary formulas are discussed in Section 3. Section 4 presents an algorithm of EC by LoR. Section 5 describes how one can apply EC by LoR if the power of a PQE solver is not sufficient to compute boundary formulas precisely. Section 6 provides experimental evidence in favor of our approach. In Section 7, some background is given. We make conclusions in Section 8.

2. Equivalence Checking By LoR And PQE

In this section, we prove the correctness of Equivalence Checking (EC) by Logic Relaxation (LoR) and relate the latter to Partial Quantifier Elimination (PQE).

A. Complete and partial quantifier elimination

In this paper, by a quantified formula we mean one with *existential* quantifiers. Given a quantified formula

$\exists W[A(V,W)]$, the problem of *quantifier elimination* is to find a quantifier-free formula $A^*(V)$ such that $A^* \equiv \exists W[A]$. Given a quantified formula $\exists W[A(V,W) \wedge B(V,W)]$, the problem of **Partial Quantifier Elimination** (**PQE**) is to find a quantifier-free formula $A^*(V)$ such that $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. Note that formula B remains quantified (hence the name *partial* quantifier elimination). We will say that formula A^* is obtained by **taking A out of the scope of quantifiers** in $\exists W[A \wedge B]$. Importantly, there is a strong relation between PQE and the notion of *redundancy* of a subformula in a quantified formula. In particular, solving the PQE problem above comes down to finding $A^*(V)$ implied by $A \wedge B$ that makes A redundant in $A^* \wedge \exists W[A \wedge B]$. Indeed, in this case, $\exists W[A \wedge B] \equiv A^* \wedge \exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$.

Importantly, redundancy in a quantified formula is *much more powerful* than that in a quantifier-free formula [9]. For instance, if formula $F(V)$ is satisfiable, every clause of F is redundant in formula $\exists V[F]$. (A **clause** is a disjunction of literals. We will use the notions of a CNF formula $C_1 \wedge .. \wedge C_p$ and the set of clauses $\{C_1,\dots,C_p\}$ interchangeably.) On the other hand, a clause C is redundant in a *quantifier-free* formula F *only* if C is implied by $F \setminus \{C\}$.

Let $G(V)$ be a formula implied by B. Then $\exists W[A \wedge B] \equiv A^* \wedge G \wedge \exists W[B]$ entails $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. In other words, clauses implied by the formula that remains quantified are *noise* and can be removed from a solution to the PQE problem. So when building A^* by resolution it is sufficient to use only the resolvents that are descendants of clauses of A. For that reason, in the case formula A is much smaller than B, PQE can be dramatically faster than complete quantifier elimination. Describing how PQE is solved is beyond the scope of this paper. A brief discussion of a PQE algorithm and recall of the necessary background is given in the technical report [7] presenting a complete version of this paper. The relevant results are described in [8], [9], [10] in more detail.

B. Proving equivalence/inequivalence by LoR

Proposition 1 below shows how one proves[1] equivalence/inequivalence of circuits by LoR. Let formula G denote $EQ \wedge F_{N'} \wedge F_{N''}$ and formula G^{rlx} denote $F_{N'} \wedge F_{N''}$. Recall from Subsection 1-D that $F_{N'}(X',Y',z')$ and $F_{N''}(X'',Y'',z'')$ specify circuits N' and N'' respectively. Formula $EQ(\boldsymbol{x'},\boldsymbol{x''})$ evaluates to 1 iff $\boldsymbol{x'}$=$\boldsymbol{x''}$ where $\boldsymbol{x'}$ and $\boldsymbol{x''}$ are assignments to variables of X' and X'' respectively.

Proposition 1: Let $H(z',z'')$ be a formula such that $\exists W[EQ \wedge G^{rlx}] \equiv H \wedge \exists W[G^{rlx}]$ where $W = X' \cup X'' \cup Y' \cup Y''$. Then formula $G \wedge (z' \not\equiv z'')$ is equisatisfiable with $H \wedge G^{rlx} \wedge (z' \not\equiv z'')$.

Note that finding formula $H(z',z'')$ of Proposition 1 reduces to taking formula EQ out of the scope of quantifiers i.e. to solving the PQE problem. Proposition 1 implies that proving *inequivalence* of N' and N'' comes down to showing that formula G^{rlx} is satisfiable under assignment $(z'=b', z''=b'')$ (where $b',b'' \in \{0,1\}$) such that $b' \neq b''$ and $H(b',b'')=1$.

[1]The proofs of propositions are given in [7].

Recall that the input variables of N' and N'' are independent of each other in formula G^{rlx}. Hence the only situation where G^{rlx} is unsatisfiable under $(z'=b', z''=b'')$ is when N' is constant $\overline{b'}$ and/or N'' is constant $\overline{b''}$. So the corollary below holds.

Corollary 1: If neither N' nor N'' are constants, they are equivalent iff $H(1,0)=H(0,1)=0$.

Reducing EC to an instance of PQE also provides valuable information when proving *equivalence* of N' and N''. Formula G^{rlx} remains quantified in $\exists W[EQ \wedge G^{rlx}] \equiv H \wedge \exists W[G^{rlx}]$. This means that to obtain formula H, it suffices to generate only resolvents that are descendants of clauses of EQ. The clauses obtained by resolving solely clauses of G^{rlx} are just "noise" (see Subsection 2-A). This observation is the basis of our algorithm for generating proofs of equivalence by induction.

3. Boundary Formulas

In this section, we discuss boundary formulas, a key notion of EC by LoR. Subsection 3-A explains the semantics of boundary formulas. Subsection 3-B discusses the size of boundary formulas. In Subsection 3-C, we describe how boundary formulas are built.

A. Definition and some properties of boundary formulas

Let M be the subcircuit consisting of the gates of N', N'' located below a cut as shown in Fig. 5. As usual, G denotes $EQ(X',X'') \wedge F_{N'} \wedge F_{N''}$ and G^{rlx} does $F_{N'} \wedge F_{N''}$.

Definition 1: Let formula H_{cut} depend only on variables of a cut. Let $\boldsymbol{q}$ be an assignment to the variables of this cut. Formula H_{cut} is called **boundary** if[2]

a) $G \rightarrow H_{cut}$ holds and
b) for every $\boldsymbol{q}$ that can be extended to satisfy G^{rlx} but cannot be extended to satisfy G, the value of $H_{cut}(\boldsymbol{q})$ is 0.

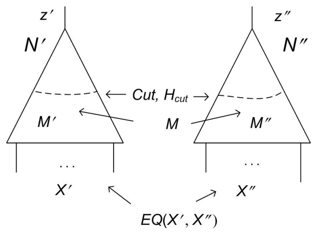


Fig. 5. Building boundary formula H_{cut}

A cut assignment $\boldsymbol{q}$ can be represented as ($\boldsymbol{q'}$,$\boldsymbol{q''}$) where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are assignments to cut variables of N' and N'' respectively. Note that Definition 1 does not specify the value of $H_{cut}(\boldsymbol{q})$ if $\boldsymbol{q}$ *cannot* be extended to satisfy G^{rlx} (and hence G). This means that H_{cut} does not have to exclude ($\boldsymbol{q'}$,$\boldsymbol{q''}$) if, say, no input $\boldsymbol{x'}$ of N' produces $\boldsymbol{q'}$. This means that H_{cut} *does not depend* on the individual complexity of N' and N''.

Formula $EQ(X',X'')$ and formula $H(z',z'')$ of Proposition 1 are actually boundary formulas with respect to cuts $X' \cup X''$ and $\{z',z''\}$ respectively. We will refer to $H(z',z'')$

[2]Since formula $(z' \not\equiv z'')$ constraining the outputs of N' and N'' is not a part of formulas G^{rlx} and G, a boundary formula of Definition 1 is not "property driven". This can be fixed by making a boundary formula specify the difference between $G^{rlx} \wedge (z' \not\equiv z'')$ and $G \wedge (z' \not\equiv z'')$ rather than between G^{rlx} and G. In this paper, we explore only boundary formulas of Definition 1 leaving property-driven ones for future research.

as an **output boundary formula**. Proposition 2 below reduces building H_{cut} to PQE.

Proposition 2: Let H_{cut} be a formula depending only on variables of a cut. Let H_{cut} satisfy $\exists W[EQ \wedge F_M] \equiv H_{cut} \wedge \exists W[F_M]$. Here W is the set of variables of F_M minus those of the cut. Then H_{cut} is a boundary formula.

Proposition 3 below extends Proposition 1 to an arbitrary boundary formula.

Proposition 3: Let H_{cut} be a boundary formula with respect to a cut. Then $G \wedge (z' \not\equiv z'')$ is equisatisfiable with $H_{cut} \wedge G^{rlx} \wedge (z' \not\equiv z'')$.

B. Size of boundary formulas

The proposition below estimates the size of a boundary formula computed for a cut if every cut variable of N' can be expressed as a function of cut variables of N''. If the number of arguments in the functions relating cut points of N' and N'' is small, these circuits can be viewed as structurally similar.

Proposition 4: Let circuits M' and M'' consist of the gates of N' and N'' located below a cut as shown in Fig. 5. Let Cut', Cut'' specify the outputs of M' and M'' respectively. Assume that for every variable v'_i of Cut' there is a set $S(v'_i) = \{v''_{i_1},\dots,v''_{i_p}\}$ of variables of Cut'' that have the following property. Knowing the values of variables of $S(v'_i)$ produced in N'' under input $\boldsymbol{x}$ one can determine the value of v'_i of N' under the same input $\boldsymbol{x}$. We assume here that $S(v'_i)$ has this property for every possible input $\boldsymbol{x}$. Let $Max(S(v'_i))$ be the size of the largest $S(v'_i)$ over variables of Cut'. Then there is a boundary formula H_{cut} where every clause has at most $Max(S(v'_i)) + 1$ literals.

Proposition 4 gives an example of boundary formulas whose complexity is exponential in the value of $Max(S(v'_i)) + 1$ rather than the cut size. This means that these boundary formulas depend only on *similarity* of N' and N'' and do not depend on how complex N' and N'' are.

Corollary 2: Let circuits M' and M'' of Fig. 5 be functionally equivalent. Then for every variable $v' \in Cut'$ there is a set $S(v') = \{v''\}$ where v'' is the variable of Cut'' that is functionally equivalent to v'. In this case, formula $EQ(Cut', Cut'')$ stating equivalence of corresponding output variables of M' and M'' is a boundary formula for the cut in question. Note that $v' \equiv v''$ can be represented as a CNF formula $(v' \vee \overline{v''}) \wedge (\overline{v'} \vee v'')$. So $EQ(Cut', Cut'')$ can be represented by $2*p$ two-literal clauses where $p = |Cut'| = |Cut''|$.

C. Computing Boundary Formulas

The key part of EC by LoR is to compute an output boundary formula $H(z', z'')$. In this subsection, we show how to build formula H *inductively* by constructing a sequence of boundary formulas $H_0,\dots,H_k$ computed with respect to cuts $Cut_0,\dots,Cut_k$ of N' and N'' (see Fig. 2). We assume that $Cut_0 = X' \cup X''$ and $Cut_k = \{z', z''\}$ (i.e. $H = H_k$) and $Cut_i \cap Cut_j = \emptyset$ if $i \neq j$.

Boundary formula H_0 is set to $EQ(X', X'')$ whereas formula H_i, $i > 0$ is computed from H_{i-1} as follows. Let M_i be the circuit consisting of the gates of N' and N'' located below i-th cut. Let F_{M_i} be the subformula of G^{rlx} specifying M_i. Let W_i consist of all the variables of F_{M_i} minus those of Cut_i. Formula H_i is built to satisfy $\exists W_i[H_{i-1} \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$ and so make the previous boundary formula H_{i-1} *redundant* in $H_i \wedge \exists W_i[H_{i-1} \wedge F_{M_i}]$. The fact that $H_1,\dots,H_k$ are indeed boundary formulas follows from Proposition 5.

```
$EC\_LoR(N', N'')$ {
1  $(N', N'') := Bufferize(N', N'')$;
2  $Cut_0 = X' \cup X''$;
3  $Cut_1,..,Cut_{k-1} := BldIntermCuts(N', N'')$;
4  $Cut_k := \{z', z''\}$;
5  $H_0 := EQ(X', X'')$;
– – – – – – – – – – – – – – – – – –
6  $for(i := 1; i \leq k; i{+}{+})$ {
7      $F_{M_i} := SubForm(G^{rlx}, Cut_i)$;
8      $W_i := Vars(F_{M_i}) \setminus Vars(Cut_i)$;
9      $H_i := PQE(\exists W_i[H_{i-1} \wedge F_{M_i}])$; }
– – – – – – – – – – – – – – – – – –
10 if $(H_k(0,1) = 1)$
11     if $(Sat(G^{rlx} \wedge \overline{z'} \wedge z''))$ return(*No*);
12 if $(H_k(1,0) = 1)$
13     if $(Sat(G^{rlx} \wedge z' \wedge \overline{z''}))$ return(*No*);
14 return(*Yes*); }
```

Fig. 6. EC by LoR

Proposition 5: Let W_i where $i > 0$ be the set of variables of F_{M_i} minus those of Cut_i. Let H_{i-1}, $i > 1$ satisfy $\exists W_{i-1}[H_0 \wedge F_{M_{i-1}}] \equiv H_{i-1} \wedge \exists W_{i-1}[F_{M_{i-1}}]$. (So H_{i-1} is a boundary formula due to Proposition 2.) Let $\exists W_i[H_{i-1} \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$ hold. Then $\exists W_i[H_0 \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$ holds and so, H_i is a boundary formula too.

4. Algorithm of EC by LoR

In this section, we introduce an algorithm called *EC_LoR* that checks for equivalence two single-output circuits N' and N''. The pseudo-code of *EC_LoR* is given in Fig. 6. *EC_LoR* builds a sequence of boundary formulas $H_0,\dots,H_k$ as described in Subsection 3-C. Here H_0 equals $EQ(X', X'')$ and $H_k(z', z'')$ is an output boundary formula. Then, according to Proposition 1, *EC_LoR* checks the satisfiability of formula $H_k \wedge G^{rlx} \wedge (z' \not\equiv z'')$ where $G^{rlx} = F_{N'} \wedge F_{N''}$.

EC_LoR consists of three parts separated by the dotted lines in Figure 6. *EC_LoR* starts the first part (lines 1-5) by calling procedure *Bufferize*. This procedure eliminates non-local connections of N' and N'' i.e. those that span more than two consecutive topological levels. (The *topological level* of a gate g of a circuit K is the longest path from an input of K to g measured in the number of gates on this path.) The presence of non-local connections makes it hard to find cuts that do not overlap. To avoid this problem, procedure *Bufferize* replaces every non-local connection spanning d topological levels ($d > 2$) with a chain of $d-2$ buffers. (A more detailed discussion of this topic is given in [7].) Then *EC_LoR* sets the initial cut to $X' \cup X''$, computes the intermediate cuts (line 3), sets the final cut to $\{z', z''\}$ and formula H_0 to $EQ(X', X'')$.

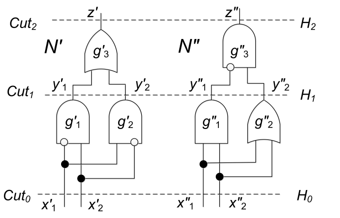


Fig. 7. An example of EC by LoR

Boundary formulas H_i, $1 \leq i \leq k$ are computed in the second part (lines 6-9) that consists of a *for* loop. In the third part (lines 10-14), EC_LoR uses the output boundary formula $H_k(z',z'')$ computed in the second part to decide whether N',N'' are equivalent. If $H_k(b',b'') = 1$ where $b' \neq b''$ and G^{rlx} is satisfiable under $z' = b', z'' = b''$, then N',N'' are inequivalent. Otherwise, they are equivalent (line 14).

Boundary formulas are computed in the *for* loop as follows. Formula H_i, $i > 0$ is obtained by taking H_{i-1} out of the scope of quantifiers in $\exists W_i[H_{i-1} \wedge F_{M_i}]$ (line 9) i.e. $\exists W_i[H_{i-1} \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$. Here F_{M_i} is the formula specifying the gates of N' and N'' below i-th cut and W_i consists of all the variables of F_{M_i} but cut variables.

Example 1: Let us explain the operation of EC_LoR by the example of Fig. 7 showing two different implementations of function $XOR(x_1,x_2)$. EC_LoR starts by executing the first part specified by lines 1-5 of Fig. 6. Since circuits N' and N'' do not have no-local connections, no buffers are inserted. EC_LoR sets the initial cut Cut_0 to $\{X',X''\}$ where $X' = \{x'_1,x'_2\}$, $X'' = \{x''_1,x''_2\}$, generates an intermediate cut $Cut_1 = \{y'_1,y'_2,y''_1,y''_2\}$ and the final cut $Cut_2 = \{z',z''\}$. EC_LoR concludes the first part by setting H_0 to $EQ(X',X'')$ i.e. to $(x'_1 \equiv x''_1) \wedge (x'_2 \equiv x''_2)$.

In the second part (lines 6-9 of Fig. 6), EC_LoR computes boundary formulas for Cut_1 and Cut_2. A boundary formula for Cut_1 is obtained by taking H_0 out of the scope of quantifiers in $\exists W_1[H_0 \wedge F_{M_1}]$ i.e. by finding formula H_1 such that $\exists W_1[H_0 \wedge F_{M_1}] \equiv H_1 \wedge \exists W_1[F_{M_1}]$. Here F_{M_1} specifies the gates located below cut Cut_1 and so $F_{M_1} = F_{g'_1} \wedge F_{g'_2} \wedge F_{g''_1} \wedge F_{g''_2}$ where F_g specifies gate g. For instance, $F_{g'_1} = (x'_1 \vee \overline{x'}_2 \vee y''_1) \wedge (\overline{x'}_1 \vee \overline{y'}_1) \wedge (x'_2 \vee \overline{y'}_1)$. Set W_1 consists of all variables of F_{M_1} minus the variables of Cut_1 i.e. $W_1 = X' \cup X''$. Formula H_1 obtained by a PQE-solver implementing the algorithm of [10] consists of five clauses: $C_1 = y'_1 \vee y'_2 \vee y''_1 \vee \overline{y''}_2$, $C_2 = \overline{y'}_1 \vee \overline{y''}_1$, $C_3 = \overline{y'}_1 \vee y''_2$, $C_4 = \overline{y'}_2 \vee y''_2$, $C_5 = \overline{y'}_2 \vee \overline{y''}_1$.

A boundary formula for cut Cut_2 is obtained by taking H_1 out of the scope of quantifiers in $\exists W_2[H_1 \wedge F_{M_2}]$. Here F_{M_2} specifies the gates located below cut Cut_2, so $F_{M_2} = F_{M_1} \wedge F_{g'_3} \wedge F_{g''_3}$. Set W_2 consists of the variables of F_{M_2} minus those of Cut_2, so $W_2 = W_1 \cup \{y'_1,y'_2,y''_1,y''_2\}$. Formula H_2 obtained by the PQE-solver mentioned above is equal to $(z' \vee \overline{z''}) \wedge (\overline{z'} \vee z'')$. This means that $H(0,1) = H(1,0) = 0$. So after executing its last part (lines 10-14 of Fig. 6), EC_LoR reports that N' and N'' are equivalent.

Let us take a closer look at formula H_1. On one hand, as a boundary formula, H_1 excludes every assignment to Cut_1 that can be produced by applying an input $(\boldsymbol{x'},\boldsymbol{x''})$ where $\boldsymbol{x'} \neq \boldsymbol{x''}$ but cannot be produced if input assignments are constrained by $EQ(X',X'')$. For instance, input $(x'_1 = 0, x'_2 = 0, x''_1 = 0, x''_2 = 1)$ produces cut assignment $y'_1 = 0, y'_2 = 0, y''_1 = 0, y''_2 = 1)$ that cannot be produced by an input assignment $(\boldsymbol{x'},\boldsymbol{x''})$ where $\boldsymbol{x'}=\boldsymbol{x''}$. This cut assignment falsifies clause $C_1 = y'_1 \vee y'_2 \vee y''_1 \vee \overline{y''}_2$ of H_1. On the other hand, H_1 is simpler that formula Img_1 that excludes *every* assignment to Cut_1 that cannot be produced by an input $(\boldsymbol{x'},\boldsymbol{x''})$ where $\boldsymbol{x'}=\boldsymbol{x''}$. Formula Img_1 is logically equivalent to $\exists W_1[H_0 \wedge F_{M_1}]$ i.e. it is obtained from the latter by *complete* quantifier elimination. By applying our program of [9], we obtain formula Img_1 equal to $H_1 \wedge C_6 \wedge C_7$ where $C_6 = \overline{y'_1} \vee \overline{y'_2}$, $C_7 = \overline{y''}_1 \wedge y''_2$. Note that C_6, C_7 do not relate variables of N' and N''. Instead, they exclude some cut assignments that cannot be produced in N' or N''. For instance, clause C_6 excludes cut assignment $(y'_1 = 1, y'_2 = 1)$ that cannot be produced in N'.

5. Computing Boundary Formulas By Current PQE Solvers

To obtain boundary formula H_i, one needs to take H_{i-1} out of the scope of quantifiers in formula $\exists W_i[H_{i-1} \wedge F_{M_i}]$. The size of the latter grows with i due to formula F_{M_i}. So a PQE solver that computes H_i must have good scalability. On the other hand, the algorithm of [10] does not scale well yet. The main problem here is that learned information is not re-used in contrast to SAT-solvers effectively re-using learned clauses. Fixing this problem requires some time because bookkeeping of a PQE algorithm is more complex than that of a SAT-solver. (In more detail, this topic is discussed in [7].) In this section, we describe two methods of adapting EC by LoR to a PQE-solver that is not efficient enough to compute every boundary formula *precisely*. Both methods are illustrated experimentally in Section 6.

One way to reduce the complexity of computing H_i is to use only a subset of F_{M_i}. For instance, one can discard the clauses of F_{M_i} specifying the gates located between cuts Cut_0 and Cut_p, $0 < p < i$. In this case, boundary formula H_i is computed *approximately*. A downside of this is that condition b) of Definition 1 does not hold anymore and so EC by LoR becomes *incomplete*. Namely, if $H_k(b',b'') = 1$ where $b' \neq b''$ and H_k is an output boundary formula, the fact that G^{rlx} is satisfiable under $z' = b', z'' = b''$ does not mean that N' and N'' are inequivalent. Nevertheless, even EC by LoR with approximate computation of boundary formulas can be a powerful tool for proving N' and N'' *equivalent* for the following reason. If $H_k(1,0) = H_k(0,1) = 0$, circuits N' and N'' are proved equivalent even if intermediate formulas H_i are built approximately. Importantly, computing boundary formulas *inductively* still provides a powerful way to *structure* a proof of equivalence. Formula H_i (i.e. a "sufficient" set of clauses relating variables of i-th cut) is still obtained by taking H_{i-1} out of the scope of quantifiers in $\exists W_i[H_{i-1} \wedge F_{M_i}]$. Only now formula F_{M_i} is simplified by discarding some clauses.

Another way to adapt EC by LoR to a PQE solver that is not efficient enough to compute *every* boundary formula precisely is as follows. Suppose that the power of a PQE solver

TABLE I
Computing cut image and boundary formulas. Time limit = 1 hour

#bits	#quan. vars	#free vars	cut image formula (QE)		boundary formula (PQE)	
			result size	(s.)	result size	(s.)
8	32	84	3,142	4.0	**242**	**0.1**
9	36	104	4,937	13	**273**	**0.2**
10	40	126	7,243	51	**407**	**0.3**
11	44	150	9,272	147	**532**	**0.5**
12	48	176	14,731	497	**576**	**0.6**
13	52	206	19,261	1,299	**674**	**0.9**
14	56	234	$*$	$*$	**971**	**1.5**
15	60	266	$*$	$*$	**1,218**	**2.0**
16	64	300	$*$	$*$	**1,411**	**3.0**

is sufficient to build *one* intermediate boundary formula H_i precisely. From Proposition 3 it follows that formula α equal to $G \wedge (z' \not\equiv z'')$ is equisatisfiable with formula β equal to $H_{cut} \wedge G^{rlx} \wedge (z' \not\equiv z'')$. So, to show that N' and N'' are inequivalent it is sufficient to find an assignment satisfying β. As we argued in Subsection 1-D, finding such an assignment for β is easier than for α.

6. Experiments

In the experiments, we employed the PQE algorithm published in [10] in 2014. We will refer to this algorithm as **PQE-14**. As we mentioned in Section 5, PQE-14 does not scale well yet. So building a full-fledged equivalence checker based on *EC_LoR* would mean simultaneously designing a new EC algorithm and a new PQE solver. The latter is beyond the scope of our paper. On the other hand, PQE-14 is efficient enough to make a few important points experimentally. In the experiments described in this section, we used a new implementation of PQE-14 [6].

The experiment of Subsection 6-A compares computing a cut image formula and a boundary formula. Recall that a cut image formula is satisfied by a cut assignment iff the latter can be produced in N' and N'' by some input satisfying $EQ(X', X'')$. This experiment also contrasts complete quantifier elimination (employed to compute a cut image formula) with PQE. In Subsection 6-B, we apply *EC_LoR* to a non-trivial instance of equivalence checking that is hard for *ABC*, a high-quality synthesis and verification tool [20]. In Subsection 6-C, we show that computing boundary formulas is beneficial for proving inequivalence.

In the experiments, circuits N' and N'' to be checked for equivalence were derived from a circuit computing a median output bit of an s-bit multiplier. We will refer to this circuit as Mlp_s. Our motivation here is as follows. In many cases, the equivalence of circuits with simple topology and low fanout values can be efficiently checked by a general-purpose SAT-solver. This is not true for circuits involving multipliers. In all experiments, circuits N' and N'' were bufferized to get rid of long connections (see Section 4).

A. Image computation versus building boundary formulas

In the experiment of this subsection, we compared computation of a boundary formula H_{cut} and a cut image formula Img_{cut}. We used two identical copies of circuit Mlp_s as circuits N' and N''. As a cut of N', N'' we picked the set of variables of the first topological level (every variable of this level specifies the output of a gate fed by input variables of N' or N''). Formula Img_{cut} is logically equivalent to $\exists W[EQ(X', X'') \wedge F_M]$ where $W = X' \cup X''$ and formula F_M specifies the gates of the first topological level of N' and N''. So, computing Img_{cut} comes down to solving the quantifier elimination problem. Computing a boundary formula reduces to finding H_{cut} such that $\exists W[EQ \wedge F_M] \equiv H_{cut} \wedge \exists W[F_M]$ i.e. solving the PQE problem.

The results of the experiment are given in Table I. Abbreviation QE stands for Quantifier Elimination. The value of s in Mlp_s is shown in the first column. The next two columns give the number of quantified and free variables in $\exists W[EQ \wedge F_M]$. To compute formula Img_{cut} we used our quantifier elimination program presented in [9]. Formula H_{cut} was generated by PQE-14. When computing image formula Img_{cut} and boundary formula H_{cut} we recorded the size of the result (as the number of clauses) and the run time in seconds. As Table I shows, formulas H_{cut} are much smaller than Img_{cut} and take much less time to compute.

B. Proving equivalence by LoR

In this subsection, we ran an implementation of *EC_LoR* introduced in Section 4 on circuits N' and N'' shown in Fig. 8. (The idea of this EC example was suggested by Vigyan Singhal [19].) These circuits were derived from Mlp_s by adding one extra input h. Either circuit produces the same output as Mlp_s when $h = 1$ and output 0 if $h = 0$. So N' and N'' are logically equivalent. Note that the value of every internal variable of N' depends on h whereas this is not the case for N''. So N' and N'' have no functionally equivalent internal variables. On the other hand, N' and N'' satisfy the notion of structural similarity introduced in Subsection 3-B to prove Proposition 4. Namely, the value of every internal variable v' of N' is specified by that of h'' and some variable v'' of N''. (So, in this case, for every internal variable v' of N' there is a set $S(v')$ defined in Proposition 4 consisting of only two variables of N''.) In particular, if v' is an internal variable of Mlp'_s, then v'' is the corresponding variable of Mlp''_s. Indeed, if $h'' = 1$, then v' takes the same value as v''. If $h'' = 0$, then v' is a constant (in the implementation of Mlp_s we used in the experiments). The objective of the experiment below was to show that *EC_LoR* can check for equivalence structurally similar circuits that have no functionally equivalent internal points.

Cuts $Cut_0, \dots, Cut_k$ used by *EC_LoR* were generated according to topological levels. That is every variable of Cut_i specified the output of a gate of i-th topological level. Since N' and N'' were bufferized, $Cut_i \cap Cut_j = \emptyset$ if $i \neq j$. The version of *EC_LoR* we used in the experiment was slightly different from the one described in Fig. 6. We will refer to this version as *EC_LoR**. (A detailed description of *EC_LoR** is given in [7]). The main change was that boundary formulas were computed in *EC_LoR** *approximately*. That is formula

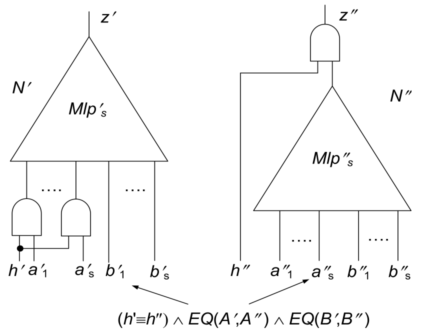


Fig. 8. Equivalence checking of N' and N'' derived from Mlp_s

TABLE II
EC of N' and N'' derived from Mlp_s. Time limit = 6 hours

#bits	#vars	#clauses	#cuts	*EC_LoR** (s.)	*ABC* (s.)
10	2,844	6,907	37	**4.5**	10
11	3,708	8,932	41	**7.1**	38
12	4,726	11,297	45	**11**	142
13	5,910	14,026	49	**16**	757
14	7,272	17,143	53	**25**	3,667
15	8,824	20,672	57	**40**	11,237
16	10,578	24,637	61	**70**	$> 21{,}600$

H_i was obtained by taking H_{i-1} out of the scope of quantifiers in formula $\exists W_i[H_{i-1} \wedge F_{M_i}]$ where only a subset of clauses of F_{M_i} was used. Nevertheless, *EC_LoR** was able to compute an output boundary formula $H_k(z', z'')$ that implied $(z' \equiv z'')$ thus proving that N' and N'' were equivalent.

One more difference between *EC_LoR* and *EC_LoR** was that the latter built formula H_i by solving a sequence of small PQE problems rather than one large PQE problem (line 9 of Fig. 6). Each PQE problem of this sequence was meant to find clauses relating the output of a gate g' of N' of Cut_i to its "siblings" of N'' that are in Cut_i. A gate g'' of N'' was considered a sibling of g' if inputs of g' and g'' were related by a clause of H_{i-1}. After solving the sequence of small PQE problems above, *EC_LoR** checked a *cut termination condition*. That is *EC_LoR** verified that $\exists W_i[H_{i-1} \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$ and so the set of clauses accumulated in H_i was indeed a boundary formula for i-th cut.

In Table II, we compare *EC_LoR** with *ABC* [20]. The first column gives the value of s of Mlp_s used in N' and N''. The next two columns show the size of formulas $EQ(X', X'') \wedge F_{N'} \wedge F_{N''} \wedge (z' \not\equiv z'')$ specifying equivalence checking of N' and N'' to which *EC_LoR** was applied. (N' and N'' were fed into *ABC* as circuits in the BLIF format.) Here $X = \{h, a_1, \ldots, a_s, b_1, \ldots, b_s\}$ denotes the set of input variables of circuits N' and N''. The fourth column shows the number of topological levels in circuits N' and N'' and so the number of cuts used by *EC_LoR**. The last two columns give the run time of *EC_LoR** and *ABC*.

The results of Table II show that equivalence checking of N' and N'' derived from Mlp_s was hard for *ABC*. On the other hand, *EC_LoR** managed to solve all instances in a reasonable time. Most of the run time of *EC_LoR** was taken by PQE-14 when checking cut termination conditions mentioned above. So, PQE-14 was also the reason why the run time of *EC_LoR** grew quickly with the size of Mlp_s. The performance of *EC_LoR** with a more efficient PQE-solver should have a weaker dependency on the value of s.

C. Using boundary formulas for proving inequivalence

In the experiment of this subsection, we checked for equivalence circuits N' and N'' that were correct and buggy versions of Mlp_{16} respectively. Since *EC_LoR** described in the previous subsection computes boundary formulas approximately, one cannot directly apply it to prove inequivalence of N' and N''. In this subsection, we show that the precise computation of even *one* boundary formula corresponding to an intermediate cut can be quite useful for proving inequivalence. Let α and β denote formulas $EQ(X', X'') \wedge F_{N'} \wedge F_{N''} \wedge (z' \equiv z'')$ and $H_i \wedge F_{N'} \wedge F_{N''} \wedge (z' \equiv z'')$ respectively. Here H_i is a boundary formula precisely computed for the cut of N' and N'' consisting of the gates with topological level equal to i. According to Proposition 3, α and β are equisatisfiable. Proving N' and N'' inequivalent comes down to showing that β is satisfiable. Intuitively, checking the satisfiability of β the easier, the larger the value of i and so the closer the cut to the outputs of N' and N''. In the experiment below, we show that computing boundary formula H_i makes proving inequivalence of N' and N'' easier even for a cut with a small value of i.

TABLE III
Sat-solving of formulas α and β by Minisat. Time limit = 600 s.

formula type	#solv-ed	total time (s.)	median time (s.)
α	95	$> 3{,}490$	4.2
β	**100**	**1,030**	**1.0**

Bugs were introduced into circuit N'' *above* the cut (so N' and N'' were identical *below* the cut). Let M'_i and M''_i denote the subcircuits of N' and N'' consisting of the gates located below the cut (like circuits M' and M'' in Fig. 5). Since M'_i and M''_i are identical they are also functionally equivalent. Then Corollary 2 entails that formula H_i equal to $EQ(Cut'_i, Cut''_i)$ is boundary. Here Cut'_i and Cut''_i specify the output variables of M'_i and M''_i respectively. Derivation of $EQ(Cut'_i, Cut''_i)$ for identical circuits M'_i and M''_i is trivial. However, *proving* that H_i equal to $EQ(Cut'_i, Cut''_i)$ is indeed a boundary formula is *non-trivial* even for identical circuits. (According to Proposition 2, this requires showing that $\exists W_i[EQ(X', X'') \wedge F_{M_i}] \equiv H_i \wedge \exists W_i[F_{M_i}]$ where F_{M_i} specifies the gates of M'_i and M''_i and W_i consists of all the variables of F_{M_i} but the cut variables.) In the experiment, we used the cut with $i = 3$ i.e. the gates located below the cut had topological level less or equal to 3. Proving that $EQ(Cut'_i, Cut''_i)$ is a boundary formula takes a fraction of a second for $i = 3$ but requires much more time for $i = 4$.

We generated 100 buggy versions of Mlp_{16}. Table III contains results of checking the satisfiability of 100 formulas α and β by Minisat 2.0 [5], [21]. Similar results were observed for the other SAT-solvers we tried. The first column of Table III shows the type of formulas (α or β). The second column gives the number of formulas solved in the time limit of 600 s. The third column shows the total run time on all formulas. We charged 600 s. to every formula α that was not solved within the time limit. The run times of solving formulas β include the time required to build H_3. The fourth column gives the median time. The results of this experiment show that proving satisfiability of β is noticeably easier than that of α. As we mentioned above, using formula β for proving inequivalence of N' and N'' should be much more beneficial if formula H_i is computed for a cut with a greater value of i. However, this will require a more powerful PQE solver than PQE-14.

7. Some Background

The EC methods can be roughly classified into two groups. Methods of the first group do not assume that circuits N' and N'' to be checked for equivalence are structurally similar. Checking if N' and N'' have identical BDDs [4] is an example of a method of this group. Another method of the first group is to reduce EC to SAT and run a general-purpose SAT-solver [15], [18], [5], [2]. A major flaw of these methods is that they do not scale well with the circuit size.

Methods of the second group try to exploit the structural similarity of N', N''. This can be done, for instance, by making transformations that produce isomorphic subcircuits in N' and N'' [1] or make simplifications of N' and N'' that do not affect their range [14]. The most common approach used by the methods of this group is to generate an inductive proof by computing simple relations between internal points of N', N''. Usually, these relations are equivalences [11], [12], [17]. However, in some approaches the derived relations are implications [13] or equivalences modulo observability [3]. The main flaw of the methods of the second group is that they are very "fragile". That is they work only if the equivalence of N' and N'' can be proved by derivation of relations of a very small class.

The machinery of boundary formulas introduced in this paper can be related to interpolation [16]. As far as propositional logic is concerned, interpolation and an interpolant are a special case of logic relaxation and a boundary formula respectively [7].

8. Conclusions

We introduced a new framework for Equivalence Checking (EC) based on Logic Relaxation (LoR). The appeal of applying LoR to EC is twofold. First, EC by LoR provides a powerful method for generating proofs of equivalence by induction. Second, LoR gives a framework for proving inequivalence without generating a counterexample. The idea of LoR is quite general and can be applied beyond EC. LoR is enabled by a technique called partial quantifier elimination and the performance of the former strongly depends on that of the latter. So building efficient algorithms of partial quantifier elimination is of great importance.

Acknowledgment

I would like to thank the anonymous reviewers of this paper for their helpful feedback. I am thankful to Harsh Raju Chamarthi for reading the first version of the manuscript. I am especially grateful to Mitesh Jain who has read several versions of this paper and made detailed and valuable comments. This research was supported in part by NSF grants CCF-1117184 and CCF-1319580.

References

[1] H.R. Andersen and H. Hulgaard. Boolean expression diagrams. *Inf. Comput.*, 179(2):194–212, 2002.
[2] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
[3] D. Brand. Verification of large synthesized designs. In *ICCAD-93*, pages 534–537, 1993.
[4] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
[5] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, Santa Margherita Ligure, Italy, 2003.
[6] E. Goldberg. On efficient methods for partial quantifier elimination. To be published.
[7] E. Goldberg. Equivalence checking by logic relaxation. Technical Report arXiv:1511.01368 [cs.LO], 2015.
[8] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.
[9] E. Goldberg and P. Manolios. Quantifier elimination via clause redundancy. In *FMCAD-13*, pages 85–92, 2013.
[10] E. Goldberg and P. Manolios. Partial quantifier elimination. In *Proc. of HVC-14*, pages 148–164. Springer-Verlag, 2014.
[11] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts And Heaps. *DAC*, pages 263–268, 1997.
[12] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. CAD*, 21:1377–1394, 2002.
[13] W. Kunz. Hannibal: An efficient tool for logic verification based on recursive learning. In *ICCAD-93*, pages 538–543, 1993.
[14] H. Kwak, I. MoonJames, H. Kukula, and T. Shiple. Combinational equivalence checking through function transformation. In *ICCAD-02*, pages 526–533, 2002.
[15] J. Marques-Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, 1996.
[16] K. L. Mcmillan. Interpolation and sat-based model checking. In *CAV-03*, pages 1–13. Springer, 2003.
[17] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to combinational equivalence checking. In *ICCAD-06*, pages 836–843, 2006.
[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC-01*, pages 530–535, New York, NY, USA, 2001.
[19] V. Singhal. Private communication.
[20] ABC. http://www.eecs.berkeley.edu/ alanmi/abc/.
[21] Minisat2.0. http://minisat.se/MiniSat.html.

Minimal unsatisfiable core extraction for SMT

Ofer Guthmann[1], Ofer Strichman[2], Anna Trostanetski[1]
[1] Computer science, Technion, Haifa, Israel. {ofer.guthmann,annat}@cs.technion.ac.il
[2] Information Systems Engineering, IE, Technion, Haifa, Israel. ofers@ie.technion.ac.il

Abstract*—Finding a minimal (i.e., irreducible) unsatisfiable core (MUC), and high-level minimal unsatisfiable core (also known as group MUC, or GMUC), are well-studied problems in the domain of propositional satisfiability. In contrast, in the domain of SMT, no solver in the public domain produces a minimal or group-minimal core. Several SMT solvers, like Z3, produce a core but do not attempt to minimize it. The SMT solver MATHSAT has an option to try to make the core smaller, but does not guarantee minimality. In this article we present a method and tool, HSMTMUC, for finding MUC and GMUC for SMT solvers. The method is based on the well-known deletion-based MUC extraction that is used in most propositional MUC extractors, together with several new optimizations such as *theory-rotation*, and an adaptive activation strategy based on measurements, during execution, of the time consumed by various components, combined with exponential smoothing. We implemented HSMTMUC on top of Z3 and MATHSAT, and evaluated its performance with hundreds of SMT-LIB benchmarks.

I. INTRODUCTION

Given an unsatisfiable formula in conjunctive normal form (CNF), an unsatisfiable core (UC) is any subset of these clauses that is still unsatisfiable. In the case of propositional formulas, the problem of finding a minimum core (or rather, the decision problem associated with it) is a Σ_2-complete problem [23] and there were several attempts to cope with it in practice [28], [39]. Given the high-complexity of this problem, there were several attempts to just find *small* cores, without a guarantee of minimality [42], [11], [20]. A large body of work has been dedicated to finding a *minimal* (i.e., irreducible) unsat core (MUC), e.g., [34], [30], [31], [32], which is easier than finding the minimum core, and at least gives the user the guarantee that no single constraint can be removed without making the formula satisfiable. Indeed the only competition ever held in this domain (as part of the SAT competition in 2011) focused on MUC extractions, and now there are several tools that provide this feature, such as MUSER2 [8], HAIFAMUC [31] and MCS-MUS [3]. Most applications of core extraction do not rely on the core being minimal or minimum per-ce, although a small core is desirable; hence striking a balance between efficiency and size of the core is a popular strategy.

The applications of minimal/minimum/small cores of propositional formulas are numerous, including abstraction refinement for model checking [2], [24], [6], formal equivalence verification [26], [16], decision procedures [12], bounded model-checking of multi-threaded systems [22] and functional bi-composition [13] — see [36], [30], [15] for extensive surveys.

When it comes to satisfiability Modulo Theories (SMT), we are aware of several SMT solvers that produce a core, including Z3 [18], CVC3 [5] and YICES [19] but do not attempt to minimize it. A method suggested by Cimatti et al. [15] and implemented in MATHSAT attempts to make the core smaller, but still does not guarantee minimality. We will describe this method in detail and our implementation and experiments with it in Sec. IV. We are aware of one tool, called DFS-FINDER [41], [40], for extracting minimal SMT cores. That tool and the benchmarks it was tested with are not in the public domain. It is based on a deletion-based strategy, and was implemented on top of the SMT solver ARGOLIB [29]. Their focus is on the order in which clauses are removed, which is orthogonal to the techniques we present here.

Whether minimality is important for SMT remains to be seen. As mentioned above most applications of propositional MUC do not *rely* on minimality, but still use a minimal core extractor, since they aspire to use a small core as a 'rule of thumb', namely it is assumed that a small core is better for the rest of the application. Since SMT is used now in many applications that require a core it is reasonable to expect that a tool that finds minimal cores reasonably fast will be used. Microsoft's tool YOGI, for example, a software property-checking tool based on static analysis and testing technology, uses unsat cores in its refinement process [21][1]. Another example is the UFO software model-checker, which uses it to generalize the proof before interpolation [1]. More generally minimality is essential to avoid unnecessary effort in analyzing constraints that are irrelevant for the conflict.

In the current article we show a method for finding a minimal core of SMT formulas, based on the popular deletion-based strategy that is used in several propositional MUC extractors. The basic idea is illustrated in Fig. 1. Given an initial core C, in which all the clauses are unmarked, we remove an unmarked clause $c \in C$ and check for satisfiability of the remaining formula. If the result is SAT, we mark c as necessary for the minimal core, and introduce it back to C. Otherwise (the remaining formula is unsat), we remove clauses outside the new core and continue.

There are many possible optimizations to this basic algorithm, as surveyed in [32], with varying relevance to the case of SMT. Most of them rely on access to a proof (e.g., resolution) and cannot be implemented without changing the SMT solver itself. An exception is Belov and Marques-Silva's recursive model rotation technique [7] (from hereon—rotation), which is both effective and does not require changes to the solver. Indeed we show in Sect. II a generalization of this technique,

[1]Private communication with the authors.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

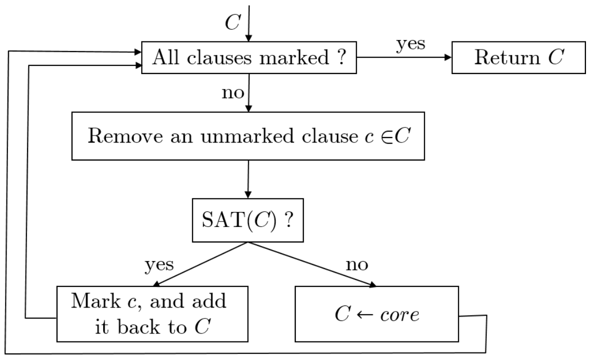


Fig. 1. Deletion-based core extraction. C is an inconsistent set of clauses, all of which are initially unmarked.

which we call *theory rotation*, to the case of SMT. It turns out to be not as effective in the SMT case owing to the cost of checking the T-consistency of assignments, as we will show in Sec. III, but it still improves the run time by about 10% on average, depending on the theory. We used a novel adaptive technique to decide when to activate it based on measurements, during execution, of the time consumed by various components.

We implemented our tool on top of Z3, while relying solely on its API.[2] We did not change Z3 itself, with the hope of supporting other SMT solvers in the future. Z3 has a large user-base and is one of the best solvers, for a large number of theories, according to the latest competitions results of the last few years, and hence is a natural choice. Our results, which we will present in Sect. III, show that our tool HSMTMUC, available from [25], reduces the size of the core by 45% on average. In addition, we support a route via MATHSAT, which turns out to be even faster. It is based on MATHSAT's implementation of Cimatti et al.'s method mentioned above [15], [14] for finding a small core, and then minimizing it with HSMTMUC. We will describe this hybrid approach towards the end of this article, in Sec. IV-E.

II. MINIMIZING SMT UNSATISFIABLE CORES

The standard input language of SMT solvers is that corresponding to the SMT-LIB2 standard [4], which is generally not clausal. A formula is stated via an `assert` statement. Multiple such statements are interpreted as a conjunction between their respective formulas. Z3 tracks which of the assertion statements were used in the proof, and this is what it returns upon a call to `(get-unsat-core)`.

We support two types of core minimizations: minimizing the set of assertions that are inconsistent, and minimizing the set of clauses that result from transforming them to CNF. The first of these goals can be thought of as an application of *High-level minimal unsatisfiable core* [30] (sometimes called *group-MUC*), a problem in which the goal is to minimize the number of high-level constraints in the core, where in this case each assertion is such a constraint. This variant is more useful for human-reading of the output, as it maintains the connection to the original input. The clausal variant can also be useful. For example, one can think of a refinement process that extracts the participating variables from the core, or an engine that extracts the interpolant from the last unsatisfiability proof rather than from the original one, which is likely to make the interpolant smaller. In such applications the mapping of the core to the original formula is not necessary.

Our implementation begins by simplifying the input formula and transforming it to CNF. This is done with the help of Z3's tactics ("simplify" and "tseitin-cnf"). The resulting CNF C is over theory literals and new auxiliary Boolean variables resulting from the Tseitin encoding. Each theory literal l is associated with a propositional variable that encodes whether this literal is true or false in the current assignment . To enjoy the incrementality of Z3, we build the formula such that each clause is guarded with an auxiliary variable, which is then passed as an assumption to the solver. We introduce the guard as a negated auxiliary variable, hence deleting a clause amounts to changing its associated assumption from TRUE to FALSE.

We follow the basic deletion-based method as explained in the introduction and illustrated in Fig. 1 (the center rectangle SAT(C) now corresponds to an SMT call). When solving the high-level variant, each time we remove the whole set of clauses that are associated with a single high-level constraint. On top of that, we implemented an optimization that we call *theory rotation*, for both the high-level and clausal variants of the problem. It does not require any change in Z3 itself. This optimization is the topic of the next subsection.

A. Theory rotation

Suppose that a set of T-clauses C is unsatisfiable, and removing an unmarked clause $c \in C$ makes the formula satisfiable. According to Fig. 1 at this point we should now mark c and put it back in C. In Alg. 1 we show a basic method by which additional clauses can potentially be marked without additional SMT calls. This method generalizes the *rotation* [37] and *recursive model rotation* [7] techniques, which were introduced and proven effective in the domain of propositional MUC extraction.

The idea is the following: given the (propositional) assignment α that satisfies $C \backslash \{c\}$, in line 3 we swap in α the value of one of the variables in c, and call the new assignment α'. This necessarily means that $\alpha' \models c$. If it happens to be the case that α' is T-satisfiable, and contradicts a single unmarked clause $c' \in C \setminus \{c\}$, then we can conclude that c' is also necessary for the core and hence can be marked. The reason is that while C is unsatisfiable, we found a T-satisfiable assignment α' such that $\alpha' \models C \backslash \{c'\}$. Hence in line 5 we mark c'. In the line that follows we call T-ROTATE$_b$ recursively with the new

[2]Specifically, we used the following API functions: Z3_solver_check_assumptions, Z3_solver_get_unsat_core, Z3_solver _reset, Z3_solver_assert, Z3_get_app_decl, Z3_get_app_num_args, Z3_get_app_arg, Z3_mk_not, Z3_mk_or, and Z3_is_eq_ast. To translate the input formula to CNF we apply simplification before and after: tactic t = tactic(ctx, "simplify"), tactic(ctx, "tseitin-cnf"), tactic(ctx, "simplify"); Z3_mk_tactic, Z3_tactic_and_then, Z3_tactic_apply.

assignment α'. The check in line 4 is done lazily, from left to right. Note that the fact that we check in line 4 that c' is unmarked guarantees termination, because 1) each clause can only be marked once, 2) a clause can never be unmarked, and 3) a recursive call happens only after marking a clause.

We continue by suggesting an improvement to this basic procedure, based on the observation that T-ROTATE$_b$ gives up once α' is not T-satisfiable. There is an obvious trade-off between the time invested in attempting to fix α' so that it becomes T-satisfiable and the time it saves by reducing the number of SMT calls. Algorithm T-ROTATE, which appears in Alg. 2, gives the user control over the amount of effort through the bound $flipThreshold$ (see line 6 and the fourth parameter of FLIP). In our winning strategy we set this threshold to two, meaning that we allow for flipping one additional variable in our attempt to make α' T-satisfiable. This strategy increases the number of marked clauses by close to 10% on average, as will be evident in the next section.

The functions T-ROTATE and FLIP in Alg. 2 are mutually recursive, and FLIP is also self-recursive. T-ROTATE is the one called from the main core minimization loop when the removal of c makes C satisfiable. T-ROTATE takes c as input and marks clauses (including c itself), i.e., mark that they belong to the MUC. For each literal $l \in c$, it calls FLIP in line 4.

The conditions checked by FLIP in line 8 are different than those checked in T-ROTATE$_b$, because we want to continue even if α' is not T-consistent (note that in such a case it is possible that $UnsatSet(C, \alpha') \equiv \emptyset$). In such a case we call FLIP recursively with each of the literals in the core (lines 12 – 13), with the hope that after flipping it the new assignment will satisfy all the conditions required for reaching line 10. This invokes FLIP for each literal in the core, which justifies the low value of $flipThreshold$ mentioned above.

Some implementation details: To compute $UnsatSet(C, \alpha')$, we maintain for each literal a set of clauses in which it appears. Hence if we flip l to $\neg l$, we check for (propositional) satisfiability of all the clauses that contain l, in addition to the clause that was unsatisfiable at the entrance to FLIP (when called from T-ROTATE, that clause is guaranteed to become satisfiable after flipping l, by construction. It is not necessarily the case when T-ROTATE is called recursively). Furthermore, since each clause is potentially checked multiple times, under similar assignments, we maintain a map from each clause cl that we check, to the literal lit that satisfies it. When revisiting cl, we first check if lit is still satisfied by the current assignment. If not, we revert to scanning the clause and update the map with a new satisfied literal, if one exists.

B. Theory rotation over high-level constraints

Our implementation of theory rotation for high-level theory constraints appears in Alg. 3. It is a generalization of Alg. 2, based on the propositional high-level rotation that was described in [35], [33]. If, after removing the clauses associated with a high-level constraint H, the formula becomes satisfiable, then we find the set of literals in the intersection of all the clauses in H that are unsatisfied by the current assignment α. We then flip the assignment of each of these literals separately, and check each time if it is both T-consistent and makes a single high-level constraint H' unsat, and H' is unmarked. If both conditions are true, then H' is marked as necessary. The FLIP function that is called in line 8 has the same code as in Alg. 2, except that C is now a set of high-level constraints.

C. Adaptive activation of theory rotation

Our initial experiments showed that T-rotation is frequently not cost-effective, despite the fact that it is polynomial and saves SMT calls which are worst-case exponential. This stands in contrast to the propositional case, where rotation is generally very cost-effective. Reasons for this difference include

- the theory check in line 9 is potentially expensive (yet for most theories still polynomial),
- the success rate is smaller than in the propositional case, because of the additional requirement that the assignment α is T-consistent,
- the attempts to fix α so it becomes T-consistent (line 13) may be expensive if $flipThreshold$ is not small.

Analyzing the logs of our experiments shows that T-rotation has a large impact in the beginning (by 'beginning' we mean after having the initial core from Z3), when there are still many unmarked clauses, but it diminishes through time. The overhead of calling T-rotation while being ineffective at later steps frequently outweighs the initial gain. To overcome this problem, we attempt to stop T-rotation when it is no longer cost-effective. We experimented with two strategies:

- **fail bound**: when x consecutive activations of T-ROTATE produce no marked clauses, we stop.
- **exponential smoothing**: Let t_{smt} be the average time it takes to check T-satisfiability of $C \setminus \{c\}$, t_r the average time it takes to run T-ROTATE, and n_r the average number of clauses that it marks (not including the initial clause c). Had these figures been known and constant throughout the run, we would use T-rotation only if

$$t_{smt} > \frac{t_r}{n_r} . \tag{1}$$

Since this is not the case, then as a second best choice we measure these figures at run time, and use them as the basis for estimating (1). They are not purely monotonic, however, and hence terminating T-rotation once (1) does not hold is not a good strategy. On the other hand the number of marked clauses n_r, as hinted before, has a clear trend: in practice we see that it is reduced to near 0 after a while, when the set of unmarked clauses becomes small. The solution we chose is based on *exponential smoothing*, a known technique in statistics that was also used recently in a SAT branching heuristic [27]. The input data can be seen as a stream of tuples $\langle t^0_{smt}, t^0_r, n^0_r\rangle, \langle t^1_{smt}, t^1_r, n^1_r\rangle, \ldots$, where the superscript denotes the time index. We define $T^0_{smt} = t^0_{smt}$

Algorithm 1 A basic theory rotation algorithm.

```
1: function T-ROTATE_b(clause-set $C$, clause $c$, assignment $\alpha$)
2:     for each $l \in c$ do
3:         $\alpha' := \alpha[l \leftarrow \neg l]$;
4:         if $UnsatSet(C, \alpha') \equiv \{c'\}$ and $c'$ is unmarked and $T$-SAT($\alpha'$) then
5:             mark $c'$;
6:             T-ROTATE_b ($C, c', \alpha'$);
```

Algorithm 2 Theory rotation, in which certain effort is invested in fixing the assignment α so it becomes T-consistent and consequently leads to marking of additional clauses.

```
Require: $C$ is unsat, and $\alpha \models C \setminus c$
1: void T-ROTATE(clause-set $C$, clause $c$, assignment $\alpha$)
2:     mark $c$;
3:     for each literal $l \in c$ do
4:         FLIP($C$, $l$, $\alpha$, 0);

Require: $\alpha$ does not satisfy zero or one clauses from $C$
5: void FLIP(clause-set $C$, lit $l$, assignment $\alpha$, int $depth$)
6:     if $depth \geq flipThreshold$ then return ;                          ▷ User-defined threshold
7:     $\alpha' := \alpha[l \leftarrow \neg l]$;
8:     if ($UnsatSet(C, \alpha') \equiv \{c'\}$ and $c'$ is unmarked)  or $UnsatSet(C, \alpha') \equiv \emptyset$ then
9:         if ($T$-SAT($\alpha'$)) then                                     ▷ Theory-checking of $\alpha'$
10:            T-ROTATE ($C$, $c'$, $\alpha'$);                               ▷ $c'$ must exist here
11:        else
12:            for each literal $l'$ in $core$ do                           ▷ $core =$ unsat core of line 9
13:                FLIP($C$, $l'$, $\alpha'$, $depth + 1$);
```

Algorithm 3 High-level theory rotation, in which certain effort is invested in fixing the assignment α so it becomes T-consistent and consequently leads to marking of additional clauses.

```
Require: $C$ is unsat, and $\alpha \models C \setminus c$
1: void T-ROTATE(constraint-set $C$, constraint $c$, assignment $\alpha$)
2:     mark $c$;
3:     $intersection := Lit(c)$                       ▷ $Lit(c)$ is the set of literals that appear in $c$
4:     for each clause $cl \in c$ do
5:         if $\alpha \not\models cl$ then                      ▷ at least one such clause exists since $\alpha \not\models c$
6:             $intersection := intersection \cap Lit(cl)$      ▷ $Lit(cl)$ is the set of literals that appear in $cl$
7:     for each literal $l \in intersection$ do
8:         FLIP($C$, $l$, $\alpha$, 0);
```

and

$$T^i_{smt} = \alpha \cdot t^i_{smt} + (1 - \alpha) \cdot T^{i-1}_{smt} \ , \tag{2}$$

where α is a parameter in the range [0..1]: the closer it is to 1, the closer the value of T^i_{smt} is to the current input t^i_{smt}. Conversely the closer α is to 0, the more 'smooth' T^i_{smt} becomes. Similarly we define T^i_r and N^i_r, with respect to the t_r and n_r sequences, respectively. We continue with T-rotation while

$$T^i_{smt} > \frac{T^i_r}{N^i_r} \ . \tag{3}$$

In our experiments we used $\alpha = 0.1$.

III. Experiments

We experimented with the same 561 benchmarks used in [15], which were selected from SMT-LIB, and include instances from the quantifier-free theories LRA, UF, RDL, LIA and IDL. From those we removed 63 instances that Z3 cannot solve in 10 minutes (our timeout), i.e., solve the formula itself augmented with the auxiliary guard variables. This left us with 498 benchmarks. All experiments and graphs in this section were conducted via HBENCH [38], a performance-benchmarking platform.

For the clausal variant of the problem, Fig. 2 (left) shows a comparison of the size of the default core given by Z3 and the minimal core that our tool, HSMTMUC, emits. Overall the average reduction in core size is 45%. The plot on the right shows the impact of theory rotation on the number of iterations. Overall the average reduction in the number of iterations is 34%. Although the diagram shows that rotation always reduces the number of iterations (or at least does not increase it), in theory a point to the left of the diagonal *is* possible. This is because rotation may change the order in which clauses are chosen for removal. This, in turn, may impact the run of the SMT solver and produce a different core if the formula is unsatisfiable.

Table I shows more detailed results for the clausal variant. The 'base' configuration is a simple deletion-based strategy, and all others include rotation. The '$b\ x$' label means that we activate the 'fail bound' strategy with a bound x, and the 'exp' label means that we activate the exponential smoothing strategy, both explained in the previous section. Overall, all the rotation strategies that we used improve upon the base configuration, with the 'exp' strategy having the least number of fails within the 10 min. timeout. The 'T-conflict resolved' column presents the number of cases that lines 12–13 in T-ROTATE led to additional marked clauses. Whereas the avg. time it takes to compute the minimal core (the Time column) is close to 30 sec., the avg. time it takes Z3 to compute the initial core is ≈ 2.5 sec.

A detailed analysis shows that the effect of theory rotation depends on the theory itself (to the extent that the benchmarks themselves are representative of the theory). We refer the reader to [25] for detailed results.

We also experimented with the high-level variant. We removed benchmarks that have a single `assert` statement, which left us with 395 benchmarks, out of which 41 timed-out. The last line of the table shows our results. Note that the number of iterations in the high-level variant is an order of magnitude smaller comparing to the clausal variant, which is expected given the nature of the problem (i.e., rather than removing a single clause in each iteration, we remove many). In our experiments rotation had negligible effect in this variant, which is expected given the low number of iterations. We again refer the reader to [25] for detailed results.

Unfortunately we cannot make a full comparison to [41], [40] because neither the tool nor the benchmarks that were used to evaluate it are in the public domain. We could only compare the 15 sample benchmarks for which detailed results were published in [40]. The results show that our tool is over two orders of magnitude faster. Detailed results are available in [25]. We emphasize that [41], [40] is based on a different, less competitive SMT solver (ARGOLIB), and that they used different hardware, hence the comparison only approximates the relation between the tools, not the MUC extraction algorithms themselves.[3]

[3]It seems that they also used a different conversion to CNF, because the number of clauses that they report is different than ours (to both directions), despite the fact that we start from the same SMT-LIB benchmark.

IV. A COMPARISON TO A MINIMIZATION OF THE BOOLEAN ENCODING, AND A HYBRID APPROACH

We now describe in detail our implementation and experiments with a method suggested by Cimatti et. al in [15] that was implemented in MATHSAT, which finds a small SMT core, that is not necessarily minimal. As we will show, our implementation of this method based on Z3 is not competitive with the one in MATHSAT, but a hybrid approach, in which we run HSMTMUC to minimize the result of MATHSAT, is the best configuration we found.

A. Boolean-encoding minimization

Recall that an SMT solver combines a propositional SAT solver and a decision procedure DP_T for a conjunction of T-terms, for each supported theory T. It begins by associating with each T-literal l a new propositional variable which we denote by $e(l)$. Overloading the notation, we denote by $e(\varphi)$ a T-formula φ after all of its literals are encoded this way. Hence $e(\varphi)$ is a propositional *abstraction* of φ. We call $e(\varphi)$ the *propositional skeleton* of φ.

Cimatti et al.'s method for extracting a small unsat SMT core, is based on using a *propositional* MUC extractor for minimizing $e(\varphi) \wedge e(L)$, where L denotes the lemmas generated during the run of the SMT solver. The $e(L)$ clauses are discarded from the core, because L corresponds to theory lemmas that are by construction T-valid, and hence can always be conjoined to the formula. The rest of the core can be mapped back to a set of clauses $\varphi' \subseteq \varphi$, which is guaranteed to be unsatisfiable. This method has a major practical advantage as it leverages existing tools for minimizing propositional cores and is easy to implement if the SMT solver can emit $e(L)$ (Z3 does not support such an option, but we will explain how this can be achieved with Z3 later on). Nevertheless, as noted by the authors, this process does not guarantee minimality of the SMT core. We demonstrate this fact with two examples.

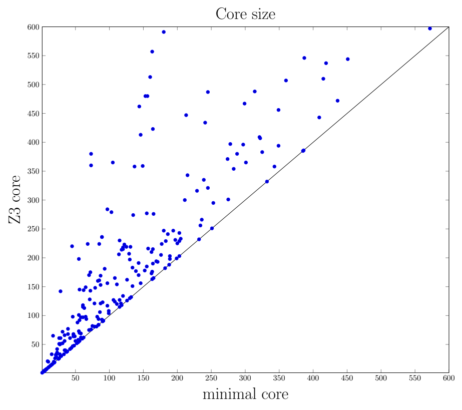
Example 1. *Quoting example 5 from [15], consider*

$$\begin{aligned}\varphi \doteq\ & ((x=0) \vee (x=1)) \wedge (\neg(x=0) \vee (x=1)) \wedge \\ & ((x=0) \vee \neg(x=1)) \wedge (\neg(x=0) \vee \neg(x=1))\ .\end{aligned} \tag{4}$$

It is clear that $e(\varphi)$ is unsatisfiable, and further that all the clauses in $e(\varphi)$ are necessary for maintaining unsatisfiability. Nevertheless the last clause is not necessary and hence this is not a minimal core of φ. $\square$

The example above demonstrates the fact that whereas T-valid clauses *cannot* be part of a *minimal* core (because they are always implied anyway and therefor can be removed), the information that they are valid is lost once the search for a core focuses on the propositional abstraction of φ. This particular problem can be easily fixed by removing T-valid clauses from the resulting core (by calling DP_T for each clause separately), but this still does not guarantee minimality as we show next.

A bigger problem is that once we minimize $e(\varphi) \wedge e(L)$, we are restricted to the lemmas in L, which are not necessarily



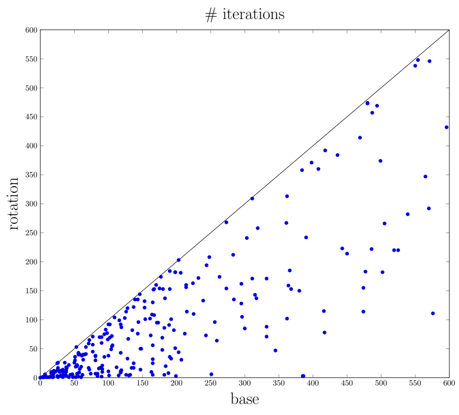


Fig. 2. (left) Z3's default core vs. the minimal core that our tool HSmtMuc emits. (right) The number of iterations with and without theory-rotation.

Config.	# fails	Time (sec.)	Iterations	Rotation Calls	Cls. Marked By Rotation	T-Conflicts Resolved	T-check Time (Sec.)	Init core	Final core
(base)	108	30.5	559.2	0.0	0.0	0.0	0.0	820.2	454.2
T-Rotate	108	29.7	372.0	472.2	203.7	20.8	1.4	820.2	454.4
T-Rotate b 5	108	**28.9**	435.9	168.8	130.8	10.2	1.0	820.2	454.5
T-Rotate b 7	109	29.2	417.1	194.4	143.2	12.3	1.2	820.2	454.4
T-Rotate exp	**107**	29.6	424.3	244.8	151.8	11.2	1.2	820.2	454.5
HL (base)	41	7.2	45.4	0.0	0.0	0.0	0.0	41.8	26.7

TABLE I
The top five rows reflect results with different configurations, over 498 SMT benchmarks. Other than the # fails column, the data reflects averages. The last two columns refers only to benchmarks solved by all configurations. 'Init core' is the size of the initial core emitted by Z3. The last row refers to the High-Level variant, and is only over the (395) benchmarks that have more than one `assert` statement.

minimal themselves. The following example demonstrates this problem.

Example 2. *For $x_1, \ldots, x_4 \in \mathcal{R}$, let*

$$\varphi \doteq (x_1 = x_2) \wedge (x_2 = x_4) \wedge (x_1 = x_3) \wedge (x_3 = x_4) \wedge \neg(x_1 = x_4)\ . \tag{5}$$

Suppose now that the following lemma, which is simply a negation of φ, was learned during the search

$$L \doteq \neg(x_1 = x_2) \vee \neg(x_2 = x_4) \vee \neg(x_1 = x_3) \vee \neg(x_3 = x_4) \vee (x_1 = x_4)\ . \tag{6}$$

This is a T-valid statement, although it is not minimal. Now $e(\varphi) \wedge e(L)$ is unsatisfiable, and a minimal core at the propositional *level, after discarding the $e(L)$ clauses, is all the clauses of φ. This core is not minimal with respect to φ, however, because, e.g., the first two clauses can be removed. This* could *have been prevented had the solver inferred the shorter lemma L':*

$$L' \doteq \neg(x_1 = x_3) \vee \neg(x_3 = x_4) \vee (x_1 = x_4)\ , \tag{7}$$

but there is no guarantee for this to happen. □

B. Z3's lemmas and proofs

As described in [14], MathSat has a built-in support for logging all the T-Lemmas produced during φ's satisfiability check. However, in the case of Z3, this logging is bypassed, and instead Z3 maintains proof objects during conflict resolution, as described in [17].

A detailed description of Z3's language and proofs has been given in [17], [9], [10]. Z3*'s language* is a many-sorted FOL based on the SMT-LIB language. Z3*'s proof terms* represent natural deduction proof currently using 34 *axioms* and *inference rules*. These inference rules range from simple rules such as **MP** (modus ponens), to complex rules that abbreviate multiple reasoning steps such as **Rewrite** for standard simplification rules, and other theory-specific reasoning, such as **Transitivity**.

Given an unsatisfiable formula φ, Z3*'s proof* is a *directed acyclic graph* (*DAG*) with a single root. Each node is labeled with a formula: leafs are labeled with either a T-valid formula or one of the original clauses in φ, internal nodes are labeled with a consequent of some T-inference rule, and the root is labeled with $\perp$, i.e., $false$. In the discussion below we will not make the distinction between a node and its label. An edge from a node n to a node n' in the proof represents the fact that n' was used as a premise of an inference rule whose consequent is n. Hence, if n has k children $n_1..n_k$, then

$$(e(n_1) \wedge \cdots \wedge e(n_k)) \rightarrow e(n) \tag{8}$$

represents an encoding of a valid T-implication. Let $e(L)$ be the set of implications of the form (8) corresponding to the entire set of internal nodes and the set of $T-$valid leafs in the proof graph. Then $e(\varphi) \wedge e(L)$ must be unsatisfiable.

C. Implementation using Z3's proof

To implement Cimatti et. el.'s method on top of Z3, we traverse the proof graph produced by Z3[4], and replace each inference with a corresponding propositional lemma (8). Having extracted $e(L)$, it is now possible to apply propositional MUC extraction on $e(\varphi) \wedge e(L)$, as described in Sect. IV-A. Finally, given the propositional MUC, we translate it back to the original T-clauses and check whether it is a minimal core with HSMTMUC. Since we are only interested in the question whether the core is already minimal, for this experiment we terminate HSMTMUC early with "not minimal" once we find a clause that can be removed.

D. Experimental results

We ran our implementation on top of Z3 of the method of [14], with the same 498 benchmarks that were mentioned in Sect. III. For the propositional MUC extractor we used HMUC [32]. We note that the fact that we ask Z3 to log the proof, has an overhead. In experiments reported in [17], it was shown that the memory overhead is $\times 3$ to $\times 40$ greater, with corresponding slowdowns of $\times 1.1$ to $\times 3$. The detailed results appear in Table II. Overall it is not competitive with the implementation in MATHSAT, as will be seen next.

E. Experiments with a hybrid solution

We tested two more configurations: the original implementation of [14] in MATHSAT, and another version in which after running MATHSAT we invoke HSMTMUC to minimize the resulting core. We refer to the two stages of this *hybrid* solution as Hybrid-M (MATHSAT) and Hybrid-H (HSMTMUC). The number of fails are:

HSMTMUC (base)	Hybrid
171	138

Note that those numbers are out of the full set of 561 benchmarks. Hence the 171 fails of HSMTMUC is made of the 108 fails reported in Table I + 63 cases in which Z3 could not produce the initial core within the time limit. The 138 fails of the hybrid approach include 98 fails of MATHSAT itself. Hence, we can see that from the perspective of the number of fails, the hybrid approach is better than HSMTMUC alone for finding a minimal core. In Table III we examine more closely the cases that all three approaches succeeded. As can be seen, we achieve a reduction of 20.9% on average in core size with the hybrid approach, comparing to MATHSAT alone (which, recall, is not necessarily minimal), and a reduction of 9% comparing to HSMTMUC. The total average time of the hybrid approach (11.2 sec. + 16.7 sec.) is larger, however, than invoking HSMTMUC alone (22.9 sec).

Comparing MATHSAT to our implementation on top of Z3, we see that the former is better: it has less fails (98 vs. 164), better run-time on those instances it completes (11.2 sec. vs. 40.4 sec) and smaller average core size (523.0 vs. 723.7). It seems that MATHSAT simply finds proofs that use a smaller number of facts from the original formula φ. It also does not have the overhead of reconstructing the proof as explained in Sec. IV-C, and it uses a different propositional extractor (MUSER2 vs. HMUC).

[4]Using the methods expr::num_args(), expr::arg(i), expr::decl(), fun_decl::decl_kind()

V. Conclusions and future work

We presented an algorithm for extracting a minimal unsatisfiable core from SMT unsatisfiable formulas, which is based on a combination of a deletion-based strategy and theory rotation. Many other optimizations exist for the propositional case, such as those published in [8], [32], but they can only be used in the context of SMT if the SMT solver itself is changed. We refrained so far from such changes, with the hope of supporting other SMT solvers that provide a similar API. A highly desirable situation is one in which the initial run of the SMT solver is already biased towards a small core, the same way that the SAT solver is biased towards finding a minimal core in HaifaMuc [32]. For example, make the theory solver return lemmas that contradict as few unmarked clauses as possible. Such an optimization requires theory-specific changes, however, which we leave for future research.

References

[1] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Computer Aided Verification - 24th International Conference, CAV*, volume 7358 of *LNCS*, pages 672–678. Springer, 2012.

[2] N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *9th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, 2003.

[3] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Computer Aided Verification - 27th International Conference, CAV*, volume 9207 of *LNCS*, pages 70–86. Springer, 2015.

[4] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at http://www.SMT-LIB.org.

[5] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[6] A. Belov, H. Chen, A. Mishchenko, and J. Marques-Silva. Core minimization in SAT-based abstraction. In *DATE*, pages 1411–1416, 2013.

[7] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, pages 37–40, 2011.

[8] A. Belov and J. Marques-Silva. MUSer2: An efficient MUS extractor. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, 8(1/2):123–128, 2012.

[9] S. Böhme. Proof reconstruction for z3 in isabelle/hol. In *7th International Workshop on Satisfiability Modulo Theories (SMT09)*, 2009.

[10] S. Böhme and T. Weber. Fast lcf-style proof reconstruction for z3. In *Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[11] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, 130(2):85–100, 2003.

[12] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *Software Tools for Technology Transfer (STTT)*, 11:95 – 104, 2009.

[13] H. Chen and J. Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC (Selected Papers)*, pages 52–72, 2011.

[14] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 334–339. Springer, 2007.

Family	# Benchmarks	# Fails	# Minimal	Time (Sec.)	# lemmas extracted	Init core	Core size
QF_IDL	71	4	43	6.2	3109.5	1120.7	529.9
QF_LIA	115	15	20	13.5	3670.1	1306.4	824.6
QF_LRA	116	14	26	46.	14134.0	1094.8	794.9
QF_RDL	81	20	14	72.2	9281.0	2398.6	1257.3
QF_UF	115	48	2	76.2	54394.6	242.4	172.6
Total	498	101	105	40.4	15686.6	1208.9	723.7

TABLE II
RESULTS OF OUR IMPLEMENTATION OF CIMATTI'S ET. AL.'S METHOD ON TOP OF Z3, OVER 498 SMT BENCHMARKS. COLUMNS 4 – 8 REFER ONLY TO BENCHMARKS THAT RAN TO COMPLETION.

Family	# Benchmarks	Original size	Core size			Time			Core
			Hybrid-M	Hybrid-H	HSMTMUC	Hybrid-M	Hybrid-H	HSMTMUC	Reduction %
QF_IDL	68	30659.5	534.3	525.8	515.9	4.5	6.0	8.5	1.6
QF_LIA	100	3490.1	614.1	496.0	567.3	18.7	13.6	11.9	22.5
QF_LRA	83	1908.3	448.7	330.2	405.5	2.2	20.8	47.6	31.7
QF_RDL	53	13353.3	872.1	779.2	813.6	26.5	16.6	28.2	15.7
QF_UF	71	2007.3	209.9	105.0	107.7	5.8	26.6	19.7	50.0
Total	375	9180.0	523.0	428.8	470.0	11.2	16.7	22.9	20.9

TABLE III
CHECKING THE HYBRID APPROACH, OVER THE 375 (OUT OF 561) BENCHMARKS SOLVED BY ALL CONFIGURATIONS TO COMPLETION. THE LAST COLUMN REFERS TO THE REDUCTION IN CORE SIZE BY THE HYBRID APPROACH, COMPARING TO MATHSAT ALONE.

[15] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res. (JAIR)*, 40:701–728, 2011.

[16] O. Cohen, M. Gordon, M. Lifshits, A. Nadel, and V. Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*, 2010.

[17] L. M. de Moura and N. Bjørner. Proofs and refutations, and z3. In *LPAR Workshops*, volume 418, pages 123–132, 2008.

[18] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[19] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV), 18th International Conference*, volume 4144 of *LNCS*, pages 81–94, 2006.

[20] R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design (FMSD)*, 33:1 – 27, 2008.

[21] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 43–56. ACM, 2010.

[22] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, pages 122–131. ACM Press, 2005.

[23] A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006.

[24] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *ICCAD*, pages 416–423, 2003.

[25] O. Guthmann, O. Strichman, and A. Trostanetski. HSmtMuc:. http://ie.technion.ac.il/~ofers/haifasolvers/site/site.html.

[26] Z. Khasidashvili, D. Kaiss, and D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143, 2009.

[27] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In N. Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *LNCS*, pages 225–241. Springer, 2015.

[28] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.

[29] F. Maric and P. Janicic. Argo-Lib: A generic platform for decision procedures. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR*, volume 3097 of *LNCS*, pages 213–217. Springer, 2004.

[30] A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 221–229. IEEE, 2010.

[31] A. Nadel, V. Ryvchin, and O. Strichman. Efficient mus extraction with resolution. In *FMCAD*, pages 197–200, 2013.

[32] A. Nadel, V. Ryvchin, and O. Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:27–51, 2014.

[33] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, volume 8561 of *LNCS*, pages 206–218. Springer, 2014.

[34] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.

[35] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *SAT*, pages 174–187, 2011.

[36] J. P. M. Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL'10*, pages 9–14, 2010.

[37] J. P. M. Silva and I. Lynce. On improving MUS extraction algorithms. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference*, volume 6695 of *LNCS*, pages 159–173, 2011.

[38] O. Strichman. HBench – a platform for performance benchmarking. http://strichman.net.technion.ac.il/hbench/.

[39] J. Zhang, S. Li, and S. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence*, volume 4304 of *LNCS*, pages 847–856. Springer, 2006.

[40] J. Zhang, S. Shen, J. Zhang, W. Xu, and S. Li. Extracting minimal unsatisfiable subformulas in satisfiability modulo theories. *Comput. Sci. Inf. Syst.*, 8(3):693–710, 2011.

[41] J. Zhang, W. Xu, J. Zhang, S. Shen, Z. Pang, T. Li, J. Xia, and S. Li. Finding first-order minimal unsatisfiable cores with a heuristic depth-first-search algorithm. In *Intelligent Data Engineering and Automated Learning - IDEAL*, volume 6936 of *LNCS*, pages 178–185, 2011.

[42] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas, 2003.

Efficient Uninterpreted Function Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho*, Pankaj Chauhan†, Pritam Roy†, Alan Mishchenko*, Robert Brayton*
*Department of EECS, University of California, Berkeley, CA, USA
{ysho, alanmi, brayton}@eecs.berkeley.edu
†Mentor Graphics, Inc., Fremont, CA, USA
{Pankaj_Chauhan2, Pritam_Roy}@mentor.com

***Abstract*—Methods for word-level model checking based on purely bit-level techniques have difficulties with heavy arithmetic logic. Word-level and SMT approaches often are limited by relying on (incomplete) bounded model checking. UFAR, a hybrid word- and bit-level approach, addresses these issues, taking advantage of modern bit-level sequential techniques while heavy arithmetic logic is addressed by word-level abstraction and the use of uninterpreted function (UF) constraints. The methods and efficiency improvements developed for UFAR enabled it to prove 2422 of a set of 2492 industrial sequential model checking problems within a 1-hour limit, while a bit-level model checker *super_prove* completed only 2115 of these within the same limit.**

I. Introduction

Model checking (MC) on a Register-Transfer-Level (RTL) word-level netlist is a necessary verification task for applications involving sequential synthesis. In this, an RTL netlist is synthesized into another through retiming, clock-gating, pipelining etc., and MC is required for proving the correctness of the result. These problems are challenging if hard arithmetic operators such as multipliers, adders, and variable shifters are involved, and correspondences between flip flops are not known.

Previous methods in this domain can be classified as follows. One directly "bit-blasts" the problem and then solves with bit-level techniques such as IC3/PDR [5], [14], interpolation [19], or BDDs [11]. Another [18] translates the problem into SMT formulas (if possible) and then directly employs SMT solvers such as Boolector [10], or Z3 [13]. A third [17] applies *predicate abstraction* [16]. *Term-level abstraction* [2], [1], [7], [6] replaces arithmetic operators with uninterpreted functions (UF), and then solves with SMT solvers. However, bit-level techniques are problematic when verifying circuits with heavy arithmetic logic. Techniques adapted from software verification are often not effective for hardware equivalence checking. Most SMT-based approaches rely on (incomplete) bounded model checking (BMC) [4] or induction [21] and may not be applicable.

UFAR (Uninterpreted Function Abstraction and Refinement), is a hybrid word- and bit-level solver, which moderates the above issues. It takes advantage of modern sequential techniques such as PDR and BMC at the bit-level, while heavy word-level logic is tackled by abstraction and the use of uninterpreted function (UF) constraints.

Such techniques are not new, even at the word level. Conventional UF abstraction [2], [1], [7], [6] methods implicitly enforce *all* possible UF constraints among the same functions. This becomes inefficient when the number of similar functions is large. Keys to UFAR's efficiency are how simulations and minimized counterexamples are used to refine abstractions, how constraints are added and removed lazily, which pairs of operators are constrained, and how UF constraints are applied between operators of the same type but with different bit widths. All this requires efficiently iterating between word-level Verilog and AIG representations as refinements are done. These techniques enable UFAR to prove problems containing hundreds of heavy word-level operators.

We prove that UFAR is a sound and complete framework for word-level counterexample guided abstraction and refinement (CEGAR) [12]. It starts with the extreme abstraction with all "problematic" word-level operators (e.g., multipliers, adders, etc) removed (i.e. operator outputs are replaced by unconstrained pseudo primary inputs). This is then bit-blasted and given to a sound and complete bit-level model checker. If a counterexample is returned, UFAR first simulates it on the original netlist to check if it is *real*. If so, UFAR terminates and reports it. Otherwise, the *spurious* counterexample is used to refine the current abstraction. Refinement is done in this context by 1) adding UF constraints between some pairs of chosen compatible operators, and 2) restoring one or more of the removed operators.

We experiment on 2492 industrial benchmarks for sequential RTL (word-level) model checking and show how different refinement methods and heuristics are complementary, each solving more problems in less time, and leading to a final algorithm which solves all but 70 of the benchmarks within a one hour time limit. We show detailed results on 19 examples having ranges of 4-475 multipliers, 21092-302277 AIG nodes, and 358-4785 flip-flops.

This paper first presents background material and formal settings in Section II. The UFAR algorithm is presented in Section III. Several optimization techniques for the algorithm are given in Section IV. Section V gives some details about the UFAR framework, including word-level representation and bit-blasting this into an AIG. Experimental results on an extensive set of industrial problems are presented in Section VI, comparing the effectiveness of the two optimizations and

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

the overall UFAR algorithm. Some conclusions and future work are discussed in Section VII.

II. Bit-Vectors and UF Constraints

In the context of Verilog and its bit-vector operators, we need to be precise about applying UF constraints between pairs of operators. A UF constraint states that for two same-type functions, if their inputs are equal then their outputs are equal. Unfortunately, this is not at all straight-forward when bit-vector operators are involved. Incorrect application of UF constraints can lead to an unsound procedure on the one hand or to a too restrictive application on the other. In this section, we discuss bit-vector operators, define what it means to be the same function, state when and how to make UF constraints valid between two same-type operators, and prove the soundness of the derived methods.

A. The MC problem

We assume that the input RTL design is in *structural Verilog*. In structural Verilog, there are 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. A bit-vector signal s can be either *signed* or *unsigned*, denoted by $signed(s)$. The *bit-width* of s is denoted by $bw(s)$. A design is modeled as a finite state machine (FSM).

Definition 1. A design in structural Verilog is a tuple $M = (I, O, S, S_0, T)$ where I is the set of inputs, O is the set of outputs, S is the set of state variables, S_0 is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, then there exists a transition from s to s' under i.

The input format is assumed to be *mitered* as a single FSM and a single output, out, representing the property to be checked. If the problem is to prove equivalence between two designs, a miter is created by merging all PIs and merging corresponding mapped FFs (if any). The output out is a Boolean signal, which is the OR of the pairwise XORs of the corresponding outputs of the two designs. Thus it is 1 if the two designs are different. Similarly for property checking, the output is a monitor which signals 1 if the property fails. In terms of linear temporal logic (LTL), the MC problem is formulated as $M \models \mathbf{G}\neg out$, meaning the miter M should never excite the signal out if the property holds.

B. Basics of word-level operators

We focus on abstracting problematic *word-level operators* in a design. The subset of operators considered are all word-level *binary* operators, such as $+, -, *, /, \%, <<, >>, <<<, >>>$. In Verilog, an operator is instantiated by a *structural statement* which only states the function type of the operator and the connection between signals[1]. An operator is modeled as a labeled node with a single output, up to two inputs, and its label of function type.

[1]Without loss of generality, we assume that each statement contains only 1 binary operator. Statements like `x = (a+b)*c` can always be rewritten to `y = a+b` and `x = y*c`.

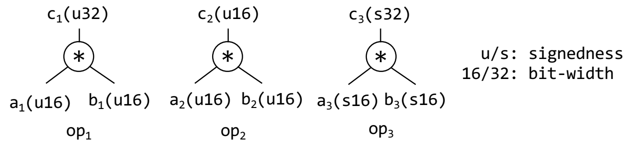


Fig. 1: Three multipliers with different functions.

Definition 2. An operator op is a tuple $op = (o, i_1, i_2, t)$ where o is the output signal, i_1 and i_2 are the input signals, and t is the label of function type.

For example, the Verilog statement, `c = a * b`, is modeled as $op = (c, a, b, *)$. Note that the inputs are *ordered* as specified in the Verilog statement. Note also that $*$ is a "function-type" and not a function, since the actual function that would be instantiated would depend on the properties of the signals to which its inputs and output are connected. The necessity of this important distinction will be clarified in the next section.

C. Functions of word-level operators

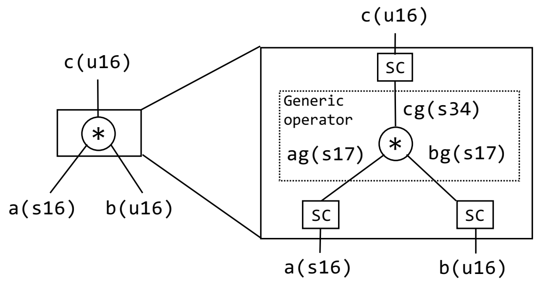
In Verilog, the actual *function* associated with an operator is determined by the bit-widths and signedness of its inputs, output, and function-type. Operators with the same function type do not necessarily have the same function; a function-type represents a set of functions. For example, the three multipliers in Figure 1 all represent different functions. Operators op_1 and op_3 are different since op_1 is *unsigned* multiplication while op_3 is *signed* multiplication. Operator op_2 is different because its output is only 16 bits.

To be precise in what follows, we need to explicitly model what a Verilog front end does when it bit-blasts a Verilog RTL design into a bit-level circuit. For this we need *generic operators* and *signal convertors*.

Definition 3 (Generic operator)**.** A generic operator is a bit-vector operator that agrees with the *integer function* of its function-type. That is, the bit-vector output, when evaluated as an integer, is consistent with the result using the integer function. It has the following properties.

- All of its inputs and output are signed.
- It is a pure arithmetic function parametrized with specified widths for its inputs and output.
- When a generic operation is implemented, its input widths should be compatible with the widths of the signals connected to them.
- The output width should be exactly large enough so as to not impose any restriction on the operation (such as truncation due to overflow).

In order to create signals used or produced by an instantiated generic function, they must be converted from unsigned to signed signals or vice versa. They also need to be converted by truncation, sign extension, or zero extension. We model this



```
                                       wire signed [15:0] a;
                                       wire        [15:0] b;
wire signed [15:0] a;        Expose    wire        [15:0] c;
wire        [15:0] b;        ━━━➡      wire signed [16:0] ag = {1'b0, a};
wire        [15:0] c = a * b;          wire signed [16:0] bg = {1'b0, b};
                                       wire signed [33:0] cg = ag * bg;
                                       assign             c = cg;
```

(b) A piece of Verilog code for exposing the generic operator.

Fig. 2: An example showing how generic operators are modeled and exposed.

by emulating what Verilog does in its assignment operator (=) and concatenation operator ({}), called signal convertors.

The benefits of explicitly exposing generic operators include 1) it agrees with the arithmetic of not only *bit-vectors* but *integers*, and 2) it unifies unsigned and signed operators. For example, the original bit-vector multiplier, `c = a * b` in Figure 2, does *not* agree with integer arithmetic, meaning that if signals (`a`, `b`, `c`) are evaluated as integers (A, B, C), then they do *not* necessarily satisfy the relation $C = A \times B$ (here $\times$ is integer multiplication). To expose the generic multiplier in Figure 2, first we observe that the original one is unsigned[2] multiplication, so the original inputs are converted to signed generic inputs with leading zeros inserted. Then a generic output of 34 bits is created to prevent overflow. Finally the generic output is converted to the unsigned original one with some upper bits truncated. The generic multiplier, `cg = ag * bg`, agrees with the integer arithmetic by construction. This way, the original multiplier is represented by its generic version and some signal convertors on the inputs and output.

With generic operators, all same-type generic operators (e.g., multipliers) are considered to have the same function since they all agree with their integer functions (e.g. integer multiplication). This is important for uninterpreted function abstraction since uninterpreted function constraints are valid only for same-function classes.

D. Uninterpreted function constraints

The theory of uninterpreted functions (UF) states that given any *function* F with its input X, and any two instances of the same function, (x_1, f_1) and (x_2, f_2), then the Property (1) holds, stating that if the inputs are equal then the two outputs must be equal.

$$(x_1 = x_2) \Rightarrow (f_1 = f_2) \tag{1}$$

This is called a UF constraint which is simply a relation implied by any pair of the same two functions.

For Verilog, we need to be more precise about "same function" and "equal inputs". By f and g being the same function we mean that f and g are instantiations of the same generic function-type. By two signals being equal, we will mean that they are signed and bit-wise equal *after* extension. Then Property (1) holds with these modifications. Thus, *a UF constraint is valid between any pair of same function-type generic operators (even if they have different bit widths).*

Definition 4. Two signals, s_1 and s_2, are said to be *equal in Verilog* if the corresponding statement, `s1 == s2`, is evaluated to 1 in Verilog.

The precise Verilog semantics for comparing two signals is as follows. It does either zero- or sign-extension for the signal with the smaller bit-width depending on their signedness. If both signals are signed, then it does sign-extension. Otherwise, zero-extension is applied. Two signals are equal if they are bit-wise equal after extension.

Definition 5. For two same function-type generic operators, $op_1 = (o_1, i_{11}, i_{12}, t)$ and $op_2 = (o_2, i_{21}, i_{22}, t)$, the UF constraint, denoted as c, is either Constraint (2) or (3).

- If t is asymmetric:

$$c \;=\; (i_{11}{==}i_{21}) \wedge (i_{12}{==}i_{22}) \Rightarrow (o_1{==}o_2) \tag{2}$$

- If t is symmetric:

$$\begin{aligned} c \;=\; & \Big((i_{11}{==}i_{21}) \wedge (i_{12}{==}i_{22}) \Rightarrow (o_1{==}o_2)\Big) \bigwedge \\ & \Big((i_{11}{==}i_{22}) \wedge (i_{12}{==}i_{21}) \Rightarrow (o_1{==}o_2)\Big) \end{aligned} \tag{3}$$

We only apply UF constraints between generic instances of same function-type operators. The constraints are created as signals first and then treated as invariant constraints to the model checking problem (see Section III-C). Thus, abstractions are created by 1) using UF constraints and 2) replacing their outputs by new primary inputs (the generic operators are "black-boxed").

Definition 6. A generic instance is said to be *black-boxed* if its output is replaced by a fresh primary input consistent with the generic's output.

Thus the new primary input is signed and has the same width as the instance output being replaced. Note that a UF constraint may be added even though the two operators involved are both white-boxed. This can still be effective as it provides a relation between operators which may not be easy to derive using bit-level operations.

[2]In Verilog, an operation is unsigned if at least one input is unsigned.

III. UFAR

In this section, the abstraction-refinement algorithm, UFAR, for solving word level model checking problems is described.

A. The algorithm

Algorithm 1 UFAR

```
Input: $M$                                   ▷ $M$: the input miter
Input: $\mathcal{S}$                         ▷ $\mathcal{S}$: the set of problematic operators
Output: status ∈ { SAT, UNSAT }
 1: $\mathcal{B} \leftarrow \mathcal{S}$     ▷ $\mathcal{B}$: the set of black-box operators
 2: $\mathcal{P} \leftarrow \emptyset$       ▷ $\mathcal{P}$: the set of UF constraints
 3: $M \leftarrow$ ExposingFunctions($M$, $\mathcal{S}$)
 4: while true do
 5:     $A \leftarrow$ CreateAbstraction($M$, $\mathcal{P}$, $\mathcal{B}$)
 6:     status, cex ← ModelChecking($A$)
 7:     if status = SAT then
 8:         if IsRealCEX($M$, cex) then
 9:             return SAT
10:         else
11:             $\Delta\mathcal{P} \leftarrow$ RefineUFPairs($A$, $\mathcal{S}$, cex)
12:             if $\Delta\mathcal{P} \neq \emptyset$ then
13:                 $\mathcal{P} \leftarrow \mathcal{P} \cup \Delta\mathcal{P}$
14:                 continue
15:             else
16:                 $\Delta\mathcal{B} \leftarrow$ RefineBlack($M$, $\mathcal{P}$, $\mathcal{B}$, cex)
17:                 $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
18:     else
19:         return UNSAT
```

Algorithm 1 provides a high level view of UFAR. It takes two inputs; one is a miter M in word-level structural Verilog and the other is $\mathcal{S}$, the set of *problematic* operators that we want to abstract (multipliers in most cases). UFAR will return SAT if a true counterexample is found; otherwise, it concludes that $M \models \mathbf{G}\neg out$ and returns UNSAT. We will prove that UFAR is a sound and complete algorithm in Section III-G.

There are two internal state sets in UFAR. The first is $\mathcal{B}$, the set of *black* operators that will be black-boxed in the abstraction. The second is $\mathcal{P}$, the set of operator pairs whose UF constraints will be added to the abstraction. Initially $\mathcal{B} = \mathcal{S}$, thereby black-boxing all problematic operators, and $\mathcal{P} = \emptyset$.

Algorithm 1 begins with the procedure of exposing generic operators (see Section III-B). It then operates in an abstraction-refinement loop (lines 4–19). Each iteration begins by creating an abstraction based on the current states of the algorithm, which will be discussed in Section III-C. The abstraction is then bit-blasted and solved by state-of-the-art bit-level engines concurrently (see Section III-D). If the solver returns UNSAT, the property is proven and UFAR terminates (line 19). Otherwise a counterexample to the abstraction (cex) exists. If cex is also a counterexample to the original miter, then the property is falsified and UFAR terminates (lines 8–9). Otherwise cex is spurious and UFAR analyzes it to refine the abstraction (lines 11–17).

Refinement is achieved in two phases. UFAR first tries to find new UF pairs that will block cex (see Section III-E). If such are found, UFAR adds them to $\mathcal{P}$ and starts a new iteration (lines 12–14). Otherwise, the second phase is started, where cex is analyzed to determine a set of critical operators ($\Delta\mathcal{B}$) that can block cex (see Section III-F). For the next iteration, UFAR will remove operators in $\Delta\mathcal{B}$ from $\mathcal{B}$ (lines 16–17) and hence these will be *white-boxed*.

B. Exposing generic operators

To expose the generic version of an operator, we modify the Verilog by inserting signed- or zero-extended signal convertors to ensure that it becomes signed and that the bit-width of its output is large enough. The procedure for each operator $op = (o, i_1, i_2, t)$ in the problematic set $\mathcal{S}$ is summarized below.

1) If one of the inputs is *unsigned*, then create zero-extension-by-1 signed signal convertors for both inputs. Denote two generic inputs as a_1 and a_2.
2) Create the generic operator $op_2 = (o_2, a_1, a_2, t)$ where o_2 is signed and has a large enough bit-width.
3) Replace the original output o with the statement `o = o2`. Note that this step creates the generic operator op_2, eliminating the original one op.

C. Creating abstractions

An abstraction (A) is created from the original miter (M), using $\mathcal{P}$ and $\mathcal{B}$, the two current states of Algorithm 1. CreateAbstraction operates in two steps:

1) For each pair $p = (op_1, op_2)$ in $\mathcal{P}$, construct a Boolean signal c as defined in UF Constraints (2) or (3). Signal $c = 1$ implies that a UF constraint is active in M between op_1 and op_2. Signal c is then treated as an invariant constraint.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, replace its output o with a fresh primary input ppi with the same signedness and bit-width, i.e. black-box it.

Note that an operator can be in a pair of $\mathcal{P}$ but not $\mathcal{B}$. For example, one benchmark contained a group of 3 multipliers where 2 UF constraints were used between them, but only one of the 3 was needed to be white-boxed for the final proof. Note also that $\mathcal{P}$ and $\mathcal{B}$ are monotone.

We claim that the model A is an abstraction of M.

Lemma 1. *Let N denote the model created after Step 1 (adding UF constraints) in* CreateAbstraction. *N and M satisfy: ($\neg out$ denotes the property)*

$$N \models \mathbf{G}\neg out \Leftrightarrow M \models \mathbf{G}\neg out.$$

Proof. Consider any constraint signal c. We have $M \models \mathbf{G}c$ since the model M satisfies any valid UF constraint. Thus,

$$M \models \mathbf{G}\neg out \Leftrightarrow M \models \mathbf{G}\neg out \wedge \mathbf{G}c$$
$$\Leftrightarrow M, \mathbf{G}c \models \mathbf{G}\neg out \Leftrightarrow N \models \mathbf{G}\neg out$$

□

Theorem 1. *The model A created by* CreateAbstraction *is an abstraction of the miter M.*

Proof. From Lemma 1, N generated by Step 1 is equisatisfiable to the miter M. In Step 2, it creates the model A by replacing some internal signals in N with fresh primary inputs, which is a known procedure for producing an abstraction. □

D. Model checking using concurrency

To verify the current abstraction at the bit level, we could use a single engine like PDR since it is efficient, sound, and complete. Also, this procedure should be parallelized to take advantage of different engines. Running a BMC engine in parallel with PDR usually finds counterexamples to the current abstraction more efficiently and thus very effective in improving the algorithm. Also, various versions (based on different implementations and parameters) of PDR and BMC complement each other.

E. Refining UF pairs

This is the first phase of refinement. Given a (spurious) counterexample cex to the abstraction, we want to find new UF pairs $\Delta\mathcal{P}$ among operators in $\mathcal{S}$ that can block cex during the next iteration. REFINEUFPAIRS operates in two steps:

1) Simulate cex on the abstraction A to derive an assignment function $\alpha : S \times \mathbb{N} \rightarrow \mathbb{Z}$ that maps every signal in A at each time frame to a concrete value.
2) Identify pairs that violate UF constraints and add them to $\Delta\mathcal{P}$. For each time frame t and every pair of operators $(op_1, op_2) : op_1, op_2 \in \mathcal{S}, op_1 \neq op_2$, if the values of the inputs are equal but the outputs are different (Formula 4), then add (op_1, op_2) to $\Delta\mathcal{P}$. Note that we consider both input orders for symmetric operators although this is not shown in Formula 4 for simplicity.

$$(\alpha(i_{11}, t) = \alpha(i_{21}, t) \wedge \alpha(i_{12}, t) = \alpha(i_{22}, t)) \bigwedge \alpha(o_1, t) \neq \alpha(o_2, t) \tag{4}$$

Next, we discuss an upper bound for the size of $\mathcal{P}$.

Theorem 2. *The size of $\mathcal{P}$ in Algorithm 1 is bounded by $|\mathcal{S}|(|\mathcal{S}| - 1)$.*

Proof. Consider the worst case where the operators in $\mathcal{S}$ are all symmetric, then there are $\binom{|\mathcal{S}|}{2}$ pairs of operators with 2 possible permutations of binary inputs. Hence the number of pairs in the algorithm cannot exceed $|\mathcal{S}|(|\mathcal{S}| - 1)$. □

F. Refining black operators

In the second phase of refinement, we want to identify a subset of operators $\Delta\mathcal{B}$ in $\mathcal{B}$ such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, cex will be blocked for the next iteration. We call the procedure of removing elements from $\mathcal{B}$ *white-boxing* and the operators in $\mathcal{S} \setminus \mathcal{B}$ *white boxes*.

A straightforward way of identifying $\Delta\mathcal{B}$ is to simulate cex on the abstraction A and collect those operators in $\mathcal{B}$ that have input-output values inconsistent with their white-box values. However, this approach often finds an overly large $\Delta\mathcal{B}$, resulting in an unnecessarily large abstraction in the next round. Hence, we propose a *proof-based* approach that often obtains a much smaller $\Delta\mathcal{B}$.

The main idea is that if cex is spurious, then the BMC Formula (5) is UNSAT. Here the functions $\beta(i, t)$ and $\beta(s, t)$ denote the assignment of input i or state s at time t derived from cex being simulated on the original miter M, k is the depth of cex, and out is the miter signal.

$$I_M(\beta(s, 0)) \wedge \bigwedge_{t=0}^{k-1} T_M(\beta(i, t), \beta(s, t), \beta(s, t + 1)) \wedge \bigvee_{t=0}^{k} out(\beta(i, t), \beta(s, t)) \tag{5}$$

Next, multiplexers are introduced to select between the concrete version (white-box) and the abstracted version (black-box) of an operator. If assumptions are made such that all the concrete ones are selected initially, then the resulting BMC formula would still be UNSAT and a modern SAT solver like MiniSat [15] will return a subset of the assumptions that is sufficient for UNSAT. This is a variation of finding an *unsat core* and the subset returned is our candidate for $\Delta\mathcal{B}$.

The procedure REFINEBLACK operates in five steps.

1) For each pair in $\mathcal{P}$, construct a UF constraint signal and treat it as an invariant constraint on M.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, introduce two fresh primary inputs, sel and ppi, where sel is a Boolean signal and ppi a bit-vector signal which is consistent with the output o_{gen} of the associated generic operator. Replace o_{gen} with $o'_{gen} = ITE(sel, o_{gen}, ppi)$ where ITE is the *if-then-else* operator. Depending on the value of sel, either the concrete operator (o_{gen}) or the abstracted one (ppi) flows to the new output o'_{gen}.
3) Denote the model created in Step 2 by N and unroll it with the values of cex plugged in, and keep sel and ppi as the remaining primary inputs. The cex values plugged in are initial states and PIs at each time frame, denoted by $\gamma(s, 0)$ and $\gamma(i, t)$ respectively.
4) Solve the BMC query (6), which is guaranteed to be UNSAT. Note that γ is the assignment function of cex, X_t is the set of sel input signals at time t, PPI_t is the set of ppi input signals at time t, and x_{tn} is the sel signal for the n-th operator at time t. By propagating $x_{tn} = 1$ for all t and n, the query (6) is reduced to (5) by construction ($sel = 1$ means that the concrete version is chosen).

$$I_N(\gamma(s, 0)) \wedge \bigwedge_{t=0}^{k-1} T_N(\gamma(i, t), X_t, PPI_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(\gamma(i, t), X_t, PPI_t, s_t) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=0}^{|X_t|} x_{tn} \tag{6}$$

5) Derive a subset ΔX of X using the assumption interface of a modern SAT solver, and determine $\Delta\mathcal{B}$ from ΔX.

Theorem 3. *The set $\Delta\mathcal{B}$ found by* RefineBlack *is not empty ($|\Delta\mathcal{B}| > 0$).*

Proof. $|\Delta\mathcal{B}| = 0$ would mean that the formula (6) is SAT, which contradicts with the fact that cex is spurious. □

G. Analysis of the algorithm

Theorem 4. *Algorithm 1 is sound and complete.*

Proof. (sketch) Algorithm 1 is sound because it returns UNSAT only if the model A satisfies $\mathbf{G}\neg out$, which implies $M \models \mathbf{G}\neg out$ from Theorem 1. As for the completeness, the algorithm returns SAT only if a counterexample is real (line 8–9). Convergence follows because for each iteration (line 4–19), the following statements are true.

- $\mathcal{B}$ and $\mathcal{P}$ are monotone. Either $\mathcal{P}$ becomes strictly bigger (line 12–13) or $\mathcal{B}$ becomes strictly smaller (Theorem 3).
- $|\mathcal{P}|$ is upper bounded by $|\mathcal{S}|(|\mathcal{S}| - 1)$ (Theorem 2) and $|\mathcal{B}|$ is lower bounded by 0 (empty set of black boxes).

Therefore the iteration must terminate implying that a definitive answer must have been found. □

IV. Optimization

In this section, we introduce two optimizations (counterexample minimization, and random simulation), each of which improves the basic version of UFAR, Algorithm 1.

A. Minimizing counterexamples

A counterexample can be minimized [20] in the sense that some inputs can be assigned as X (don't care), but the counterexample is still valid after ternary simulation. This way, the number of concrete assignments is minimized.

The main advantage of using minimized counterexamples is that Procedure RefineUFPairs in Algorithm 1 can return potentially fewer, but higher-quality pairs of constraints. This is done by modifying the condition (Formula 4) for identifying and adding a UF constraint, where we check if the inputs are equal and the outputs are different under concrete assignments. With minimized counterexamples, Xs might appear on the outputs of black-box operators (unconstrained pseudo primary inputs). We strengthen the condition by considering only *incompatible* outputs with X assignments. Two assignments are said to be *incompatible* if they have opposite values at some bit position, and *compatible* if they do not. For example, the assignments `XX01` and `X000` are incompatible while `10XX` and `100X` are compatible. With this strengthening, pairs that satisfy Formula 4 under concrete assignments might violate the new condition since their outputs become compatible after the minimization. For example, consider two operators with concrete assignments (o, in_1, in_2), (`0011`, `01`, `10`) and (`0101`, `01`, `10`), which satisfies Formula 4. After the minimization, if the assignments become (`0XX1`, `01`, `10`) and (`XXX1`, `01`, `10`), then the pair will not be added as UF constraints since it violates the strengthened condition with compatible outputs. Thus, it is likely that fewer constraints are added. Also, the constraints we drop are lower-quality in the sense that if they are added, then UFAR will still get similar counterexamples.

B. Performing random simulation

UFAR in Algorithm 1 only finds and applies UF constraints when a counterexample (CEX) is found. However, the CEX returned by a verification engine may not be unique. If UFAR were to get a different CEX, then it might find and apply a different set of UF constraints. This inherent randomness of counterexamples could cause UFAR to take a path where more white boxes are needed for a proof. Thus, random simulation is applied on the original miter to find candidates for "good" UF constraints. The idea is that if a UF constraint is useful for the final proof, then the corresponding pair of operators must be related in some way. This means that for some execution traces they would have identical input assignments.

The procedure of random simulation operates in 2 steps.

1) Determine the parameters: the number of patterns and the number of time frames. Run random simulation on the original miter.
2) For each time frame and for each pair of same function-type generic operators, count the number of times identical input patterns occur.

A threshold is then set for determining what are good candidates of UF constraints (a pair is considered good if its count is above the threshold). A threshold should be chosen carefully since there is a trade-off between the number and the quality of constraints; a lower threshold increases the chances of getting higher-quality UF constraints (in the sense that it is more difficult to find them with counterexamples), but a lower threshold also leads to a larger number of constraints.

V. The UFAR framework

UFAR involves an iteration of abstraction and refinement between two types of representations,

1) AIGs (bit-level circuit), and
2) an internal netlist format called WLC (word-level circuit), a new development in ABC [9] to represent word-level designs.

This capability includes 1) a very fast Verilog based bit-blaster, using Verilog semantics of the WLC box operators, to translate into an AIG, and 2) a duplication-based method to create different WLC netlists at the word level. These developments are critical in making UFAR efficient, to the extent that UFAR run-time is dominated by the SAT solving in the bit-level model checker.

A. Bit-blasting WLC with Verilog semantics

The framework starts with reading in a structural Verilog miter representing the model checking problem. This is translated into a WLC netlist (WLCm) using ABC's structural Verilog parser. Next, the generic operators of all designated "problematic" operators are exposed by creating a new WLC netlist, denoted as WLCg. More details of creating a new WLC netlist are described in the next subsection. It is important to note that WLCg needs to be created only once during the entire flow and represents the fully concretized problem. This is bit-blasted into an AIG, denoted by AIGg to be used later.

The next step is to create a WLC netlist, WLCa, for the current abstraction using WLCg and the state sets $\mathcal{P}$ and $\mathcal{B}$. WLCa is bit-blasted into an AIG, denoted as AIGa. During this, Verilog semantics are used to faithfully interpret the box operators of WLC netlists.

Typically the model checker, applied to AIGa, returns a counterexample which is simulated on AIGg to see if it is spurious. If so, the counterexample is first minimized, using AIGg as reference. This is analyzed to decide the state changes to $\mathcal{P}$ and $\mathcal{B}$, which will be used to block this counterexample. These are implemented by creating a new WLCa from WLCg and the current state sets. Then the next iteration proceeds.

B. *Creating abstractions WLCa*

In the iteration in the previous section, the next abstraction is constructed as a WLC netlist using inputs $\mathcal{P}$ and $\mathcal{B}$ and WLCg. This is achieved by constructing one intermediate netlist (WLCp) and the final netlist (WLCa). To activate the UF constraints in $\mathcal{P}$, WLCp is created by duplicating WLCg but attaching the UF constraints in $\mathcal{P}$ to the appropriate signals. The boxes listed in $\mathcal{B}$ need to be made black, so the outputs of each such box need to be replaced by new PIs. WLCa is built by duplicating WLCp but with the outputs of the boxes in $\mathcal{B}$ replaced by the new PIs.

VI. Experimental Results

In this section, we present the experimental results of our implementation of UFAR with different optimization methods enabled. The implementation is based on ABC [9] using its latest improvements to Verilog parsing and bit-blasting.

We ran UFAR on a set of 2492 industrial word-level Verilog designs that were synthesized by an industrial tool to be cycle-accurate with the original circuit. Multipliers are the targeted problematic operators for UFAR to abstract. All experiments were performed on a workstation of Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM.

Comparing our results against publicly available verification tools is difficult. To our knowledge, no tools exist that can parse such designs directly without requiring a major modification[3]. Also, there is no standard format for sequential word level circuits, as there is for the combinational case with SMT-LIB [3]. Therefore we compared results of running a) `super_prove` [8] on bit-blasted designs against b) three UFAR versions with different optimization settings.

For `super_prove`, we simply bit-blasted an input miter and immediately called `super_prove` to solve it. For UFAR we used three versions in this comparison:

- `opt1` means the basic version.
- `opt2` means `opt1` plus counterexample minimization.
- `opt3` means `opt2` plus random simulation.

For all UFAR versions, four bit-level verification engines were run in parallel, 3 variants of PDR and one BMC implementation. BMC is much more efficient at finding counterexamples, while the 3 versions of PDR in combination are efficient at proving a problem UNSAT.

[3]Ebmc [18] cannot handle parameterized modules or functions/tasks in Verilog. VCEGAR [17] has a more limited front-end than the one in Ebmc.

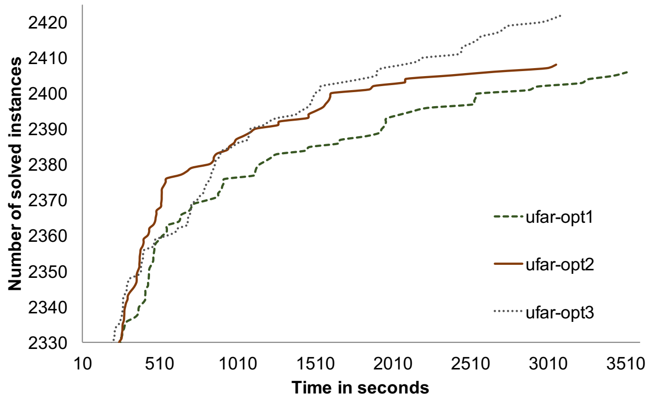


Fig. 3: Comparison of UFAR variants.

`super_prove`	ufar-opt1	ufar-opt2	ufar-opt3
2115	2398	2408	2422

TABLE I: The numbers of solved instances using different settings. 70 instances remain unsolved.

We present the results in Figure 3, where the horizontal axis represents wall-clock time and the vertical axis represents the cumulative number of solved instances. A time-out of 1 hour was enforced for each example. The result of `super_prove` is not shown in Figure 3 because its number of solved instances is 2115, well below the bottom scale of 2330. The `opt2` version is slightly better than `opt1` because the counterexample minimization prevents UFAR from applying too many constraints. The `opt3` version works best because the random simulation finds important UF constraints that can be missed by counterexamples. All solved instances are *unsat*.

Table I shows the numbers of instances finally solved by all versions within the 1-hour time-out. The three versions of UFAR outperform `super_prove`, which is often ineffective in solving problems with many arithmetic operators.

We selected 19 out of 2492 designs to present more detailed results in Table II. The selection is somewhat arbitrary but it does represent designs that are dissimilar and gives an idea of expected ranges of iterations needed, UF constraints used, and white box operators in the final abstractions. We observe the following from Table II.

1) UFAR proves most cases with a relatively small number of white-box multipliers.
2) The number and quality of UF constraints are two important factors of performance. If the number is large, then UFAR generally needs more time to run, which is why counterexample-based constraint reduction is important. If the quality is good, then UFAR may prove a problem with fewer white boxes (or none). This supports the using of the random simulation to find good constraints.
3) It takes a nontrivial number of refinements for UFAR

					`sp`	ufar-opt1		ufar-opt2		ufar-opt3	
Design	#Mults	#AIGs	#FFs	Result	Time	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$
1	60	187303	2608	unsat	306.71	173.54	1/0/364/0	1127.1	3/0/12/0	3.39	1/0/874/0
2	11	72003	358	unsat			2/1/24/9	2661.2	3/1/30/9	1209.7	1/1/6/9
3	16	115888	550	unsat		20.75	2/0/37/0		3/1/17/10	18.69	1/0/12/0
4	12	49948	819	unsat		4.01	4/0/26/0	4.47	4/0/18/0	1.33	1/0/4/0
5	102	104272	1524	unsat		60.65	3/0/641/0	65.56	6/0/82/0	98.18	2/0/1252/0
6	144	161721	2456	unsat		1225.8	4/0/2366/0	2085.1	12/0/57/0	843.09	1/0/352/0
7	8	192143	5825	unsat		725.36	1/0/14/0	488.99	2/0/4/0	833.42	2/0/11/0
8	14	21092	400	unsat		474.51	3/1/30/10	316.95	3/1/28/8	272.7	2/1/33/8
9	32	46239	972	unsat	8.93		7/3/315/5		3/2/16/4	2428.6	3/3/276/9
10	43	302277	4065	unsat		157.14	3/0/597/0	106.83	4/0/40/0	66.38	1/0/447/0
11	4	25355	1552	unsat	156.16	1.49	1/0/2/0	1.10	1/0/2/0	0.99	1/0/2/0
12	15	49718	1707	unsat		25.33	3/1/40/7	48.46	8/1/34/7	24.93	3/1/52/7
13	21	65892	1997	unsat		40.18	2/1/154/7	50.42	5/1/40/9	102.24	6/1/90/7
14	223	292183	2649	unsat		2548.9	41/8/2398/50		34/9/674/55	1670.8	7/5/1401/37
15	63	91259	875	unsat		1967.4	10/4/915/29	517.86	10/4/142/37	2457.5	3/5/374/35
16	15	184859	4785	unsat		2939.8	1/1/68/4	535.18	2/1/7/3	2169.9	1/1/73/3
17	216	128137	1661	unsat		2231.2	4/0/1107/0		16/0/1943/0	401.76	1/0/394/0
18	253	199466	3751	unsat			118/0/23825/0	6.15	5/2/78/5	686.49	82/2/8638/3
19	475	274801	4204	unsat		470.71	5/0/10556/0	150.18	8/0/172/0	158.99	1/0/268/0

TABLE II: Detailed results of 19 unsat designs. The #Mults/#AIGs/#FFs means the number of multipliers/bit-level AIG nodes/bit-level flip flops. The $i_p/i_w/n_p/n_w$ means the number of iterations of applying new UF constraints/iterations of applying new white boxes/total UF constraints/total white boxes.

to converge, implying that UFAR builds up abstractions gradually. A major challenge is to figure out how to strike a good balance between the number and quality of UF constraints and the number of white boxes needed.

4) The main effect of counterexample minimization seems to be to make the overall algorithm more efficient but solves only 2 additional benchmarks.
5) Random simulation made UFAR faster and helped solve 4 more benchmarks.

VII. Conclusion and Future Work

UFAR is an algorithm that abstracts (black-boxes) all problematic operators up front and refines them by applying UF constraints and/or white-boxing. We presented two optimization techniques for UFAR. We demonstrated UFAR's scalability on a large set of industrial problems.

For future work, we would like to understand a few of the anomalies in Table II (e.g., Designs 15 and 18) where an optimization caused quite a large slow-down in the solving. We also want to experiment on the 70 remaining unsolved benchmarks to find additional techniques to solve more problems. We plan to extend UFAR to use UF constraints across time frames and to perform refinement more gradually. For example, instead of white-boxing an entire operator, we might *grey-box* it. Last, we plan to integrate modern SMT solvers to investigate possible advantages in this setting.

VIII. Acknowledgements

This work was supported in part by SRC contract 2265.001 as well as NSA under the TRUST project. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific. for their continued support.

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] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. of DAC'04*.
[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.
[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS'99*.
[5] A. R. Bradley. Sat-based model checking without unrolling. In *Proc. of VMCAI'11*.
[6] B. A. Brady, R. E. Bryant, and S. A. Seshia. Learning conditional abstractions. In *Proc. of FMCAD'11*.
[7] 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*.
[8] R. Brayton, N. Een, and A. Mishchenko. Using speculation for sequential equivalence checking. In *Proc. of IWLS'12*.
[9] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Proc. of CAV'10*.
[10] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proc. of TACAS'09*.
[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of LICS'90*.
[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*.
[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS'08*.
[14] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proc. of FMCAD'11*.
[15] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*.
[16] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proc. of CAV'97*.
[17] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In *Proc. of DAC'05*.
[18] D. Kroening and M. Purandare. Ebmc: The enhanced bounded model checker. `www.cprover.org/ebmc`.
[19] K. L. McMillan. Interpolation and sat-based model checking. In *Proc. of CAV'03*.
[20] A. Mishchenko, N. Eén, and R. Brayton. A toolbox for counter-example analysis and optimization. In *Proc. of IWLS'13*.
[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of FMCAD'00*.

Optimizing Horn Solvers for Network Repair

Hossein Hojjat*, Philipp Rümmer†, Jedidiah McClurg‡, Pavol Černý‡, and Nate Foster*
* Cornell University, USA † Uppsala University, Sweden ‡ CU Boulder, USA
{hojjat,jnfoster}@cs.cornell.edu philipp.ruemmer@it.uu.se {jedidiah.mcclurg,pavol.cerny}@colorado.edu

***Abstract*—Automatic program repair modifies a faulty program to make it correct with respect to a specification. Previous approaches have typically been restricted to specific programming languages and a fixed set of syntactical mutation techniques—e.g., changing the conditions of *if* statements. We present a more general technique based on repairing sets of unsolvable Horn clauses. Working with Horn clauses enables repairing programs from many different source languages, but also introduces challenges, such as navigating the large space of possible repairs. We propose a conservative semantic repair technique that only removes incorrect behaviors and does not introduce new behaviors. Our proposed framework allows the user to request the *best* repairs—it constructs an optimization lattice representing the space of possible repairs, and uses a novel local search technique that exploits heuristics to avoid searching through sub-lattices with no feasible repairs. To illustrate the applicability of our approach, we apply it to problems in software-defined networking (SDN), and illustrate how it is able to help network operators fix buggy configurations by properly filtering undesired traffic. We show that interval and Boolean lattices are effective choices of optimization lattices in this domain, and we enable optimization objectives such as modifying the minimal number of switches. We have implemented a prototype repair tool, and present preliminary experimental results on several benchmarks using real topologies and realistic repair scenarios in data centers and congested networks.**

I. Introduction

Program repair is a promising approach to software development that synthesizes a modification to a faulty system to make verification succeed. A number of approaches have been explored in the literature including deductive program repair [1] and automatic patch generation [2], but these often have several limitations.

1) They target *specific types of programs*—e.g., repairing functional Scala programs, or patching PHP programs to make them pass a test suite.
2) They search for *specific types of repairs*—e.g., finding syntactically similar programs by swapping arguments to functions, or modifying the conditions on *if* statements by conjoining (or disjoining) additional conditions.
3) In general, they are not able to find repairs that are optimal with respect to a given objective function.

This paper develops a general approach to the program repair problem. Rather than developing tools customized for specific languages, we utilize a general modeling framework that can be used to encode a wide variety of software artifacts. Additionally, rather than examining specific types of repairs, we explore the space of all possible repairs, and develop techniques for doing this efficiently. Importantly, our tool also has the ability to search for optimal repairs, specified using a domain-specific objective function.

Our approach is based *Horn clauses*—a general framework that is able to model a wide variety of systems and has scalable algorithms and verification tools [3], [4], [5]. In order to use the framework in the context of program repair, we formulate the *Horn clause repair problem*: given a set of Horn clauses that violates a safety invariant, our goal is to produce a repaired set of clauses where the repair is *optimal* with respect to a domain-specific objective function. To find the optimal repair, we must search through a large (in fact, potentially infinite) space of Horn clause repairs. To do this, we construct a finite lattice that abstracts the space of possible repairs—e.g., using *Boolean* and *interval* lattices. Our algorithm for solving Horn-clause optimization problems over finite lattices combines ideas from local search with conflict-driven learning (inspired by SAT and SMT solvers) to prune parts of the optimization lattice that are guaranteed to not contain solutions.

To evaluate our approach, we show how it can be used to solve a variety of real-world problems in the domain of software-defined networking (SDN). To apply our techniques in a given domain, we need a user-defined mapping from the source language to Horn clauses (and vice-versa), and an objective function that specifies in what sense a repair is optimal. In SDN, a *configuration* consists of tables of packet-forwarding rules of the individual switches in the network. Network configurations often contain bugs—e.g., due to loops, black-holes, or access-control violations [6]. We model network configurations using Horn clauses and use an objective function that minimizes the number of switches whose configuration is modified by the repair. We show that our repair framework is able to produce optimal repairs of realistic network configurations efficiently.

II. Motivating Example

The network shown in Fig. 1(a) corresponds to a topology commonly used in large data centers—switches are grouped into three layers: core, aggregation, and ToR (top-of-rack) switches. During normal operation, packets are forwarded from a host upward through aggregation and core switches, and then back downward to the destination host. Although there are physical loops in this network a packet should take only a finite number of hops in any configuration.

In this example, the data center configuration provides service to multiple tenants: hosts H_1, H_2, and H_3 belong to one customer, and H_4 belongs to another customer. Host H_1 sends traffic to H_2 and H_3, but this traffic should not reach H_4, as

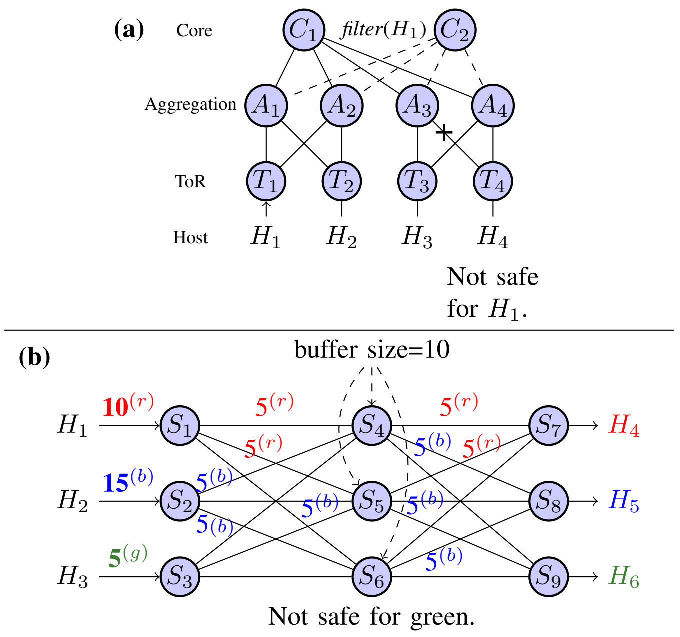


Fig. 1: a) Repair in data center, b) Repair w.r.t. bandwidth and queue sizes.

it is not owned by the customer. To implement this policy, the operator might install a forwarding rule at C_1 to filter packets from H_1 going towards A_4 and also disable the link $A_3 - T_4$ for good measure (in the figure, this disabled link is indicated by a "+" symbol.) Now assume that the network operator has brought the core switch C_2 down for maintenance—i.e., the dashed links cannot be traversed by any packet. After the maintenance task has been completed, the network operator decides to bring up C_2 to help balance load within the data center. Unfortunately, this causes the safety requirement to be violated, since there is a new path that forwards H_1 traffic to H_4. Our repair framework interactively helps the network operator bring the network back to safety. The operator can provide as input (i) a description of the network as a high-level transition system, and (ii) a set of required safety properties. Our tool then synthesizes a set of possible repairs which returns the system to safety. As a first solution, the repair engine might suggest that we either disconnect the links $A_1 - C_2$ and $A_2 - C_2$, or take C_2 offline and return the network to its initial state. The network operator could reject this "trivial" repair by stipulating that any repair must not disconnect links. The repair engine might also suggest solutions that rewrite the traffic from H_1 to another type of traffic by modifying packet headers, and the network operator could reject such solutions by stipulating that the repair engine must not modify headers. After providing such restrictions, our tool returns a solution in which filters for H_1 traffic have been added on a number of links: $\{A_1 - C_2, A_2 - C_2\}$, $\{C_2 - A_4, A_4 - T_4\}$, $\{A_4 - T_4\}$, etc. Our framework uses objective (ranking) functions to guide the repair engine to the "best" answers. For example, the network operator might be interested in solutions that modify the configurations on the smallest number of switches. By providing a suitable objective function, our tool can find an optimal correct solutions—e.g., adding a single filter on the link $C_2 - A_4$ or on the link $A_4 - T_4$.

Another important class of network configuration repairs is related to quantitative measures of bandwidth and traffic. As an example, consider Fig. 1(b) and suppose that each intermediate node can buffer at most 10 units of traffic. The hosts H$_1$, H$_2$, H$_3$ on the left receive 10 units of "red" traffic, 15 units of "blue" traffic, and 5 units of "green" traffic respectively. Red traffic should be sent to H_4, blue traffic to H_5, and green traffic to H_6. In addition, the green traffic must not traverse the intermediate node S_6. Initially, the network operator decides to send 5 units of red traffic to each of S_4 and S_5. She also decides to send 5 units of blue traffic to each of S_4, S_5, and S_6. Unfortunately, this configuration does not allow the green traffic to reach its destination since it cannot flow through S_6, and the buffers of S_4 and S_5 are already full. A correct repair might shift some of red or blue traffic (or both) to S_6 to make room for the green traffic to pass through S_4 or S_5. Our repair engine might generate a solution that sends all green traffic to S_4, and allows the red and blue traffic to be arbitrary divided between S_4, S_5, and S_6, provided the total amount of traffic does not exceed the buffer capacity.

III. Basic Definitions

a) Constraint languages: Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set Σ_f of fixed-arity function symbols, and a set Σ_p of fixed-arity predicate symbols. The interpretation of Σ_f and Σ_p is determined by a fixed structure (U, I), consisting of a non-empty universe U, and a mapping I that assigns to each function in Σ_f a set-theoretic function over U, and to each predicate in Σ_p a set-theoretic relation over U. As a convention, we assume the presence of an equality symbol "=" in Σ_p, with the usual interpretation. Given a set X of variables, a *constraint language* is a set *Constr* of first-order formulae over Σ_f, Σ_p, X. For example, the language of quantifier-free Presburger arithmetic (mainly used in this paper) has $\Sigma_f = \{+, -, 0, 1, 2, \ldots\}$ and $\Sigma_p = \{=, \leq, |\}$, with the usual semantics.

b) Horn Clauses: We consider a set R of uninterpreted fixed-arity relation symbols. The arity of a symbol $p \in R$ is denoted by $\alpha(p)$. A *Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$, where C is a constraint over Σ_f, Σ_p, X; each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in R$ to first-order terms over Σ_f, X; and H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in R$ to first-order terms, or *false*.

H is called the *head* of the clause, and $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = \mathit{true}$, we usually omit C and just write $H \leftarrow B_1 \wedge \cdots \wedge B_n$. First-order variables in a clause are implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in S$. Notions like (un)satisfiability and entailment generalize to formulae with relation symbols.

Definition 3.1: Let *HC* be a set of Horn clauses over relation symbols R. *HC* is called *(semantically) solvable* (in the structure (U, I)) if there is an interpretation σ of the relation

Algorithm 1: Generalize Procedure

```
Input: Unsolvable Horn clauses $HC$
Result: Solvable Horn clauses $HC$
1 $ok := \mathit{false}$;
2 while $\neg ok$ do
3     $(ok, \mathit{CEX}) := \textsc{Solve}(HC)$;
4     pick $\mathit{CEX}' \subseteq \mathit{CEX}$;  $HC := (HC \setminus \mathit{CEX}')$;
5     if $\neg ok$ then
6         for $h := (H \leftarrow C \wedge \bigwedge_j B_j) \in \mathit{CEX}'$ do
7             $m := \mathit{fresh\_symbol}$;
8             $HC := HC \cup (H \leftarrow C \wedge m \wedge \bigwedge_j B_j)$;
```

symbols R as set-theoretic relations such that the universal closure $Cl_{\forall}(h)$ of every clause $h \in HC$ holds in (U, I), denoted by $\sigma \models HC$; in other words, if the structure (U, I) can be extended to a model of the clauses HC.

We can practically check solvability of sets of Horn clauses by means of *predicate abstraction* [7], [8], using tools like Z3 [9], HSF [7], or Eldarica [5].

IV. Horn-Clause Repair

This section defines the Horn clause repair problem and presents our conservative approach to solving it.

Definition 4.1 (Repair): Let HC be a set of Horn clauses and ϕ a safety invariant, encoded as a Horn clause h_ϕ. Now assume that HC violates the safety invariant—i.e., $HC \cup \{h_\phi\}$ is unsolvable. The set HC' is a repair of HC if (i) $HC' \cup \{h_\phi\}$ is solvable, and (ii) the models I of the first-order variables in HC are a superset of the models I' for HC'.

Given a set of unsolvable Horn clauses, there can be many different strategies for repairing them—i.e., to make the clauses solvable—but it is important that we be able to map repairs back into the problem domain. As an example, in our case studies, we will interested in converting suggested repairs from Horn clauses back to network configurations. The relation symbols in the Horn clause representation will have a specific meaning in the problem domain (e.g., position of the packets or the distribution of traffic in network), and the clauses will have a specific meaning (e.g., forwarding across links). Our repair procedure is conservative in the sense that it does not add clauses, remove clauses, or change the structure of the relation symbols. This makes the translation of repairs back to the problem domain easy—we merely add constraints to the bodies of the clauses to make the clauses more constrained with the goal of removing bad behaviors. We show that this kind of repair corresponds to adding filters or packet-processing rules to switches, and we argue in Section VII that this strategy is not restrictive in the networking domain.

The generalization procedure in Algorithm 1 removes counterexamples to a set of Horn clauses by adding fresh relation symbols to the bodies of a subset (CEX') of the clauses that constitute the counterexample (CEX). The arguments to the fresh relation symbol m are either determined by the problem domain, or use all of the arguments from the existing relation symbols in the head and body of the clause. Algorithm 1 removes every counterexample so that the **while** loop eventually terminates. In the worst case, it conjoins fresh relation symbols to the bodies of all clauses. The fresh relation symbol added to the body of each clause are trivially satisfiable, since the symbols can be set to *false*. However, our Horn optimization problem attempts to synthesize more interesting solutions.

V. Horn-Clause Repair Optimization

We now develop a general framework for formulating and solving optimization problems subject to Horn constraints. The framework is a good match for a range of analysis and synthesis tasks, and in particular, for the purpose of repairing networks. In this setting, side conditions in the form of Horn clauses are used to represent the network, its desired correctness properties, and the space of possible network repairs, while the optimization objective captures preferences about the generated repair—e.g., the smallest number of switches should be updated. Since multiple incomparable solutions may exist in general, we arrange the search space as a lattice.

Definition 5.1 (Optimization lattice): Suppose again that R is a set of uninterpreted fixed-arity relation symbols, and that

$$S_R = \{\sigma : R \to \mathcal{P}(U^*) \mid \sigma(p) \subseteq U^{\alpha(p)}\}$$

is the space of possible interpretations of the R symbols as set-theoretic relations over the universe U. An *optimization lattice* is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a mapping $\mu : L \to \mathcal{P}(S_R)$ from elements of $\langle L, \sqsubseteq_L \rangle$ to sets of interpretations of the R symbols, such that:

1) the bottom element is mapped to $\mu(\bot) = S_R$, the set of all interpretations; and
2) μ is anti-monotonic, i.e., $a \sqsubseteq_L b$ implies $\mu(a) \supseteq \mu(b)$.

The lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ is Horn-definable if there is a function π mapping elements $l \in L$ to finite sets $\pi(l)$ of Horn clauses over relation symbols $R \cup R'$, such that $\mu(l) = \{\sigma|_R \mid \sigma \models \pi(l)\}$ for every $l \in L$.

Given a set HC of Horn clauses, we call a lattice element $l \in L$ *feasible* if there is an interpretation $\sigma \in \mu(l)$ with $\sigma \models HC$; in other words, if the clauses are satisfied by some interpretation associated with l. Since μ is anti-monotonic, feasibility is an anti-monotonic predicate on optimization lattices as well: if a node is infeasible, all of its successors are also infeasible. An element $l \in L$ is *maximal feasible* if l is feasible, but all of its successors are infeasible.

Definition 5.2: A *Horn optimization problem* is defined by a set HC of Horn clauses over relation symbols R, an optimization lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ over R, and a monotonic function $\mathit{obj} : L \to D$ to a totally ordered domain D. A solution is a lattice element $l_{max} \in L$ such that

1) l_{max} is maximal feasible for HC; and
2) $\mathit{obj}(l_{max}) = \max\{\mathit{obj}(l) \mid l \in L \text{ is feasible for } HC\}$.

Example 5.1: Consider the topology shown in Fig. 2 and suppose we want to implement IP multicast from H to I_1 and I_2 with TTL scoping. As background, the TTL (time-to-live) field is initialized to a default value (e.g., 64) and is

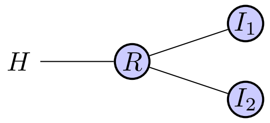


Fig. 2: Multicast router with TTL scoping.

decremented at every hop. Packets with TTL 0 are dropped, which prevents forwarding loops. In TTL scoping, the operator assigns a TTL threshold to each output port on all multicast routers. The routers only forward packets whose TTL value is greater than or equal to the configured threshold. In this example, we will consider a stronger version of TTL scoping with upper and lower bounds. To represent multicasting of a packet to the hosts I_1 and I_2 using Horn clauses, we assign relation symbols R to the router, and I_1, I_2 to the destination hosts (Section VI shows how to encode networks as Horn clauses). Now suppose the network operator wants to disable multicasting by allowing only traffic to I_1 or I_2 (but not both) by adding a filter on TTL values for traffic coming from H. Representing the newly added filter using the relation symbol f, we obtain the following Horn clauses:

$$\begin{aligned} &R(t) \leftarrow f(t) \\ &I_1(t') \leftarrow R(t) \wedge (t' = t-1) \wedge (t' \geq 3) \\ &I_2(t') \leftarrow R(t) \wedge (t' = t-1) \wedge (1 \leq t' \leq 2) \end{aligned}$$

The safety specification is $\mathit{false} \leftarrow I_1(t) \wedge I_2(t')$.

A. Optimization in Boolean Lattices

We discuss two lattices that are frequently useful for defining Horn optimization problems: Boolean lattices, defined as the powerset lattice of some finite set, and interval lattices, which can capture value or address ranges to be enabled or blocked in network repair problems (Sect. V-B). One can construct more complicated optimization lattices—e.g., by taking the Cartesian product of lattices.

We first consider powerset lattices $\langle \mathcal{P}(B), \subseteq \rangle$ of some finite base set B. The bottom element of such a lattice is the empty set $\emptyset$, while the top element is the full set B. This kind of lattice is useful for modeling optimization problems of discrete character, and also covers (weighted) first-order Max Horn SAT problems—i.e., the problem of satisfying a maximum subset of some set of Horn constraints [10].

To convert $\langle \mathcal{P}(B), \subseteq \rangle$ into an optimization lattice, a mapping π_B from $\mathcal{P}(B)$ to sets of Horn clauses can be defined as a homomorphism $\pi_B(A) = \bigcup_{x \in A} \pi_B(x)$, given a π_B that maps every element of B to a (finite) set of Horn clauses. In other words, every element $x \in B$ is responsible for enabling some Horn constraints. The mapping π_B induces an anti-monotonic mapping $\mu_B(A) = \{\sigma \mid \sigma \models \pi_B(A)\}$ to sets of interpretations, and an optimization lattice $(\langle \mathcal{P}(B), \subseteq \rangle, \mu_B)$.

Example 5.2: Recall Example 5.1. We will show how to convert this system into a Horn optimization problem. To start, we choose a base set of clauses

$$B = \left\{ \begin{matrix} f(t) \leftarrow t < 2, \; f(t) \leftarrow t = 2, \; f(t) \leftarrow t = 3, \\ f(t) \leftarrow t = 4, \; f(t) \leftarrow t > 4 \end{matrix} \right\}$$

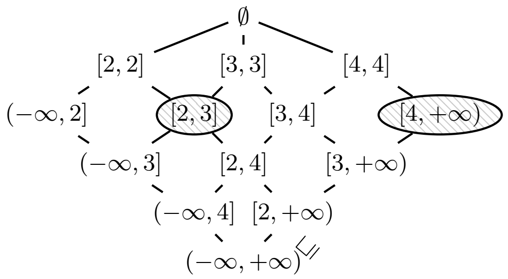


Fig. 3: Example interval lattice $\langle I_2^4, \sqsubseteq_2^4 \rangle$.

and generate a 32-element lattice $\langle \mathcal{P}(B), \subseteq \rangle$. Since each lattice element is identified with a set of Horn clauses, the mapping π_B can be defined as the identity function. Each element of B describes constraints on considered solutions of f, and maximal feasible elements correspond to solutions where f accepts as many TTL values t as possible. The maximal feasible elements are:

$$\begin{aligned} m_1 &= \{f(t) \leftarrow t < 2, \; f(t) \leftarrow t = 2, \; f(t) \leftarrow t = 3\}, \text{ and} \\ m_2 &= \{f(t) \leftarrow t < 2, \; f(t) \leftarrow t = 4, \; f(t) \leftarrow t > 4\}, \end{aligned}$$

i.e., f must filter either values $t \geq 4$, or values $t \in [2, 3]$.

B. Optimization in Interval Lattices

Boolean lattices tend to grow rapidly in practice (as in the previous example). As a more compact (though more coarse-grained) representation, lattices of *intervals* are more useful. Given integers $a, b \in \mathbb{Z}$ ($a \leq b$), we define the lattice $\langle I_a^b, \sqsubseteq_a^b \rangle$:

$$\begin{aligned} I_a^b &= \{\emptyset\} \cup \{(-\infty, \infty)\} \cup \\ &\quad \{[x, y] \mid x, y \in \mathbb{Z}, a \leq x \leq y \leq b\} \cup \\ &\quad \{(-\infty, x], [x, \infty) \mid x \in \mathbb{Z}, a \leq x \leq b\} \\ \sqsubseteq_a^b &= \{(I, J) \in I_a^b \times I_a^b \mid I \supseteq J\} \end{aligned}$$

where $[x, y], (-\infty, x]$, etc., denote non-empty intervals of integers. The bottom element of the lattice is the full interval $(-\infty, \infty) = \mathbb{Z}$, and the top element is the empty set $\emptyset$. As an example, the 14-element lattice $\langle I_2^4, \sqsubseteq_2^4 \rangle$ is given in Fig. 3.

A lattice $\langle I_a^b, \sqsubseteq_a^b \rangle$ can naturally be used to express network repairs that consist of blocking certain ranges (of packet types, addresses, ports, etc.). For instance, given a unary Horn predicate p, a mapping π_p from interval lattice elements to Horn clauses can be defined by

$$\pi_p(I) = \{p(z) \leftarrow z \notin I\} \qquad (\text{for } I \in I_a^b) \; .$$

The clause $\pi_p(I)$ implies that p holds for all values outside of the interval I, while p can be false for values within the interval.[1] As before, π_p induces an anti-monotonic mapping $\mu_p(I) = \{\sigma \mid \sigma \models \pi_p(I)\}$, and therefore gives rise to an optimization lattice $(\langle I_a^b, \sqsubseteq_a^b \rangle, \mu_p)$. Preference of some intervals over others (e.g., minimizing the lower bound of solution intervals) can be captured by adding a suitable monotonic objective function obj.

[1]For the opposite situation, constraining p to be true for all values *within* some interval, a dual lattice can be constructed in which the empty set $\emptyset$ forms the bottom element, and the full interval $(-\infty, \infty)$ is top.

Algorithm 2: Optimization Procedure

```
Input: Horn clauses $HC$, optimization lattice
       $(\langle L, \sqsubseteq_L\rangle, \mu)$, objective function $obj : L \to D$
Result: Set $Sol$ of all solutions of optimization problem
1 $Sol := \emptyset$;   $SubOpt := \emptyset$;   $B := -\infty$;
2 while there is a feasible $l \in L$ that is incomparable with
    $Sol \cup SubOpt$ do
3     $m$ or $so :=$
          $boundedMaximize(HC, (\langle L, \sqsubseteq_L\rangle, \mu), obj, l, B)$;
4     if $m$ was returned, and $obj(m) > B$ then
5         $SubOpt := SubOpt \cup Sol$;
6         $Sol := \{m\}$; $B := obj(m)$;
7     else
8         $Sol := Sol \cup \{m\}$ or $SubOpt := SubOpt \cup \{so\}$;
9 return $Sol$;
```

Example 5.3: We again use the system from Example 5.1, and the lattice $\langle I_2^4, \sqsubseteq_2^4\rangle$ in Fig. 3 as illustration. With the mapping π_f defined as in (V-B), and the Horn constraints from Example 5.1, the maximal feasible elements are $[2, 3]$ and $[4, \infty)$, which are marked in Fig. 3. Note that those solutions correspond to the ones identified in Example 5.2, but that the interval lattice is more compact than the Boolean lattice.

Since there are multiple maximal feasible elements, we can use a monotonic objective function obj to disambiguate—e.g., such a function could return the negated upper endpoint, which would express a preference for $[2, 3]$ over $[4, \infty)$:

$$obj(I) = \begin{cases} -y & \text{if } I = [x, y] \text{ or } I = (-\infty, y] \\ -\infty & \text{if } I = [x, \infty) \\ \infty & \text{if } I = \emptyset \end{cases}$$

C. Effective Optimization for Finite Lattices

We now present our algorithm for solving Horn optimization problems over finite lattices. The algorithm combines ideas from local search (e.g., [11]) with conflict-driven learning (inspired by SAT and SMT solvers) to prune parts of the optimization lattice that are guaranteed to not contain solutions. The algorithm is partly derived from an earlier search procedure for optimal Craig interpolants [12].

Example 5.4: We first illustrate the procedure using Example 5.1, and the interval lattice in Example 5.3. The two maximal feasible elements in the lattice (Fig. 3) are $[2, 3]$ and $[4, \infty)$. Interval $[2, 3]$ has cost $obj([2, 3]) = -3$, and is the optimal solution (obj from Example 5.3).

Our algorithm starts by choosing an arbitrary feasible lattice element, and then walks upward in the lattice until a maximal feasible element is reached. In the example, we can choose the bottom element $(-\infty, +\infty)$, since if any lattice element is feasible, then so is bottom; suppose that maximizing this element (walking upward as long as feasible successors exist) yields $[2, 3]$, which also happens to be the global optimum.

After identifying $[2, 3]$ as a possible solution, optimality must be verified. For this, we make the observation that every

Algorithm 3: $boundedMaximize$ $(HC, (\langle L, \sqsubseteq_L\rangle, \mu), obj, l, B)$

```
Input: Horn clauses $HC$, feasible lattice element $l \in L$,
       optimization bound $B$
Result: $m \in L$  s.t.  $l \sqsubseteq_L m$,  $m$ is maximal feasible,
                          and $obj(m) \geq B$   or
        $so \in L$  s.t.  $l \sqsubseteq_L so$,  $obj(so) < B$, and all
                          successors of $so$ are infeasible.
1  $upperBound := \top$;
2  for all immediate successors $s$ of $l$ do
3      if $s \sqsubseteq_L upperBound$ then
4          if $s$ is feasible then
5              $l := s$;   Restart loop at line 2;
6          else if $\exists b.\ feasibilityBound(l, s, b)$ then
7              $upperBound := upperBound \sqcap b$;
8              if  $obj(upperBound) < B$ then
9                  return $so := upperBound$;
10             if $upperBound$ is feasible then
11                 return $m := upperBound$;
12 if $obj(l) < B$ then  return $so := l$ ;
13 return $m := l$;
```

further solution has to be *incomparable* to $[2, 3]$, since elements above $[2, 3]$ are infeasible, and elements below are not maximal. Our procedure therefore picks an arbitrary feasible incomparable element, and then again walks upward towards a maximal feasible element. To find feasible incomparable elements, we enumerate all *minimal* incomparable elements, and check whether any of them is feasible (otherwise, no feasible incomparable element can exist). Here, the two minimal elements incomparable to $[2, 3]$ are $(-\infty, 2]$ and $[3, \infty)$, and we suppose that the latter (the feasible one) is picked.

To walk upward, we check whether $[3, \infty)$ has a feasible successor. Suppose we first consider $[3, 4]$, which turns out to be infeasible. Our algorithm utilizes this information to derive a *feasibility bound:* since $[3, \infty)$ is feasible and $[3, 4]$ infeasible, it follows that every feasible element above $[3, \infty)$ has to be below or equal to $[4, \infty)$, i.e., further search can be bounded by $[4, \infty)$. Since $obj([4, \infty)) = -\infty < obj([2, 3])$, we can conclude that no solution can possibly exist above $[3, \infty)$, and the search must backtrack. Note that it is not relevant whether $[4, \infty)$ itself is feasible.

At this point, the feasibility bound $[4, \infty)$ can be used to prune further search, since no solutions can exist above or below $[4, \infty)$. We search for further feasible elements that are incomparable to both $[2, 3]$ and $[4, \infty)$. The minimal incomparable elements are now $(-\infty, 2]$ and $[3, 4]$, both of which are infeasible. It follows that no further feasible incomparable elements exist, and that $[2, 3]$ is the (unique) solution. ■

The pseudo-code of the optimization procedure is shown in Alg. 2 and 3. The main loop in Alg. 2 maintains a set Sol of solutions, a set $SubOpt$ of blocking elements, and cost B of the best solution so far. In each iteration, Alg. 2 computes a feasible element l that is incomparable to all elements in $Sol \cup SubOpt$ (i.e., neither above nor below any element in

$Sol \cup SubOpt$, line 2), and then searches for a maximal feasible element above l using *boundedMaximize* (line 3).

To update the variable *upperBound* (line 7 in Alg. 3), the algorithm exploits the fact that a feasible lattice element l with an infeasible successor s has been found. Given a pair $l \sqsubseteq_L s$ such that l is feasible and s is infeasible, we define what it means for an element $b \in L$ to be a *feasibility bound*:

$$\mathit{feasibilityBound}(l, s, b) \equiv \begin{cases} l = s \sqcap b, \text{ and} \\ l \sqsubseteq x \text{ implies } x \sqsubseteq b \text{ for every feasible element } x \in L. \end{cases}$$

Given a feasible element l with infeasible successor s of l, the predicate *feasibilityBound* provides an upper bound b for every feasible successor of l. This allows the subsequent maximization to ignore parts of lattice that are not underneath b. Derivation of feasibility bounds is discussed in Sect. V-D.

Feasibility bounds often enable our procedure to prune away large parts of the search space. As the experiments in Sect. VIII show, the algorithm can in practice handle optimization lattices with more than 10^{30} elements, only needing to inspect a tiny fraction of the lattice to find all solutions. The procedure is furthermore an "anytime procedure," which can at any point provide (possibly sub-optimal) solutions, should time run out. The procedure is also complete:

Theorem 5.1: When applied to a finite optimization lattice, Alg. 2 terminates and returns the set of all solutions.

D. Feasibility Bounds

The predicate *feasibilityBound* can often be defined generically for a lattice $\langle L, \sqsubseteq_L\rangle$, without taking the actual set HC of clauses into account. For Boolean lattices $\langle \mathcal{P}(B), \subseteq\rangle$, correct *feasibilityBound* statements can be derived using the rule

$$\frac{x \notin A}{\mathit{feasibilityBound}(A, A \cup \{x\}, B \setminus \{x\})} .$$

For interval lattices $\langle I_a^b, \sqsubseteq_a^b\rangle$, the predicate can be defined by:

R1. $\mathit{feasibilityBound}([x, x], \emptyset, [x, x])$
R2. $\mathit{feasibilityBound}([x, y], [x + 1, y], [x, x])$
R3. $\mathit{feasibilityBound}((-\infty, y], [a, y], (-\infty, a])$
R4. $\mathit{feasibilityBound}([x, y], [x, y - 1], [y, y])$
R5. $\mathit{feasibilityBound}([x, \infty), [x, b], [b, \infty])$
R6. $\mathit{feasibilityBound}([x, \infty), [x + 1, \infty), [x, x])$
R7. $\mathit{feasibilityBound}((-\infty, \infty), [a, \infty), (-\infty, a])$
R8. $\mathit{feasibilityBound}((-\infty, y], (-\infty, y - 1], [y, y])$
R9. $\mathit{feasibilityBound}((-\infty, \infty), (-\infty, b], [b, \infty])$

For instance, **R2** says if $[x, y]$ is feasible and $[x + 1, y]$ is infeasible, it can be concluded that every feasible interval I above $[x, y]$ must be below (or equal to) $[x, x]$. Clearly, if $I \sqsupseteq_a^b [x, y]$ is feasible, it must be the case that $I \not\sqsupseteq_a^b [x+1, y]$ (since $[x+1, y]$ is infeasible, and feasibility is anti-monotonic), which implies that I must include the value x; $I \sqsubseteq_a^b [x, x]$.

VI. Software-Defined Networking

To demonstrate the usefulness of our approach in practice, we apply it in the context of software-defined networking. In this paper, we consider a packet to be a bounded natural number $pkt \in \mathbb{N}$ $(0 \leq pkt < 2^b)$ where b is the total required number of bits to represent the header fields. A packet with a value outside the admitted bound (e.g., $pkt = -1$) is an invalid packet, and any switch immediately drops it.

A switch has a forwarding table consisting of a set of rules. Each rule has a pattern which is a predicate on headers. When a packet matches a pattern, the switch forwards it to an output port (with possibly updates to some header fields). If there are multiple matching rules, the switch is free to pick any of them, and if there are no matching rules, it drops the packet.

A. Single-packet Transition System

A *single-packet transition system* is a tuple $S = \langle pkt, trc, Q, Q_i, Q_f, T\rangle$ in which $pkt \in \mathbb{N}$, $trc : [Q]$ (trace of states); Q is a set of states ($Q_i \subseteq Q$ start, $Q_f \subseteq Q$ final); $T \in (Q \times \Phi(pkt, pkt') \times Q)$ is the transition relation from state q to q', written as $q \xrightarrow{\phi} q'$. The label $\phi \in \Phi$ is a Presburger formula over pkt (value of pkt in q) and pkt' (value of pkt in q'). Each state $q \in Q$ of a single-packet transition system normally corresponds to a switch in the network. We show the source of a transition with src, destination with dst, and label with ℓ. A transition updates the trc value $trc' = trc \lhd q'$.

a) Drop State: We assume that there is a special state $q_d \in Q_f$ that represents dropping a packet. For any $q \notin Q_f$, there is a transition to the drop state for the invalid packets: $q \xrightarrow{(pkt<0 \ \vee \ pkt \geq 2^b)} q_d$. The condition on this transition is weaker for a switch that drops more packets in the space of admissible packets.

b) Local Progress: We assume that for any packet pkt, there is always a transition out of a non-final state:

$$\forall q \notin Q_f . \forall pkt \in \mathbb{N} . \exists t \in T . \exists pkt' \in \mathbb{N} . (src(t) = q) \wedge \ell(t)(pkt, pkt')$$

Intuitively, this means that a non-final state either forwards a packet to the next or the drop state. The local progress property along with the drop state helps us specify reachability in terms of safety constraints. If there are no forwarding loops in a network, having local progress ensures that a packet is either received at the drop state or a final host.

c) Path: A path of a single-packet transition system $S = \langle pkt, trc, Q, Q_i, Q_f, T\rangle$ is a sequence $\langle pkt_0, trc_0, q_0\rangle \xrightarrow{\phi} \langle pkt_1, trc_1, q_1\rangle \xrightarrow{\phi'} \cdots \xrightarrow{\phi^{(n-1)}} \langle pkt_n, trc_n, q_n\rangle$ where $q_0 \in Q_i$ is an initial state, and $q_n \in Q_f$ is a final state.

d) Invariant: A single-packet transition system $S = \langle pkt, trc, Q, Q_i, Q_f, T\rangle$ satisfies an invariant $\psi(trc)$ (written as $S \models \psi$) if and only if every path trc satisfies ψ.

e) Horn-Clause Translation: We associate a relation symbol s_q with arity 2 to any $q \in Q$. The following Horn clause represents the transition relation $q \xrightarrow{\phi} q'$:

$$s_{q'}(pkt', trc') \leftarrow s_q(pkt, trc) \wedge \phi(pkt, pkt') \wedge (trc' = trc \lhd q') .$$

If in a start state q_i, a packet has an initial value pkt_i, then we add the following clause: $s_{q_i}(pkt, trc) \leftarrow (pkt = pkt_i)$.

We can describe some invariants of interest in the network domain using Horn clauses, such as the following.

Non-dropping—no packet is dropped: $\mathit{false} \leftarrow q_d(\mathit{pkt}, \mathit{trc})$.
Non-Reachability—for a non-dropping network, the traffic from a given source q_a must not reach a certain destination: $\mathit{false} \leftarrow q_f(\mathit{pkt}, \mathit{trc}) \wedge q_a \neq \mathit{trc.head}$.
Way-pointing—a specific switch q_a must be traversed: $\mathit{false} \leftarrow q_f(\mathit{pkt}, \mathit{trc}) \wedge q_a \notin \mathit{trc}$.

B. Bandwidth Constraints

In some repair scenarios, the properties of interest are related to bandwidth capacities of the links, congestion avoidance, or buffer overflows in packet queues. To model traffic sizes, we use a technique based on counter abstraction. The basic idea is to use tokens to represent the sizes of the flows that enter the network. Tokens here are merely used to model bandwidth usage and should not be confused with the actual packets. The token counters get updated whenever a flow of packets travels through a link.
A *bandwidth transition system* is a tuple $S = \langle Q, Q_i, Q_f, M, M_0, T\rangle$ in which Q is a set of states ($Q_i \subseteq Q$ start, $Q_f \subseteq Q$ final); M is the distribution of the traffic tokens in the network at any time. For a state $q \in Q$ and a traffic type $\tau \in \mathbb{N}$, the value of $M(q, \tau)$ is the number of tokens of traffic type τ at state q, M_0 is the initial distribution of tokens in the network, $T \in (Q \times \Phi(M, M') \times Q)$ is the transition relation from state q to q', written as $q \xrightarrow{\phi} q'$. The label ϕ determines how the distribution of the tokens $M(q)$ and $M(q')$ changes during the transition.
a) Invariant: Invariants in bandwidth transition systems are similar to single-packet transition systems, the difference being that the property ψ talks about the distributions of tokens in the network. As an example, if a state q is not safe for traffic type *typ*, then an invariant for the network specifies the number of tokens for *typ* to be 0 at any time.
b) Horn-Clause Translation: Assume that there are n types of traffic in a network, namely $\{\mathit{typ}_1, \cdots, \mathit{typ}_n\}$. For a bandwidth transition system $S = \langle Q, Q_i, Q_f, M, M_0, T\rangle$, we use a single relation symbol s that holds counters to store the number of tokens for each flow at any $Q = \{q_1, \cdots, q_m\}$ position in the network: $s(c_{q_1}^{\mathit{typ}_1}, \cdots, c_{q_1}^{\mathit{typ}_n}, \cdots, c_{q_m}^{\mathit{typ}_1}, \cdots, c_{q_m}^{\mathit{typ}_n})$. Similar to the single-packet case, we add clauses to capture the transitions of T and the updates to M.

VII. Network Repair Problem

A network repair problem $\mathcal{U} = (S, \psi, \rho)$ has the following inputs: S is a single-packet or bandwidth transition system, ψ is an invariant such that $S \not\models \psi$, and objective ρ is a ranking on the space of transition systems. A solution to the repair problem updates the transition relation T in S to obtain S', such that $S' \models \psi$ and if S'' is another transition system that satisfies the above conditions then $\rho(S') \geq \rho(S'')$. Objectives of interest in networking are, e.g., touching a minimal number of switches, filtering fewer traffic paths in the network, etc.

Let HC be the translation of S to Horn clauses. We formulate a Horn optimization problem for single-packet and bandwidth transition systems.

Benchmarks	#Nodes	#Links	#Rels.	#Lattice	#Eld	Time(s)
Gridnet	9	20	–	–	–	–
Cesnet200304	29	33	3	2.22×10^{10}	145	4.98
Arpanet19706	9	10	3	2.22×10^{10}	91	2.98
Oxford	20	26	8	3.89×10^{27}	664	16.70
Garr200902	54	71	6	4.92×10^{20}	3045	107.62
Getnet	7	8	2	7.90×10^{6}	61	1.45
Surfnet	50	73	3	2.22×10^{10}	101	3.49
Itnet	11	10	1	2.81×10^{3}	17	0.18
Garr199904	23	25	1	2.81×10^{3}	19	0.33
Darkstrand	28	31	5	1.75×10^{17}	425	14.81
Carnet	44	43	2	7.90×10^{6}	37	0.49
Atmnet	21	22	1	2.81×10^{3}	15	0.67
HiberniaCanada	13	14	11	8.63×10^{37}	1795	84.56
Evolink	37	45	1	2.81×10^{3}	14	0.20
Dfn	58	87	–	–	–	–
Ernet	30	32	4	6.23×10^{13}	140	4.94
Bren	37	38	6	4.92×10^{20}	974	25.14
Niif	36	41	2	7.90×10^{6}	48	0.92
Renater2001	24	27	3	2.22×10^{10}	101	3.56
Latnet	69	74	2	7.90×10^{6}	47	0.64

Fig. 4: Repairing 20 benchmarks from Topology Zoo [13] on a 1.4 GHz AMD Opteron$^{\text{TM}}$ Processor with 32 Gigabytes of memory (time-out is set to 2 minutes in this experiment).

a) Single-packet Transition System: We use Alg. 1 to add new fresh symbols $m(\mathit{pkt})$ to the bodies of some clauses in HC to get HC' (or $m(\mathit{pkt}, \mathit{pkt}')$ when the source switch can rewrite packets). Assuming the size of the header in a packet is b, to each $m(\mathit{pkt})$ we associate an interval lattice $I_0^{2^b-1}$ (e.g., lattice in Fig. 3) that represents the packets that should be filtered out. The lattice of repair solutions is the product of all the interval lattices for m relations. For an objective function ρ that more highly ranks solutions that filter fewer traffic types, we use an objective function obj in the Horn optimization problem that assigns the lowest rank to the solution that assigns $(-\infty, \infty)$ to every m.
b) Bandwidth Transition System: The formulation of network repair in this case is similar to single-packet transition systems, with the difference being that the lower-bound of the intervals for added fresh symbols is 0, and the upper-bound is the maximum number of tokens for each type.
c) Generality of Repair: We assume that the reason for a violation of the safety property is that the network configuration is under-constrained. In other words, there are forwarding behaviors in the network that should be restricted—e.g., by adding filters on the links. Furthermore, we assume that during the repair procedure, no new switches or links are added to the network. These assumptions are not overly restrictive in practice—if the network operator wants to add new switches to the network, she can connect the new switch to the rest of the network without any constraints: the new switch behaves as a repeater. It is also possible to add links to the network and send all the traffic through the new links. The repair procedure may then restrict forwarding of packets through these links.

VIII. Implementation and Experiments

We have implemented the prototype tool *Marham* (Minimal repair for Horn clause systems) that operates on top of the

Eldarica [5] verifier. To evaluate Marham, we considered two main questions. First, we studied the applicability of our approach to several interesting repair scenarios from the network domain. Second, we benchmarked the performance of our tool against a dataset of real topologies. For the first question, we considered the network properties introduced in Section II using the data center topology shown in Fig. 1(a) with a non-dropping criterion. We used $[0, 7]$ as intervals, with 0 representing SSH traffic. Our tool found the correct repair by suggesting that a filter be added on A_4. We also repaired a way-pointing scenario by removing the path through the way-point and then repairing. For the Fig. 1(b) example, Marham produces a repair by sending the green traffic to s_4.

For the second question, for topologies from the Internet Topology Zoo set [13], we generated Horn clauses to connect a set of random vertices (Topology Zoo contains data network topologies from around the world). We non-deterministically selected a node and made it unsafe for a certain flow by adding a clause specifying that this type of traffic should not reach that particular point. We considered the objective function that minimizes the number of filtered paths relative to the original configuration. Table in Fig. 4 shows the results of executing Marham for repairing 20 representative topologies from the Topology Zoo. The table reports the number of nodes, links, and synthesized relation symbols, as well as the size of the lattice, number of calls to Eldarica, and total time.

IX. Related Work

Although *optimizing SMT solvers* have been proposed in previous work [14], to the best of our knowledge, our framework is the first to provide such optimization functionality in a Horn clause solver. Our approach also differs from *MaxSAT solvers,* which search for solutions satisfying maximum sets of clauses, in the generic way that optimization lattices and objectives are formulated.

A number of approaches to repair are based on finding **similar expressions**—e.g., by using a game-based approach in which winning strategies correspond to choosing a correct expression [15], adding nondeterministic expressions at problematic locations and using a SAT solver to find a deterministic program that satisfies the specification [16], using a cost function to select a correct expression [17], or using deductive approaches based on *guided* synthesis [1].

Other repair approaches target **specific languages** (e.g., Boolean programs, which are essentially a restricted form of C programs [18]) or **specific types of fixes** (e.g., atomicity violations [19]). Our repair framework is different in that (i) it is not language-specific so it can be used in a variety of settings, (ii) it places no restrictions on the type of repairs that can be made, and (iii) it allows the programmer to repair with respect to a safety property as well as an objective function.

In regards to the problem of synthesizing repairs for network configurations, the closest to our work is [20]. Our work is more general in several aspects. Their specification language is based on regular expressions, and updates are specified as end-to-end paths from the old to new configuration using regular expressions. Our Horn clause specification language gives us the power to consider more general properties such as loop freedom, bandwidth constraints, etc.

X. Conclusion

This paper introduces a framework for repairing a set of Horn clauses, and presents an optimization technique to search the space of repairs efficiently. We have implemented our repair engine in the Marham tool—to investigate its applicability to real world problems, we perform experiments using the Internet Topology Zoo dataset. The generality of Horn clauses in describing problems from various domains makes our proposed approach suitable for repairing various systems.

Acknowledgments

We thank the FMCAD reviewers for helpful and constructive comments. Our work is supported by the National Science Foundation under grants CNS-1111698, CNS-1413972, CCF-1421752, CCF-1422046, CCF-1253165, and CCF-1535952; the Office of Naval Research under grant N00014-15-1-2177; and gifts from Cisco, Facebook, Fujitsu, Google, and Intel.

References

[1] E. Kneuss, M. Koukoutos, and V. Kuncak, "Deductive program repair," in *CAV*, pp. 217–233, 2015.

[2] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, pp. 298–312, ACM, 2016.

[3] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II*, pp. 24–51, Springer, 2015.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko, "On solving universally quantified horn clauses," in *SAS*, pp. 105–125, Springer, 2013.

[5] H. Hojjat, F. Konečný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A verification toolkit for numerical transition systems (tool paper)," in *FM*, 2012.

[6] J. McClurg, H. Hojjat, P. Cerný, and N. Foster, "Efficient synthesis of network updates," in *PLDI*, pp. 196–207, 2015.

[7] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.

[8] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for Horn-clause verification," in *CAV*, pp. 347–363, 2013.

[9] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2012.

[10] B. Jaumard and B. Simeone, "On the complexity of the maximum satisfiability problem for Horn formulas," *Information Processing Letters*, vol. 26, no. 1, pp. 1 – 4, 1987.

[11] H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., 2004.

[12] J. Leroux, P. Rümmer, and P. Subotic, "Guiding Craig interpolation with domain-specific abstractions," *Acta Informatica*, 2015.

[13] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765–1775, 2011.

[14] N. Bjørner, A. Phan, and L. Fleckenstein, "νz - an optimizing SMT solver," in *TACAS*, pp. 194–199, 2015.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, pp. 226–238, Springer, 2005.

[16] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using SAT," in *TACAS*, pp. 173–188, Springer, 2011.

[17] R. Samanta, O. Olivo, and E. A. Emerson, "Cost-aware automatic program repair," in *SAS*, pp. 268–284, Springer, 2014.

[18] A. Griesmayer, R. Bloem, and B. Cook, "Repair of boolean programs with an application to C," in *CAV*, pp. 358–371, Springer, 2006.

[19] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," *PLDI*, vol. 46, no. 6, pp. 389–400, 2011.

[20] S. Saha, S. Prabhu, and P. Madhusudan, "Netgen: Synthesizing data-plane configurations for network policies," in *SOSR '15*, 2015.

On ∃ ∀ ∃! Solving: A Case Study on Automated Synthesis of Magic Card Tricks

Susmit Jha Vasumathi Raman
United Technology Research Center, Berkeley
jhask, ramanv@utrc.utc.com

Sanjit A. Seshia
EECS, UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*—In formal synthesis, the goal is to find a composition of components from a finite library such that the composition satisfies a given logical specification. In this paper, we consider the problem of synthesizing magic card tricks from component actions, where some of the actions depend on non-deterministic choices made by the audience. This problem can be naturally represented as a quantified logical formula of the form: *Exists* a composition, *Forall* nondeterministic choices, *Uniquely-Exists* intermediate and final outputs satisfying a logical specification, that is, an (∃∀∃!) satisfiability problem. We present a novel approach to solve this problem that exploits the unique-existence of intermediate and final outputs for any given composition and choice values. We illustrate how several popular magic card tricks can be recovered using this approach. These tricks evolved through human ingenuity over decades, but we demonstrate that formal synthesis can generate a number of novel variants of these tricks within minutes. In contrast, a direct encoding to quantified SMT problem fails to find a solution in hours.**

I. Introduction

Composition has played a central role in enabling configurable and scalable design of efficient systems across different domains. Automated formal synthesis of systems as composition of elementary components using satisfiability solving techniques has proved to be useful in creating non-intuitive artefacts such as bit-vector programs using bit-twiddling operations [1] and deobfuscating code [2]. Synthesis from components can also be carried out by selecting a syntactically expressible family of compositions such as a sketch with a fixed skeleton but free choice of components [3], [4]. These approaches are, however, restricted to deterministic functional components. In this paper, we consider the problem of synthesizing composition of components that represent primitive actions such as shuffling, cutting a card deck or turning over of a set of cards. Some of these component actions are nondeterministic, and their output depends on uncontrollable external choices. This formulation also applies to synthesis of strategies against unknown non-deterministic choices made by an adversary, synthesis of circuits with stochastic (abstracted as non-deterministic) components, in addition to the application to synthesizing magic card tricks considered in this paper. Magic card tricks are often developed over decades and use significant human ingenuity. These tricks involve interesting applications of number theory and discrete mathematics, and hence, their explanation has received significant attention from mathematicians [5], [6]. In particular, they have been found to be useful in teaching computer science [7]. To the best of our knowledge, this is the first application of formal methods to synthesize new magic card tricks. Beyond this unconventional application domain and an attempt to automate a long-standing bastion of human creativity, we believe this application will also add interesting benchmarks to fuel further work on automated synthesis of systems with non-deterministic components.

In previous work [1], we have shown that the synthesis of composition using components can be expressed in first-order logic by eliminating the existential quantification over the predicate corresponding to composition, using integer parameters to encode the set of possible compositions for a finite library of components. We make the following novel contributions in this paper:

- The synthesis problem using non-deterministic component actions is directly representable as ∃∀∃! problem and it is shown to have a corollary of the ∃∀ form. This dual representation enables the use of counterexample guided inductive synthesis.
- We model the component actions in popular magic card tricks logically and use the Z3 SMT solver [8] to discover novel magic card tricks as compositions of these actions. While human discovery of these variants takes decades, we discover new magic tricks within minutes.

In the rest of the paper, we first present a brief background on magic card tricks in Section II. We formulate the synthesis of magic tricks as a composition synthesis problem and describe its reduction to a satisfiability problem in Section III. We present results including new magic tricks discovered by our approach in Section IV, and conclude with discussion of future work in Section V.

II. Background on Magic Card Tricks

Magic card tricks consist of sequences of actions in which the performer manipulates the cards through acts such as shuffling or turning over cards. Usually, these action sequences also involve audiences making a number of choices independently and at their discretion, and sometimes in secret without the performer knowing the exact choice. These choices are intended to make the audience believe that they are in control of the manipulation of cards. Consequently, the performer's ability to achieve some deterministic goal such as finding a particular card or ensuring that the cards end up in a specific

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

pattern surprises the audience and makes the entire process appear *magical*.

The secret sauce behind these magic card tricks often lies in some mathematical invariant of the sequence of actions which allows the performer to ensure a deterministic output irrespective of the exact choice made by the audience. Such sequences of actions are naturally hard to generate, and consequently, each sequence is passed on as training from magician to magician. We use an example of a very simple magic card trick to give insight into the underlying mathematics that ensures the outcome of the trick is independent of audience choices. This magic trick was invented by magician Bob Hummer and popularized as Face Up/Face Down Mysteries (1941)[9]. It has the following sequence of actions.

1) Take any ten cards face-down and hold them as if you are going to deal the cards.
2) Go through the following procedure which mixes the cards face-up and face-down based on audience choice: Spread the top two cards off and turn them over together, placing them back on top. Let an audience member give the deck of cards a straight cut [1]. The cut can be at any position (odd or even) of the deck. Repeat this "turn-two and audience-directed cut" procedures at random as often as you like. The cards will be in an *apparent* unpredictable mess.
3) To find the order in the mess, proceed as follows: Go through the card deck, reversing every second card (the cards in positions 2, 4, 6, 8, and 10). You will find exactly five cards face-up, no matter how many times the "turn two and cut at random" procedure was repeated.

The ten cards with faces-up or down can end up in $2^{10} \times 10!$ possible arrangements, which is more than 3.6 billion. The *magical* surprise is due to the fact that we always end up in a state satisfying a deterministic property irrespective of the choices made by the audience. The explanation for this lies in the following observation. The actions in step 2 (also called Hummer Shuffle) has the following invariant - the number of cards facing up/down at even places in the stack is respectively, equal to the number of cards facing up/down at odd places after each Hummer Shuffle. Let m be the number of face-up cards in even places, and by the above invariant, also in odd places. The last step of turning over even cards will leave us with $5 - m$ face-up cards in even places. Combined with m face-up cards in odd places, the total number of face-up cards will always be five irrespective of the audience choices.

A more popular derivative of this trick called the Royal Hummer invented by Steve Freeman decades later is used to produce a royal flush from a deck of cards that has been apparently shuffled randomly by the audience. The Hummer Shuffle consisting of just two actions: turn two and audience-directed cut, was invented more than seventy years ago but it still remains the core of many magic card tricks. Our goal is to automatically synthesize even longer (and hence, even more surprising and non-intuitive) sequences which are guaranteed to satisfy some relevant deterministic property on the final state irrespective of the choices made by the audience.

[1]Audience chooses a random position in the deck and all the cards below this position are put contiguously on the top of the deck. See [9], [10] for a glossary of card trick terms.

III. Composition Based Synthesis of Card Tricks

Automated formal synthesis of systems (particularly programs) from high-level specification has received significant attention over the past decade. For brevity, we refer to a recent papers [3], [11] and references therein for detailed discussion on state of the art in synthesis. We focus on the problem of formal modelling and synthesis of magic card tricks in this section. Given a deck of cards, we characterize the configuration of cards as a state $s \in S$ where S is all possible configurations the cards can be put in. The states would capture relevant details for a trick such as whether a card is facing up or down, the position of each card and color or kind of cards in each position. For the Hummmer trick described in Section II, the state space is given by whether the cards are facing-up or down in each of the 10 positions. We denote the library of primitive actions, such as turning over a card or cutting the card deck, by $L = \{A_1, A_2, \dots, A_n\}$ where each action is either deterministic or non-deterministic. For notational uniformity, we associate both kinds of actions A_i with a state transformation function $T_i : S \times C \rightarrow S$ where C is the set of choices. Execution of actions A_i takes the cards from state s to $s' = T_i(s, c)$ where $c \in C$ is the choice parameter. Deterministic actions ignore the second argument, that is, the choice c. In order to generate action sequences of varying length, we include k copies of noop (no operation) actions $A_{n+1} \dots A_{n+k}$ in the library L such that the transformation function for each noop action is identity, that is, $T_{n+1}(s, c) = \dots T_{n+k}(s, c) = s$. A schematic of a magic trick as a sequence of actions is shown in Figure 1.

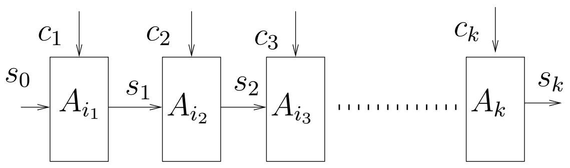


Fig. 1. Magic Scheme as Sequence of Actions

From the scheme presented above and using the first-order representation of the connection predicate [1], [2], the problem of synthesizing magic trick can be formulated as follows, assuming a unique initial state s_0 (generalization to multiple initial states is straightforward). We assume that k is the upper bound on the length of the magic trick. Since $k \leq n$, a trick may not use all actions in the library. Further, we include k copies of noop action with identity state transformation function so that the sequence may use these functions which can be eliminated as post-processing to generate shorter magic sequences.

Let the vector corresponding to a sequence of actions $i_1 i_2 \dots i_k$, where $i_j \in [0, n]$, be denoted by $\texttt{conn}, c_1, c_2, \dots c_k$

with $c_j \in C$ be denoted by $\mathtt{choices}$, and $s_1, \dots s_k$ with $s_j \in S$ be denoted by $\mathtt{states}$. Further, T is a parametrized state transformer which uses i_m as its first argument to select the corresponding action A_{i_m} and thus, the transformer T_{i_m}. We can now rewrite the above constraints as

$$\mathcal{F}_{des}: \quad \exists \mathtt{conn}\, \forall \mathtt{choices}\, \exists ! \mathtt{states} \quad \phi_{des}(\mathtt{conn}, \mathtt{choices}, s_0, \mathtt{states}) \wedge \phi(\mathtt{states})$$

where $\phi_{des}(\mathtt{conn}, \mathtt{choices}, s_0, \mathtt{states}) = \bigwedge_{m=1}^{k} (s_m = T(i_m, s_{m-1}, c_m))$ and $\phi(\mathtt{states}) = \phi_{spec}(s_k)$ is the deterministic requirement on the final state. Since the possible choices constitute a finite (albeit very large) set, say $\{\mathtt{choices_1}, \mathtt{choices_2}, \dots, \mathtt{choices_L}\}$, and there exists exactly one state vector $\mathtt{states}_l$ for each choice vector $\mathtt{choices}_l$, we can rewrite $\mathcal{F}_{des}$ as follows:

$$\mathcal{F}_{sat}: \quad \exists \mathtt{conn}\, \exists \mathtt{states_1} \exists \mathtt{states_2} \dots \exists \mathtt{states_L} \quad \bigwedge_{l=1}^{L} \phi_{des}(\mathtt{conn}, \mathtt{choices}_l, s_0, \mathtt{states}_l) \wedge \phi(\mathtt{states})$$

The challenge in solving $\mathcal{F}_{sat}$ is that the number of choices is huge. But the above formulation can be used to find a sequence of actions conn that is consistent with a subset of example choice vectors. Let us consider another constraint:

$$\mathcal{F}_{ver}: \quad \exists \mathtt{conn}\, \forall \mathtt{choices}\, \forall \mathtt{states} \quad \phi_{des}(\mathtt{conn}, \mathtt{choices}, s_0, \mathtt{states}) \Rightarrow \phi(\mathtt{states})$$

We observe that $\mathcal{F}_{ver}$ is true if $\mathcal{F}_{des}$ is true, that is, $\mathcal{F}_{des} \Rightarrow \mathcal{F}_{ver}$ due to the unique existence of the states for any given choice vector in $\mathcal{F}_{des}$. For any given sequence of component actions conn, a counterexample obtained by solving the satisfiability problem for $\exists \mathtt{choices} \exists \mathtt{states} \ \neg(\phi_{des}(\mathtt{conn}, \mathtt{choices}, s_0, \mathtt{states}) \Rightarrow \phi(\mathtt{states}))$ can thus be used to solve the original synthesis problem in $\mathcal{F}_{des}$ using component-based program synthesis [1], [2]. The generalization from examples is done by solving the satisfiability problem for $\mathcal{F}_{sat}$ including only the choices found by solving $\mathcal{F}_{ver}$.

IV. Results

In this section, we present our results on automated synthesis of magic card tricks. We consider three different known magic card tricks from [9], [10] and synthesize a number of novel variants for each of these tricks. These novel variants are new sequence of actions which ensure that the final state meets the given deterministic requirement irrespective of the choices made by the audience for non-deterministic actions. Our experiments used the Z3 [8] SMT solver for checking satisfiability and all experiments were run on a 2.9 GHz Intel Core i7 with 16 GB RAM. Finding the action sequence for the example presented in Section II took 3 minutes 18 seconds.

A. Baby Hummer

In this magic trick, the audience selects a card unknown to the magician, and three other cards creating a card deck of 4 cards with the selected card at the bottom. All the cards are put in the deck face down. The magician instructs the audience to take a sequence of actions - some of which involve making choices, such that the card deck appears to be randomly shuffled. At the end, the magician states that all the cards in the deck except one must face in the same direction - either up or down. The deck must have only one odd-facing card, and this card is the one selected by the audience. We build a glossary of actions used in this trick before describing the actions in the original card trick and the synthesized novel variants. The finite library used in the automated synthesis process consists of four copies of each action listed below, and noop actions as described in Section III.

Action	Description
`turntop`	Turn over the face of the top card.
`turntop2`	Take off the top two cards (keeping them together), turn them over together and place them back on top.
`toptobottom`	Place the top card of the deck to the bottom.
`top2tobottom`	Place the top two cards of the deck to the bottom keeping them together and in-order
`cut`	Audience-choice directed straight-cut of the card deck.

The sequence of actions in the original magic card trick are as follows:

- `toptobottom, turntop, cut, turntop2, cut, turntop2, cut, turntop2, turntop, top2tobottom, top2tobottom, turntop`

The reader can refer to Fig 4-13 in Chapter 1 of [9] for pictorial illustrations of this magic trick.

We used the synthesis approach presented in Section III to generate new action sequences of length at most 15 steps. Our initial experiments yielded non-interesting sequences where the `cut` action was never used. We, therefore, added constraints to ensure that the `cut` action was always included in the sequence. This led to sequences where all the occurrences of `cut` were either at the start of the sequence or at the end of the sequence. These are also not interesting since the `cut` actions do not turn the cards and hence, do not achieve any shuffling of face-up and face-down cards unless mixed with turning-over actions.

After adding constraints to mix the card-turning actions with cut, we generated a number of new sequences which also ensure that the audience-selected card is the odd-facing card at the end, irrespective of the choices made by audience. We report six of these (with the noops removed) below:

- `toptobottom, turntop, cut, cut, turntop, toptobottom, turntop2, toptobottom, turntop, toptobottom`
- `turntop2, turntop, toptobottom, toptobottom, cut, turntop, toptobottom, toptobottom, turntop, cut, turntop2, turntop2`
- `turntop2, toptobottom, toptobottom, turntop, cut, toptobottom, cut, turntop2, turntop2, cut, turntop, turntop`
- `toptobottom, turntop, cut, toptobottom, toptobottom, turntop2, cut, cut, turntop, toptobottom, turntop, turntop2`
- `toptobottom, turntop, cut, toptobottom, toptobottom, cut, turntop2, cut, turntop, toptobottom, turntop, turntop2`
- `toptobottom, toptobottom, toptobottom, turntop, cut, top2tobottom, cut, turntop2, turntop2, cut, turntop, turntop`

The runtime for synthesis was 12m 23sec, 12m 04sec, 10m 30sec, 5m 43 sec, 11m 11 sec and 11m 39 sec. respectively. A direct $\exists\forall\exists$ encoding of the problem timed out after 2 hours and could not find any satisfying action sequence.

B. Elmsley Shuffles

Shuffles are very common component actions of many magic card tricks. We consider two perfect shuffle actions: in-shuffle and out-shuffle. Both in-shuffle and out-shuffle begin by splitting the deck into two equal piles by splitting it in the middle, followed by shuffling perfectly so that the new deck has cards alternately from both piles. The original top card is the second card in in-shuffle and it is the top card in out-shuffle. In this magic trick, the performer asks the audience to randomly shuffle the cards and select one card which is put on top of the stack but not revealed to the performer. The performer then does a number of in-shuffles and out-shuffles which make the stack appear to be randomly shuffled, but the performer is able to select a card from a particular position in the stack, and this card is revealed to be the one originally selected by the audience. Since these shuffles do not depend on the initial state of the card deck, simple enumeration could also work in this case. However, more interesting shuffles will not be possible through enumeration. Let `inS`, `outS` denote the two actions of in-shuffle and out-shuffle respectively. Starting with 8 cards, two action sequences that would put the top card at position 6 are as follows: `inS`, `inS`, `outS`; and `inS`, `outS`, `outS`, `outS`, `inS`, `outS`. Similarly, two action sequences which would put the top card at position 5 are as follows: `inS`, `outS`, `inS`; and `inS`, `outS`, `outS`, `outS`, `outS`, `inS`. The runtimes for synthesis were 6m 42sec, 8m 52sec, 5m 18sec and 6m 08sec.

C. "Mind Reading" of Cards

In this magic trick, the performer starts with a prearranged card deck with 8 cards which are passed to the audience. Let this sequence be AH, 5D, 6H, 2S, 5S, KC, 7H, 8S where H,D,C,S stands for hearts, diamonds, clubs and spades, respectively. The color sequence for this stack is RRRBBBRB. Any three audience members are asked to make a sequence of random cuts on this stack after which each of them select (in order) the top card as their chosen card in secret from the performer. The performer is required to find these cards. He does so by asking the audience members with red cards to stand up. Depending on who stand up, the performer can tell the exact card selected by the audience. This trick works because any cyclic subsequence with length 3 of RRRBBBRB is unique. So, when the audience members stand up, the performer can find the subsequence (RRR, RRB, RBR, RBB, BRR, BRB, BBR, BBB). Once this subsequence is known, the performer uses his knowledge of the prearranged stack to tell the exact cards picked by the audience. The automated synthesis requires coming up with this initial prearranged card deck. The subsequent actions performed by the audience are fixed in this trick. Clearly, not all prearranged card decks will work. The patterns needed for this trick are in face de Bruijn sequences also used in robot localization [12]. For the variant of this trick with 5 audience members and a card deck with 32 cards (there are 32 possible sub-sequences of length 5 which can be possibly decoded using the color pattern for the 5 audiences), we synthesized the following possible color patterns. The runtime for the two searches was 8 m 34s and 9m 18s respectively. Note that the exact card sequence is not important and any cards consistent with these color patterns can be used as long as the performer remembers the original sequence, to be able to tell the exact cards at the end irrespective of the choice of cuts by the audience.

- RRRRRBRBRRBRRRBBBBBRBBBRRBBRBRBB
- RRRRRBRRBRBBRRBBBBBRRRBBRBBBRBRB

V. Conclusion

In this paper, we consider the problem of synthesizing a composition of non-deterministic components and show how its solution can be used to generate new magic tricks. We believe computer-aided design of magic tricks can provide interesting examples in computer science courses. Class-room exercises on automatically synthesizing new magic card tricks can be used to introduce combinatorial search, quantified satisfiability solving and formal methods. We also plan to improve the quality of our results by adding constraints to rule out subsequences that reduce to identity operations without user-input. We also plan to investigate more complex magic tricks including probabilistic tricks such as Kruskal's count trick [13]. Further, the logical modelling of magic tricks is similar to that used for computer programs from component functions and for plans from component actions in artificial intelligence. In future work, we plan to investigate the application of this approach to planning in the presence of adversarial choice.

Acknowledgement: We thank Ashish Tiwari, Sonali Sinha and Claudio Pinello for very helpful discussions. The third author was supported in part by NSF grant CCF-1139138.

References

[1] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.

[2] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," ser. ICSE '10, 2010, pp. 215–224.

[3] R. Alur, R. Bodik, 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*, October 2013.

[4] A. Solar-Lezama, L. Tancau, R. Bodk, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs." in *ASPLOS*, 2006, pp. 404–415.

[5] M. Gardner, *Mathematics, magic and mystery*. Courier Corp., 2014.

[6] C. Mulcahy, "Mathematical card tricks," *Whats New in Mathematics*, 2000.

[7] J. F. Ferreira and A. Mendes, "The magic of algorithm design and analysis: Teaching algorithmic skills using magic card tricks," ser. ITiCSE '14, 2014, pp. 75–80.

[8] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[9] P. Diaconis and R. Graham, *Magical Mathematics: The Mathematical Ideas That Animate Great Magic Tricks*. Princeton University Press, 2011.

[10] J. Hugard and J. J. Crimmins, *Encyclopedia of card tricks*. Courier Corp.

[11] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *CoRR*, vol. abs/1505.03953, 2015.

[12] J. Pagès, J. Salvi, C. Collewet, and J. Forest, "Optimised de bruijn patterns for one-shot shape acquisition," *Image and Vision Computing*, vol. 23, no. 8, pp. 707–720, 2005.

[13] S. Humble, "Magic math cards," *The Mathematics Enthusiast*, vol. 5, no. 2, pp. 327–336, 2008.

Property-Directed k-Induction

Dejan Jovanović
SRI International
dejan.jovanovic@sri.com

Bruno Dutertre
SRI International
bruno.dutertre@sri.com

***Abstract*—IC3 and k-induction are commonly used in automated analysis of infinite-state systems. We present a reformulation of IC3 that separates reachability checking from induction reasoning. This makes the algorithm more modular, and allows us to integrate IC3 and k-induction. We call this new method property-directed k-induction (PD-KIND). We show that k-induction is more powerful than regular induction, and that, modulo assumptions on the interpolation method, PD-KIND is more powerful than k-induction. Moreover, with k-induction as the invariant generation back-end of IC3, the new method can produce more concise invariants. We have implemented the method in the SALLY model checker. We present empirical results to support its effectiveness.**

I. Introduction

IC3 and k-induction are two commonly used methods in automated analysis of infinite-state systems. IC3 was originally developed for finite systems [7], [20], but it was soon adapted to the infinite-state case by relying on SMT solvers as reasoning engines [22], [9], [24], [10]. These IC3 variants have been successfully used in analysis of software. Similarly, k-induction was initially introduced in the finite-state setting [31] and was then extended to infinite-state systems [15], with similar success in software verification [16].

At its core, IC3 is based on induction. To show that a property is invariant, IC3 tries to incrementally construct an inductive strengthening of the property. Surprisingly, the relative power of k-induction and induction-based methods, with respect to the capability to construct a strengthening, has not been studied in detail.[1] It is folklore knowledge that k-induction can be stronger than induction, but this has, to the best of our knowledge, never been formally accounted for. In this paper, we show that k-induction can be strictly more powerful than regular induction, making a case for an IC3-style method that is based on k-induction. The additional reasoning power is particularly important when one works within an expressive logical theory such as the theory of arrays [21], [23]).

Although additional deductive power is desirable, we are also concerned with practical effectiveness. With this in mind, we start with IC3, a practically effective algorithm, and break it into its constituents: satisfiability checking, reachability checking, and generation of inductive invariants. Isolating these functionally independent modules allows us to replace the inductive core with k-induction, producing a method that is a natural combination of IC3 and k-induction. This method is effective in practice, and can be shown to be at least as powerful as k-induction, provided the interpolation procedure satisfies a natural property that we call finite-covering.

To summarize, the main contributions of the paper are as follows. We show that k-induction can be more powerful, and more concise, than regular induction (Section III). We decompose IC3 into functionally relevant parts (Section IV) and adopt k-induction as the core reasoning step (Section IV-C). We isolate a fundamental property of interpolation (Section IV-A) that allows us to prove the new method more powerful than k-induction. We provide experimental evidence that the new method is effective in practice (Section V). [2]

II. Background

We assume a finite set of typed variables $\vec{x}$ called *state variables*. To each variable $x \in \vec{x}$, we associate its primed version x' of the same type. We call any quantifier-free formula $F(\vec{x})$ over the state variables a *state formula*, and any quantifier-free formula $T(\vec{x}, \vec{x}')$ a *state-transition formula*. A *state* s is a type-consistent interpretation of $\vec{x}$ that assigns to each variable $x \in \vec{x}$ a value $s(x)$ over its domain. A state formula $F(\vec{x})$ holds in a state s (written $s \vDash F$) if the formula evaluates to true under the state's assignment.

A *state-transition system* is a pair $\mathfrak{S} = \langle I, T \rangle$, where $I(\vec{x})$ is a state formula describing the initial states and $T(\vec{x}, \vec{x}')$ is a state-transition formula describing the system's evolution. A state s' is a successor of a state s in $\mathfrak{S}$ if the formula $T(\vec{x}, \vec{x}')$ evaluates to true when we interpret each $x \in \vec{x}$ as $s(x)$ and each $x' \in \vec{x}'$ as $s'(x)$. A state s is *k-reachable* if there exists a sequence of states $\sigma = \langle s_0, \ldots s_k \rangle$ such that, $s = s_k$, the state s_0 satisfies I, and each s_{i+1} is a successor of s_i. We call σ a *concrete trace* of the system. We also say that a state formula F is reachable in k steps if there is a k-reachable state s such that $s \vDash F$.

Given a state formula P (*the property*), we want to determine whether all the reachable states of $\mathfrak{S}$ satisfy P. If this is the case, P is an *invariant* of $\mathfrak{S}$, which we denote by $\mathfrak{S} \vDash P$. We also write $\mathfrak{S} \vDash_a^b P$ to denote that P is true in all k-reachable states for $a \leq k \leq b$. If P is not invariant, there is a concrete trace, called a *counter-example*, that reaches $\neg P$.

The research presented in this paper has been supported by NASA Cooperative Agreements NNX14AI05A and NNA10DE73C, by DARPA under agreement number FA8750-16-C-0043, and by NSF grant 1528153.

[1]Some IC3 variants, such as PDR [22], are not guaranteed to terminate even if the property is already inductive.

[2]Due to space limitations proofs we omit the proofs in this paper. The full paper with proofs and additional experimental data is available from the authors as a technical report.

Definition II.1 ($\mathcal{F}$-Induction)**.** *Given a set $\mathcal{F}$ of state formulas such that $P \in \mathcal{F}$, P is* $\mathcal{F}$-inductive[3] *with respect to $\mathfrak{S}$ if*

$$I(\vec{x}) \Rightarrow \mathcal{F}(\vec{x}) \ , \qquad \text{(init)}$$

$$\mathcal{F}(\vec{x}) \wedge T(\vec{x}, \vec{x}') \Rightarrow P(\vec{x}') \ . \qquad \text{(cons)}$$

If $\mathcal{F} = \{P\}$, we say that P is inductive.

If P is inductive then it is also invariant. Since invariants are in general not inductive, a common approach to prove that P is invariant is to find a *strengthening* of P. Such a strengthening is a set of formulas $\mathcal{F}$ such that $P \in \mathcal{F}$ and $\mathcal{F}$ is inductive. If such a strengthening exists, then P is invariant.

Definition II.2 ($\mathcal{F}^k$-Induction)**.** *Given a set $\mathcal{F}$ of state formulas such that $P \in \mathcal{F}$, P is* $\mathcal{F}^k$-inductive *with respect to $\mathfrak{S}$ if*

$$I(\vec{x_0}) \wedge \bigwedge_{i=0}^{l-1} T(\vec{x}_i, \vec{x}_{i+1}) \Rightarrow \mathcal{F}(\vec{x}_l) \ \ , \textit{for } 0 \leq l < k \ \ , \qquad \text{(k-init)}$$

$$\bigwedge_{i=0}^{k-1} \left(\mathcal{F}(\vec{x}_i) \wedge T(\vec{x}_i, \vec{x}_{i+1})\right) \Rightarrow P(\vec{x}_k) \ . \qquad \text{(k-cons)}$$

When $\mathcal{F} = \{P\}$, we say that P is k-inductive.

A property that is inductive is 1-inductive by definition. It is also k-inductive for any k. In the other direction, if a property P is k-inductive and the logical theory underlying the system admits quantifier elimination, then we can construct an inductive strengthening of P by eliminating quantifiers.[4] For such theories, induction and k-induction have the same deductive power but k-induction may give more succinct strengthenings. If the base theory does not admit quantifier elimination then k-induction can be more powerful than induction.[5]

III. Relative Power of Induction and k-Induction

We present examples that illustrate the deductive power of k-induction in relation to induction. To simplify the presentation, we describe transition systems as programs. We use the quantifier-free fragment of the extensional theory of arrays [27], denoted by $\mathcal{T}_{\mathbf{arr}}$, and an extension of $\mathcal{T}_{\mathbf{arr}}$ with array constants [13], denoted by $\mathcal{T}_{\mathbf{arr}}^{\mathbf{c}}$. The theory $\mathcal{T}_{\mathbf{arr}}$ is axiomatized as $\textsc{write}(a,i,v)[i] = v$, $i \neq j \Rightarrow \textsc{write}(a,i,v)[j] = a[j]$, and $a[i] \neq b[i] \Rightarrow a \neq b$. The extended theory $\mathcal{T}_{\mathbf{arr}}^{\mathbf{c}}$ is obtained by adding a construct $\mathbf{c}(v)$ that represents the constant array with value v, and the axiom $\mathbf{c}(v)[i] = v$.

The quantifier-free fragments of both theories are decidable [32], [13] and are very useful in practice. For example, one can model integer sets in the theory $\mathcal{T}_{\mathbf{arr}}^{\mathbf{c}}$ as arrays that map integers to Booleans, and define set operations as $x \in a \equiv a[x]$, $\emptyset \equiv \mathbf{c}(\mathbf{false})$, $a \cup \{x\} \equiv \textsc{write}(a, x, \mathbf{true})$.

```
1  int i, j;
2  map<int,int> a; // int -> int
3
4  // I: write 0 at a[0]
5  i = j = 0;
6  a[0] = 0;
7
8  // T: write 0 at a[i] from a[j]
9  while (true) {
10   j = rand() % (i+1);
11   i = (i+1) % N;
12   a[i] = a[j];
13 }
```

Fig. 1. Writing 0 to array a in a circular fashion, resetting every N steps.

Although their quantifier-free fragments are decidable, the full array theories are not decidable [8] and therefore do not admit quantifier elimination.[6] This makes them good candidates for relating the powers of induction and k-induction.

Example III.1. *The program in Figure 1 sets the elements $a[0], \ldots, a[N-1]$ to 0 in a circular fashion. This program can be encoded as a transition system in theory $\mathcal{T}_{\mathbf{arr}}$, with initial states defined by lines 5-6, and transition relation defined by lines 10-12. Now, consider the property $P \equiv (a[0] = 0)$. This property is clearly an invariant of the system. One can show that it is $(N+1)$-inductive: Any sequence of $N+1$ states must include a transition that resets i to 0. From then on, all transitions increment i to an integer no more than $N-1$, pick a j between 0 and i, and copy $a[j]$ into $a[i]$. If P holds at all states in this sequence, and if $i = N-1$ in the last state of this sequence, then $a[0], \ldots, a[N-1]$ are all 0 in this state. If $i \neq N-1$ in the last state then the next transition keeps $a[0]$ unchanged. These observations are enough to conclude that P is $(N+1)$-inductive. On the other hand, P is not k-inductive for any $k \leq N$. Moreover,* any inductive strengthening of P must have size at least N, *such as for example $P \wedge \bigwedge_{k=1}^{N-1} (a[k] = 0 \vee i < k)$.*

Lemma III.1. *There exists a sequence of state-transition systems $\mathfrak{S}_N$ and a property P, such that for any N the property P is N-inductive in $\mathfrak{S}_N$, but the shortest inductive strengthening of P has a size larger than N.*

The relationship between induction and k-induction is explored in [5], where the authors show that induction is as powerful as k-induction for theories that admit "feasible interpolation." Feasible interpolation ensures that the inductive strengthening is polynomial in the size of the proof. This is in line with Lemma III.1, since the proof itself can be exponential, resulting in potential blowup of the invariant.

Example III.2. *Consider the program in Figure 2. This program sets the elements $a[0], \ldots, a[i], \ldots$ to 0 in rounds of N steps. Variable c counts the number of steps in the current round and variable J stores the indices of the elements of a that have been written to. The program can be encoded as a transition system in theory $\mathcal{T}_{\mathbf{arr}}^{\mathbf{c}}$, with initial states defined*

[3]This is the same idea as induction relative to $\mathcal{F}$ used in [7].

[4]If P is k-inductive then $P \wedge \circ P \wedge \circ\circ P \wedge \ldots \wedge \overbrace{\circ\circ\cdots\circ}^{k-1} P$ is inductive, where $\circ$ stands for "next state".

[5]We postulate that, for theories such as pure Boolean logic or linear arithmetic, k-induction is exponentially more succinct than induction. Proving this postulate would entail new complexity results on quantifier elimination (e.g., [33]) and would require a much more sophisticated analysis.

[6]For example, $\exists i \ . \ a[i] = 0$ does not have a quantifier-free equivalent.

```
 1  int c, i, j;
 2  set<int> J;       // int -> bool
 3  map<int, int> a; // int -> int
 4
 5  // I: write 0 at a[0]
 6  c = 0;
 7  a[0] = 0;
 8  J.insert(0)
 9
10  // T: write 0 at a[i] from a[j]
11  while (true) {
12    c = (c+1) % N;
13    if (c == 0) {
14      J.clear();
15      i = 0;
16      a[i] = 0;
17    } else {
18      i = rand();
19      j = J.pick_rand();
20      a[i] = a[j];
21    }
22    J.insert(i)
23  }
```

Fig. 2. Writing 0 to array a while keeping a set J of written-to indices, resetting every N steps.

by lines 6-8, and transition relation expressed by lines 12-22. As previously, let the property be $P \equiv (a[0] = 0)$ then P is invariant. By the pigeon-hole principle, P is $(N+1)$-inductive: Any sequence of $N+1$ states must include a "reset" that sets $i = 0$ and $J = \{0\}$. From this reset point, all transitions sets some $a[i]$ to 0 and adds i to J. However, there is no inductive strengthening P' of P. *For P' to be inductive, it would need to ensure that for all $j \in J$ we have $a[j] = 0$ but this can not be expressed in the quantifier-free fragment of theory $\mathcal{T}^{c}_{arr}$.*

Lemma III.2. *There exists a state-transition system $\mathfrak{S}$ and a property P, such that the property P is k-inductive for $k > 1$ but there is no inductive strengthening of P.*

The additional power of k-induction comes with a computational price: checking whether a property is k-inductive requires $k+1$ satisfiability checks and a potentially expensive unrolling of the transition relation. The method we propose in this paper alleviates this issue by using an incremental approach that minimizes the need for unrolling.

IV. Algorithm

We reason about transition systems in the satisfiability modulo theories (SMT) framework [2]. Specifically, we assume that the transition system is described in a theory where quantifier-free satisfiability is decidable.

A. Satisfiability Checking

Given a state formula F, we denote with $T[F]^k$ the unrolling of T to length k where F holds in the intermediate states. For $k > 1$, $T[F]^k(\vec{x}, \vec{x}')$ is then defined as

$$T(\vec{x}, \vec{w}_1) \wedge \bigwedge_{i=1}^{k-1} (F(\vec{w}_i) \wedge T(\vec{w}_i, \vec{w}_{i+1})) \wedge T(\vec{w}_{k-1}, \vec{x}')$$

where $\vec{w}$ are the state variables in the intermediate states. For $k = 0$ and $k = 1$, we set $T^0[F](\vec{x}, \vec{x}') \equiv (\vec{x} = \vec{x}')$ and $T^1[F](\vec{x}, \vec{x}') \equiv T(\vec{x}, \vec{x}')$. When $F \equiv \mathbf{true}$, we omit it and simply write T^k.

A basic step in our algorithms is to check the satisfiability of formulas of the form

$$A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y}) \ , \qquad (1)$$

where A, B, and C are state formulas. We denote by CHECK-SAT$(A, T[B]^k, C)$ an (SMT-based) procedure that checks satisfiability of formula (1), and returns a model if the formula is satisfiable. In addition, we require two artifacts from the SMT solver: *interpolants* and *generalizations*.

Definition IV.1 (Interpolant)**.** *If the formula* (1) *is unsatisfiable, a formula $J(\vec{y})$ is a* state interpolant *if*

1) *$A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{w}, \vec{y}) \Rightarrow J(\vec{y})$, and*
2) *$J(\vec{y})$ and $C(\vec{y})$ are inconsistent.*

Definition IV.2 (Generalization)**.** *If the formula* (1) *is satisfiable, we call a formula $G(\vec{x})$ a* state generalization *if*

1) *$A(\vec{x})$ and $G(\vec{x})$ are consistent, and*
2) *$G(\vec{x}) \Rightarrow \exists \vec{w}, \vec{y} \,.\, T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y})$.*

Interpolation provides forward learning. An interpolant over-approximates the set of states reachable from A via $T[B]^k$ and is enough to refute C. Generalization is the dual and provides backward learning. A generalization G is consistent with A and under-approximates the set of states that can reach C via $T[B]^k$.

Our notion of a state interpolant is more specific than the one usually considered in general interpolation, and our definition can be easily satisfied: formula $\neg C$ is always an interpolant (so interpolants exists in our case even if the underlying theory does not support general interpolation). Similarly, one can construct a generalization G from a model v of formula (1) by substitution (i.e., the formula $(A \wedge T[B]^k)[\vec{w}/v(\vec{w}), \vec{y}/v(\vec{y})]$ is a trivial generalization). Although correct, trivial interpolants and generalizations are not ideal for practical applications. In particular, they do not satisfy the following property.

Definition IV.3 (Finite Cover Property)**.** *An interpolation (resp. generalization) procedure has the* finite cover property *(is finite-covering) when, for a fixed $T[B]^k$ and A (resp C), it can only produce a finite number of distinct interpolants (resp. generalizations).*

Interpolation is a well-studied topic [28], [12] and it is available in several SMT solvers. Effective generalization in SMT was introduced in [24] (as model-based projection) for specific use in a PDR engine. There are known generalization methods for the theories of linear arithmetic [24], arrays [23], and algebraic data-types [6]. These methods have the finite cover property. On the other hand, most interpolation procedures are proof-based and do not ensure finite covering.

Both interpolation and generalization approximate quantifier elimination. For theories that admit quantifier elimination, one can construct precise interpolants by eliminating $\vec{x}$ and $\vec{w}$ from $A \wedge T[B]^k$ and precise generalizations by eliminating $\vec{w}$ and

$\vec{y}$ from $T[B]^k \wedge B$. Such precise procedures have the finite cover property, and it is not unreasonable to expect the same from interpolation and generalization. This is the case for pure SAT problems. In SAT, interpolants can always be expressed as clauses, while generalizations can be expressed as prime implicants, both of which guarantee the finite cover property.[7] The finite-cover property for interpolants is an incremental form of the related notion of uniform interpolation [30]: uniform interpolation requires a single interpolant instead of a finite set.

Example IV.1. *Consider the system $\mathfrak{S} = \langle I, T\rangle$ defined as $I \equiv (x = 0)$, $T \equiv (x' = x + 1)$ where x is a real-valued variable. Let P be the formula $0 \le x \vee x \ge 1$. To check whether P is inductive, we can ask the following satisfiability query to the SMT solver*

$$\overbrace{(0 \le x \vee x \ge 1)}^{A_1} \wedge \overbrace{(x' = x + 1)}^{T^1} \wedge \overbrace{\neg(0 \le x' \vee x' \ge 1)}^{B_1 \equiv \neg A_1} \ .$$

This formula is satisfiable in a model $x \mapsto -0.5$, $x' \mapsto 0.5$. We can generalize this model to $G \equiv (-1 < x < 0)$; any state that satisfies G is a counterexample to induction of P. We can then check whether G intersects with the initial states and whether G is reachable in one step, by making two separate queries

$$\overbrace{(x = 0)}^{A_2} \wedge \overbrace{(x' = x)}^{T^0} \wedge \overbrace{(-1 < x' < 0)}^{B_2} \ ,$$
$$\underbrace{(x = 0)}_{A_3} \wedge \underbrace{(x' = x + 1)}_{T^1} \wedge \underbrace{(-1 < x' < 0)}_{B_3} \ .$$

Both queries are unsatisfiable. From the first query, we can get an interpolant $J_0(x') \equiv (x' \ge 0)$ that refutes G in the initial states. From the second query, we can get an interpolant $J_1(x') \equiv (x' \ge 1)$ that refutes G after one transition. Although P itself is not inductive, the two interpolants give us a strengthening: the formula $P' \equiv P \wedge (J_0(x) \vee J_1(x))$ is inductive.

B. Reachability

Problem IV.1 (k-reachability)**.** *Given a state formula F that is not reachable in fewer then k steps, check whether F is reachable in k steps.*

The reachability problem can be solved by bounded model checking [3], but we discuss an alternative method that does not require unrolling the transition relation. We introduce the concept of k-interpolation as a way to learn from failures of k-reachability.

Definition IV.4 (k-interpolant)**.** *Given a system $\mathfrak{S}$ and state formula F that is unreachable in $\le k$ steps (system is k-inconsistent with F), a state formula J is a state k-interpolant for F if*

$$\mathfrak{S} \models_0^k J \ , \qquad J \textit{ and } F \textit{ are inconsistent} \ .$$

[7]For arithmetic theories, finite-covering interpolants can be generated using model-based procedures such as MCSat [14].

As with regular interpolation, the formula $\neg F$ itself is a trivial k-interpolant. We can also construct a k-interpolant by calling a standard interpolation procedure $k + 1$ times: If $\mathfrak{S}$ and F are k-inconsistent, then $I(\vec{x}) \wedge T^i(\vec{x}, \vec{w}, \vec{x}') \wedge F(\vec{x}')$ is unsatisfiable for $0 \le i \le k$. From these inconsistencies we can obtain interpolants $J_0, \ldots, J_k$, and the formula $J \equiv (J_0 \vee \ldots \vee J_k)$ is a k-interpolant for F. Moreover, if the interpolation procedure has the finite-cover property then so does the k-interpolation procedure.

To check k-reachability, we adopt the incremental depth-first reachability method of IC3, which relies on local reasoning. The procedure maintains a sequence $\mathcal{R}_0$, $\mathcal{R}_1$, …of *reachability frames*. Frame $\mathcal{R}_i$ is a set of state formulas that over-approximates the set of states reachable in i steps or less. This implies that $\mathfrak{S} \models_0^i \mathcal{R}_i$. Unlike IC3/PDR, we *do not require the frames to be monotonic*; we may have $\mathcal{R}_{i+1} \not\subseteq \mathcal{R}_i$.[8]

This setup allows us to build k-interpolants efficiently provided an extra local condition holds. If $k = 0$, we just take the interpolant of $I \wedge T^0 \wedge F$. If $k > 0$ and the formula F is not reachable in up to k steps, and if, in addition, F is not reachable in one step from $\mathcal{R}_{k-1}$, then both $I \wedge T^0 \wedge F$ and $\mathcal{R}_{k-1} \wedge T \wedge F$ are inconsistent. We can then obtain interpolants J_1 and J_2 for these two formulas and $(J_1 \vee J_2)$ is a k-interpolant for F. This k-interpolant, which we denote by EXPLAIN$(\mathfrak{S}, k, F)$, is potentially more concise than the one described before and it is obtained by local reasoning only. Although EXPLAIN has an additional precondition, our algorithm ensures that this holds whenever EXPLAIN is called.

Lemma IV.1. *Starting from a fixed finite frame sequence $\mathcal{R}_0, \mathcal{R}_1, \ldots$, if the only formulas we add to the frames are obtained through the* EXPLAIN *procedure, and the interpolation procedure is finite-covering, then the* EXPLAIN *procedure is also finite-covering.*

Algorithm 1 Check k-reachability of F.

Require: $\mathfrak{S} \models_0^i \mathcal{R}_i$ for $0 \le i \le k$, $\mathfrak{S} \models_0^{k-1} \neg F$.
Ensure: $\mathfrak{S} \models_0^i \mathcal{R}_i$ for $0 \le i \le k$. If not reachable, $\mathcal{R}_{k-1} \wedge T \wedge F$ is unsatisfiable.

```
function REACHABLE($\mathfrak{S}$, $k$, $F$)
    if $k = 0$ then return CHECK-SAT($I, T^0, F$)
    loop
        if CHECK-SAT($\mathcal{R}_{k-1}, T, F$) then
            $G \leftarrow$ GENERALIZE($\mathcal{R}_{k-1}, T, F$)
            if REACHABLE($\mathfrak{S}, k-1, G$) then
                return true
            else
                $E \leftarrow$ EXPLAIN($\mathfrak{S}, k-1, G$)
                $\mathcal{R}_{k-1} \leftarrow \mathcal{R}_{k-1} \cup \{E\}$
        else return false
```

Finally, our reachability routine REACHABLE$(\mathfrak{S}, k, F)$ performs a step-wise search for a concrete trace by using a depth-first search strategy. It tries to reach the initial states

[8]From an implementation perspective, this gives flexibility in garbage collection. We can remove any subset of formulas from any frame $\mathcal{R}_i$ without compromising correctness.

backwards. To reach F at frame k, we check first whether F can be reached in one transition from the previous frame $\mathcal{R}_{k-1}$. If no such transition is possible, then F is not reachable. Otherwise, we get a state s that satisfies $\mathcal{R}_{k-1}$ and from which F is reachable in one step. The generalization procedure gives us a formula G that generalizes s: every state that satisfies G has a successor that satisfies F. We then recursively check whether G is reachable. The recursive call will either find a path from the initial states to G, in which case F is reachable, or determine that G is not reachable, in which case we can learn an explanation E of the reachability failure and eliminate G. Learning E eliminates G as a potential step backward, and we continue.

Lemma IV.2. *Algorithm 1 solves the k-reachability problem. If F is not reachable then, upon completion, either $k = 0$ and F is inconsistent with I, or $k > 0$ and F is not reachable in one step from $\mathcal{R}_{k-1}$. In addition, if the interpolation or the generalization procedure is finite-covering, then the procedure always terminates.*

We use a variant of the REACHABLE procedure to check whether F is reachable in steps k_1 to k_2. We denote this by $(r, l) = \text{REACHABLE}(\mathfrak{S}, k_1, k_2, F)$. This extension of the REACHABLE procedure is a straightforward loop from k_1 to k_2, and has the same precondition as the single check version (on k_1). In the return value, r denotes the reachability result (true/false), and, if r is true, l is the length of the shortest trace that can reach F. The postcondition (and hence Lemma IV.2) of the iterative extension is also the same (on k_2).

C. Property-Directed k-Induction

We now present the main procedure of PD-KIND. This procedure checks whether a property P is invariant for a system $\mathfrak{S} = \langle I, T \rangle$. It does so by iteratively trying to construct a k-inductive strengthening of P for some $k > 0$. The overall idea behind the procedure is simple. Assume a set of formulas $\mathcal{F}_{\text{ABS}}$ that is a strengthening of P and is valid in $\mathfrak{S}$ for up to n steps. In other words, $\mathcal{F}_{\text{ABS}}$ is an over-approximation of states reachable in n steps or less. Then, the set $\mathcal{F}_{\text{ABS}}$ satisfies (k-init) for all $1 \leq k \leq n+1$. We can pick any such k and try to show that $\mathcal{F}_{\text{ABS}}$ is k-inductive by checking whether it also satisfies (k-cons). Each iteration of the procedure PD-KIND does this check. The core of our algorithm is procedure PUSH that either finds a counter-example to P or produces a new strengthening $\mathcal{G}_{\text{ABS}} \subseteq \mathcal{F}_{\text{ABS}}$. This new set $\mathcal{G}_{\text{ABS}}$ satisfies (k-cons) with respect to $\mathcal{F}_{\text{ABS}}$. The set $\mathcal{G}_{\text{ABS}}$ is then $\mathcal{F}^k_{\text{ABS}}$-inductive. If $\mathcal{G}_{\text{ABS}} = \mathcal{F}_{\text{ABS}}$, we can conclude that P is invariant. Otherwise, we know that $\mathcal{G}_{\text{ABS}}$ is valid (at least) up to index $n + 1$. Procedure PUSH actually returns an integer n_p such that $\mathcal{G}_{\text{ABS}}$ is valid up to n_p. This index n_p is the length of the shortest trace of $\mathfrak{S}$ that reaches $\neg\mathcal{F}_{\text{ABS}}$; it is guaranteed to be at least $n+1$ but it may be larger. At this point, we repeat the loop with $\mathcal{G}_{\text{ABS}}$ as our current strengthening and n_p as our new index.

In addition to the set of formulas $\mathcal{F}_{\text{ABS}}$, procedure PD-KIND associates with each each $F_{\text{ABS}} \in \mathcal{F}_{\text{ABS}}$ information about a potential counter-example to P that the formula F_{ABS} eliminates. The set $\mathcal{F}_{\text{ABS}}$ and this additional information is represented in the form of an induction frame. Let $\mathbb{F}$ denote the set of all state formulas in our theory.

Algorithm 2 Main PD-KIND procedure.

```
Require: $\mathfrak{S} = \langle I, T \rangle$ and $I \Rightarrow P$
function PD-KIND($\mathfrak{S}, P$)
    $n \leftarrow 0$
    $\mathcal{F} \leftarrow \{(P, \neg P)\}$
    loop
        pick $k$-induction depth $1 \leq k \leq n + 1$
        $\langle \mathcal{F}, \mathcal{G}, n_p \rangle \leftarrow$ PUSH($\mathfrak{S}, \mathcal{F}, P, n, k$)
        if $P$ marked invalid then return invalid
        if $\mathcal{F} = \mathcal{G}$ then return valid
        $n \leftarrow n_p$
        $\mathcal{F} \leftarrow \mathcal{G}$
```

Definition IV.5 (Induction Frame)**.** *A set of tuples $\mathcal{F} \subset \mathbb{F} \times \mathbb{F}$ is an* induction frame *at index n if $(P, \neg P) \in \mathcal{F}$ and the following holds for all $(F_{\text{ABS}}, F_{\text{CEX}}) \in \mathcal{F}$:*

1) *F_{ABS} is valid up to n steps and refutes F_{CEX}, and*
2) *F_{CEX}-states can be extended to a counter-example to P.*

If $I \Rightarrow P$, then the set $\mathcal{F} = \{(P, \neg P)\}$ is an induction frame at index 0. Given an induction frame $\mathcal{F}$, we denote by $\mathcal{F}_{\text{ABS}}$ the strengthening represented by $\mathcal{F}$, i.e., $\mathcal{F}_{\text{ABS}} = \{F_{\text{ABS}} \mid (F_{\text{ABS}}, F_{\text{CEX}}) \in \mathcal{F}\}$. With this in mind, the procedure PD-KIND is presented in Algorithm 2.

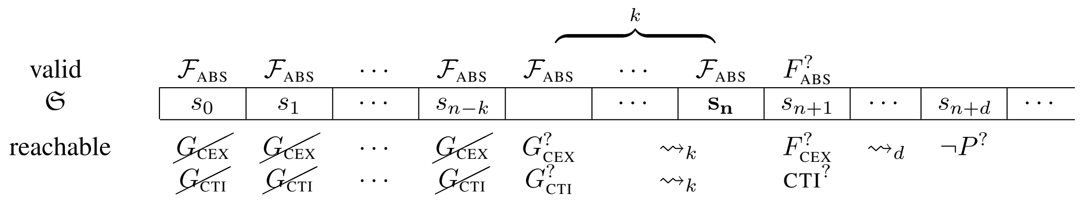
D. The PUSH Procedure

The core of the PD-KIND algorithm is the PUSH procedure (Algorithm 3). This procedure takes as input an induction frame $\mathcal{F}$ at index n, and tries to push formulas of the frame using k-induction where $1 \leq k \leq n + 1$. Figure 3 illustrates the formulas and frame indices over which PUSH operates.

Since $\mathcal{F}$ is an induction frame at n, we know that $\mathcal{F}_{\text{ABS}}$ is valid up to index n. In each iteration, the procedure picks one yet unprocessed $(F_{\text{ABS}}, F_{\text{CEX}})$ from $\mathcal{F}$. Both F_{ABS} and $\neg F_{\text{CEX}}$ hold up to index n in $\mathfrak{S}$.

First, the procedure checks whether F_{ABS} is $\mathcal{F}^k_{\text{ABS}}$-inductive (lines 9-12). If so, then we know that F_{ABS} is valid at least up to position $n + 1$. We call this a successful push and we add $(F_{\text{ABS}}, F_{\text{CEX}})$ to the set of pushed obligations $\mathcal{G}$, and continue with the next obligation. If the k-induction check fails, then we have a model (counterexample to induction) m_{CTI}. This is a trace of length $k + 1$ in which $\mathcal{F}_{\text{ABS}}$ holds for the first k states but F_{ABS} is false in the last state.

The procedure does not use m_{CTI} yet. Instead, it checks whether the counterexample formula F_{CEX} is reachable from $\mathcal{F}_{\text{ABS}}$ (lines 15-24). If the query at line 15 is satisfiable, it has a model m_{CEX}. Like m_{CTI}, this model is a trace of length $k + 1$; it starts with k states that satisfy $\mathcal{F}_{\text{ABS}}$ and ends with a state that satisfies F_{CEX} (thus, from the first state of m_{CEX} we can reach $\neg P$). At this point, we generalize m_{CEX} to a formula G_{CEX}. From any state that satisfies G_{CEX}, one can reach $\neg P$. Formula G_{CEX} is then a potential counterexample for P. We check whether G_{CEX} is reachable from the initial



valid
$\mathfrak{S}$
reachable
k
$\mathcal{F}_{\text{ABS}}$
$F^?_{\text{ABS}}$
s_0 s_1 s_{n-k} $\mathbf{s_n}$ s_{n+1} s_{n+d}
$\cancel{G_{\text{CEX}}}$ $G^?_{\text{CEX}}$ $\rightsquigarrow_k$ $F^?_{\text{CEX}}$ $\rightsquigarrow_d$ $\neg P^?$
$\cancel{G_{\text{CTI}}}$ $G^?_{\text{CTI}}$ $\rightsquigarrow_k$ $\text{CTI}^?$

Fig. 3. Illustration of the formulas and frame indices over which PUSH operates.

states of $\mathfrak{S}$. Because we know that F_{CEX} is not n-reachable, G_{CEX} can't be reached in less than $n-k+1$ steps. So we check reachability of G_{CEX} at positions $n-k+1\ \ldots n$. If G_{CEX} is reachable, then so is $\neg P$ and we mark P as invalid. Otherwise, we call the EXPLAIN procedure, which returns a new fact G_{ABS} that eliminates G_{CEX}. The new fact G_{ABS} is true up to position n, and refutes G_{CEX}, so we can add the new induction obligation $(G_{\text{ABS}}, G_{\text{CEX}})$ to $\mathcal{F}$, strengthening $\mathcal{F}$, and try again with a potential counter-example eliminated.

In the remaining case, we have a counterexample m_{CTI} to the k-inductiveness of F_{ABS}. Since the query at line 15 is not satisfiable and $\mathcal{F}_{\text{ABS}} \Rightarrow \neg F_{\text{CEX}}$, we know that $\neg F_{\text{CEX}}$ is $\mathcal{F}^k_{\text{ABS}}$ inductive. We first apply generalization to m_{CTI} to construct a formula G_{CTI}. From any state that satisfies G_{CTI}, we can reach $\neg F_{\text{ABS}}$ in k steps. If G_{CTI} is reachable in $\mathfrak{S}$ then $\neg F_{\text{ABS}}$ is also reachable, so F_{ABS} can't be part of a valid strengthening of P. This check is performed at line 28; as previously, it is enough to check reachability of G_{CTI} at positions $n-k+1, \ldots, n$. If G_{CTI} is reachable, we can't push F_{ABS}. Instead, we replace the triple $(F_{\text{ABS}}, F_{\text{CEX}})$ by the weaker obligation $(\neg F_{\text{CEX}}, F_{\text{CEX}})$. This new obligation can be immediately pushed to $\mathcal{G}$. On the other hand, if G_{CTI} is not reachable then we strengthen F_{ABS} with a new fact G_{ABS} learned from procedure EXPLAIN. This eliminates the counterexample to k-induction and the procedure continues.

At lines 30-31 of the procedure, we know that $\neg F_{\text{ABS}}$ is reachable in $\mathfrak{S}$ and that this requires at least $n+1$ transitions. It is useful to make this more precise by computing the actual length of the shortest path to $\neg F_{\text{ABS}}$ (line 30). This length is stored in variable n_p (if it's smaller than n_p's current value).

After the loop terminates, PUSH returns the set of successfully pushed induction obligations $\mathcal{G}$, the modified set $\mathcal{F}$ of k-induction assumptions for $\mathcal{G}$, and the shortest refutation length n_p for any $F_{\text{ABS}} \in \mathcal{F}_{\text{ABS}}$ that was not successfully pushed. The procedure does not only add to the original set $\mathcal{F}$, it also actively modifies it (line 37). Unlike existing IC3-based procedures where frames are explored in succession, keeping track of n_p allows us to perform "jumps" that move to deeper frames faster. This is because $\mathcal{F}_{\text{ABS}}$ is valid up to position $n_p - 1 \geq n$, and the facts in $\mathcal{G}_{\text{ABS}}$ are valid up to position $n_p \geq n+1$.[9]

Assuming that PD-KIND terminates, it is not hard to show that it returns the correct result. If PD-KIND terminates with P marked invalid, then we have found a counter-example

[9]We have observed significant frame jumps in practice although this is problem-specific.

Algorithm 3 Push $\mathcal{F}$ with k-induction.

Require: $\mathcal{F}$ is a valid frame for P at position n, $1 \leq k \leq n+1$, $(P, \neg P) \in \mathcal{F}$.
Ensure: $\mathcal{F}$ is a valid frame for P at position $n_p - 1 \geq n$, $\mathcal{G} \subseteq \mathcal{F}$ is $\mathcal{F}^k$-inductive, and P marked invalid or $(P, \neg P, 0) \in \mathcal{F}_p$.

```
1  function PUSH($\mathfrak{S}$, $\mathcal{F}$, $P$, $n$, $k$)
2      push elements of $\mathcal{F}$ to $\mathcal{Q}$            ▷ $\mathcal{Q}$ is a priority queue.
3      $\mathcal{G} \leftarrow \{\}$           ▷ Pushed facts, i.e. $\mathcal{G}_{\text{ABS}}$ is $\mathcal{F}^k_{\text{ABS}}$-inductive.
4      $n_p \leftarrow n + k$       ▷ Keeps track of the shortest CTI position.
5      while $P$ not marked invalid, $\mathcal{Q}$ not empty do
6          pop $(F_{\text{ABS}}, F_{\text{CEX}})$ from $\mathcal{Q}$
7
8          ______________                         ▷ Is $F_{\text{ABS}}$ $\mathcal{F}^k_{\text{ABS}}$-inductive?
9          $(sat_{\text{CTI}}, m_{\text{CTI}}) \leftarrow$ CHECK-SAT$(\mathcal{F}_{\text{ABS}}, T[\mathcal{F}_{\text{ABS}}]^k, \neg F_{\text{ABS}})$
10         if not $sat_{\text{CTI}}$ then
11             $\mathcal{G} \leftarrow \mathcal{G} \cup \{(F_{\text{ABS}}, F_{\text{CEX}})\}$     ▷ $\mathcal{G}_{\text{ABS}}$ is $\mathcal{F}^k_{\text{ABS}}$-inductive.
12             continue
13
14         ________________                          ▷ Is $F_{\text{CEX}}$ reachable?
15         $(sat_{\text{CEX}}, m_{\text{CEX}}) \leftarrow$ CHECK-SAT$(\mathcal{F}_{\text{ABS}}, T[\mathcal{F}_{\text{ABS}}]^k, F_{\text{CEX}})$
16         if $sat_{\text{CEX}}$ then
17             $G_{\text{CEX}} \leftarrow$ GENERALIZE$(m_{\text{CEX}}, T^k, F_{\text{CEX}})$
18             if REACHABLE$(\mathfrak{S}, n-k+1, n, G_{\text{CEX}})$ then
19                 mark $P$ invalid      ▷ $I \rightsquigarrow G_{\text{CEX}} \rightsquigarrow_k F_{\text{CEX}} \rightsquigarrow \neg P$.
20             else
21                 $G_{\text{ABS}} \leftarrow$ EXPLAIN$(\mathfrak{S}, n, G_{\text{CEX}})$
22                 $\mathcal{F} \leftarrow \mathcal{F} \cup \{(G_{\text{ABS}}, G_{\text{CEX}})\}$        ▷ Eliminate CEX.
23                 push $(G_{\text{ABS}}, G_{\text{CEX}})$, $(F_{\text{ABS}}, F_{\text{CEX}})$ to $\mathcal{Q}$
24             continue
25
26         ___________                    ▷ Analyze the induction failure.
27         $G_{\text{CTI}} \leftarrow$ GENERALIZE$(m_{\text{CTI}}, T^k, \neg F_{\text{ABS}})$
28         $(r_{\text{CTI}}, n_{\text{CTI}}) \leftarrow$ REACHABLE$(\mathfrak{S}, n-k+1, n, G_{\text{CTI}})$
29         if $r_{\text{CTI}}$ then                  ▷ $I \rightsquigarrow_{n_{\text{CTI}}} G_{\text{CTI}} \rightsquigarrow_k \neg F_{\text{ABS}}$.
30             $(r_{\text{CTI}}, n_{\text{CTI}}) \leftarrow$ REACHABLE$(\mathfrak{S}, n{+}1, n_{\text{CTI}}{+}k, \neg F_{\text{ABS}})$
31             $n_p \leftarrow \min(n_p, n_{\text{CTI}})$
32             $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\neg F_{\text{CEX}}, F_{\text{CEX}})\}$
33             $\mathcal{G} \leftarrow \mathcal{G} \cup \{(\neg F_{\text{CEX}}, F_{\text{CEX}})\}$
34         else                                ▷ $I \not\rightsquigarrow_{\leq n} G_{\text{CTI}} \rightsquigarrow_k \neg F_{\text{ABS}}$.
35             $G_{\text{ABS}} \leftarrow$ EXPLAIN$(\mathfrak{S}, n, G_{\text{CTI}})$      ▷ $G_{\text{ABS}} \Rightarrow \neg G_{\text{CTI}}$.
36             $G_{\text{ABS}} \leftarrow F_{\text{ABS}} \wedge G_{\text{ABS}}$               ▷ $G_{\text{ABS}} \Rightarrow \neg F_{\text{CEX}}$.
37             $\mathcal{F} \leftarrow \mathcal{F} \cup \{(G_{\text{ABS}}, F_{\text{CEX}})\} \setminus \{(F_{\text{ABS}}, F_{\text{CEX}})\}$
38             push $(G_{\text{ABS}}, F_{\text{CEX}})$ to $\mathcal{Q}$.
39
40     return $\langle \mathcal{F}, \mathcal{G}, n_p \rangle$
```

to the property. On the other hand, if PD-KIND terminates when the inductive frames become equal, i.e. $\mathcal{F} = \mathcal{G}$, then $\mathcal{F}_{\text{ABS}}$ is a k-inductive strengthening of P and P is therefore valid. In general, for infinite domains, even termination of the PUSH procedure is not guaranteed. But, a finite-covering interpolation procedure ensures termination: the number of new facts that PUSH can learn is finite, and this bounds both the number of possible refinement steps, and the number of new counter-examples that can be found in line 15.

The PD-KIND procedure, as presented, has the freedom to choose the induction depth k in each iteration (line 5). We call a strategy for picking the depth increasing if it guarantees that, for every k, PD-KIND eventually picks induction depths k' larger than k.

Lemma IV.3. *If the interpolation procedure is finite-covering, then the* PUSH *procedure terminates. If the property P is k-inductive for some $k > 0$, and* PD-KIND *uses an increasing strategy for k, then the* PD-KIND *procedure terminates.*

Proof. (Sketch) If the property P is k inductive, then PD-KIND will eventually pick only depths $k' \geq k$. For any such k' no counterexamples can be found at line 15, because any m_{CEX} could be extended to a counter-example of P, violating the assumption that P is k-inductive. If no new counter-examples can be found then, as PD-KIND goes from frame to frame, the only new facts that can be added to the frame are either obtained from refinement on line 35, where existing facts are replaced with stronger facts, or line 36, where facts are weakened to a counter-example refutation. Since we know that no new counter-examples can be found, the latter can only happen a finite number of times. Therefore, the size of the frame can not increase indefinitely, and will eventually converge to a state where $\mathcal{F} = \mathcal{G}$. □

V. Experimental Evaluation

We have implemented PD-KIND in the SALLY model-checker.[10] The implementation of the procedure itself is rather small (1.2 Kloc of C++) and follows the presentation of the paper. As our back-end SMT solver we combine YICES2 [17], [18] and MATHSAT5 [11]. YICES2 is used for all satisfiability queries and for generalization, while MATHSAT5 is used for interpolation. We use two solvers because YICES2 supports model-based generalization (but not interpolation), and MATHSAT5 supports interpolation (but not generalization at the time we started implementing SALLY). This combination incurs some overhead as we solve every unsatisfiable problem twice, once with YICES2 to know that the problem is unsatisfiable and once with MATHSAT5 to get an interpolant.[11] The default strategy for picking the parameter k in PUSH is to increment by one in each iteration.

We have evaluated the new procedure on a range of existing and new benchmarks. Several of our benchmarks are related to fault-tolerant algorithms (`oral-messages` [25], `tte-synchro` and `tta-startup` [19], `unified-approx` [29], `azadmanesh-kieckhafer` [1], `approximate-agreement` [26], and `hacms` problem sets). We also used benchmarks from software model checking (`cav12` [9], `ctigar` [4]). The `lustre` benchmarks are from the benchmark suite of the KIND model-checker, and `cons` are simple concurrent programs. Some of the benchmarks were obtained from an existing repository [12]. Since our tool does not yet handle integer reasoning properly, we converted all the integer variables to the real type. All problems are flat transition systems and we translated them to the input languages of other tools in a straightforward manner. The software benchmarks come from a public repository and were already encoded in SMTLIB2 as flat transition systems.[13]

To put the results in context, we compare PD-KIND with other state-of-the-art, infinite-state model checkers, namely, Z3 [22], NUXMV [10], and SPACER [24]. The results are presented in Table 4. Each solver was run with a timeout of 20 minutes. Each column of the table corresponds to a different model-checking engine, and each row corresponds to a different problem set. For each problem set and tool combination we report the number of problems that the tool has solved, how many of the solved problems were valid and invalid properties, and the total time (in seconds) that the tool took to solve those problems.

The evaluation shows that the new method is effective and robust on real-world problems: on all problem sets, PD-KIND either solves the most problems or is very close to the best engine. PD-KIND is good at both proving properties correct and finding counter-examples. When proving invariants, PD-KIND benefits from k-induction and can in some cases prove the properties using a substantially smaller strengthening than the inductive engines. Moreover, PD-KIND is the only engine that can prove all properties that are already k-inductive. On the other hand, PD-KIND is also effective as a bug-finder due to the longer steps of k-induction. As an example of this, in one of the `hacms` examples, PD-KIND finds a counter-example of length 60 already at frame 15. The comparison is not exhaustive or definitive: we have not included tools such as KIND, and we used all the model checkers with default options. It is likely that they could be tuned to perform better on the particular benchmarks we have chosen.

VI. Conclusion

The k-induction principle is a popular method for proving safety of infinite-state systems. We have shown that k-induction can be more powerful than regular induction for expressive theories such as the theory of arrays. With this in mind, we proposed PD-KIND, a reformulation of the IC3 method that allows us to integrate k-induction into the method. We have implemented the new procedure in the SALLY tool, and the experimental evaluation shows that our prototype

[10]SALLY is open source and available at http://sri-csl.github.io/sally/.

[11]On the other hand, with no burden of proof-production, YICES2 is much faster on satisfiable queries.

[12]https://es-static.fbk.eu/people/griggio/vtsa2015/

[13]All benchmarks can be downloaded from http://csl.sri.com/~dejan/sally-benchmarks.tar.gz.

Fig. 4. Experimental evaluation. Each row corresponds to a different problem set. Each column corresponds to a different engine. Each table entry shows the number of problems that the engine solved, how many of those were valid and invalid, and the total time it took for the solved instances.

	Z3			SPACER			NUXMV			PD-KIND		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
approximate-agreement (9)	9	8/1	213	7	6/1	1150	9	8/1	2174	**9**	**8/1**	**164**
azadmanesh-kieckhafer (20)	20	17/3	3404	20	17/3	4678	20	17/3	294	**20**	**17/3**	**192**
cav12 (99)	69	48/21	2102	71	49/22	3529	**72**	**50/22**	**7443**	71	49/22	4990
conc (6)	4	4/0	128	4	4/0	655	**6**	**6/0**	**421**	4	4/0	270
ctigar (110)	64	44/20	1683	72	52/20	4249	76	56/20	1342	**77**	**57/20**	**2823**
hacms (5)	1	1/0	11	1	1/0	4	4	3/1	388	**5**	**3/2**	**1661**
lustre (790)	757	421/336	1888	763	427/336	2263	760	424/336	7660	**774**	**438/336**	**3494**
oral-messages (9)	9	7/2	16	9	7/2	44	9	7/2	161	**9**	**7/2**	**2**
tta-startup (3)	1	1/0	9	**1**	**1/0**	**8**	1	1/0	17	**1**	**1/0**	**8**
tte-synchro (6)	6	3/3	969	6	3/3	445	5	2/3	405	**6**	**3/3**	**21**
unified-approx (11)	8	5/3	2928	11	8/3	589	**11**	**8/3**	**139**	11	8/3	217

is effective at solving real-world problems. In addition, the new method is more powerful then k-induction, which is a novel and theoretically pleasing property: if the property being checked is k-inductive (for some k), then (modulo a requirement on the interpolation procedure) the method always terminates. It can also prove properties that are not k-inductive. When limiting the induction depth $k = 1$, the method can be seen as an effective instance of the IC3/PDR class of algorithms.

References

[1] M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

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

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[4] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification*, pages 831–848, 2014.

[5] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. 2015.

[6] N. Bjørner and M. Janota. Playing with quantified satisfaction. Logic for Programming, Artificial Intelligence and Reasoning, 2015.

[7] A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, 2011.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442, 2006.

[9] A. Cimatti and A. Griggio. Software model checking via IC3. In *Computer Aided Verification*, pages 277–293, 2012.

[10] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. 2014.

[11] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. 2013.

[12] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.

[13] L. De Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, pages 45–52, 2009.

[14] L. De Moura and D. Jovanović. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–12, 2013.

[15] L. De Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. *Lecture notes in computer science*, pages 14–26, 2003.

[16] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Static Analysis*, pages 351–368. 2011.

[17] B. Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744, 2014.

[18] B. Dutertre. Solving exists/forall problems with Yices. In *SMT Workshop*, 2015.

[19] B. Dutertre, A. Easwaran, B. Hall, and W. Steiner. Model-based analysis of timed-triggered ethernet. In *Digital Avionics Systems Conference*, pages 9D2–1, 2012.

[20] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design*, pages 125–134, 2011.

[21] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *Automated Reasoning*, pages 22–29. 2010.

[22] K. Hoder and N. Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing*, pages 157–171. 2012.

[23] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Formal Methods in Computer-Aided Design*, pages 89–96, 2015.

[24] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification*, pages 17–34, 2014.

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[26] N. A. Lynch. *Distributed algorithms*. 1996.

[27] J. McCarthy. Towards a mathematical science of computation. In *Program Verification*, pages 35–56. 1993.

[28] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[29] P. Miner, A. Geser, L. Pike, and J. Maddalon. A unified fault-tolerance protocol. In *FORMATS/FTRTFT*, pages 167–182, 2004.

[30] A. M. Pitts. On an interpretation of second order quantification in first order intuitionistic propositional logic. *The Journal of Symbolic Logic*, 57(01):33–52, 1992.

[31] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 127–144, 2000.

[32] A. Stump, C. W. Barrett, and D. L. Dill. A decision procedure for an extensional theory of arrays. In *In 16th IEEE Symposium on Logic in Computer Science*, 2001.

[33] V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1):3–27, 1988.

Lazy Proofs for DPLL(T)-Based SMT Solvers

Guy Katz, Clark Barrett
New York University

Cesare Tinelli, Andrew Reynolds
The University of Iowa

Liana Hadarean
Synopsys Inc.

***Abstract*—With the integration of SMT solvers into analysis frameworks aimed at ensuring a system's end-to-end correctness, having a high level of confidence in these solvers' results has become crucial. For unsatisfiable queries, a reasonable approach is to have the solver return an independently checkable proof of unsatisfiability. We propose a lazy, extensible and robust method for enhancing DPLL(T)-style SMT solvers with proof-generation capabilities. Our method maintains separate Boolean-level and theory-level proofs, and weaves them together into one coherent artifact. Each theory-specific solver is called upon lazily, a posteriori, to prove precisely those solution steps it is responsible for and that are needed for the final proof. We present an implementation of our technique in the CVC4 SMT solver, capable of producing unsatisfiability proofs for quantifier-free queries involving uninterpreted functions, arrays, bitvectors and combinations thereof. We discuss an evaluation of our tool using industrial benchmarks and benchmarks from the SMT-LIB library, which shows promising results.**

I. Introduction

Many different tools for system analysis and verification exploit the reasoning capabilities of SMT solvers. Typically, these tools dispatch satisfiability queries to an SMT solver and then use the returned results to prove or disprove various system properties. Thus, one's ability to rely on the outcome of the analysis depends on the level of confidence in the results returned by the underlying SMT solver. Unfortunately, obtaining the high level of trust required for, e.g., safety-critical systems can be difficult, as the solvers themselves are highly complex tools and may contain errors.

One reasonable approach to increasing one's level of confidence in an SMT solver's answers is to have it produce solution certificates checkable by simpler, external tools. In the case of a satisfiable (quantifier-free) query, a natural certificate is a *satisfying assignment* for the input formula, which typically can be checked by straightforward means. In the unsatisfiable case, the natural counterpart of a satisfying assignment is a *proof* certificate, which details how to derive a contradiction from the input assertions using a reasonably small set of trusted inference rules. Proof certificates can then be checked by a small trusted proof-checker, thus removing the need to trust the SMT solver.

Proof certificates provide several additional benefits. For instance, they can be used for interpolant generation [23] and certified compilation [11]. Notably, they can be used also to improve the performance of *skeptical proof assistants*. The proof assistant discharges subgoals to the SMT solver and then uses the proof certificates to internally reconstruct a proof [1], [7], [13].

The project or effort depicted was sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-13-2-0241 and FA8750-15-C-0113. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL or DARPA.

Instrumenting SMT solvers to generate proofs is a complex task. One challenge is that modern solvers reason about their input on multiple levels: typically an underlying SAT engine performs Boolean reasoning, whereas multiple dedicated *theory solvers* (e.g. array, arithmetic, and bitvector solvers) perform theory-specific deductions. The various components interact with each other in subtle ways—the theory solvers interact with the SAT engine and also with each other—and all of these interactions need to be properly captured in the produced proofs. Another challenge is to produce *fine-grained* proofs, i.e., proofs that are sufficiently detailed to be checked by simple means.

In previous work, we presented a proof generation technique for input queries in the logic of quantifier-free fixed-width bitvectors [15]. A main limitation there was that the technique was specifically tailored for that particular logic. In this work we make three major contributions that considerably enhance our previous approach:

1) We present and formalize a general approach for fine-grained proof generation in DPLL(T)-style SMT solvers. This approach is not limited to one specific theory (e.g, fixed-width bitvectors); in fact, it even supports proof generation for *combinations of theories*. We explain the approach in terms on an abstract description of DPLL(T) and also discuss ways to implement it in practice.
2) We demonstrate how our approach can be realized using *lazy proof generation*, which incurs a lower overhead. During search, an SMT solver will often generate a multitude of lemmas that are not actually needed to derive a contradiction from the input. Our lazy approach postpones proof construction for such lemmas until after the contradiction has been found, and then generates proofs just for those lemmas that were actually used.
3) We present lazy proof generation procedures for the *theory of uninterpreted functions with equality* and the *extensional theory of arrays*.

For evaluation purposes, we implemented our technique in CVC4, a state-of-the-art SMT solver [2]. We conducted extensive experiments using the relevant benchmarks from the SMT-LIB library [4]. Our tool was able to produce proofs in the vast majority of cases.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Before describing our work, we give a high-level description of the DPLL(T) framework for SMT solvers in Section II. Next, in Section III, we explain how proofs of unsatisfiability can be generated in a DPLL(T) setting. In Section IV we discuss our approach to lazy proof production, and in Section V we cover proof production for three theories: uninterpreted functions, arrays, and fixed-width bitvectors. An experimental evaluation of our approach is summarized in Section VI, followed by a discussion of related work in Section VII, and a few concluding remarks in Section VIII.

II. DPLL(T)-Based SMT Solvers

In its most general formulation, SMT is the problem of determining the satisfiability of a set of formulas in some background theory T. This work focuses on quantifier-free formulas and on SMT solvers based on the DPLL(T) architecture [21], which modularly combines a generic CDCL SAT solver (the *SAT engine*) with one or more reasoners (the *theory solvers*). Each theory solver decides the satisfiability of *constraints* (i.e., conjunctions of ground literals), in a specific background theory. Commonly supported theories include equality over uninterpreted functions (T_{UF}), linear arithmetic over the integers (T_{LIA}) or the reals (T_{LRA}), fixed-width bitvectors (T_{BV}), arrays (T_{AX}), and their combinations.

*Abstract DPLL(*T*) Framework.* We follow a recent abstract formalization of DPLL(T)-style SMT solvers by Reynolds *et al.* [22], which in turn is an elaboration of the one first introduced by Nieuwenhuis *et al.* [21]. We consider a background theory T that is a combination of m theories $T_1, \dots, T_m$ with respective many-sorted (i.e., typed) signatures $\Sigma_1, \dots, \Sigma_m$. For convenience, and without loss of generality, we assume that the theories have no predicate symbols besides equality[1] and that they all have the same set $\mathbf{S}$ of sort symbols. We also assume that the theories share no function symbols except for a set $\mathcal{C} = \bigcup_{S \in \mathbf{S}} \mathcal{C}_S$ of constant symbols (functions of arity 0), where each $\mathcal{C}_S$ is a distinguished infinite set of *free* (i.e., uninterpreted) constants of sort S.

DPLL(T) solvers can be formalized abstractly as state transition systems defined by a set of transition rules. The states of the transition system are either the distinguished state fail or triples of the form $\langle M, F, C \rangle$, where

- M, the current *context*, is a sequence of literals and *decision points* $\bullet$,
- F is a set of ground clauses derived from the original input formula, and
- C is either the empty set or a singleton set containing a ground clause, the current *conflict clause*.

Each context M can be factored uniquely into a concatenation of the form $M_0 \bullet M_1 \bullet \cdots \bullet M_n$, where the M_i's contain no decision points. For every $0 \leq i \leq n$ we call M_i the *i'th decision level* of M, and denote with $M^{[i]}$ the subsequence $M_0 \bullet \cdots \bullet M_i$. Each atom of a clause in $F \cup C$ is *pure*, in the sense that it has signature Σ_i for some $i \in \{1, \dots, m\}$. Note that two atoms in the same clause can have different signatures, and when they do they share at most the constants in $\mathcal{C}$. Input formulas can always be converted to this form while preserving satisfiability in T.

The initial state of the transition system is $\langle \emptyset, F_0, \emptyset \rangle$, where F_0 is a given set of clauses to be checked for satisfiability (i.e., the input formula). The expected final states are either fail, when F_0 is unsatisfiable in T, or $\langle M, F, \emptyset \rangle$ where M is satisfiable in T, F is equisatifiable with F_0 in T, and M propositionally entails F.

The possible behaviors of the system are defined by a set of non-deterministic transition rules that specify a set of successor states for any given state. These rules are depicted in Figure 1 in *guarded assignment form* [17].[2] A rule applies to a state s if all of its premises hold for s.

In the rules, M, F, and C denote, respectively, the context, clause set, and conflict component of the current state. The conclusion describes how each component is changed, if at all. We write $\bar{l}$ to denote the complement of literal l and $l \prec_{\mathsf{M}} l'$ to indicate that l occurs before l' in M. The function lev maps each literal of M to the (unique) decision level in which it occurs. The set Lit_{F} (resp., Lit_{M}) consists of all literals in F (resp., in M) and their complements. For $i = 1, \dots, m$, the set $\text{Lit}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of Lit_{M}. Int_{M} is the set of all *interface literals* of M: the equalities and disequalities between *shared constants*, where the set of shared constants is $\{c \mid \text{constant } c \text{ occurs in } \text{Lit}_{\mathsf{M}}|_i \text{ and } \text{Lit}_{\mathsf{M}}|_j, \text{ for some } 1 \leq i < j \leq m\}$. The index i for the rules Prop_i, Confl_i, Learn_i, and Expl_i ranges from 1 to m. In those rules, $\models_i$ denotes validity in the theory T_i. Clauses are implicitly processed modulo associativity, commutativity and idempotency of $\vee$.

Modeling Solver Behavior. Rules Dec, Prop, Expl, Confl, Fail, Learn, and Backj model the behavior of the SAT engine, which treats atoms as Boolean variables. In particular, Confl and Expl model the conflict discovery and analysis mechanism used by CDCL SAT solvers [18]. The remaining rules model the interaction between the SAT engine and the individual theory solvers within the overall SMT solver. The rules maintain the invariant that every conflict clause and learned clause is entailed in T by the initial clause set.

Generally speaking, the system uses the SAT engine to construct the context M as a truth assignment for the clauses in F, as if those clauses were propositional. However, it periodically asks the solver of each theory T_i to check if the set of Σ_i-constraints in M is unsatisfiable in T_i or entails some yet-undetermined literal from $\text{Lit}_{\mathsf{F}} \cup \text{Int}_{\mathsf{M}}$. In the first case, the theory solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by rule Confl_i. The propagation of entailed theory literals and the extension of the conflict analysis mechanism to them is modeled by rules Prop_i and Expl_i. We assume (as in [21]) that each T_i-solver provides an $\texttt{explain}_i$ method with the property that if l is a

[1]Other predicate symbols can be expressed as function symbols with return sort Bool, interpreted as the Booleans in each theory.

[2]To simplify the presentation, we do not consider here rules that model the forgetting of learned lemmas or restarts of the SMT solver.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

$$\mathsf{Dec}\ \frac{l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \bullet l} \qquad \mathsf{Confl}\ \frac{\mathsf{C} = \emptyset \quad l_1 \vee \cdots \vee l_n \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M}}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n\}} \qquad \mathsf{Fail}\ \frac{\mathsf{C} \neq \emptyset \quad \bullet \notin \mathsf{M}}{\mathsf{fail}}$$

$$\mathsf{Prop}\ \frac{l_1 \vee \cdots \vee l_n \vee l \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\ l} \qquad \mathsf{Backj}\ \frac{\mathsf{C} = \{l_1 \vee \cdots \vee l_n \vee l\} \quad \mathsf{lev}\ \bar{l}_1, \ldots, \mathsf{lev}\ \bar{l}_n \leq\ i < \mathsf{lev}\ \bar{l}}{\mathsf{C} := \emptyset \quad \mathsf{M} := \mathsf{M}^{[i]}\ l}$$

$$\mathsf{Expl}\ \frac{\mathsf{C} = \{\bar{l} \vee D\} \quad l_1 \vee \cdots \vee l_n \vee l \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} l}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n \vee D\}} \qquad \mathsf{Learn}\ \frac{\mathsf{C} \neq \emptyset}{\mathsf{F} := \mathsf{F} \cup \mathsf{C}}$$

$$\mathsf{Expl}_i\ \frac{\mathsf{C} = \{\bar{l} \vee D\} \quad \models_i l_1 \vee \cdots l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} l}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n \vee D\}} \qquad \mathsf{Confl}_i\ \frac{\mathsf{C} = \emptyset \quad \models_i l_1 \vee \cdots \vee l_n \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M}}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n\}}$$

$$\mathsf{Prop}_i\ \frac{\begin{array}{c} l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \\ \models_i l_1 \vee \cdots \vee l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M} \quad l, \bar{l} \notin \mathsf{M} \end{array}}{\mathsf{M} := \mathsf{M}\ l} \qquad \mathsf{Learn}_i\ \frac{\begin{array}{c} l_1, \ldots, l_n \in \mathrm{Lit}_{\mathsf{M}}|_i \cup \mathrm{Int}_{\mathsf{M}} \cup L_i \\ \models_i \exists \mathbf{x}\, (l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}]) \end{array}}{\mathsf{F} := \mathsf{F} \cup \{l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]\}}$$

Figure 1: State transition rules. In Learn_i, $\mathbf{x}$ is a (possibly empty) tuple of variables; $\mathbf{c}$ is a tuple of fresh constants from $\mathcal{C}$ of the same sort as $\mathbf{x}$.

M	F	C	Rule
	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Dec
$\bullet\, 1$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Prop $(1 \vee \bar{2})$
$\bullet\, 1\, \bar{2}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Prop $(2 \vee 3)$
$\bullet\, 1\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Confl $(\bar{3} \vee 2)$
$\bullet\, 1\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\bar{3} \vee 2$	Expl $(2 \vee 3)$
$\bullet\, 1\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	2	Expl $(\bar{1} \vee \bar{2})$
$\bullet\, 1\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\bar{1}$	Learn $(\bar{1})$
$\bullet\, 1\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bar{1}$	Backj $(\bar{1})$
$\bar{1}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Prop $(1 \vee \bar{2})$
$\bar{1}\, \bar{2}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Prop $(2 \vee 3)$
$\bar{1}\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Confl $(\bar{3} \vee 2)$
$\bar{1}\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bar{3} \vee 2$	Expl $(2 \vee 3)$
$\bar{1}\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	2	Expl $(1 \vee \bar{2})$
$\bar{1}\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	1	Expl $(\bar{1})$
$\bar{1}\, \bar{2}\, 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bot$	Fail
	fail		

Figure 2: An execution using only propositional rules.

literal propagated by the solver, then $\mathtt{explain}_i(l)$ returns a subset $\{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n\}$ of M, such that $\models_i l_1 \vee l_2 \vee \cdots \vee l_n \vee l$. The inclusion of the interface literals $\mathrm{Int}_{\mathsf{M}}$ in rules Dec and Prop_i achieves the effect of the Nelson-Oppen combination method [10], [25]. Rule Learn_i models theory solvers following the splitting-on-demand paradigm [5]. When asked about the satisfiability of the set of Σ_i-literals in M, such solvers may return instead a *splitting lemma*, a clause encoding a guess that needs to be made about those literals before the solver can determine their satisfiability. The set L_i in the rule is a finite set consisting of *additional* literals, i.e., not present in the original formula in F, which may be generated by splitting-on-demand theory solvers.

III. Generating Proofs in DPLL(T)

One can prove that the transition rules defined in Section II are *refutation sound*: if an execution starting with $\langle \emptyset, F_0, \emptyset \rangle$ ends with fail, then F_0 is unsatisfiable in T. We discuss below how to generate unsatisfiability proofs from such executions.

Example 1: Figure 2 shows an example of an execution from an initial state to fail, using only propositional rules. In the figure, we abstract clause atoms by numbers to stress that they are treated purely propositionally by these rules. The *Rule* column shows the rule used for each transition, together with the clause the rule was applied to. We observe that Fail could have been applied right after the second application of Confl; however, we show instead a longer execution that regresses (with Expl) the conflict clause $\bar{3} \vee 2$ to the empty clause $\bot$. As we discuss later, the applications of Expl are needed for proof generation. Note that the second occurrence of $\bar{3} \vee 2$ as a conflict could have been avoided by learning the conflict clause 2 as soon as it was generated. Then, a shorter execution leading to fail would have been possible.

A. *Proof Generation for Propositional Unsatisfiability.*

Given a failed execution from an input set F_0 that uses only propositional clauses, as in Example 1, one can construct a proof that F_0 is (propositionally) unsatisfiable. Intuitively, we can understand a failed execution as trying to construct a *refutation tree*: a tree of clauses built from the leaves, which are either clauses in F_0 or learned clauses, down to the root $\bot$, where each non-leaf node is a propositional resolvent of its children. Thus, a failed execution can be translated into a Boolean resolution proof in a straightforward manner.

Observe, however, that a refutation tree provides only part of the full proof, since it only shows the unsatisfiability of the initial clause set *plus* some set of learned clauses. Thus, to complete the proof one also needs to prove that each learned clause is a consequence of the initial clause set. This can be performed similarly to how conflict analysis is performed in CDCL solvers [16]: every learned clause is the result of an application of the Confl rule and possibly a series of Expl rules. A sequence of resolution applications to the clauses to which these rules were applied produces the learned clause.

Figure 3 depicts a refutation tree for the execution in Figure 2. The tree shows the final resolution proof once all

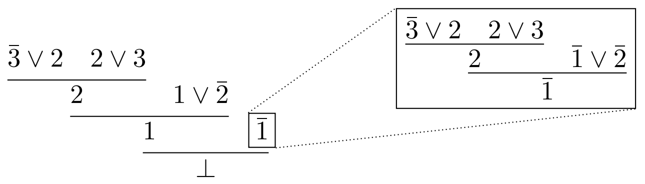


Figure 3: A refutation tree (on the left) with a sub-proof for a learned clause (on the right).

the needed clauses have been learned. Its leaves are the input clauses $\bar{3} \vee 2$, $3 \vee 2$ and $1 \vee \bar{2}$, and the learned clause $\bar{1}$. The tree itself is constructed simply by revisiting the applications of rules Confl and Expl that led to the conflict clause $\bot$, since each application of Expl produces a new conflict clause as the resolvent of the current conflict clause and an initial or learned clause. A separate proof is constructed for the learned clause $\bar{1}$, from the applications of Confl and Expl that generated it. In general, this recursive proof-tree generation process always terminates because each learned clause is derived from initial clauses and previously learned ones. It can be implemented in practice by keeping track of the various applications of Expl.

B. *Proof Generation for Unsatisfiability Modulo Theories.*

Executions ending in fail that involve the use of the non-propositional transition rules can also be seen as attempts to construct a refutation tree. This time, however, the leaves of the tree can include, in addition to initial and propositionally learned clauses, also *theory lemmas*—a name we give to clauses that come from the Confl_i, Learn_i, and Expl_i rules. Thus, the full proof tree requires combining propositional resolution proofs, produced by the SAT engine, with theory-specific proofs for each theory lemma.

To make this possible, we require each T_i-solver to provide a method $\texttt{provideProof}_i$ that takes as input a theory lemma and returns a proof of that lemma using theory-specific proof rules.[3] Then, a full proof tree can be constructed as before, by visiting the application of rules that led to the final conflict clause $\bot$. When visiting applications of Expl_i, the conflict clause $l_1 \vee \cdots \vee l_n \vee D$ is obtained by resolving $\bar{l} \vee D$ with the theory lemma $E = l_1 \vee \cdots l_n \vee l$. We then call $\texttt{provideProof}_i$ on E to obtain the missing part of the proof. Rule Confl_i adds a conflict clause $C = l_1 \vee \cdots \vee l_n$, which may end up as a leaf in a refutation tree. Thus, C is also a theory lemma and we call $\texttt{provideProof}_i$ on it if we encounter it during proof construction. Finally, rule Learn_i adds the clause $D = l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]$ directly to F, with the consequence that D can act as an input clause. Thus, if we encounter it during proof construction, we call $\texttt{provideProof}_i$ on D to obtain its theory-specific proof.

Thanks to the use of pure literals in clauses and the controlled exchange of information between the various theory solvers through the use of interface literals, Expl_i and $\texttt{provideProof}_i$, which are local to the T_i-theory solver for

M	F	C	Rule
	...		
$1\ 2\ 3 \bullet \bar{4} \bullet 5$	F_0	$\emptyset$	Prop_1 $(\bar{1} \vee \bar{2} \vee \bar{5} \vee 6)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6$	F_0	$\emptyset$	Prop_2 $(\bar{3} \vee \bar{6} \vee 7)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$\emptyset$	Confl_1
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$4 \vee \bar{6} \vee \bar{7}$	Expl_2 $(\bar{3} \vee \bar{6} \vee 7)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$\bar{3} \vee 4 \vee \bar{6}$	Expl_1 $(\bar{1} \vee \bar{2} \vee \bar{5} \vee 6)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	C	Learn
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0, C	C	Backj
$1\ 2\ 3 \bullet \bar{4}\ \bar{5}$	F_0, C	$\emptyset$	...

$C = \bar{1} \vee \bar{2} \vee \bar{3} \vee 4 \vee \bar{5}$

Figure 4: An execution using theory rules.

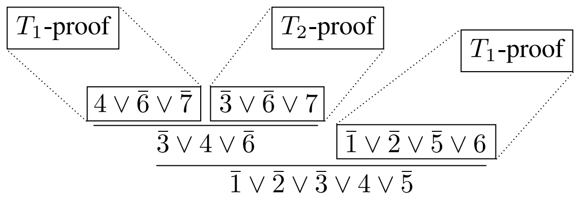


Figure 5: Using theory-specific proofs in proving a lemma.

each i, are enough to construct complex SMT proofs that involve several theories.

Example 2: Suppose T is the combination of the theory of uninterpreted functions (T_1 is T_{UF}) and the theory of arrays with extensionality (T_2 is T_{AX}), and consider an initial clause set F_0 containing the atoms:

$$\begin{array}{ll} 1: c_3 = f(c_1) & 3: c_5 = (a[c_3] := c_1)[c_4] \\ 2: c_4 = f(c_2) & 4: g(c_3, c_5) = g(c_4, c_1) \end{array}$$

where a is an array, $c_1, \ldots, c_5$ are shared constants, and f and g are uninterpreted functions. The expression $a[i]$ denotes the result of *reading* an array a at index i, and $a[i] := b$ denotes the result of *writing* value b at index i of a. Suppose that literals $1, 2, 3$ occur as unit clauses in F_0 while 4 occurs in some longer clause. Then, a possible execution from F_0 might look like the one in Figure 4 where 5, 6, and 7 are the following interface literals:

$$5: c_1 = c_2 \qquad 6: c_3 = c_4 \qquad 7: c_5 = c_1 \ .$$

If that execution eventually ends in fail and uses the learned clause $C = \bar{1} \vee \bar{2} \vee \bar{3} \vee 4 \vee \bar{5}$, then a proof certificate for F_0 will need a proof of C. The proof tree for C generated from the given execution is shown in Figure 5, with the proofs of the various theory lemmas omitted. Note that C, which has both Σ_1- and Σ_2-literals, is valid in T. However, it is not a lemma of either component theory. Proving it valid in T really requires a collaboration between the two theory solvers.

In practice, concrete implementations of this framework do not pass to the SAT engine the theory lemmas used in Expl_i steps, to avoid polluting the engine with unnecessary clauses. This means that in the example above, for instance, to obtain a proof for the learned clause C, we must be able to reconstruct the theory lemmas used in each Expl_i step. To do this, we record for each learned clause a *proof sketch*: a list of theory propagations, each performed by a specific theory solver, that together justify the learned clause. A clause's proof sketch can

[3] We give a few examples of theory-specific proofs for theory lemmas in Section V, when we discuss specific theory solvers.

be used later to produce a full proof as needed: each individual propagation is converted into a theory lemma via a call to the relevant solver's `explain`$_i$ method, and then a proof for that propagation is obtained via a call to `provideProof`$_i$. These intermediate proofs are then composed into a proof for the learned clause, using resolution as in the example above. By keeping these proof sketches we have enough information to construct complete proofs later on. This process facilitates lazy proof generation for learned clauses, as we discuss next.

IV. Lazy Proof Production

In the previous section we saw that in order to produce proofs in a DPLL(T) setting, each T_i-solver must be able to justify the theory lemmas it generates. In this section, we discuss a complementary question: *when* should it provide these justifications?

One approach, found in some solvers that support various forms of proof production [3], [6], is to prove each theory lemma *eagerly*, at the time it is generated. This has the advantage that proof production for each theory step typically incurs only a small overhead, and often boils down to recording the internal deductive process that the theory solver follows when generating the lemma. However, this greedy approach can be inefficient. During the solution phase, theory solvers usually produce numerous lemmas that end up not being used in deriving the empty clause, and so do not make it into the final refutation tree. Hence, any proofs produced for such lemmas are a waste of effort. As an alternative, we advocate a *lazy* approach where no proofs for theory lemmas are generated until the final refutation tree has been found. Then, the `provideProof`$_i$ methods are invoked only for those theory lemmas that appear as leaves in the tree.

For many of the benchmarks we tried, only a fraction of the thousands of theory lemmas generated during the solving phase are used in the final proof, so the savings from producing proofs for theory lemmas lazily can be significant. A disadvantage is that theory lemmas occurring in the final proof end up being processed twice: once when they are originally generated, and then again when producing the proof. Typically, this means that in addition to generating the proof, the theory solver will have to redo the deductive work that was required to generate each lemma in the first place.

Choosing an appropriate strategy depends on the particular theory solver in question. For some theory solvers reproving lemmas is cheap, making the lazy approach more suitable; for others, an eager approach may yield better results. Our experiments (in Section VI) indicate that, in the cases of T_{UF} and T_{AX}, the lazy approach fairs better. We discuss the particulars of our implementation in Section V.

Lazy Proofs and Rewrite Rules. Modern SMT solvers make use of a large arsenal of rewrite rules aimed at simplifying formulas. These rules specify how and when to replace atoms and terms with simpler but equivalent versions, and applying them can significantly improve the performance of solvers. However, the simplification of even a single atom that appears in a theory lemma can interfere with lazy proof production, as illustrated by the following example, encountered while attempting to produce proofs for the SMT-LIB benchmarks [4] in the theory T_{ABV} combining arrays and bitvectors.

Example 3: Suppose that the T_{AX}-solver generates the theory lemma $L_1 : (b{+}1 = 1) \vee ((a[b{+}1] := x)[1] = a[1])$, where a is an array and b is a fixed-width bitvector (for conciseness, we give here the lemma in non-purified form). Intuitively, this lemma says that if $b+1 \neq 1$, then writing x to $a[b+1]$ does not alter the value of $a[1]$. L_1 is valid in T_{AX}, and so the T_{AX}-solver should be able to prove it.

In the lazy approach, the T_{AX}-solver is not asked to provide a proof for L_1 right away. Now, suppose that during subsequent processing of the theory lemma, a bitvector rewrite rule is invoked, simplifying the atom $b + 1 = 1$ to $b = 0$, and consequently transforming lemma L_1 into $L_2 : (b = 0) \vee ((a[b + 1] := x)[1] = a[1])$. This lemma is valid in T_{ABV}, but not in T_{AX}. Thus, when the time comes to produce a proof and the T_{AX}-solver is asked to prove L_2, it will fail to do so.

We can overcome this difficulty as follows. First, we extend the abstract DPLL(T) framework with the following, general rule, which allows theory solvers to rewrite literals:

$$\mathsf{Rewrite}_i \quad \frac{\mathsf{C} = \{l \vee D\} \qquad \models_i \bar{l}_1 \vee \cdots \vee \bar{l}_n \vee (l = l') \quad l_1, \ldots, l_n \in \mathsf{M}}{\mathsf{C} := \{l' \vee D\}}$$

We call the clause $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee (l = l')$ above a *rewrite lemma*. During the solution phase, we keep track of the application of these rewrite rules to theory atoms. Whenever a theory atom that participates in a lemma is rewritten, we record this information in the lemma's proof sketch. Then, if and when we need to prove the (rewritten) lemma, we can separately prove the original lemma and each specific rewrite lemma used to rewrite it, and then combine their proofs into a proof for the rewritten lemma. In our example above, when we need to prove L_2, we first have the T_{AX}-solver prove the original lemma L_1, and then separately ask the T_{BV}-solver to provide a proof for the equivalence $(b + 1 = 1) = (b = 0)$. These two proofs can then be combined to prove L_2, which is the actual leaf in the refutation tree. Observe that this technique is applicable even if there is a series of rewrites involving multiple theory solvers, because, according to the $\mathsf{Rewrite}_i$ rule, each rewrite lemma used is valid in some individual theory.

Besides enabling proof production when rewrite rules are applied, this process also has a beneficial effect on *modularity*: it separates proofs for rewrite rules from those of the theory lemmas, thus simplifying proof production and improving proof legibility.

V. Theory-Specific Proofs

In the purely propositional case (as in Example 1), a proof can always be constructed that consists of a sequence of applications of Boolean resolution, starting from the input clauses. In the non-propositional case, we saw that each theory solver must provide proofs for its theory lemmas. This requires additional instrumentation in the theory solvers as well as additional deduction rules and axioms beyond Boolean resolution. In this section, we discuss the construction of proof-producing theory solvers for three common theories: uninterpreted functions with equality (T_{UF}), arrays with extensionality (T_{AX}) and fixed-width bitvectors (T_{BV}). In all theory solvers, it is more convenient to prove a theory lemma $l_1 \vee \cdots \vee l_n$ by first proving the unsatisfiability of the set $\{\bar{l}_1, \ldots, \bar{l}_n\}$; so we focus on the latter kind of proof here.

Uninterpreted Functions. A general scheme for a proof-producing T_{UF}-solver was proposed by Fontaine *et al.* [14]. We follow a similar approach, briefly summarized below. Decision procedures for T_{UF} are normally based on congruence closure: the solver maintains an *equality graph* which partitions the terms appearing in the input constraints into equivalence classes. As the search progresses, equivalence classes get merged. Unsatisfiability is derived when two terms a and b from an input constraint $a \neq b$ end up in the same equivalence class.

To produce a refutation tree, the T_{UF}-solver keeps track of all previously performed merges of equivalence classes. When it is asked to prove that $a = b$ is a consequence of some of the input constraints (contradicting the input constraint $a \neq b$), it backtracks through these merges and constructs a chain $a = x_1 = \cdots = x_n = b$, where each link is the result of an input constraint or an application of the congruence rule (deriving, for instance, $f(x) = f(y)$ from $x = y$) [14]. This chain can then be transformed into a proof tree whose leaves are input assertions and whose internal nodes are generated by the application of one of the following rules:

Transitivity:	from $x = y$ and $y = z$ derive $x = z$
Congruence:	from $\mathbf{x} = \mathbf{y}$ derive $f(\mathbf{x}) = f(\mathbf{y})$
Symmetry:	from $x = y$ derive $y = x$

Figure 6 depicts a refutation of the negation of the T_{UF} theory lemma $(x \neq y) \vee (z \neq f(y)) \vee (f(x) = z)$ using those rules.

$$\dfrac{f(x) \neq z \qquad \dfrac{\dfrac{x = y}{f(x) = f(y)}\,\text{Cong.} \qquad \dfrac{z = f(y)}{f(y) = z}\,\text{Symm.}}{f(x) = z}\,\text{Trans.}}{\bot}$$

Figure 6: A refutation of $\{x = y,\, z = f(y),\, f(x) \neq z\}$.

A convenient way to implement eager T_{UF} proof production is to instrument the T_{UF}-solver's `explain` function to produce, apart from an explanation clause, also a proof for that clause. However, T_{UF} is a prime candidate for lazy proof production: since the decision procedure in this case is very efficient, reproving previous lemmas is cheap. In the lazy approach, during proof construction, if we encounter a T_{UF} theory lemma $l_1 \vee \ldots \vee l_n$, we assert its negation to a *fresh* proof-producing instance of the T_{UF}-solver. This solver then constructs the proof as it derives a contradiction. Our experimental evaluation (see Section VI) suggests that the lazy approach is superior to the eager approach for T_{UF}.

Arrays with Extensionality. We now show how we can build on the procedure for T_{UF} to produce proofs for T_{AX}. An efficient decision procedure for T_{AX} [12] uses congruence closure and maintains an equality graph, similarly to the T_{UF} case; however, it merges equivalence classes also as the result of array-specific axioms (proof rules with no premises):

1) Read-over-write 1: for any array a, indices i and j and element x, if $i \neq j$ then $(a[i] := x)[j] = a[j]$.
2) Read-over-write 2: $(a[i] := x)[i] = x$.

The first axiom guarantees that writing to index i does not change the value at a different index j, and the second guarantees that written values persist. A third axiom states that disequal arrays must differ in at least one cell:

3) Extensionality: for any two arrays a and b, if $a \neq b$ then there exists a k such that $a[k] \neq b[k]$.

Observe that, unlike in the T_{UF} case, an unsatisfiable set of constraints here does not have to include one of the form $a \neq b$, since disequalities can also be deduced by the extensionality axiom. A contradiction is reached when two contradictory literal, $a = b$ and $a \neq b$, are derived.

Instrumenting a T_{AX}-solver to produce proof trees based on these axioms again consists of collecting the reasons for the merges of equivalence classes. In particular, any application of Read-over-write 1 and Extensionality contains a sub-proof for the axiom's guard—respectively, $i \neq j$ and $a \neq b$.

Figure 7 depicts a refutation of the negation of the T_{AX} theory lemma $(i = j) \vee ((a[j] := y)[i] \neq x) \vee (a[i] = x)$ using the first read-over-write (*RoW*) axiom.

$$\dfrac{\dfrac{i \neq j \qquad (a[j] := y)[i] = x}{a[i] = x}\,\text{RoW 1} \qquad a[i] \neq x}{\bot}$$

Figure 7: Refutation of $\{i \neq j,\, (a[j] := y)[i] = x,\, a[i] \neq x\}$.

Eager proof production can be achieved as in the T_{UF} case. For lazy proof production, we can again instantiate a fresh copy of the solver for every lemma that we need to prove. However, in this case, reproving lemmas from scratch does not suffice. The problem is due to the Extensionality axiom. Consider a case where we need to reprove an instance $(a = b) \vee (a[k] \neq b[k])$ of that axiom, where k is a free constant witnessing the disequality $a \neq b$. If we attempt to lazily prove this lemma by instantiating a fresh T_{AX}-solver and asserting to it the set $\{a \neq b,\, a[k] = b[k]\}$, it will be unable to refute it (simply because, by itself, it is not unsatisfiable). This problem can be overcome by some simple bookkeeping during the solution phase: whenever the Extensionality axiom is used, we record that k is a witness for $a \neq b$; later, during lazy proof production, we ensure that the same k is used to witness $a \neq b$ in the fresh solver. Again, our experiments (see Section VI) suggest that, despite this extra bookkeeping, the lazy approach is superior to the eager approach for T_{AX}.

Bitvectors. We discuss proof generation for the theory T_{BV} of fixed-width bitvectors thoroughly in our previous work [15], so we provide here only a short recap, for completeness. Theory solvers for this theory make extensive use of *bit-blasting*: they transform a bitvector formula φ into an equisatisfiable propositional formula φ^{BB}, in which fresh Boolean variables represent the values of individual bitvector bits. An internal SAT solver then checks the satisfiability of φ^{BB}, and a proof for its unsatisfiability can be translated into a proof for the unsatisfiability of the original φ.

A small example appears in Figure 8. It depicts a bit-blasting refutation for the negation of T_{BV} lemma $(b_1 \neq b_2) \vee (b_2 \neq 10) \vee (b_1 \neq 00)$, where b_1 and b_2 are bitvectors of size 2. The three equalities in the lemma are bit-blasted, via application of the *BB* rule, to derive equalities over some of their constituent bits (denoted here by an array-like notation); these equalities are then used to derive a contradiction.

$$\dfrac{\dfrac{\dfrac{b_1 = b_2}{b_1[1] = b_2[1]}\,\text{BB} \qquad \dfrac{b_2 = 10}{b_2[1] = 1}\,\text{BB}}{b_1[1] = 1}\,\text{Trans.} \qquad \dfrac{b_1 = 00}{b_1[1] = 0}\,\text{BB}}{\bot}$$

Figure 8: A refutation of $\{b_1 = b_2,\, b_2 = 10,\, b_1 = 00\}$.

Bitvector lemmas are proved semi-lazily, in the following sense. During the solution phase, the T_{BV}-solver's internal SAT solver is instrumented to eagerly record any conflict that it discovers. Later, when a lemma needs to be proved because it appears in the refutation tree, the bit-level conflicts that prove it have already been recorded and can be used. While most of the work is thus done eagerly, one part is still performed lazily: the proof of the bit-blasting process itself, i.e., the part of the proof connecting φ to φ^{BB}, is reproduced lazily only for participating lemmas.

Our motivation for eagerly recording the internal SAT solver's conflicts is that reproducing a T_{BV} theory lemma with no information would require re-bit-blasting and re-solving, a potentially very expensive process.

VI. Evaluation

For evaluation purposes we implemented our proof generation approach in CVC4 [2]. Proof generation for T_{BV} was implemented as part of previous work [15]. For this evaluation, we extended CVC4 with both eager and lazy proof generation capabilities for T_{UF} and T_{AX}. We also completed the instrumentation of the DPLL(T) engine as described in Section III, enabling it to handle any combination of the three theories above. Support for proving rewrite rules is still under development, and so for the purposes of this evaluation rewrite rules are treated as axioms, i.e. are given without fine-grained justification. However, the rewrite rules do appear in separate lemmas outside the main proof as discussed in Section IV, and their usage in other parts of the proof is checked for correctness. All changes have been integrated into the master branch of CVC4, which is available online through CVC4's GitHub repository at https://github.com/CVC4.

CVC4 outputs the proofs it generates as terms in the *Logical Framework with Side Conditions* (*LFSC*) [24]. Based on a simply typed λ-calculus with dependent types, LFSC reduces proof checking to type checking: proof rules are encoded as (higher-order) constants, with their premises and conclusions encoded as types, and a proof is a term whose constants are proof-rule names. An LFSC checker takes as input a proof term t and a *signature* S, a collection of type and constant declarations that includes the various proof rules, and checks that t is well-typed with respect to S. We extended the signature S from Hadarean *et al.* [15] to support the T_{UF} and T_{AX} rules mentioned in Section V.

We first compared the lazy and eager proof generation approaches for T_{UF} and T_{AX}. Figure 9 shows the results on all QF_UF and QF_AX benchmarks from the SMT-LIB library [4]. For QF_UF benchmarks, the eager approach was slower than the lazy one on almost all instances and incurred an average performance overhead of 30%. For QF_AX benchmarks, the eager approach was 25% slower on average. Both cases thus indicate a clear advantage for the lazy approach.

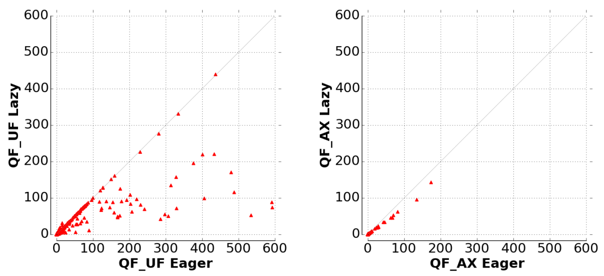


Figure 9: Eager vs. Lazy proof production runtimes, in seconds.

We then ran a more extensive experiment to test our ability to correctly generate and check proofs (lazily for the T_{UF} and T_{AX} solvers) for unsatisfiable benchmarks from all the relevant logics (including theory combinations) in the SMT-LIB library [4]: QF_UF, QF_AX, QF_BV, QF_UFBV, QF_ABV and QF_AUFBV. Table I shows the results. The *Default* columns describe the performance of CVC4 with proof production disabled; the *Generate and Check Proof* and *Generate Proof* columns describe performance when producing a proof with and without checking it, respectively. Also shown in the table are results on a set of industrial QF_ABV benchmarks encoding symbolic execution problems, which were provided to us by collaborators from GrammaTech, Inc. These results appear in the row labeled *Symbolic Execution*.

CVC4 was able to produce proofs for over 99% of all instances that it could solve without proof generation. We were similarly able to check most of the generated proofs using LFSC's external proof checker. In the future, we plan to improve proof checking time by optimizing the LFSC checker and using more efficient LFSC encodings for our proofs.

Benchmark Category	Default		Generate Proof		Generate and Check Proof	
	Solved	Time	Solved	Time	Solved	Time
QF_UF	4083	7523	4067	19097	4029	61650
QF_AX	277	450	264	3170	260	3193
QF_BV	20517	49884	20430	67072	17602	132975
QF_UFBV	12	1391	12	2623	4	170
QF_ABV	4487	16223	4410	19900	4127	22768
QF_AUFBV	31	93	31	245	30	1751
Symbolic Execution	94	1735	89	4364	71	2348

Table I: Producing and checking proofs. All times are in seconds. Experiments were run with a 600 second timeout.

VII. Related Work

Various SMT solvers have taken different approaches to proof production over the years (see Barrett *et al.* [3] for a recent survey). To the best of our knowledge, the only other SMT solver that is both actively maintained and able to produce independently-checkable proofs is veriT [9], which supports eager proof-production for T_{UF} and the theory of linear arithmetic. Our approach for eager proof production in T_{UF} is similar to that of veriT [14]. However, veriT does not support lazy proof production or proofs for T_{AX} or T_{BV}.

The Z3 solver produces *proof traces*, essentially a record of propositional inferences plus a listing of theory lemmas used [6]. Extending such a proof trace to a full proof requires an external tool capable of proving theory lemmas independently, which can be quite challenging, for instance for bitvector theory lemmas [8]. Our approach differs from Z3's approach in that it produces full, fine-grained proofs that are checkable by simple checkers.

The LFSC format [24] allows us to use a generic LFSC checker to check proofs. Other approaches for checking SMT-generated proofs include using custom checkers [20] or skeptical interactive theorem provers such as HOL Light [19] or Isabelle/HOL [14].

VIII. Conclusion and Future Work

Adding proof production capabilities to complex tools like SMT solvers can greatly increase our level of confidence in their results. We presented here a technique that allows DPLL(T)-style SMT solvers to produce unsatisfiability proofs for queries involving combinations of theories. Our approach requires that each theory solver provide proofs for its theory-specific deductions; and these sub-proofs are then interwoven into a complete, cohesive proof by the main SAT engine. Our approach is modular and extensible in the sense that any new proof-producing solver can be readily integrated with existing ones. We also explored *lazy* proof generation and demonstrated its advantages for T_{UF} and T_{AX}.

For the near future, we plan to improve CVC4's ability to prove rewrite steps, as discussed in Section IV. Another planned enhancement is the addition of proof support for arithmetic and *quantified logics*—with the aim of eventually being able to produce proofs for unsatisfiable formulas in the full input language supported by CVC4.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, 2011.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[3] C. Barrett, L. de Moura, and P. Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 2015.

[4] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org.

[5] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. In *LPAR*, 2006.

[6] N. Bjørner and L. de Moura. Proofs and Refutations, and Z3. In *LPAR*, 2008.

[7] J. Blanchette, S. Böhme, and L. Paulson. Extending Sledgehammer with SMT Solvers. *J. of Automated Reasoning*, 2013.

[8] S. Böhme, A. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *CPP*, 2011.

[9] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an Open, Trustable and Efficient SMT-Solver. In *CADE*, 2009.

[10] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. *Annals of Mathematics and Artificial Intelligence*, 2009.

[11] J. Chen, R. Chugh, and N. Swamy. Type-Preserving Compilation of End-to-End Verification of Security Enforcement. In *PLDI*, 2010.

[12] L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *FMCAD*, 2009.

[13] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. Reynolds, and C. Tinelli. Extending SMTCoq, a Certified Checker for SMT. In *HATT*, 2016.

[14] P. Fontaine, J. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In *TACAS*, 2006.

[15] L. Hadarean, C. Barrett, A. Reynolds, C. Tinelli, and M. Deters. Fine-grained SMT Proofs for the Theory of Fixed-width Bit-vectors. In *LPAR*, 2015.

[16] M. Heule and A. Biere. Proofs for Satisfiability Problems. *All about Proofs, Proofs for All*, 2015.

[17] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FROCOS*, 2007.

[18] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 1999.

[19] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. In *PDPAR*, 2005.

[20] M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In *TACAS*, 2008.

[21] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. of the ACM*, 2006.

[22] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite Model Finding in SMT. In *CAV*, 2013.

[23] A. Reynolds, C. Tinelli, and L. Hadarean. Certified Interpolant Generation for EUF. In *SMT*, 2011.

[24] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT Proof Checking Using a Logical Framework. *Formal Methods in System Design*, 2012.

[25] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *FROCOS*, 1996.

Verifiable Hierarchical Protocols with Network Invariants on Parametric Systems

Opeoluwa Matthews
Department of ECE
Duke University
luwa.matthews@duke.edu

Jesse Bingham
Intel Corporation
jesse.d.bingham@intel.com

Daniel J. Sorin
Department of ECE
Duke University
sorin@ee.duke.edu

***Abstract*—We present Neo, a framework for designing pre-verified protocol components that can be instantiated and connected in an arbitrarily large hierarchy (tree), with a guarantee that the whole system satisfies a given safety property. We employ the idea of *network invariants* to handle correctness for arbitrary depths in the hierarchy. Orthogonally, we leverage a parameterized model checker (*Cubicle*) to allow for a parametric number of children at each internal node of the tree. We believe this is the first time these two distinct dimensions of configuration have been together tackled in a verification approach, and also the first time a proof of an *observational preorder* (as required by network invariants) has been formulated inside a parametric model checker. Aside from the natural up/down communication between a child and a parent, we allow for peer-to-peer communication, since many real protocol optimizations rely on this paradigm. The paper details the Neo theory, which is built upon the *Input-Output Automata* formalism, and demonstrates the approach on an example hierarchical cache coherence protocol.**

I. Introduction

Formal verification of large-scale, modern systems protocols is currently challenging. Although theorem proving is theoretically able to verify arbitrary protocols, the manual effort required to guide a theorem prover through the verification of a modern protocol is prohibitive. Model checkers are more widely used, but they cannot handle complex, large-scale protocols. As a result of the state explosion problem, model checking proofs are successful for only a handful of protocol components—generally not sufficient to exercise all the behaviors exhibited in industrial-scale systems. Hence, there is strong motivation for architects to design protocols specifically to be verifiable with state-of-the-art model checking tools. Our solution is to construct a set of protocol components, instances of which are composed into an arbitrary hierarchy, where each component instance is independently scaled. The components are pre-verified in such a way as to guarantee that the resulting large and complex system is always correct.

Our approach involves the combination of two distinct ideas from the model checking literature: *network invariants* and *parameterized model checking*. Consider the hierarchical protocol depicted in Fig. I. We would like to design the leaf (L), internal (I), and root (R) nodes[1] so that any arbitrary nesting in the vertical direction, and any arbitrary (and independent) branching degree (number of children) at each internal and root node, yields a system that is correct. Arbitrary nesting is handled by network invariants; in particular we require (and verify) that L is a network invariant. This means that the observational behaviors of L subsumes that of any larger composition of components. For instance, the behavior along communication channel $c2$ over-approximates that of $c1$, $c3$, etc. We formulate network invariants in a novel way that not only captures the observational behaviors (messages) across an interface, but also captures what we call the *summary state* of a sub-hierarchy. These summary states are integral in defining the safety property, which, like the system itself, is hierarchically defined.

Beyond the hierarchical nesting afforded by network invariants, we employ parameterized model checking to allow arbitrary branching degrees. This entails that we prove the observational pre-order containment required of network invariants *parametrically* in a model checker; we believe this is novel. Hence, L serves as a network invariant for not just a particular I, but for all members of an infinite family $I(1), I(2), I(3), \ldots$, where $I(n)$ is an internal node configured to connect to n children.

We emphasize that network invariants and parameterized model checking are both necessary ingredients in this story; neither is capable of solving what the other does. Network invariants deal with the connection of instances of components into arbitrarily complex hierarchies, with relatively simple interfaces between constituents; parameterized model checkers typically do not support such a notion of "parameterization" when the structure is nontrivial (e.g. a tree). On the other hand, network invariants are not appropriate to deal with an internal node that is parameterized on the number of children. An example is a directory in a cache coherence protocol—the directory is an array, with one entry per child. There is no clear way to formulate this type of tightly coupled parameterization as the composition of components along with a network invariant. Fortunately, parameterized model checkers are usually targeted at precisely this style of parameterization.

Previous research on network invariants [12], [15], [1], [16], [32] tends to focus on "flat" compositions of processes with rather trivial structure; processes are arranged in a linear or circular array with only neighbor-to-neighbor communication, or the other extreme wherein each process talks to all others. The work of Clarke, Grumberg, and Jha [7], is the most closely

[1]We use the terms *component* and *node* interchangeably.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

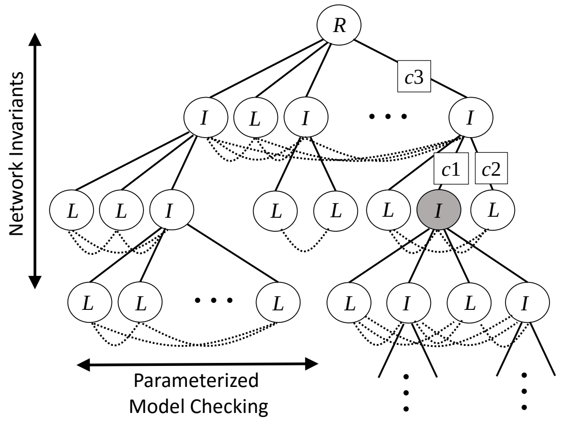


Fig. 1. A Neo hierarchy. Nodes labels R, I, and L respectively indicate root, internal and leaf nodes. Solid lines indicate parent/child communication channels, while dotted lines indicate peer/peer communication. Arbitrary nesting of tree structures in the vertical direction is handled by network invariants. Arbitrary branching widths in the horizontal direction is handled by parameterized model checking. The leaf L acts as a network invariant, which means for example that the behavior along communication channel $c2$ over-approximates that of $c1$, $c3$, etc.

related to us, since they allow hierarchical structures. Like us, they require that a small process serves as a network invariant for all (larger) composite processes.[2]. However, we extend their work in several ways:

- As mentioned above, we use parameterized model checking to facilitate arbitrary branching degree
- We use an asynchronous/interleaved execution semantics (I/O automata), while Clarke et al. use a synchronous.
- We make a (modest, but important) extension to allow processes to be given "identifiers."
- Our example cache protocol is significantly more complex than their example (a protocol that computes a parity function over the leaves of a tree).
- We express our state invariant property using summary functions, which makes the property's structure naturally echo that of the Neo system's hierarchy.

Other related work looks at the problem of verifying hierarchical protocols with two levels, using abstraction and assume/guarantee reasoning [5]. Similar to us, smaller systems are verified to conclude coherence of a system for which model checking is intractable, but the approach involves manual effort and it's unclear if it scales to more elaborate hierarchies.

Parameterized model checking approaches have been widely explored in the literature [13], [2], [9]. The research includes disparate techniques such as: assume/guarantee-style abstraction [21], [22], [6], [14], [33], predicate abstraction [17], invisible invariants [26], flows [23], regular sets [4], Satisfiability Modulo Theories (SMT) [11]. We elected to use *Cubicle* [8] as it has a clean language, has published encouraging results, and is being actively maintained. Though our work is rather agnostic to the underlying model checking technique, we believe our leveraging of a parametric model checker to parametrically prove an observational pre-order is novel.

[2]In cases where a single terminal (what we call a *leaf*) process fails to be a networks invariant, they are able to instead employ a non-terminal, but suitably small, composition of processes.

Some prior work has proposed designing systems from pre-verified components to enable scalable verification. Zhang et al. propose designing cache coherence protocols such that caches are organized in a tree hierarchy, with any scale of the system being observationally equivalent to a pre-verified small-scale system [34]. Unfortunately, [34] is not rigorously formalized. Furthermore, the definition of observational equivalence used focuses only on matching states and ignores actions, which could permit safety violations in a larger scale system. [31] and [30] present performance optimizations to [34] and [20] adapts [34] to designing verifiable power management protocols. Hence, these works inherit [34]'s flaws.

Beu et al. propose a template that allows one to link pre-verified cache coherence protocols into a hierarchy by allowing directories of lower tiers to seek permissions from higher tiers [3]. However, the work is also not rigorously formalized. Also, the pre-verified protocols are not verified in an environment where they interact with higher tiers, which could permit incoherence when they are actually linked into a hierarchy.

To illustrate our verification methodology, we design a hierarchical cache coherence protocol called NeoGerman by composing a parameterized German protocol [6] into a Neo hierarchy. We prove that our protocol is a Neo system, which implies that it behaves correctly for any arbitrary configuration of the hierarchy. While the flat German protocol is trivial, we are not aware of any work that verifies an arbitrary-dimension hierarchical version. We believe our framework is applicable to more sophisticated protocols, and we pick the hierarchical German protocol only to illustrate our approach.

We note that several proofs have been omitted due to length constraints; these proofs can be found in the complete version of this paper [19].

II. Formalizing the Neo Framework

Our framework can be thought of as a class of transition systems for which certain properties hold, as a result of which any member of this class is amenable to a much simpler verification methodology. We hope many systems protocols can be shown (or designed) to fit this class and thus inherit the simplified verification. In this section, we will define this class of transition systems and prove that given some automatedly verifiable antecedents, all members of this class are *safe*.

For any $n \geq 0$, we define $\mathbb{N}_n = \{0, 1, \ldots, n-1\}$; note that $\mathbb{N}_0 = \emptyset$. Also, if $x = (x_0, \ldots, x_k)$ is a tuple or list, we denote x_i by $x[i]$.

A. I/O Automata Theory

We start by giving a short description of the well-known I/O automata theory upon which our framework is formalized. We will only go into enough detail as is sufficient for our work; for a more complete description of I/O automata, see [29].

An action signature S is a partition of a set $acts(S)$ of *actions* into three disjoint sets: $in(S)$, $out(S)$, and $int(S)$, respectively called the *input*, *output*, and *internal* actions. The set $int(S) \cup out(S)$ is denoted by $local(S)$. An *I/O automaton* (IOA) A consists of the following:[3]

- an action signature S, denoted $sig(A)$
- a set $states(A)$ called the *states*
- a nonempty set $start(A) \subseteq states(A)$ called the *start states*
- a transition relation $steps(A) \subseteq states(A) \times acts(S) \times states(A)$

$acts(S)$ is also referred to as $acts(A)$, $in(S)$ is also referred to as $in(A)$, etc. The set $in(A) \cup out(A)$ of *external actions* is referred to as $ext(A)$.

An *execution fragment* e of A is a sequence $e = s_0, a_1, s_1, \ldots, a_k, s_k$ such that, for each i, $(s_i, a_{i+1}, s_{i+1}) \in steps(A)$. If $s_0 \in start(A)$, then e is an *execution* of A. The set of executions of A is denoted by $execs(A)$. If a state s is the final state of an *execution*, then s is said to be *reachable*.

A set $\{S_0, \ldots, S_{n-1}\}$ of action signatures is said to be *compatible* if for all $i \neq j$, $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$. A set of IOA are said to be compatible if their action signatures are compatible.

The n-way composition $S = \prod_{i=0}^{n-1} S_i$ of compatible action signatures $\{S_0, \ldots, S_{n-1}\}$ is an action signature with $in(S) = \bigcup_{i=0}^{n-1} in(S_i) \setminus \bigcup_{i=0}^{n-1} out(S_i)$, $out(S) = \bigcup_{i=0}^{n-1} out(S_i) \setminus \bigcup_{i=0}^{n-1} in(S_i)$, and $int(S) = \bigcup_{i=0}^{n-1} int(S_i) \cup (\bigcup_{i=0}^{n-1} out(S_i) \cap \bigcup_{i=0}^{n-1} in(S_i))$ [4].

The n-way composition $C = \prod_{i=0}^{n-1} C_i$ of compatible IOA $\{C_0, \ldots, C_{n-1}\}$ is an IOA with the following:

- $sig(C) = \prod_{i=0}^{n-1} sig(C_i)$
- $states(C) = states(C_0) \times \cdots \times states(C_{n-1})$
- $start(C) = start(C_0) \times \cdots \times start(C_{n-1})$
- $steps(C)$ is a set of tuples of the form $(s, a, s') \in states(C) \times acts(C) \times states(C)$ that satisfy the following for all i:
  - $a \in acts(C_i)$ implies $(s[i], a, s'[i]) \in steps(C_i)$
  - $a \notin acts(C_i)$ implies $s[i] = s'[i]$

For IOA $C = \prod_{i=0}^{n-1} C_i$, for $s \in states(C)$ and for all i, define $s|C_i = s[i]$. Let $e = s_0, a_1, s_1, \ldots, a_k, s_k$ be an execution of C. Then, for all i, define $e|C_i$ as the sequence derived by modifying e as follows. Delete each a_j, s_j if $a_j \notin acts(C_i)$. Then, replace all remaining s_j with $s_j|C_i$.

Lemma 1. *Let IOA $C = \prod_{i=0}^{n-1} C_i$ and $e \in execs(C)$. Then, for all i, $e|C_i \in execs(C_i)$.*

Proof. See Tuttle et al. [29]. $\square$

B. Defining Neo Systems

We now formalize our framework by defining a class of IOA and expressing what properties we require of processes in the class. We will eventually show that these properties can be verified automatedly and indeed imply safety of the system.

For any set of actions Σ, we define $\Sigma(n) = \Sigma \times \mathbb{N}_n$ and $\Sigma(n, m) = \Sigma \times \mathbb{N}_n \times \mathbb{N}_m$. Let U be a finite set of *upward interface actions*, let D be a finite set of *downward interface actions*, and let P be a finite set of *peer-to-peer interface actions*. We identify some classes of IOA below.

- An IOA I is an (n, m) (or n-child and m-peer) *internal node* if it supports communication with a parent, n children, and $m-1$ peers. Formally, $out(I) = U \cup D(n) \cup P(m-1)$ and $in(I) = D \cup U(n) \cup P(m-1)$.
- An IOA L is a *leaf node* if it is a $(0, m)$ internal node, for some m. Hence, $out(L) = U \cup P(m-1)$ and $in(L) = D \cup P(m-1)$. A leaf is a degenerate internal node with no children.
- An IOA R is an n-child *root node* if $out(R) = D(n)$ and $in(R) = U(n)$. An n-child root node caps a Neo hierarchy, and hence has no peer or parental communication, but still has n children.

For example, the node shaded grey in Fig. I is a $(4, 3)$-internal node. For each $i \in \mathbb{N}_{m-1}$ and process A, we define the function ϕ_i that derives a new process $\phi_i(A)$ with *tag* i by modifying A's action signature as follows. Let $shift(i, j) = j$ if $j < i$, otherwise $j + 1$.

- If a is an external action with $a \in U \cup D$, or a is an internal action, it is replaced with (a, i).
- Each input action $(p, j) \in P(m-1)$ is replaced with $(p, shift(i, j), i)$.
- Each output action $(p, j) \in P(m-1)$ is replaced with $(p, i, shift(i, j))$.

Intuitively, each upward or downward interface action is now augmented with $\phi_i(A)$'s tag i, allowing it to communicate uniquely with its parent. Each internal action is also augmented with $\phi_i(A)$'s tag i, so as to ensure disjoint sets of internal actions in compositions, as required by IOA theory. To facilitate unique peer-to-peer communication, each peer-to-peer interface action p in $\phi_i(A)$ appears in the form (p, src, dst), where src is the tag of the source process and dst is the tag of the destination process.

Given a set of leaves $Ls = \{L(1), L(2), \ldots\}$, where each $L(m)$ is an m-peer leaf, a set of internal nodes Is, and a set of root nodes Rs, we define the notions of *open Neo systems* and *closed Neo systems* inductively as follows.

- Each $L(m)$ is an m-peer open Neo system, supporting communication with $m-1$ peers.
- Given n n-peer open Neo systems $\Omega_0, \ldots, \Omega_{n-1}$ and an n-child node $A \in Is \cup Rs$, the $(n+1)$-way IOA composition
$$\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i) \qquad (1)$$
is an m-peer open Neo system (if $A \in Is$) or a closed Neo system (if $A \in Rs$).

We will simply write *Neo system* if we are not concerned about whether Ω is open or closed. Where Ω is an open Neo system,

[3]We deviate from Tuttle's thesis [29] in two ways: we preclude $part(A)$, and we don't require input-enabledness. This is justified since both notions are only relevant for fair executions, which for us is purely future work.

[4]Unlike in Tuttle's thesis [29], by default, we hide messages sent between component processes as internal.

we characterize IOA $\phi_i(\Omega)$, for some $i < m$, as a *tagged* open Neo system and Ω as an *untagged* open Neo system.

Lemma 2. *For Neo system Ω (1), if Ω is an open Neo system, then $in(\Omega) = D \cup P(m-1)$ and $out(\Omega) = U \cup P(m-1)$. If Ω is a closed Neo system, then $ext(\Omega) = \emptyset$.*

C. Neo System Safety

1) Summary of States: Let Sum be a finite set of *summary states* that contains a distinguished state bad. We associate *summary functions* with each $L(m)$ and the elements of Is and Rs as follows

- $sum_{L(m)}$ has type $states(L(m)) \to Sum$
- For each n-child $A \in Is \cup Rs$, $sum_A : states(A) \times Sum^n \to Sum$ is a "bad preserving" function, i.e. $bad \in \{s_0, \ldots, s_{n-1}\}$ implies $sum_A(s, s_0, \ldots, s_{n-1}) = bad$.

We extend the above elemental sum_* functions to summarize the state of an arbitrary non-leaf Neo system Ω as follows. Ω is (1), where $A \in Is \cup Rs$ and $\Omega_0, \ldots \Omega_{n-1}$ are open Neo systems. Then $sum_\Omega : states(\Omega) \to Sum$ is defined by

$$sum_\Omega(s_a, s_0, \ldots, s_{n-1}) = \\ sum_A(s_a, sum_{\Omega_0}(s_0), \ldots, sum_{\Omega_{n-1}}(s_{n-1}))$$

2) Summary Sequence of Executions: Given an execution $e = s_0, \alpha_1, \ldots, \alpha_k, s_k$ of a Neo system Ω, we define the summary sequence $sum(e)$ as follows. Let $\alpha'_i = \alpha_i$ if $\alpha_i \in ext(\Omega)$, otherwise $\alpha'_i = \lambda$. We start with the sequence

$$sum_\Omega(s_0), \alpha'_1, \ldots, \alpha'_k, sum_\Omega(s_k)$$

and delete all $\alpha'_i, sum_\Omega(s_i)$ such that $\alpha'_i = \lambda$ and $sum_\Omega(s_i) = sum_\Omega(s_{i-1})$.

3) Safety Definition: For Neo system Ω and state $s \in states(\Omega)$, we say that s is *safe* if $sum_\Omega(s) \neq bad$. We say that Ω itself is safe if all its reachable states are safe. The primary goal of this paper is to establish that *all* Neo systems are safe, by only proving a handful of lemmas about the "ingredient" IOAs (Ls, Is, Rs) and their summary functions.

D. Neo Pre-order $\preceq$

We define a pre-order $\preceq$ on open Neo systems. Given two m-peer open Neo systems Ω_1 and Ω_2 that are either both tagged or untagged, the relation $\Omega_1 \preceq \Omega_2$ holds if, for all executions e_1 of Ω_1, there exists an execution e_2 of Ω_2 such that $sum(e_1) = sum(e_2)$.

Lemma 3. *$\preceq$ is transitive.*

Lemma 4. *$\Theta \preceq \Omega$ if and only if $\phi_i(\Theta) \preceq \phi_i(\Omega)$.*

Lemma 5. *Let Neo systems $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ and $\Theta = A \cdot \prod_{i=0}^{n-1} \phi_i(\Theta_i)$, where $A \in Is \cup Rs$. Suppose for some k, $\Omega_k \preceq \Theta_k$ and for all $i \neq k$, $\Omega_i = \Theta_i$. Then, for all executions e of Ω, there exists an execution e' of Θ such that $sum(e) = sum(e')$.*

Lemma 6. *Let $\Theta = A \cdot \prod_{i=0}^{n-1} \phi_i(\Theta_i)$ and $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ be open Neo systems such that $\Theta_i \preceq \Omega_i$, for all i. Then, $\Theta \preceq \Omega$.*

Lemma 7. (Leaf as a Network Invariant) *Suppose that the m-peer open Neo system $\Omega_L = A \cdot \prod_{i=0}^{n-1} \phi_i(L(n))$ satisfies $\Omega_L \preceq L(m)$. Then, for any m-peer open Neo system Ω, $\Omega \preceq L(m)$.*

Proof. If Ω is an m-peer leaf, then $\Omega = L(m) \preceq L(m)$. Otherwise, let $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ be an m-peer open Neo system. Assuming that each $\Omega_i \preceq L(n)$ (inductive hypothesis), we will prove, by structural induction on the construction of Ω, that $\Omega \preceq L(m)$. By Lemma 6 and inductive hypothesis, $\Omega \preceq \Omega_L$. By transitivity of $\preceq$ (Lemma 3) and $\Omega_L \preceq L(m)$ (assumption in lemma statement), $\Omega \preceq L(m)$. □

Theorem 1. (Every Neo system is safe.) *Suppose that for each n-child node $A \in Rs \cup Is$, $\Omega_L = A \cdot \prod_{i=0}^{n-1} \phi_i(L(n))$ is safe. Furthermore, suppose that if A is an m-peer internal node, then $\Omega_L \preceq L(m)$. Then all Neo systems are safe.*

Proof. Let Ω be an (open or closed) Neo system (1). From the assumptions of this lemma and Lemma 7, $\Omega_i \preceq L(n)$ for all i. Let e be an arbitrary execution of Ω. By an n-fold application of Lemma 5, there exists an execution e' of Ω_L such that $sum(e') = sum(e)$. By definition of sum, if no state in e' summarizes to bad, then no state in e summarizes to bad. Therefore Ω is safe. □

The significance of Theorem 1 is that if we establish Ω_L's safety, for all $A \in Rs \cup Is$, and $\Omega_L \preceq L(m)$, for all $A \in Is$, then any configuration of the Neo nodes (which would typically be closed) is safe. Parameteric model checking comes into play, since when the elements of $Rs \cup Is$ are paramterized (by number of children n and number of peers m), these safety and preorder checks are parameterized verification problems.

III. Mapping Protocols' Safety to Neo Safety

We have defined safety of a Neo system (Sect. IV-D1) to mean that no reachable state summarizes to bad, which is somewhat removed from the actual invariant one might be interested in. Here, we illustrate how an invariant of interest can be expressed in the form of the Neo safety definition. The key is that the summary functions must be forced to return *bad* whenever the specific safety property of interest is violated.

A. Cache Coherence

In a typical MOESI cache coherence protocol [27], $Sum = \{I, S, O, E, M, bad\}$, [5] and cache coherence means that if any leaf summarizes to M or E, then all other leaves must summarize to I. To ensure that Neo system safety (no reachable state summarizes to bad) implies all reachable states are cache coherent, we require some simple constraints on sum_A for each $A \in Is \cup Rs$. Let us define an ordering $<$ on Sum by $I < S, O < E, M < bad$. Recalling that sum_A has type $states(A) \times Sum^n \to Sum$, where n is the arity of A, the *cache coherence constraint* on sum_A is as follows[6]:

[5]The reasoning of this section can be extended to handle cases where Sum includes more than just these 6 elements.

[6]Note that the constraints presented here are the weakest set of constraints that allow us to prove Lemma 8 below; they do not preclude, e.g., that inconsistencies between the state s_a of A and $s_0, \ldots, s_{n-1}$ yield bad.

- Whenever there exists distinct $i, j \in \mathbb{N}_n$ such that $s_i \in \{M, E\}$ and $s_j \neq I$, we require $sum_A(s_a, s_0, \ldots, s_{n-1}) = bad$, and
- For all $i \in \mathbb{N}_n$, $s_i \leq sum_A(s_a, s_0, \ldots, s_{n-1})$ (i.e. sum_A is monotonically increasing with $<$)

Lemma 8. *If sum_A satisfies the cache coherence constraint for all $A \in Is \cup Rs$, then Neo safety implies cache coherence.*

Proof. Let s be a state of a Neo system Ω. We argue, by structural induction on Ω, that whenever s contains a cache coherency violation, $sum_\Omega(s) = bad$. The base case $\Omega \in Ls$ holds vacuously, since a leaf in isolation cannot violate cache coherency. Now choose $A \in Is \cup Rs$ with arity n. Then $s = (s_a, s_0, \ldots, s_{n-1})$, where $s_a \in states(A)$ and $s_i \in states(\Omega_i)$, $0 \leq i < n$. If s contains a cache coherency violation then there exists a leaf L in s that summarizes to M or E, and a distinct leaf L' that summarizes to something other than I. If L and L' are both components of Ω_i for some i, then from our inductive hypothesis, $sum_{\Omega_i}(s_i) = bad$, and from Sect. II-C1, we have that $sum_\Omega(s) = bad$. On the other hand, suppose L and L' are respectively components in Ω_i and Ω_j with $i \neq j$. Since the cache coherency constraint requires sum to be monotonic, it follows that $E \leq sum_{\Omega_i}(s_i)$ and $S \leq sum_{\Omega_j}(s_j)$, and again the cache coherency constraint requires $sum_\Omega(s) = bad$. □

We envision that other types of Neo systems will need similar side arguments to relate Neo safety to a more concrete property of interest, and such arguments will be as straightforward as what was required to prove Lemma 8 above.

B. Distributed Lock Management (DLM)

Distributed Lock Management (DLM) protocols are used to ensure safe access to shared resources such as disks and files. Several DLM protocols are based on the DEC VMS's DLM implementation [28], including the Oracle Cluster File System (OCFS2) that appears in the Linux Kernel [24] [18]. VMS's DLM has 6 permissions—Null (NL), Concurrent Read (CR), Concurrent Write (CW), Protected Read (PR), Protected Write (PW), and Exclusive (EX). The following combinations of permissions are prohibited: (CR,EX), (CW,EX), (CW,PW), (CW,PR), (PR,EX), (PR,PW), (PW, EX), (PW,PW), (EX,EX).

For scalable resource management, one can organize nodes in a cluster as a hierarchy according to the Neo framework. This would facilitate verification, for arbitrary system sizes, that no two nodes hold a prohibited combination of permissions. We could set $Sum = \{NL, CR, CW, PR, PW, EX, bad\}$ and define a partial order $<$ such that $NL < CR < PR < PW < EX < bad$ and $NL < CR < CW < EX < bad$; $<$ does not order PW and CW. Then, imposing similar constraints to the sum functions of Sect. III-A, one can show that sum not evaluating to bad implies that the system never violates DLM safety.

IV. Case Study: The NeoGerman Protocol

To illustrate our verification methodology, we design and verify a hierarchical cache coherence protocol called NeoGerman. Using a parametric model checker, we verify that NeoGerman is a Neo System and, consequently, satisfies the coherence invariant for arbitrary configurations.

A. NeoGerman Description

German's protocol is a simple, directory-based caching protocol proposed as a challenge for parameterized verification [10]. To make the protocol hierarchical, we made significant modifications. In particular, the directory was modified to communicate with a parent, hence serving as an internal node.[7]

1) The German Protocol: The German protocol is a flat cache coherence protocol. We use the version specified in [6], which is parameterized to have a single directory connected to an arbitrary number of private caches. Each cache block is in one of three states: I(nvalid), S(hared), or E(xclusive). The protocol uses the directory to maintain the invariant that no two caches are simultaneously in (S, E) or (E, E). The directory maintains a list of all nodes in S or E, called sharers.

If a cache sends a message to the directory to request S ($GetS$) when there is a cache in E, the cache in E gets sent an $Invalidate$ message, and the directory collects an invalidation acknowledgement ($InvAck$) from it. The directory then sends a $GrantS$ message to the requesting cache to grant it S permissions. If the directory receives a $GetE$, it invalidates all sharers and collects all their $InvAck$'s before sending a $GrantE$ message to the requesting cache.

2) Modifications to German: To turn German into an open Neo system, we modify the directory so it behaves like a private cache along a (previously non-existent) communication channel shared with a parent. Upon receiving requests from its children, the directory now has the ability to seek permissions from *its* parent. We will refer to this modified directory as the *internal directory*, to distinguish it from the original German directory that we use as a root node to close the Neo hierarchy.

The internal directory maintains a variable called $Permissions\_O$, which summarizes the permissions of the open Neo system it heads as that of a single private cache. The intent is that if, for example, $Permissions\_O$ is in I and the internal directory receives a $GetS$ from a child, the internal directory forwards the request to its parent. Upon receiving a subsequent $GrantS$ from its parent, $Permissions\_O$ changes to S and the internal directory sends a $GrantS$ to the requesting child and makes it a sharer. If the internal directory receives an $Invalidate$ from its parent, it invalidates all sharing children and collects all $InvAck$s. Finally, the internal directory sends an $InvAck$ to its parent and updates $Permissions\_O$ to I.

B. Tying NeoGerman to the Neo Framework

In NeoGerman, we have $U = \{GetS, GetE, InvAck\}$, $D = \{GrantS, GrantE, Invalidate\}$ and $P = \emptyset$. The private caches correspond to tagged leaf processes of the form $\phi_i(L)$, where L is a leaf node. Each individual leaf is tagged with a parameter that enables unique communication with its

[7]The directory was used unmodified to create the root node.

directory. The directory/memory $R(n)$ of the original German protocol constitutes an n-child root node of NeoGerman. The directory $I(n)$ of the NeoGerman protocol constitutes an n-child internal node. Armed with $R(n)$, $I(n)$, L and the Cubicle process composition methods we discussed above, we have all the ingredients to build NeoGerman as a Neo system.

C. Modeling NeoGerman in a Model Checker

We modeled NeoGerman in Cubicle [8]. Cubicle is a symbolic model checker used to verify parameterized array-based systems by using a backwards reachability algorithm and an SMT solver. Its support for parametric verification allows us to verify safety properties for arbitrary configurations of a Neo hierarchy. Cubicle's processes are parameterized by indices of a built-in type *proc*. The state of an arbitrary number of processes is represented by arrays indexed by *proc*. Even though neither Cubicle nor our framework impose size restrictions on communication buffers, we model NeoGerman with a single-entry communication buffers for simplicity.

1) Representing a Process: To illustrate how we model processes in Cubicle, let set $B = \{\phi_i(A) : i \in \mathbb{N}_n\}$, where A is a Neo leaf node with $steps(\phi_i(A)) = \{(s_0, (a_0, i), s_1), (s_1, (a_1, i), s_2), (s_2, (a_2, i), s_0)\}$. Let $start(A_i) = \{s_0\}$, $in(A_i) = \{(a_0, i)\}$, $out(A_i) = \{(a_1, i)\}$, and $int(A_i) = \{(a_2, i)\}$. We would model B in Cubicle as follows [8]:

```
type state_a = s0|s1|s2 //state type declaration
array State_A[proc]:state //state array variable declaration

init (i)
{State_A[i]=s0//initialize each proc's state to start state}

transition a0(i) //input transition
requires {State_A[i]=s0} //guard
{ State_A[i]:=s1; } //state update

transition a1(i) //output transition
requires {State_A[i]=s1}
{ State_A[i]:=s2; }

transition a2(i) //internal transition
requires {State_A[i]=s2}
{ State_A[i]:=s0; }
```

2) Representing Composition: For IOA B, let $steps(B) = \bigcup_{i=0}^{n-1}\{(S_0, (a_0, i), S_1), (S_1, (a_1, i), S_2), (S_2, \pi, S_0)\}$. Let $start(B) = \{S_0\}$, $out(B) = \{(a_0, i)\}$, $in(B) = \{(a_1, i)\}$, and $int(B) = \{\pi\}$. By combining the guards and state updates of transitions with identical names, we represent the composition $C = B \cdot \prod_{i=0}^{n-1} A_i$ as follows:

```
type state_b = S0|S1|S2; type state_a = s0|s1|s2
var State_B:state_p; array State_A[proc]:state_a

init (i)
{ State_B=S0 & State_A[i]=s0 }

transition a0(i) //internal transition
requires {State_B=S0 & State_A[i]=s0}
{ State_B:=S1; State_A[i]:=s1; }

transition a1(i) //internal transition
requires {State_A[i]=s1 & State_B=S1}
{ State_A[i]:=s2; State_B:=S2; }

transition π() //internal transition
requires {State_B=S2}
{ State_B:=S0; }

transition a2(i) //internal transition
requires {State_A[i]=s2}
{ State_A[i]:=s0; }
```

D. Proving the NeoGerman Hierarchy is Coherent

We leverage the Neo framework and Cubicle's parametric verification to prove that any NeoGerman configuration is coherent. Our strategy is to first define $sum_{A(n)}$ for each $A \in \{R, I\}$ and $n \geq 1$ such that it satisfies the constraints of Section III-A. Then, we prove the conditions of Theorem 1 and Lemma 8, from which coherency of Ω follows. Let $\Omega_{A(n)} = A(n) \cdot \prod_{i=0}^{n-1} \phi_i(L)$, and let $Sum = \{I, S, E, bad\}$, ordered $I < S < E < bad$. To leverage Theorem 1 and Lemma 8, we model $\Omega_{R(n)}$ and $\Omega_{I(n)}$ in Cubicle and prove the following for all n and each $A \in \{R, I\}$

$$\Omega_{A(n)} \text{ is safe} \tag{2}$$

$$\forall i : s_i \leq sum_{A(n)}(s, s_0, \ldots, s_{n-1}) \tag{3}$$

$$\begin{aligned}\forall i \neq j : s_i \in \{M, E\} \wedge s_j \neq I \\ \Rightarrow sum_{A(n)}(s, s_0, \ldots, s_{n-1}) = bad\end{aligned} \tag{4}$$

$$\Omega_{I(n)} \preceq L \tag{5}$$

First, we define $sum_{R(n)}$ and $sum_{I(n)}$. Unlesss the cache coherence constraints (Sec. III-A) require *bad*, $sum_{R(n)}(s, s_0, \ldots, s_{n-1}) = E$. Likewise, unless the cache coherence constraints require *bad*, $sum_{I(n)}(s, s_0, \ldots, s_{n-1}) = Permissions\_O$, where $Permissions\_O$ is a Cubicle variable of $I(n)$. Hence, $Permissions\_O$ is a function of $states(I(n))$.

1) Safety and Monotonicity of Sum: [9] To prove (2), we parametrically model check $\Omega_{R(n)}$ and $\Omega_{I(n)}$; after each $\Omega_{I(n)}$ transition, a variable Sum_Output_O representing the output of sum_* is updated to $Permissions\_O$. The following is specified as a safety violation:

```
unsafe(i,j) CacheState[i]=Bad || (CacheState[i]=E &
    CacheState[j]!=I)
```

where, for all i, $CacheState[i] \equiv sum_L(\phi_i(L))$.

For (3), $sum_{R(n)}$ is monotonic by definition. To prove $sum_{I(n)}$ is monotonically increasing, we model check $\Omega_{I(n)}$, specifying the following as safety violations:

```
unsafe(i) {Sum_Output_O!=E & CacheState[i]=E}
unsafe(i) {Sum_Output_O=I & CacheState[i]=S}
```

To prove (4), we model check $\Omega_{I(n)}$ with the following:

[8]For all Cubicle code in this paper, we deviate slightly from Cubicle syntax for conciseness.

[9]As a result of the limitations of Cubicle, we need to *prove* (3) and (4) for reachable states, rather than writing code that clearly satisfies these constraints for any state.

```
unsafe (i j) {not ((CacheState[i]=E &
       CacheState[j]!=Invalid) => Sum_Output_O=Bad)}
```

2) Observational Process Pre-order: To prove (5), we employ a similar approach to Park et al. [25], with an important difference that we generalize to a parametric setting to verify *our* pre-order. Park et al. show how to prove that a process A implements a process B in a model checker by expressing B as a function. A is model-checked and, on each transition t, B's function is called to give B's next state, given A's state at the start of t. An assertion checks that a simulation relation holds, given t's action and the states of A and B at the start and end of t. Cubicle does not support functions and in-line assertions due to its underlying algorithm, so we must rely only on safety properties. As a result of these limitations, we must prove a stricter pre-order $\preceq_c$ based on a slightly different function sum_c defined below.

Let IOA A execution $e = s_0\alpha_1 s_1 \dots \alpha_k s_k$. Then, $sum_c(e)$ is a sequence derived as follows. Replace each s_i with $sum_A(s_i)$. Replace each $\alpha_i \in int(A)$ with the symbol λ. For IOA A_1 and A_2, $A_1 \preceq_c A_2$ implies for any execution e_1 of P_1, there exists an execution e_2 of A_2 such that $sum_c(e_1) = sum_c(e_e)$.

Lemma 9. *$\Theta \preceq_c \Omega$ implies $\Theta \preceq \Omega$.*

The definition of sum_c implies that, to prove $\Omega_{I(n)} \preceq_c L$, we must match every $\Omega_{I(n)}$ execution with an equal-length execution of L. Hence, we make a trivial modification to the L IOA in NeoGerman by adding λ to $int(L)$ such that, for all $s \in states(L)$, $(s, \lambda, s) \in steps(L)$. This allows L to make as many stuttering steps as needed to match execution fragments of $\Omega_{I(n)}$ that have only internal steps with no change in summary state.

The key to our approach in proving the pre-order is that in the same Cubicle file, we model both $\Omega_{I(n)}$ and L [10] and instrument the code of both processes such that they transition in lockstep, starting with $\Omega_{I(n)}$. Our instrumentation also guides L to pick transitions that match each $\Omega_{I(n)}$ transition. We use a safety property to check that, after each L transition, the states and actions of L and $\Omega_{I(n)}$ correspond as required for sum_c to be equal. We also use a safety property to check that, after each $\Omega_{I(n)}$ transition, there always exists an L transition that *can* fire. If both safety checks pass, then we know that $\Omega_{I(n)} \preceq_c L$ and, thus, $\Omega_{I(n)} \preceq L$ (Lemma 9).

Matching Executions:

1) To force $\Omega_{I(n)}$ and L to transition in lockstep, a variable *L_to_run* is initialized to false. It is set to true after each $\Omega_{I(n)}$ transition and set to false after each L transition. Then, the expression *L_to_run=False* is conjuncted to the guard of each $\Omega_{I(n)}$ transition and *L_to_run=True* is conjuncted to the guard of each L transition.

2) To access the most recent actions of L and $\Omega_{I(n)}$, we update a variable *O_action* only at the end of each $\Omega_{I(n)}$ transition and variable *L_action* only at the end of each L transition. For external transitions, *O_action* and *L_action* are updated to the transition's name. Otherwise, they are updated to *lambda*.

3) To guide L to make a matching external step to each external step of $\Omega_{I(n)}$'s, we conjunct to the guard of each L external transition named *trans_name* the expression *O_action=trans_name*.

4) To guide L to make a matching step to each internal step of $\Omega_{I(n)}$'s, a variable *Forced_Transition* is updated to some value *int_name* after each $\Omega_{I(n)}$ internal transition. Then, the guard of the desired L internal transition is conjuncted with the expressions *Forced_Transition=int_name* and *O_action=lambda*.

Note that the above modifications maintain the integrity of the pre-order check. All modifications to L's guards involve logical conjunctions, which could only restrict L's transitions. And the only modification to $\Omega_{I(n)}$'s guards is conjuncting *L_to_run=False*, which holds after every L transition.

Safety Checks:

We must check that, after each L transition, the actions and summaries of states of L and $\Omega_{I(n)}$ match. Where $sum_L(S_L) \equiv$ *Cache_State_L*, the following illustrates our safety checks.

```
unsafe() {Sum_Output_O!=Cache_State_L & L_to_run=False}
unsafe() {O_action!=L_action & L_to_run=False}
```

Finally, we must check that after each $\Omega_{I(n)}$ transition, there exists an L transition that can fire. To do that, we express a safety property that says that if *L_to_run=True*, the conjunction of the guards of all L transitions must not evaluate to *False*. With both safety checks passing, we can conclude that $\Omega_{I(n)} \preceq_c L$, and, consequently, $\Omega_{I(n)} \preceq L$ (Lemma 9).

This completes our proof that NeoGerman is a Neo hierarchy and thus CCsatisfies coherence for any arbitrary configuration. The full NeoGerman Cubicle model and proofs can be viewed at: http://people.duke.edu/~om26/papers/FMCAD16.

V. Characterizing the Scope of our Framework

To characterize the scope of our framework, we define a fragment of first order formulas over leaf states that we can verify using our approach and define summary functions that are guaranteed to verify a given property. Let $LP = \{\ell_1, \dots, \ell_m\}$ be a set of predicates over the leaf states $states(L)$. We show how we can verify any invariant of the form

$$\forall x_1, \dots, x_k. Distinct(x_1, \dots, x_k) \Rightarrow P(x_1, \dots, x_k) \qquad (6)$$

where the x_i's range over leaves, $Distinct(x_1, \dots, x_k)$ indicates that the x_i's are pairwise not equal, and $P(x_1, \dots, x_k)$ is a propositional formula over the atoms $\{\ell_j(x_i) | 1 \le j \le m \wedge 1 \le i \le k\}$. For example, where $LP = \{E, S, I\}$, the cache coherence invariant we verified for NeoGerman could be expressed as $\forall x_1, x_2. Distinct(x_1, x_2) \Rightarrow (E(x_1) \Rightarrow I(x_2))$.

To verify that (6) is invariant, we construct summary functions such that the state of a NEO hierarchy summarizes to *bad*

[10]Observe that L is identical to each $\phi_i(L)$, except L has no indices in transitions and its state is not a *proc* array.

if and only if (6) is false of the system's state. The summary functions have co-domain Sum, where Sum is the (finite) set

$$Sum = \left(2^{LP} \rightarrow \{0, \ldots, k\}\right) \cup \{bad\}$$

When bad is returned to node A's summary function, this indicates that (6) fails to hold of the sub-hierarchy rooted at A. Otherwise, a function f is returned, with the interpretation that $f(LP')$ is the number of distinct leaves under A with states satisfying exactly the predicates $LP' \subseteq LP$; if there are k or more such leaves, $f(LP') = k$.

The leaf summary function sum_L simply returns the function that maps all sets to 0, except the exact subset of LP that holds of the leaf's state, which is mapped to 1. However, if $k = 1$ and P does not hold of the leaf's state, bad is returned. Where A is an n-child internal or root node, it is relatively straightforward to define how sum_A (which is independent of its first argument $s_A \in states(A)$) depends on arguments $(g_0, \ldots, g_{n-1}) \in Sum^n$ and under what conditions it should return bad. sum_A returns the function that maps each LP' to $g_0(LP') + \cdots + g_{n-1}(LP')$, saturating at k, unless any g_i is bad or the counts of $(g_0, \ldots, g_{n-1})$ for each LP' indicate that some x_i's below A violate $P(x_1, \ldots, x_k)$, in which case bad is returned.

VI. Conclusion

We present the Neo framework that leverages network invariants and parameterized model checking together to enable the design and automated verification of hierarchical (tree) protocols that, for any size or configuration of the hierarchy, satisfy a safety property. We use our framework to design and verify a hierarchical cache coherence protocol called NeoGerman, using Cubicle as our parameteric model checker. Significantly, we prove an observational pre-order in a parametric setting. We believe there are no fundamental limitations that prevent our framework from being used to design and verify more complex, industrial-strength *hierarchical* protocols, especially given that model checkers like Cubicle have already been used to parametrically verify several industrial-strength *flat* protocols.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant CCF-142-1167. We thank John Erickson and Kenneth McMillan for providing helpful advice for this work.

References

[1] P. A. Abdulla and B. Jonsson. On the existence of network invariants for verifying parameterized systems. In *Correct System Design*. Springer, 1999.

[2] 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.

[3] J. G. Beu, J. A. Poovey, E. R. Hein, and T. M. Conte. High-speed formal verification of heterogeneous coherence hierarchies. In *HPCA*. IEEE, 2013.

[4] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, 2000.

[5] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *HLVDT*. IEEE, 2007.

[6] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*. Springer, 2004.

[7] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*. Springer, 1995.

[8] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *CAV*. Springer, 2012.

[9] Z. Ganjei, A. Rezine, P. Eles, and Z. Peng. Abstracting and counting synchronizing processes. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2015.

[10] S. German. Personal correspondence. 2008.

[11] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. 2010.

[12] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *CONCUR*. Springer, 2002.

[13] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. Incremental, inductive coverability. In *CAV*. Springer, 2013.

[14] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[15] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *PODC*. ACM, 1989.

[16] M. Kyas. Verifying a network invariant for all configurations of the futurebus+ cache coherence protocol. *Electronic Notes in Theoretical Computer Science*, 2001.

[17] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, 2004.

[18] LWN.net. The ocfs2 filesystem. http://lwn.net/Articles/137278/.

[19] O. Matthews, J. Bingham, and D. J. Sorin. Verifiable hierarchical protocols with network invariants on parametric systems. Extended version of this paper (with proofs), 2016. http://people.duke.edu/~om26/papers/FMCAD16/fmcad16_neo_extended.pdf.

[20] O. Matthews, M. Zhang, and D. J Sorin. Scalably verifiable dynamic power management. In *HPCA*. IEEE, 2014.

[21] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, 1999.

[22] K. L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *CHARME*. Springer, 2001.

[23] J. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *FMCAD*, 2009.

[24] oss.oracle.com. General-purpose cluster file system. https://oss.oracle.com/projects/ocfs2/.

[25] S. Park, S. Das, and D. L. Dill. Automatic checking of aggregation abstractions through state enumeration. *Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[26] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.

[27] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[28] D. W. Thiel. The VAX/VMS distributed lock manager. *VAX cluster Systems*, page 29, 1987.

[29] M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, 1987.

[30] G. Voskuilen and T.N. Vijaykumar. Fractal++: Closing the performance gap between fractal and conventional coherence. In *ISCA*. IEEE, 2014.

[31] G. Voskuilen and T.N. Vijaykumar. High-performance fractal coherence. In *ASPLOS*. ACM, 2014.

[32] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*. Springer, 1990.

[33] M. Zhang, J. D. Bingham, J. Erickson, and D. J Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *HPCA*. IEEE, 2014.

[34] M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *MICRO*. IEEE, 2010.

Modular Specification and Verification of a Cache-Coherent Interface

Kenneth McMillan
Microsoft Research

***Abstract*—We consider the problem of constructing a modular specification for a cache coherence protocol implementing a weakly consistent shared memory model. That is, we wish to specify the interface between components in a way that, if all components locally satisfy their interface specifications, the components collectively implement the desired memory semantics. The problem we face is that the semantics involves an existential quantifier over memory orderings that cannot be witnessed locally. We solve this problem using a specification idiom based on reference objects and circular assume-guarantee reasoning. The specification is written using a language and a tool called Ivy. We use Ivy to specify the TileLink coherent memory interface protocol and to prove compositionally that interconnections of TileLink components implement the memory semantics correctly. The specification is also used for modular specification-based testing of RTL components.**

I. Introduction

Modular specifications have many advantages. Most importantly, they allow us to reduce global reasoning about a system to local reasoning about components and their local specifications. But what if the system specification itself refers to actions of all the components? How, then, do we write local specifications?

This problem arose in a writing a modular specification for TileLink, a coherent memory interface protocol implementing a weakly consistent shared memory model. The specification of memory consistency requires that for every system execution, there exists a consistent global ordering of all the memory operations occurring in system components. This ordering must respect a "happens-before" relation that defines the consistency model. The difficulty this introduces is in constructing the required global ordering. By simply writing down a witness function for this object, we have already destroyed the locality of the proof, since the witness depends on the history of every system component.

While we consider a cache consistency protocol here, the same problem could arise in other kinds of systems in which distributed processes collectively implement some global semantics. This class could include, for example, distributed file systems and hash tables. We will refer to the general problem addressed here as the *collective semantics problem*. The problem is, in essence, to specify the role of a single component within the larger computation, without reference to the system as a whole.

TileLink [6] is a generic interface protocol that is implemented by a variety of components, such as processor cores, hierarchical caches, snooping hubs, directories, memory banks, crossbars, and so on. It is proposed as a standard memory architecture for systems based on the RISC-V instruction set architecture [17]. The intent is that components satisfying the protocol specification can be composed into correct memory hierarchies of arbitrary size. For this reason, two important criteria for a modular TileLink specification are *genericity* and *scalability*. That is, we don't want distinct specifications for caches, directories and so forth. Rather, we want all TileLink components to obey the *same* generic specification. Moreover, given a collection of components satisfying the specification, we should be able to verify in scalable way that arbitrarily large hierarchies constructed from the components implement the weak memory model correctly. A final criterion is that a the specification should be *testable*, that its, it should be possible to automatically test a component against its specification.

We satisfy these criteria for TileLink by introducing a particular assume-guarantee idiom. In this idiom, we define a *reference object* that constructs the memory ordering as the system executes. The components call into this object to indicate when particular memory events should be serialized. We then specify the allowed events at each system interface *relative to* the reference object. The history of events at the interface constrains the allowed serializations. These specifications are stitched into a modular assume-guarantee proof that ensures the constructed memory ordering is consistent.

Applying our assume-guarantee idiom, we arrive at a specification for TileLink that is modular, generic, scalable and testable. This specification can act as a reference document for engineers implementing TileLink components. Moreover, engineers without a formal methods background can use the specification for rigorous testing of components. We implemented a modular specification-based constrained random testing framework, and used it to find latent bugs in TileLink components under development.

Finally, we validated the specification by formally verifying that a number of abstract components implement it. These proofs are parametric in several dimensions, including address space, data width and cache line size. This gives the first modular proof that a cache coherence protocol implements a weak memory model.

II. Case study: TileLink

TileLink [6] is an open protocol specification that is intended to connect the memory hierarchy components of a

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

system-on-chip. These components can include CPU's, memories, caches, directories, and adapters to other protocols. TileLink components are connected using a common interface protocol. Our goal is to give a modular specification that allows us to prove that arbitrary hierarchies built from such components implement a weakly consistent memory model. The protocol itself is too complex to give its full specification here. Instead, we will focus on the high-level structure of the specification and in particular how we solve the collective semantics problem. The formal specification is given in a language called Ivy. Full source code for the specification and the Ivy tool are available online [14].

A. The reference specification

To begin with, we must define the memory model. A *memory history* is a sequence of *operations*. A operation can be a read, a write, or an atomic memory operation, such as compare-and-swap (CAS). An operation has a *locale* that identifies the processor that executes it. A memory history is *consistent* if the data value of every read matches that of the most recent write of the same address (and if the semantics of any other atomic operations is similarly respected).

A memory history H is *sequentially consistent* if there is some permutation π such that πH is consistent and π preserves the relative order of operations with the same locale (a permutation operates on a sequence by re-ordering its elements). We will call π a *consistent serialization*. Sequential consistency is seldom implemented, since it rules out some common optimizations. In a weak consistency model, the serialization is allowed to re-order some events with the same locale. It must, however, preserve the "happens-before" relation between operations. Here, we say that operation A happens-before B if they have the same locale and either they have the same address, or one is an atomic operation. This is sufficient to guarantee Partial Store Ordering (PSO) if the atomic operations are fences, and Release Consistency (RCsc) [10] if they are lock acquires and releases. The high-level specification approach is not dependent on the exact happens-before relation, however.

The chief difficulty in writing a modular specification using this model is that the correctness criterion is global and existential: it requires that a consistent serialization *exists* for each global behavior of the system. Because the serialization is global, we cannot provide a witness for it that refers to only one system component. Thus, there is no obvious way to localize the proof.

We solve this problem using by creating an *abstract service* that progressively constructs a consistent serialization. This imaginary service can be called by components of the system to indicate that a given memory operations should be serialized at the current global time (or to constrain the serialization in other ways). Thus, the system components cooperate through the abstract service to build the witness for the existential quantifier in the correctness criterion. If at any point the partially constructed serialization becomes inconsistent, the abstract service fails. We will call the object implementing this service the *reference object*. A simplified version of the reference object is show in the Ivy language in Fig. 1.

```
1  object reference = {
2
3      instance evs(T:ltime) : memop
4      function mem(A:address) : data
5
6      relation happensBefore(T1,T2) =
7          T1 < T2
8          & evs(T1).loc = evs(T2).loc
9          & (evs(T1).addr = evs(T2).addr
10                 | evs(T1).atomic | evs(T2).atomic)
11
12     method serialize(lt:ltime, loc:locale) = {
13
14         assert ~evs(T).serialized
15         assert happensBefore(T,lt) -> evs(T).serialized
16
17         evs(lt).serialized := true;
18
19         var a : address := evs(lt).addr
20
21         if evs(lt).op = read {
22             evs(lt).data := mem(a)
23         }
24         else if evs(lt).op = write {
25             mem(a) := evs(lt).data
26         }
27     }
28 }
```

Fig. 1. Reference object specifying consistency model.

The state of the reference object consists of a map *evs* from local time (as measured by the local CPU clocks) to memory operations, and a map *mem* from addresses to data, representing the current state of memory. Here, the types *ltime*, *address* and *data* are *uninterpreted*. Thus, this specification is independent of the size of the address the address space or data words. We assume only that *ltime* is totally order by $<$.

The happens-before relation is defined as above, with respect to local time. The *serialize* method tells the reference object to add one event to the serialization. It has an additional parameter *loc* that gives the locale of the serializing component. It isn't used here, but as we will see later, it is important for constructing assume-guarantee specifications. The *serialize* method makes key assertions at lines 14 and 15. These state that, when a memory operation *lt* is serialized, it has not previously been serialized, and all operations that happen-before *lt* have already been serialized (the variable *T* here is implicitly universally quantified). This ensures that the set of serialized operations is always weakly consistent. If the operation is a read, its data value is taken from the current state of memory, or, if it is a write, the current state of memory is updated (the assignment at line 25 mutates the map *mem*). In the Ivy language, all methods execute atomically, so multiple calls to *serialize* cannot interleave.

Typically, a cache or memory component will call *serialize* when a memory operation is executed. These calls are "ghost"

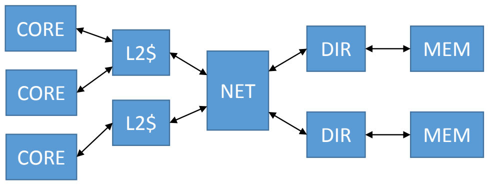


Fig. 2. Example TileLink hierarchy. Arrows are TileLink interfaces.

code, which we add to witness the serialization. Other kinds components, such as store buffers, can also serialize memo operations. A store buffer requires a richer interface, since it may constrain a read to be serialized after the future serialization of a write. We will not discuss this generalization, however. Also, we need a method for processors to create memory events. From the point of view of the memory system, however, *evs* is an arbitrary sequence of memory events, about which we make no assumptions.

Our next task will be to specify the component interfaces in a way that allows us to prove the assertions in *reference* using modular assume-guarantee reasoning.

B. The TileLink protocol

The TileLink protocol has two roles: *client* and *manager*. Roughly speaking, clients act like processors and managers act like memories. Some components can take both roles, however. For example, a first-level (L1) cache plays the role of manager when speaking to a processor, and client when speaking to the second-level (L2) cache. We can also create a hub or crossbar that routes messages between multiple clients and managers. By layering clients on top of managers we can create an arbitrarily deep memory hierarchy. An example of such a hierarchy is shown in Fig. 2.

In the simplest case, the TileLink interface connects one client and one manager. The manager provides three methods: *Acquire*, *Finish* and *Release* while the client provides two: *Grant* and *Probe*. In hardware, these are implemented by signaling across the interface using a simple handshake. These atomic methods are combined into larger, non-atomic transactions. The typical flow of transactions is shown in Fig. 3. A client calls *Acquire* to request a shared or exclusive copy of a cache line. The manager makes a sequence of calls to *Grant* providing the data for that cache line. When the entire cache line is received, the client calls *Finish* to acknowledge completion of the transaction. If another client requires exclusive access to the cache line, the manger calls the *Probe* method of the client. The client responds by calling *Release* to indicate it has given up its copy of the line, and if necessary, return modified data. Alternatively, the client may call *Release* voluntarily, in which case the manager acknowledges with a special *Grant* call.

Since multiple transactions can interleave, each is given a transaction ID (txid) that remains unique while the transaction is in progress. Multiple transactions on the same cache line may be merged into a single transaction with a single txid. This may occur, for example, if a read to a line is followed

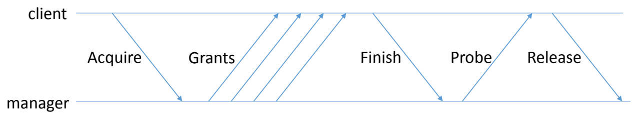


Fig. 3. Transaction flow in TileLink.

immediately by a write, which requires a rapid upgrade from shared to exclusive privileges.

The protocol also supports clients without caches. Such a client submits memory operations directly to the manager using special *Acquire* calls. The manager returns results using special *Grant* calls. The manager is allowed to cache the data, and to coalesce multiple operations from a client into single operations transmitted to its own manager.

The protocol is designed to be resilient to reordering of messages transmitted between client and server. Much of the subtlety in the protocol is involved in managing concurrent transactions (especially the management of txid's) and in managing race conditions. For example, the protocol must handle the case in which the *Finish* that completes a transaction is overtaken by the *Acquire* that starts the next transaction with the same txid, and the case in which a *Probe* message and a voluntary *Release* message cross in the channel.

C. The interface specification

Now we would like to give a formal specification of the TileLink interface protocol. The primary purpose of a cache coherence protocol is to coordinate the serialization of memory events. Thus, our specification will define not only the correct histories of events at the interface, but also the *relation* between these interface events and serialization events.

The specification consists of a collection of temporal properties and assume-guarantee relationships between the properties. Here, we will use the shorthand $C: \ \phi \rightarrow \psi$ to mean that component C guarantees always ψ assuming always ϕ. This could also be read as "the proof that ϕ implies ψ is localized to component C". The specification is in the "circular" assume-guarantee style [12]. That is, we prove a collection of temporal properties by mutual induction over time. We will write $\phi^{-} \rightarrow \psi$ to mean that, always, if ϕ held always in the strict past, then ψ holds now. An example of a valid "circular" inference would be the following:

$$\frac{\begin{array}{c} C_1 : \ \phi^{-} \rightarrow \psi \\ C_2 : \ \psi^{-} \rightarrow \phi \end{array}}{C_1, C_2 : \ \text{true} \rightarrow (\phi \wedge \psi)}$$

That is, we prove locally that neither ϕ nor ψ is the first property to fail, thus both are always true.

Since the protocol is too complex to formally specify here, we will simply state a few representative properties informally in English. The most basic properties are the ones that define the protocol rules, without consideration of semantics. For example:

Prop. $C[0]$: A new cached Acquire *may not have the same block address and privileges as a pending* Acquire.

Prop. $M[0]$*: A new* Grant *may not acknowledge a* Rel *if there is a pending* Acquire *for the same cache block which at least one* Grant *has been issued.*

Here, we use $C[i]$ for guarantees of the client side of interface and $M[i]$ for guarantees of the manager side. In full specification there are approximately 40 properties s as these that relate purely to the interface protocol.

A further group of properties relates events at the inter to the reference model, in order to ensure consistency. example, for cached operations, we have:

Prop. $M[1]$*: If a cached* Grant *has data for address* a *it must match* reference.mem(addr).

Prop. $C[1]$*: If a* Release *has data for address* addr*, it must match* reference.mem(addr).

That is, data exchanged in cached transactions must always be up-to-date with respect to all serialized operations. For uncached clients, an *Acquire* is really a request for the manager to perform an operation. We have, for example

Prop. $C[2]$*: If an uncached* Acquire *is issued for operation* lt*, then* lt *has not been serialized, and every operations that happens-before* lt *has previously been requested.*

This property represents a key design decision of TileLink: uncached clients must take charge of maintaining the happens-before order at the interface. In practice, since channels may re-order requests, this means that the client must wait for the *Grant* of any operations that happen-before an operation before requesting it. Also, notice that this property refers to the local time *lt* of an *Acquire*. This is a "ghost" parameter of the method that is added to aid in the specification.

Prop. $M[2]$*: If an uncached* Grant *is issued for operation* lt*, then* reference.evs(lt).serialized *is true.*

That is, a *Grant* indicates that the manager has serialized the requested event. We also require data correctness for these operations:

Prop. $C[3]$*: If an uncached* Acquire *is issued for operation* lt *with data, then the data match* reference.evs(lt).

Prop. $M[3]$*: If an uncached* Grant *is issued for operation* lt *with data, then the data match* reference.evs(lt).

There are approximately 20 properties of this type, relating interface events to the reference object.

Finally, we have properties relating interface events to serialization events. At its core, the purpose of the protocol is to coordinate serializations among the system components. This is the function of the cache permission states, *invalid*, *shared* and *exclusive*. The interface specification records the permission state for each cache block, based on the history of *Grant* and *Release* operations. The permission state reflects the capabilities of all the components on the client side of the interface. In particular, we have:

Prop. $SC[0]$*: Components on the client side of the interface may serialize read operations only if the permission state is* shared *or* exclusive *and write operations only if the permission state is* exclusive.

Prop. $SM[0]$*: Components on the manager side of the interface may serialize read operations only if the permission state is* shared *or* invalid *and write operations only if the permission state is* invalid.

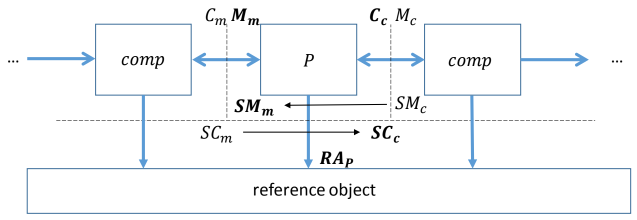


Fig. 4. Assume-guarantee flow for TileLink. Boldface indicates guarantees for P. Arrows indicate "non-strict" dependency.

Recall that the *serialize* method has an additional parameter giving the locale of the serialized component. This is what allows us to state properties $SC[0]$ and $SM[0]$. The locales are partially ordered in a way that respects the hierarchy, with lower locales on the client side and higher locales on the manager side. The client-side components have locales lesser than the interface and manager-side components have locales greater. The locale parameter of *serialize* allows us to determine which side the call is from. This still leaves a question: how do we localize the proof of these properties, since they seem to depend on many components? We will deal with this problem shortly.

For uncached operations, the situation is slightly different. A pending *Acquire* requesting an operation *lt* gives permission to the manager side to serialize just the one operation *lt*.

D. Assume-guarantee

With the properties specified, we must now give the assume-guarantee relationships. For interface i in the system, we'll use C_i, M_i, SC_i and SM_i to represent the four groups of properties at that interface (that is, client guarantees, manager guarantees, client-side serialization guarantees and manager-side serialization guarantees). Now consider a component P with one client interface c and one manager interface m (for example, P could be an L2 cache module). This situation is depicted in Fig. 4. The localizations for component P are as follows:

$$\begin{array}{ll} P, R: & C_m^-, M_c^- \rightarrow C_c, M_m \\ P, R: & SC_m, C_m^-, M_c^- \rightarrow SC_c \\ P, R: & SM_c, C_m^-, M_c^- \rightarrow SM_m \\ P, R: & C_m^-, M_c^-, SM_c^-, SC_m^- \rightarrow RA_P \end{array}$$

Here R is the reference object. Each localization contains one system component and the reference object. The first says that to prove the protocol guarantees of P, we assume the protocol guarantees of its neighbors in the strict past. In other words, we show that component P is not the first to violate the protocol. The second and third localizations show how we deal with the collective semantic properties. This is essentially by induction over the hierarchy. In the second line, we are showing that all the serializations performed on the client side of interface c are correct with respect to c. We get to assume that those on the client side of m are correct with respect to m. Component P must then guarantee that they are also correct with respect to c (among other things, this means that the permissions at m must be a subset of the permissions at c). If it can further show that its own serializations are correct, then we have SC_c. Notice here that we are assuming SC_m to hold at the current time and not just in the strict past. This is essential, since P can only guarantee properties of its own outputs. The third line gives a similar argument for the manager-side serialization properties. The main difference is that this proof flows from the manager-side to the client side.

Finally the last line shows that the reference object assertions hold. Here RA_P means that these assertions are true when *serialize* is called by P. If this is true for all components P, the reference object assertions hold always (this is called a "temporal case split" in [13]).

When we instantiate the above assume-guarantee specifications for all the components in the hierarchy, they should form a valid assume-guarantee proof. In other words there should be no dependency cycles that do not contain a strict past operator. This is easy to check algorithmically for arbitrarily large hierarchies (thus the specification is *scalable*). Intuitively, the proof is valid because the non-strict dependency chains flow only from left to right or right to left, and therefore cannot form cycles. All such paths terminate either in processors (which have only client ports) or memories (which have only manager ports).

The key point about this specification idiom is how it deals with the collective semantic properties SC and SM. The proofs of these are localized by a combination of induction over the hierarchy and circular compositional reasoning. In this way, we arrive at localized specifications which together imply global semantic correctness using a scalable machine-checkable argument.

E. Specifications in Ivy

The TileLink specification described above was written in the Ivy language [14]. Ivy supports modular assume-guarantee specifications in a flexible way that allows us to apply our specification idiom and to check the assume-guarantee proof.

Properties in Ivy are expressed not as temporal logic formulas but as monitors. A monitor is an object with internal state that can synchronize its actions with method calls of other objects. For example, here is one way to write a monitor that reflects property $SC[0]$:

```
module sc0(intf) = {
    before reference.serialize(lt:ltime,loc:locale) {
        var state := intf.state(block(reference.evs(lt).addr))
        var op := reference.evs(lt).op
        assert loc < intf.loc ->
            (op = read -> (state = shared | state = exclusive)) &
            (op = write -> state = exclusive)
    }
}
```

Here, the monitor specifies an action that is to be executed before every call to *reference.serialize*. The assertion effectively specifies a temporal safety property.

Monitor objects can contain state variables that store history information. For example, the TileLink interface specification stores in history variables information about pending transactions across the interface. This makes it possible to specify properties such as $C[0]$ that depend on the history of the interface. Ivy can verify that monitors are non-interfering, that is, their actions always terminate, and do not modify the state of the object specified.

Assume-guarantee relationships are mostly implicit in Ivy. That is, usually an assertion made as a pre-condition to a method will be a guarantee of the caller, while a post-condition will be a guarantee of the callee. In some case, however, an explicit declaration is required. Consider, for example, the properties SC_m and SC_c in Fig. 4. These are both pre-conditions to method *reference.serialize*, relating it respectively to the manager and client interfaces of component P. When proving SC_c, we need to assume that SC_m holds for the same call. In Ivy, we can specify that the monitor action for SC_c be scheduled *before* the monitor action for SC_m. This, in effect, gives the desired assume-guarantee relationship.

Given a configuration of components with their respective assume-guarantee specifications, Ivy can check the validity of the over-all proof and also allows us to locally verify that each component meets its specification.

F. Component proofs

With the TileLink specification defined, we can formally verify that some simple abstract component designs meet their specifications. The primary motivation for this is to validate the specification itself. Without verifying components, it would be quite easy to write a specification that is unrealizable, or rules out certain intended implementations or optimizations.

For example, a key objective of the protocol design is to tolerate communication over unordered channels. To test this, we built a model of an unordered channel that implements the client role on one side and the manager role on the other. Actions initiated on one side are delayed arbitrarily before being reproduced on the other. We can then attempt to verify that this model actually satisfies the component specification. This means that we can insert a reordering channel at any point in the hierarchy without affecting correctness.

This proof was carried out using the Ivy tool, first with bounded model checking, then by constructing an inductive invariant interactively [15]. The proof was done for arbitrary address space, data word and cache line size, as well as unbounded message buffers. The Ivy language guarantees that all the verification conditions, for both bounded checking and inductive invariant checking remain in a decidable fragment of first-order logic called the Bernays-Schönfinkel fragment, or EPR. This made it possible to check proofs about an infinite-state system reliably, without relying on fragile quantifier instantiation heuristics.

Proving this one simple component uncovered many errors in the formal specification itself (in fact, 25 were discovered after bounded model checking). It should be noted that these were errors in the *formal* specification. It is difficult to tell to what extent this specification corresponds to the designer's informally stated intentions.

The unbounded proof required the generation of a fairly large number of auxiliary invariants. This was done interactively, by considering graphically represented counterexamples to induction. About 30 of these formulas are generic invariants of the interface specification. Here are a few, stated informally:

Inv. I[0]: A voluntary Release *cannot be pending while an cached* Acquire *is awaiting a* Grant *for the same block.*

Inv. I[1]: While a Release *is pending, the interface state of its cache block is* invalid*.*

Inv. I[2]: If an uncached Acquire *has received a* Grant*, its operation has been serialized.*

These invariants are stated as universally quantified first-order formulas in terms of the state variables of the interface specification.

In addition, about 100 invariant formulas were needed relating the buffer state to the state of its two interfaces. Here are a few representative examples:

Inv. B[0]: For any cache block, the privileges on the manager interface are a subset of the privileges on the client interface.

Inv. B[1]: Every Acquire *pending on the client interface is pending on the manager interface.*

Inv. B[2]: Two Acquires *for the same address cannot exist in the buffer.*

Some of these properties, such as $B[0]$, are generic, while others are specific to the unordered buffer. By proving the the buffer model satisfies its specification, we show that in fact the protocol as formally specified is resilient to message reordering.

In a similar way, we proved that a generic hierarchical cache (with one manager and multiple client ports) and a generic directory (a backing store with multiple client ports) and a generic CPU (with one manager port) satisfy the protocol specification. These proofs also required interactive invariant construction, with an additional approximately 50 invariant formulas.

Having verified these components, we can now construct and prove arbitrary hierarchies built from these components with no additional complexity or manual effort. Showing correctness of systems of generic components was an important step in developing the specification, clarifying many issues in the informal protocol definition.

G. Modular specification-based testing

A modular specification for TileLink, once established, can be used as a reference by designers of components. One important aspect of this is testing. The actual RTL-level TileLink components, such as the L2 cache bank or the snooping hub, are complex designs. Formally verifying them against their specification would be a substantial undertaking requiring significant expertise in formal methods. On the other hand, the formal specification can be used to rigorously test the components, for example, using constrained random testing methods. This can be accomplished by engineers without formal methods expertise.

The modular specification allows us to automatically produce testers that both generate legal inputs for the design and act as an oracle to validate the design outputs. While formal specifications have been used for testing before [4] what is new here is modularity. That is, because we have a proof that the component specifications imply system correctness, we know that any system error must be reflect in a specification violation by some component. Thus, all system-level errors are exposed to component-level testing.

To apply this idea, we built a tool that can extract from the Ivy specification a constrained random test generator and a test oracle (both in C++) for TileLink components. The test generator uses the Z3 SMT solver [7] to generate inputs for the component that are legal relative to the current interface state, and the oracle checks correctness of the outputs. This allows us to unit-test RTL designs in simulation against the formal specification, rather than using an ad-hoc test bench. Perhaps not surprisingly, this revealed a number of ways in which the formal specification diverged from the designer's intention. It also revealed a number of *latent bugs* in the component designs. These are protocol violations that cannot be stimulated in integration tests of the current system, but may crop up later when the component is integrated in other systems.

As an example, the protocol allows the *Release* of a cache block to occur before the final *Finish* message for the *Acquire* has arrived. This is unavoidable, since the *Release* may bypass the *Finish* in a channel. The release has to be handled to avoid potential resource deadlocks. Specification-based testing of the L2 cache bank design revealed that the *Release* was not handled. A subsequent redesign assigned the task of handling the unexpected *Release* the to state machine handling the *Acquire*. Testing again revealed cases in which the *Release* was handled improperly. These errors and others that were discovered did not arise in integration testing (simulating instruction sequences executing on the processor cores) because the existing L1 cache design could not produce the timing that stimulated the errors in the L2 cache. Therefore, these bugs would have waited to be discovered in some future application of the L2 cache design, perhaps after fabrication. Using modular compositional testing, these errors were discover for the most part within the first 100 clock cycles of testing.

This illustrates a general advantage of unit testing. That is, with direct control of the inputs signal of a component, it is much easier to stimulate corner-case behaviors. The disadvantage of unit testing is that it is biased by the designer of the unit tests and his or her understanding (or misunderstanding) of the interface specification. Passing informal unit tests thus give no direct evidence of system correctness. Using a modular formal specification removes this bias, allowing us to gain confidence in system correctness from unit testing.

III. Related work

Many prior works have addressed verification of parameterized cache coherence protocols. An early effort was [11] which includes a parameterized proof using network invariants. The present approach can be seen to have some similarities to network invariants. That is, the collective semantics properties are proved in effect by induction over the hierarchy. There is considerably more flexibility here, however, as induction in two directions is combined with circular assume-guarantee (that is, mutual induction over time). Moreover, the finite-state approach in [11] was not able to prove memory consistency.

This issue can be seen in other work on cache coherence verification. For example, some recent work exploits compositionality to reduce the complexity of hierarchical cache protocol verification [2], [3]. These methods can prove protocol properties such as coherence, but cannot prove *semantic* properties such as weak consistency, in part because they are based on finite-state methods, and in part because they lack the ability to specify *relative* to a reference object. A key enabler for the present methodology is the ability to do assume-guarantee reasoning with arbitrary temporal properties relating various components of a system. A strictly hierarchical approach to compositional verification does not allow this.

Recent work uses Coq to specify the behavior of a memory hierarchy [16]. This is at a much higher level of abstraction, however, and does not provide a modular specification for RTL-level components such as caches, hubs and directories. Moreover, the approach we take here allows us to apply a much higher level of automation, using an SMT solver to check the localized verification conditions.

It is also interesting to consider to what extent existing tools and compositional frameworks would support the specification idiom applied here. We can divide these roughly into two categories we might call *procedure-modular* and *process-modular*. The former are based on Hoare logic and include systems such as Dafny [9] and VCC [5]. They allow us to abstract a procedure by a logical expression of its transition relation. Because there is no direct support for temporal assume-guarantee reasoning, it would be challenging to apply the methodology described here (though perhaps not impossible, for example by encoding the necessary temporal reasoning into the system's logic as in [8]). In a process-modular system, the basic abstraction is the process, which communicates with other processes by signals, messages or shared variables. A good example is the Reactive Modules framework of Alur and Henzinger *et al.* [1]. This supports a hierarchical approach, which, as noted above, lacks the flexibility to express the relational specifications used here. Cadence SMV [12] would support our general specification pattern. However, it lacks procedural abstraction. In developing the TileLink specification, procedures were found to be a convenient and powerful tool for structuring the specification. As an example, it is easy in Ivy to describe a component that serializes two memory operations in a single atomic action, but this would be quite awkward in Cadence SMV. It is not clear how difficult it would be to do the component proofs in Cadence SMV. It is possible that model checking and abstraction could be applied to reduce some of the manual effort (approximately one person week to produce the invariants).

IV. Conclusion

The TileLink case study illustrates the advantages of a modular specification, both as a design artifact and as a proof construct. However, TileLink is also an example of a class of systems that performs an abstract computation in a distributed way. It is difficult to localize the proof of such systems because the *witness* for their correctness (in this case a serialization of memory operations) cannot be constructed locally. To solve this problem, we introduced a specification idiom. In this idiom, a reference object takes the role of constructing the witness, based on input from all of the system components. The required collective properties of the system were proved in an inductive manner, using "circular" assume-guarantee reasoning. The result is a local specification for each component, in terms of assume-guarantee relations. The local specifications collectively form a global proof of semantic correctness that can be checked in a scalable way. It seems plausible that this paradigm might be applied to other similar systems, such as distributed file systems, replication protocols and so on.

The modular specification, once obtained, can be used for a variety of purposes. It can serve as a reference for designers, and can be used for modular specification-based testing, to reveal latent design errors that cannot be stimulated in integration testing. The hope is that such a methodology will result in more robust re-usable components that can be rapidly and reliably assembled into systems-on-chip. Ultimately, we can use the specification to formally verify the implemented components, if this effort is deemed justifiable.

The specification was constructed using a tool called Ivy that supports the necessary forms of compositional reasoning, and allows bounded and unbounded proof using decidable logics. The tool also provides for generation of constrained-random test benches from formal specifications. Work is in progress on an extension of the system to handle liveness, both in formal proofs and in a testing scenario (a preliminary liveness specification for TileLink has also been developed).

A larger goal of this work is to find ways to use formal methods to benefit engineers that may lack the skills or resources to develop formal specifications themselves. One way to do this is to provide specifications of common protocols that are formally vetted and can be used in rigorous testing.

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] Xiaofang Chen. *Verification of Hierarchical Cache Coherence Protocols for Futuristic Processors*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 2008. AAI3322423.

[3] Xiaofang Chen, Yu Yang, Michael Delisi, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 107–114. IEEE, 2007.

[4] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[5] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[6] Henry Cook. Productive design of extensible on-chip memory hierarchies. Technical Report UCB/EECS-2016-89, EECS Department, University of California, Berkeley, May 2016.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[8] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[9] K. Rustan M. Leino. Developing verified programs with Dafny. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1488–1490. IEEE / ACM, 2013.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, , and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25:63–79, March 1992.

[11] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[12] Kenneth L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 342–345. Springer, 1999.

[13] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[14] Oded Padon and Kenneth L. McMillan. Microsoft Ivy. https://github.com/Microsoft/ivy. TileLink files are found in examples/tilelink.

[15] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *PLDI*, 2016. To appear.

[16] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2015.

[17] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume I: User-level ISA version 2.0. Technical Report UCB/EECS-2014-52, EECS Department, University of California, Berkeley, May 2014.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Proof Certificates for SMT-based Model Checkers for Infinite-state Systems

Alain Mebsout Cesare Tinelli
The University of Iowa, Iowa City, IA, USA

***Abstract*—We present a dual technique for generating and verifying proof certificates in SMT-based model checkers, focusing on proofs of invariant properties. Certificates for two major model checking algorithms are extracted as k-inductive invariants, minimized and then reduced to a formal proof term with the help of an independent proof-producing SMT solver. SMT-based model checkers typically translate input problems into an internal first-order logic representation. In our approach, the correctness of translation from the model checker's input to the internal representation is verified in a lightweight manner by proving the observational equivalence between the results of two independent translations. This second proof is done by the model checker itself and generates in turn its own proof certificate. Our experimental evaluation show that, at the price of minimal instrumentation in the model checker, the approach allows one to efficiently generate and verify proof certificates for non-trivial transition systems and invariance queries.**

I. Introduction

Model checkers are perhaps among the most successful formal methods tools in term of industrial use, particularly for the development of safety-critical systems. In addition to traditional applications in hardware design, they are increasingly used in model-based software development to analyze, for instance, models of embedded systems in the aerospace or automotive industry. One clear strength of model checkers, as opposed to proof assistants, say, is their ability to return precise *error traces* witnessing the violation of a given safety property. In addition to being invaluable to help identify and correct bugs, error traces also represent a checkable unsafety certificate. In contrast, most model checkers are currently unable to return any form of corroborating evidence when they declare a safety property to be satisfied by a system under analysis. This is unsatisfactory in general since model checker are complex tools, based on a variety of sophisticated algorithms and search heuristics, and so are not immune to errors.

To mitigate this problem, a possible approach is to use a model checker whose correctness has been formally verified [10]. An alternative is to instrument the model checker so that it is *certifying*, *i.e.* it accompanies its safety claims with a *proof certificate*, an artifact embodying a proof of the claim [16]. The certificate can then be validated by a trusted *certificate checker*. While the former approach may seem better at first, based on the fact that the model checker is verified once and for all, it has a number of disadvantages. To start, the effort is normally enormous since there are no general frameworks for verifying modern model checkers. Moreover, any modifications to the originally verified tool requires proofs to be redone. In more extreme cases (*e.g.*, an in-depth modification) one may have to invest the same amount of effort as for the original correctness proof. The main advantage of the second approach is that it requires a much smaller human effort. A disadvantage of course is that every safety claim made by the model checker incurs the cost of generating and then checking the corresponding certificate. This is feasible in general only if such certificates are small and/or simple enough to be checkable by a target certificate checker in a reasonable amount of time (say, with at most an order of magnitude slowdown).

By reducing the trusted core to the certificate checker, certifying model checking facilitates the integration of formal method tools into safety critical processes such as those endorsed by the DO-178C guidelines for avionics software. In the spirit of the *de Bruijn criterion* [4], traditionally applied to theorem provers, it redirects tool qualification requirements from a complex tool, the model checker, to a much simpler one, the proof checker.

We present an approach for generating and verifying proof certificates for SMT-based model checkers. These tools use a variety of model checking techniques and some of them even employ a portfolio approach by running several engines in parallel. Input models are typically represented internally as transition systems encoded in some fragment of first-order logic. Safety properties are expressed as invariant properties and reasoning about invariance is reduced to checking the satisfiability of formulas in certain logical theories such as integer or real linear arithmetic. The latter problem is then delegated to off-the-shelf SMT solvers.

We describe how to generate intermediate certificates that show that a given safety property is satisfied the internal transition system. These certificates are designed to be checkable by an SMT solver. Since SMT solvers themselves are complex artifacts, we also show how to reduce the validity of these certificates to proof objects obtained by a proof-producing SMT solver. This reduction capitalizes specifically on the recent proof production capabilities of the SMT solver CVC4 [5] and the availability of an efficient proof checker for its proofs, which are generated in LFSC format [24]. Most model checkers do allow users to specify system models directly in this relatively low-level logical representation. Instead, they support some pre-existing modeling language (such as Simulink, Lustre, Promela, SMV, or even just C). To account for possible problems in the translation from the input modeling language to the

This work is partially supported by the National Aeronautics and Space Administration under NASA contract number NNL14AA06C.

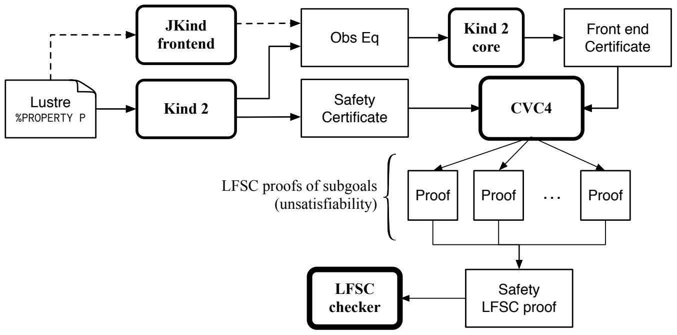


Figure 1: Process for proof certificates generation and verification in Kind 2.

internal logical representation, we include a second phase which produces an additional proof certificate providing some level of confidence in the correctness of the translation.

While the techniques we have developed are general enough to be applicable to arbitrary SMT-based model checkers, we have implemented them in a specific one: Kind 2 [7], an SMT-based, multi-engine, symbolic model checker that can prove or disprove safety properties of synchronous reactive systems expressed in the Lustre language [11]. As a consequence, we will describe our work in terms of Lustre and Kind 2, but with the assumption that a knowledgeable reader will be able to see how it generalizes to other SMT-based model checkers. In more detailS, this work contains the following specific contributions:

(1) *A technique for generating proof certificates for safety properties of transition systems*. We show how to extract and simplify k-inductive invariants that are sufficient to summarize proofs generated by the different kinds of SMT-based model checking methods (in Section II) and how proofs can be reconstructed (in Section IV).

(2) *An approach to increase trust in the translation from the external modeling language to an internal representation language*, described in Section III. A translation certificate is generated in the form of observational equivalence between two internal representations generated by independently developed front ends. Their equivalence is recast as an invariant property; checking that yields itself a second proof certificate from which a global notion of safety can be derived and incorporated in the LFSC proof. We improve on similar previous approaches [19], [20] by adopting a weaker, property-based notion of observational equivalence, which is enough for our purposes.

(3) *An implementation of these techniques in Kind 2*. The first certificate summarizes the work of its different engines: bounded model checking (BMC), k-induction, IC3, as well as additional invariant generation strategies. The certification of the translation is applied to the Lustre language. The intermediate certificates are SMT-LIB 2 scripts checked by CVC4. CVC4's own proof objects are used to construct an LFSC proof term providing an overall proof of safety.

The full certification process for Kind 2 is depicted in Figure 1. Kind 2 generates two sorts of safety certificates, in the form of SMT-LIB 2 scripts: one certifying the faithfulness of the translation from the Lustre input model to the internal encoding,

```
node add_two (a, b : real) returns (c : real) ;
  var v : real;
let
  v = a + b ;

  c = 1.0 -> if (pre c) > v then (pre c) else v ;
  --%PROPERTY (a > 0.0 and b > 0.0) => c > 0.0 ;
tel
```

Figure 2: Lustre model of running example.

and one certifying the invariance of the input properties for the internal encoding of the input system. These certificates are checked by CVC4, then turned into LFSC proof objects by collecting CVC4's own proofs and assembling them to form an overall proof that can be efficiently verified by the LFSC proof checker. Our initial experimental evaluation indicates that, at the price of minimal instrumentation in the model checker, this approach allows one to efficiently generate and check proofs for non-trivial transition systems and invariance queries.

To illustrate our different techniques, we will rely on the toy model in Figure 2. In Lustre, reactive components are modeled as *nodes*. The node `add_two` in the figure encodes a component that initially outputs 1.0, in variable `c`, and at each execution step afterwards outputs the maximum between the previous value of `c` and the sum of the current values of input variables `a` and `b`. The model is annotated with an invariance property stating that the output `c` is positive whenever both inputs are.

A. *Technical Preliminaries*

We define a *transition system* as a tuple $\mathcal{S} = (\mathbf{x}, I, T)$ where $\mathbf{x}$ is a tuple of distinct (typed) variables; I is a formula of typed first-order logic with free variables from $\mathbf{x}$, which characterizes the initial states of the system; and T is a formula with free variables from $\mathbf{x}$ and a renamed copy $\mathbf{x}'$ of $\mathbf{x}$, which describes the system's transition relation. If F is a formula with free variables from $\mathbf{x}$, we write $F[\mathbf{y}]$ to denote the instance of F obtained by replacing its free variables by the corresponding ones in $\mathbf{y}$. We write $T[\mathbf{y}, \mathbf{y}']$ similarly for T. We adopt the usual notions and notations of first-order logic. In particular, for an intepretation $\mathcal{M}$ and a formula φ, we write $\mathcal{M} \models \varphi$ to mean $\mathcal{M}$ satisfies the formula φ. We also write $\models$ for the logical entailment in a theory (such as integer and real arithmetic) that encodes the data types used in the transition system. A *state* of the system $\mathcal{S} = (\mathbf{x}, I, T)$ is a model that gives an interpretation to the variables of $\mathbf{x}$. A state $\mathcal{M}$ of a system $\mathcal{S} = (\mathbf{x}, I, T)$ is said to be *reachable* iff there exists an $i \in \mathbb{N}$ such that, $\mathcal{M} \models \exists \mathbf{x}_0 \ldots \mathbf{x}_{i-1}.\ I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-1}, \mathbf{x}_i]$. *State properties* for a system $\mathcal{S}$ are described by first-order formulas whose free variables are from $\mathbf{x}$. Let P be a state property for $\mathcal{S} = (\mathbf{x}, I, T)$. P *holds* in, or is an *invariant* of, $\mathcal{S}$ if every reachable state $\mathcal{M}$ of $\mathcal{S}$ is a model of P. Property P is k-inductive for some $k > 0$ if (*i*) $I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-2}, \mathbf{x}_{i-1}] \models P[\mathbf{x}_{i-1}]$ for all $i = 1, \ldots, k$, and (*ii*) $T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge P[\mathbf{x}_0] \wedge \ldots \wedge P[\mathbf{x}_{k-1}] \models P[\mathbf{x}_k]$. A *$k$-inductive strengthening* Q of P is a k-inductive formula $Q[\mathbf{x}]$ such that $Q[\mathbf{x}] \models P[\mathbf{x}]$. One can show that k-inductive state properties are invariant. It follows that every state property having a k-inductive strengthening is invariant.

II. k-inductive Safety Certificates

In this section, we focus on transition systems and present a certificate generation approach general enough to capture the information produced by different SMT-based model checking engines while proving invariance properties of a system $\mathcal{S} = (\mathbf{x}, I, T)$. We show that k-inductive strengthenings of original properties are an adequate summary of the reasoning resulting from the combination of these engines. We also show how to combine and simplify them with the aim of generating the most easily verifiable objects.

A. Extracting and Verifying Certificates

Kind 2 converts internally input models and properties, expressed in Lustre, to a transition system that captures the same input/output behavior. The translation is relatively straightforward for single-node models, and is based on having state variables corresponding to the node's input and output variables as well as any terms of the form `pre t`.[1] For multi-node models, the transition systems for the individual nodes are combined according to Lustre's synchronous parallel composition semantics.

Certificate extraction. In Kind 2, an input property P can be proved invariant by one of two main model checking methods: k-induction [22] and IC3 [6], each implemented in an independent engine. The job of either engine is facilitated by a number of auxiliary invariant generation engines, which discover and pass along auxiliary invariants that might be helpful in proving the main property. Often these are local invariants, for instance specific to a sub-component of the input system. All of these engines, which run concurrently, generate *safety certificates* of the form (k, ϕ) where k is a positive number and ϕ is a k-inductive strengthening of some state property. The content of the certificate depends on the engine:

- The k-induction engine tries to prove that the input property P is invariant by proving that it is k-inductive for some $k > 0$. When this succeeds, P is its own k-inductive strengthening and a possible certificate is the pair (k, P).
- The IC3 engine also tries to prove that an input property P is invariant. It succeeds when it is able to construct a conjunction ϕ of formulas such that $\phi \wedge P$ is 1-inductive. In this case, a possible certificate is $(1, \phi \wedge P)$.
- The invariant generation engines are based on variations of the previous techniques. Every auxiliary invariant used in the proof of an input property P is provided with its own certificate, also of the form (k, ϕ).

Certificate combination. Kind 2 accepts as input multiple properties for a given model, and attempts to verify them individually. This means that it normally produces individual certificates for a collection of user-specified and internally generated properties. These safety certificates are combined together thanks to the following easily provable result.

Proposition 1. *If (k_i, ϕ_i) is a k_i-inductive strengthening of property $P_i[\mathbf{x}]$ for $i = 1, 2$, then $(k, \phi_1 \wedge \phi_2)$ with $k = max(k_1, k_2)$ is a k-inductive strengthening of $P_1[\mathbf{x}] \wedge P_2[\mathbf{x}]$.*

Verifying Certificates. Checking a (combined) certificate (k, ϕ) for a (conjunctive) property P reduces to verifying that ϕ is indeed a k-inductive strengthening of P. This can be done using any tool that can prove the following entailments:

$$I[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{i-2}, \mathbf{x}_{i-1}] \models \phi[\mathbf{x}_{i-1}] \text{ for } i \in [1, k] \quad (base_k)$$
$$T[\mathbf{x}_0, \mathbf{x}_1] \wedge \ldots \wedge T[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge \phi[\mathbf{x}_0] \wedge \ldots \wedge \phi[\mathbf{x}_{k-1}] \models \phi[\mathbf{x}_k] \quad (step_k)$$
$$\phi[\mathbf{x}] \models P[\mathbf{x}] \quad (implication)$$

Using an SMT solver to prove $(base_k)$, $(step_k)$, and $(implication)$, effectively moves the burden of trust from the model checker to the solver. As we describe in Section IV, the latter can in turn be removed from the trusted core if it can provide an LFSC proof of the three entailments.

B. Simplifying Certificates

Good certificates need to be simple and easily checkable by an independent tool or method. In particular, there is an expectation that checking a certificate should not take more time than proving the original property. A common approach in the certificate production literature is to simplify and/or reduce the certificate *a posteriori* [2], [8], [25]. This extra effort at construction time can pay large dividends at checking time. In our case, a safety certificate (k, ϕ) can be simplified by reducing the value of k or the size/complexity of ϕ, or both. Currently, Kind 2 tries to reduce k before simplifying ϕ. Empirical evaluation, discussed in Section V, suggests that this sort of post-processing is always worth the overhead.

Reducing k. Referring back to the entailments $(base_k)$ and $(step_k)$ from the previous section, because of the k checks in $(base_k)$, checking a certificate (k, ϕ) requires a number of sub-checks proportional to k. Each of sub-checks in turn take time proportional to k, making the whole process quadratic in k. Due to the concurrent nature of Kind 2, proofs obtained by its k-induction engines are not guaranteed to have a minimal k. Consequently, lowering k can often be the most effective way of simplifying a certificate. To do that, after it constructs an initial combined certificate (k, ϕ), Kind 2 will replay the inductive step $(step_k)$ for ϕ for values k' smaller than k, following one of three different strategies, chosen heuristically:

- *forward enumeration*: progressively try all values of k' from 1 to k and stop at the first where k'-inductiveness holds;
- *backward enumeration*: try values of k' from k down to 1, stopping as soon as k'-inductiveness is lost;
- *binary search:* partition $[1, k]$ into subintervals $[1, k']$ and $[k' + 1, k]$ of similar size and recursively consider the first or the second interval depending on whether φ is k'-inductive or not.

Simplifying ϕ. Because of how combined certificates (k, ϕ) are generated, the invariant ϕ, which is a conjunction $\psi_1 \wedge \ldots \wedge \psi_n$

[1]For each non-initial execution step, `pre t` denotes the value of `t` in the previous step.

Algorithm 1. Two-phase simplification of invariants

```
Input: $R = \{\psi_1, \dots, \psi_n\}$: invariant set to be reduced,
       $P$: input property set,  $T$: Transition relation

Function trim($R$, $P$)
    if $R(0..k-1) \wedge P(0..k-1) \wedge T(0..k) \models P(k)$ then
        // $P$ is $k$-inductive wrt $R$
        $U$ = get-unsat-core();
        $R' = \{\psi \in R \mid \psi$ occurs in $U\}$;
        if $R'(0..k-1) \wedge P(0..k-1) \wedge T(0..k) \models R'(k) \wedge P(k)$ then
            // $R' \wedge P$ is $k$-inductive
            return $R' \cup P$
        else // $R'$ is not strong enough
            trim($R \setminus R'$, $R' \cup P$)
    else error "Not $k$-inductive";

Function cherry-pick($R$, $P$)
    if $P(0..k-1) \wedge T(0..k) \models P(k)$ then
        // $P$ is $k$-inductive
        return $P$
    else
        // Find cex to induction
        $\mathcal{M}$ = get-cex();
        // ... and a blocking invariant
        $\psi$ = choose($\{\psi \in R \mid \mathcal{M} \not\models \psi\}$);
        cherry-pick($R \setminus \{\psi\}$, $P \cup \{\psi\}$)

cherry-pick( trim($\{\psi_1, \dots, \psi_n\}$, $P$),  $P$ );
```

of formulas, can contain unnecessary information (redundancy, useless auxiliary invariants, etc.). We tighten ϕ with a process based on two fixpoint computations applied in sequence and described in Algorithm 1. There, we use the notation $\varphi(i)$, $\varphi(0..i)$ and $T(0..i)$ as an abbreviation, respectively, of $\varphi[\mathbf{x}_i]$, $\varphi[\mathbf{x}_0] \wedge \cdots \wedge f[\mathbf{x}_i]$ and $T[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge T[\mathbf{x}_{i-1}, \mathbf{x}_i]$. Also, we treat finite sets of formulas as the conjunction of their elements. For entailment checks, we assume the availability of a function `get-unsat-core` that returns an unsatisfiable core of the premises and the negated conclusion of the entailment when the entailment holds, and a function `get-cex` that returns a counterexample when the entailment does not hold. Both of these functionalities are provided by most SMT solvers.

Algorithm 1 uses two functions, `trim` and `cherry-pick`, both of which take a set P of properties and a set R of auxiliary invariants for P. Function `trim` aims at identifying and removing from R invariants that are not needed to prove P k-inductive. It relies on unsat cores to progressively reduce the set R as long as $R \cup P$ remains k-inductive. Function `cherry-pick` recursively checks that P is k-inductive and, if it is not, adds to it any of the auxiliary invariants from R that eliminate the k-induction counter-example found by the SMT solver. One can prove that each function, and so the whole process, is terminating—the main point being that the input set R is finite and gets strictly smaller with each recursive call. The process is also sound in the sense that its returned formula is a k-inductive strengthening of P whenever the input $\phi = \psi_1 \wedge \cdots \wedge \psi_n \wedge P$ is. However, it is not guaranteed to yield the smallest k-inductive strengthening of P contained. This is intentional, for practical efficiency.

Practical considerations. In principle, applying `trim` is computationally expensive because of the cost of its entailment checks. In practice, it terminates after a very small number of iterations—generally less than three on our benchmarks. Moreover, it is very effective at removing large unnecessary parts of the certificate. Considering that certificates with hundreds of conjuncts are common, the cost of running `cherry-pick` on the original certificate can become prohibitive. In our experiments, it was always beneficial to apply the coarse reduction performed by `trim` before calling `cherry-pick`.

We observe that the effect of `trim` is similar to one of the reduction steps proposed by Irvii *et al.* [14] for invariants produced by SAT-based IC3-like model checkers. While potentially increasing precision, many of their other steps require a number of satisfiability checks linear in the size of ϕ, which is already prohibitive for the SMT case.

It could be useful to try to reduce k and simplify ϕ at the same time in the hope of getting closer to a minimal k than we do currently with our algorithm. This, however, would be more expensive, so further empirical evidences would be needed to assess the practical effectiveness of more sophisticated approaches in practice.

III. Front End Certification

The certificates discussed in the previous section are produced for Kind 2's internal FOL representation of the input system and properties. Although the translation to this internal representation from the Lustre input is fairly direct, Kind 2's front end also applies a number of optimizations and simplifications to the input, such as slicing, constant propagation, and so on. This raises the question of whether the front end can be trusted to be correct. We rule out the option of formally proving its correctness for the reason we gave in Section I. In alternative, we have the translation phase generate certificates of its own.

Comparing independent translations. Our goal is to keep the whole certification process lightweight and entirely automatic. As a consequence, instead of proving a semantic preservation between the input Lustre model and its internal representation as a transition system, we prove the *observational equivalence* of two internal representations obtained *independently* from the same input. This technique for certifying translations has already been employed in the SAT based toolchain of Prover Technologies [20] and in the Systerel Smart Solver [19]. In our case, instead of developing another front end for Kind 2 we can rely on a pre-existing third-party tool: JKind, a Lustre model checker inspired by Kind but developed independently at Rockwell Collins [21]. JKind too converts input models to an FOL representation. It is a good candidate because it is sufficiently different from Kind 2: it has a completely different code base (it is written in Java whereas Kind 2 is written in OCaml) and was developed independently by a different team. While our approach does not actually guarantee the correctness of the Kind 2 translation, it provides some formal evidence of its trustworthiness.

Our certificate encodes the claim that the transition relations constructed by the two independent front ends are behaviorally equivalent over a set of *relevant* state variables. In essence, the certificate consists of a transition system that observes the internal states of the two systems generated by each front end. This *observer* system feeds its two subsystems the same inputs and verifies that their externally visible behavior is the same.

For $i = 1, 2$, let $\mathcal{S}_i = (\mathbf{x}_i, I_i[\mathbf{x}_i], T_1[\mathbf{x}_i, \mathbf{x}'_i])$ be the internal transition system, and P_i the property, respectively generated

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

by JKind and Kind 2, with $\mathbf{x}_1$ and $\mathbf{x}_2$ sharing no components. We construct an observer system $\mathcal{S}_{\text{obs}}$ and a safety property $P_{\text{obs}} = (\mathcal{S}_1, P_1) \sim (\mathcal{S}_2, P_2)$ expressing a suitable notion of observational equivalence ($\sim$) between the two systems. Then we check the correctness of this observer in *the same way* as we would check the correctness of $\mathcal{S}_2$ with respect to the original safety property. This process is illustrated as part of Figure 1, where Obs Eq is the observer described below and the module **Kind 2 Core** is the core part of Kind 2, which works directly with the internal FOL representation of a transition system.

Observational equivalence. A standard definition of observational equivalence would require the two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ to produce the same outputs when given the same inputs at each step. This is, however, is unnecessarily stringent for our purposes and, depending on how different the two translations are, it might not even be the case. A better notion of equivalence is *property-based*: we consider $\mathcal{S}_1$ and $\mathcal{S}_2$ equivalent if, for the same input, they agree at each step on the truth value they assign to their respective version of the original input property in the Lustre model. For $j = 1, 2$, let $\mathbf{i}_j$ be the subtuple of $\mathbf{x}_j$ that corresponds to the input variables of the Lustre model. Then P_{obs} and $\mathcal{S}_{\text{obs}} = (\mathbf{x}_{\text{obs}}, I_{\text{obs}}, T_{\text{obs}})$ are defined as follows:

$$\begin{array}{ll} P_{\text{obs}} = (P_1[\mathbf{x}_1] \Leftrightarrow P_2[\mathbf{x}_2]) & I_{\text{obs}} = \mathbf{i}_1 \approx \mathbf{i}_2 \wedge I_1[\mathbf{x}_1] \wedge I_2[\mathbf{x}_2] \\ \mathbf{x}_{\text{obs}} = \mathbf{x}_1, \mathbf{x}_2 & T_{\text{obs}} = \mathbf{i}'_1 \approx \mathbf{i}'_2 \wedge T_1[\mathbf{x}_1, \mathbf{x}'_1] \wedge T_2[\mathbf{x}_2, \mathbf{x}'_2] \end{array}$$

where, for two tuples $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$, the expression $\mathbf{a} \approx \mathbf{b}$ denotes the formula $\bigwedge_{i=1,\ldots,n} a_i = b_i$. The set of state variables of the observer system $\mathcal{S}_{\text{obs}}$ is the (disjoint) union of the variables of $\mathcal{S}_1$ and $\mathcal{S}_2$. The system itself is effectively the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ after their corresponding input variables have been pairwise identified.

Front end certificates. To recap, the equivalence observer $\mathcal{S}_{\text{obs}}$ and the associated property P_{obs} constitute an intermediate certificate of Kind 2's translation from the input Lustre model and properties to Kind 2's internal representation. Checking it consists in proving that the property P_{obs} is invariant for $\mathcal{S}_{\text{obs}}$. Since $\mathcal{S}_{\text{obs}}$ and P_{obs} are generated in a format that corresponds directly to Kind 2's internal representation of transition systems and properties, this invariance proof can be done by Kind 2 itself without relying on its front end. Moreover, the proof is provided with its own safety certificate, which we call a *front end certificate*, of the sort discussed in Section II.

One possible problem with this approach is the small likelihood that the property P_{obs} is k-inductive for $\mathcal{S}_{\text{obs}}$, and for a small k, so as to be easily provable by Kind 2. We mitigate this by identifying pairs of corresponding state variables from $\mathbf{x}_1$ and $\mathbf{x}_2$ and suggesting their equality as a *candidate* auxiliary invariant for Kind 2 to try. Some of these equalities may indeed be proven invariant and so they can potentially help in the proof of P_{obs}. Note that while this harks back to the stronger notion of observational equivalence we mentioned earlier, it is not the same since the equivalence between certain non-input variables is only suggested, not required.

Example 1. *Consider again the Lustre model and property of Figure 2. The systems $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively generated by JKind 2.1*[2] *and Kind 2 from that model are the following, in abstract syntax and modulo variable renaming:*

$\mathcal{S}_1$	$\mathcal{S}_2$
$\mathbf{x}_1 = \{a_1, b_1, c_1, v_1\}$	$\mathbf{x}_2 = \{i, a_2, b_2, c_2, v_2\}$
$I_1 = R[\top, \mathbf{x}_1, \mathbf{x}'_1]$	$I_2 = (i \ \wedge v_2 = a_2 + b_2 \wedge c_2 = 1)$
$T_1 = R[\bot, \mathbf{x}_1, \mathbf{x}'_1]$	$T_2 = (\neg i' \ \wedge v'_2 = a'_2 + b'_2 \wedge$ $c'_2 = ite(c_2 > v'_2, c_2, v'_2))$
$R[g, \mathbf{x}_1, \mathbf{x}'_1] = (v'_1 = a'_1 + b'_1 \wedge$ $c'_1 = ite(g, \frac{10}{10}, ite(c_1 > v'_1, c_1, v'_1)))$	$P_2 = a_2 > 0 \wedge b_2 > 0 \Rightarrow c_2 > 0$
$P_1 = a_1 > \frac{0}{10} \wedge b_1 > \frac{0}{10} \Rightarrow c_1 > \frac{0}{10}$	

The equivalence observer $\mathcal{S}_{\text{obs}}$ is defined by

$$\begin{array}{ll} \mathbf{x}_{\text{obs}} = \mathbf{x}_1, \mathbf{x}_2 & I_{\text{obs}} = (a_1 = a_2 \wedge b_1 = b_2 \ \wedge \ I_1 \ \wedge \ I_2) \\ P_{\text{obs}} = (P_1 \Leftrightarrow P_2) & T_{\text{obs}} = (a'_1 = a'_2 \wedge b'_1 = b'_2 \ \wedge \ T_1 \ \wedge \ T_2) \end{array}$$

Suggested auxiliary invariants in this case will be the equalities $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2$, and $v_1 = v_2$ between corresponding state variables in the two systems. □

IV. From Certificates to LFSC Proofs

The last step of our approach, once the various safety certificates have been produced and checked, is to gather the proofs of the various entailment checks performed by the SMT solver and assemble them into a self-contained overall proof of safety for the original system.

LFSC proofs. The entailment proofs are obtained specifically from CVC4 as proof terms in LFSC, an extension of the Edinburgh Logical Framework (LF) [12] with side conditions [25]. In LFSC, which is in essence a dependently typed λ-calculus, proof systems are encoded as type systems. Proof checking then reduces to type checking, performed by a highly optimized checker developed by Stump *et al.* [24]. This particular LFSC checker takes as input a type system S and a term t in that system, and checks whether t is well typed in S. The efficiency of this framework for proof checking lies in the use of side-conditions, defined as small functional programs, which can be pre-compiled by the checker. Using proof rules with side conditions generally leads to both smaller proof sizes and faster proof checking times.

A proof system is formally defined in LFSC through *signatures*, which contain a definition of the system's language together with axioms and proof rules. The proof system used by CVC4 is defined over a number of signatures, which are included in its source code distribution. Those relevant to this work include signatures for propositional logic and resolution (sat.plf); first-order terms and formulas, with rules for CNF conversion and abstraction to propositional logic (smt.plf); equality over uninterpreted functions (th_base.plf); and real and integer linear arithmetic (th_int.plf and th_real.plf).

Extending CVC4's proof system. We have extended CVC4's proof system with an additional signature (kind.plf) for k-inductive reasoning, invariance and safety.[3] This signature also specifies the encoding for state variables, initial states, transition

[2]We produce $\mathcal{S}_1$ by having JKind 2.1 write a dump file from which we can extract its internal representation.

[3]The LFSC checker with all the necessary signatures are distributed with Kind 2 and publicly available.

$$\textsc{Inv+Obs}\;\dfrac{\textsc{InvImpl}\;\dfrac{\textsc{K-Ind}\;\dfrac{k\in\mathbb{N}\qquad \textsc{Smt}\;\dfrac{\vdots}{B_k\models\bot}\qquad \textsc{Smt}\;\dfrac{\vdots}{S_k\models\bot}}{\mathsf{invariant}(I,T,\phi)}\qquad \textsc{Smt}\;\dfrac{\vdots}{\phi\models P}}{\mathsf{invariant}(I,T,P)}\qquad \textsc{ObsEq}\;\dfrac{\textsc{InvImpl}\;\dfrac{\textsc{K-Ind}\;\dfrac{\vdots}{\mathsf{invariant}(I_o,T_o,\phi_o)}\qquad \textsc{Smt}\;\dfrac{\vdots}{\phi_o\models P_o}}{\mathsf{invariant}(I_o,T_o,P_o)}}{\mathsf{woe}(I,T,P,I',T',P')}}{\mathsf{safe}(I,T,P)}$$

Figure 3: Sketch of derivation tree for LFSC proofs of safety produced by Kind 2

$$\textsc{InvImpl}\;\dfrac{\forall k\in\mathbb{N}.\,P_1(k)\models P_2(k)\qquad \mathsf{invariant}(I,T,P_1)}{\mathsf{invariant}(I,T,P_2)}$$

$$\textsc{K-Ind}\;\dfrac{k\in\mathbb{N}\qquad B_k\models\bot\qquad S_k\models\bot}{\mathsf{invariant}(I,T,P)}\;\begin{bmatrix}B_k=\mathsf{base}(I,T,P,k)\\ S_k=\mathsf{step}(T,P,k)\end{bmatrix}$$

$\mathsf{unroll}(T,P,k)$ = match k with
  $|\,0\mapsto P(0)$
  $|\,1\mapsto P(0)\wedge T(0,1)$
  $|\,\_\mapsto \mathsf{unroll}(T,P,k-1)\wedge P(k-1)\wedge T(k-1,k)$

$\mathsf{step}(T,P,k)=\mathsf{unroll}(T,P,k)\wedge\neg P(k)$

Figure 4: A sample of LFSC rules for k-induction proofs

relations, and property predicates. State variables are encoded as functions from natural numbers to values. This way, the unrolling of the transition relation done in ($base_k$) and ($step_k$) does not need the creation of several copies of the state variable tuple $\mathbf{x}$. For example, for the state vector $\mathbf{x}=(y,z)$ with y of type real and z of type integer, the LFSC encoding will make y and z functions from naturals to reals and integers, respectively. So we will use the tuples $(y(0),z(0))$, $(y(1),z(1)),\ldots$ instead of (y_0,z_0), $(y_1,z_1),\ldots$ where $y_0,y_1,\ldots,z_0,z_1,\ldots$ are (distinct) variables. Correspondingly, our LFSC encoding of a transition relation formula $T[\mathbf{x},\mathbf{x}']$ is parametrized by two natural variables, the index of the pre-state and that of the post-state, instead of two tuples of state variables. Similarly, I, P and ϕ are parametrized by a single natural variable.

The signature defines several derivability judgments, including one for proofs of invariance, which has the following type:

$$\mathsf{invariant} : \Pi\ I:\mathbb{N}\to formula.\ \Pi\ T:\mathbb{N}\to\mathbb{N}\to formula.\ \Pi\ P:\mathbb{N}\to formula.\ Type$$

It also contains rules to build proofs of invariance by k-induction, as illustrated in Figure 4 in abstract syntax. There, proof rule InvImpl states that weakenings of invariants are invariants. Rule K-Ind encodes the k-induction principle as presented in Section II. It has two side-conditions that compute formulas for the subgoals of k-induction. As an example, we provide the definition of step, which uses an auxiliary function to compute unrollings of the transition relation.

This signature also specifies how to encapsulate proofs for the front-end certificates by providing a additional judgment, $\mathsf{safe}(I,T,P,I',T',P')$, which can be derived only when $\mathsf{invariant}(I,T,P)$ is derivable and the observational equivalence between (I,T,P) and (I',T',P') is provable (judgment woe). Self contained proofs of safety follow the sketch depicted in Figure 3, where Smt stands for an unsatisfiability rule whose proof tree is obtained, with minor changes, from a proof produced by CVC4.

In practice, running Kind 2 in proof production mode on a Lustre model generates an LFSC proof (in a text file) that can be then fed together with the various signature files ({sat,smt,th_int,th_real,kind}.plf) to the LFSC proof checker.

V. Experimental Evaluation

We evaluated our certificate generation and checking techniques on a set of academic benchmarks and a smaller set of industrial-grade benchmarks.[4] They come from different sources (academic and industrial users, published case studies, *etc.*) and are of various nature (memory coherence protocols, reactive controllers from railway and aerospace industry, counter systems, simulation of systems, …). We selected only benchmark problems consisting of a Lustre model with properties that Kind 2 could prove with a 5 minutes timeout.

We first focus on the effect of minimization on intermediate certificate checking by the SMT solver CVC4 and then evaluate our complete certification chain, including front end certification and LFSC proof checking.

We ran our tests on a Linux machine with two 12-core 64-bits AMD Opteron processors and 32GB of memory. We used a certifying version of Kind 2 based on Kind 2 `v0.8`. The CVC4 binary was from version `1.5-prerelease` (git proofs `7ba546df`). Tools were given a timeout of 5 minutes.

Certificate simplification. The plot in Figure 5 focuses on the effects of the certificate simplification techniques presented in Section II. It shows how many problems a particular configuration can cumulatively process within a certain amount of time. We compare various measures: S measures the time needed by Kind 2 to solve the model checking problem and generate an initial safety certificate, i.e., before simplification;[5] mE measure the time to reduce the safety certificate using the *easy* simplification technique (*i.e.*, only `trim` in Algorithm 1); m is the time to do the full simplification (*i.e.* both `trim` and `cherry-pick`); finally, cvc4 measures the time necessary for CVC4 to check the safety certificate—we exclude front end certificates in this analysis. We can see from the plot that

[4]Kind 2 is available at https://kind.cs.uiowa.edu and benchmarks are available at https://github.com/kind2-mc/kind2-benchmarks/tree/fmcad16.

[5]We do not show the time to just solve the problem because its difference with S is negligible.

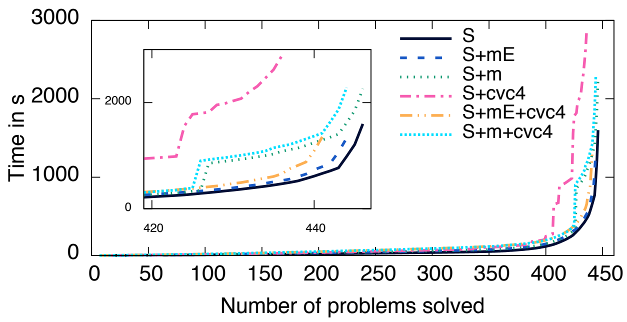


Figure 5: Overhead and improvements of minimization.

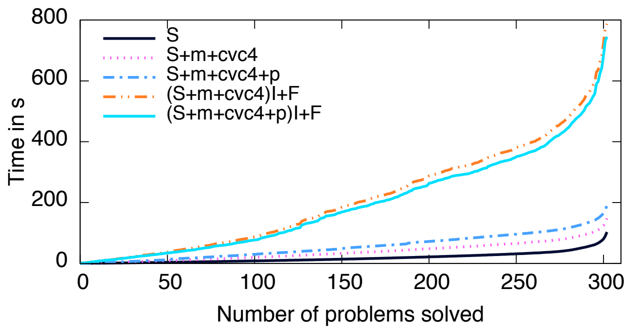


Figure 6: Evaluation of proof certification chain.

without any simplification (S+cvc4) we can check a lot less certificates and take much more time than with simplification. We can also see that, even if the full simplification process is more expensive (S+m vs. S+mE), it yields a larger number of checked certificates within the time limit (S+m+cvc4 vs. S+mE+cvc4). The superiority of full simplification is confirmed by an analysis of the full results. It reduces the size of the invariants on average by 74% (removing on average 19 invariants per certificate) for 42% of the benchmarks. For one benchmark, it removes 236 invariants out of 243. The value of k is reduced in 11% of the benchmarks, by 10 on average, the maximum being a reduction from 36 down to 2. The bump at 428 is due to the simplification overhead for a single benchmark, which is larger than the solving time. However, even with this outlier, the cumulative benefit of full simplification on certificates is clear.

Checking full certificates. The plot in Figure 6 refers to the complete proof certification chain. The measurements show the time necessary up to produce the proofs (S+m+cvc4) (which involve an intermediate checking phase, cvc4, with CVC4) and to check them with the LFSC proof checker (p). The second and third curves are for the invariance property while the last two also include the overhead for the front end proof (l+F). The latter includes the time to: prove the input property; fully minimize its safety certificate and generate the corresponding proof; construct the equivalence observer, including the time to call JKind and extract its transition system; model check the observer with Kind 2; minimize and produce the proof for the front end certificate; and finally check the combined resulting proof with LFSC.

We are able to generate and check the proof of invariance for around 80% of benchmarks that Kind 2 succeeds in verifying; we produce and check a complete proof including the front end for 60% of them. Most of the cases where we fail to generate the proof are due to CVC4's current limitations in its proof producing capabilities. The biggest bottlenecks are the model checking of the equivalence observer and the simplification of certificates. Despite that, the time cost of the full certification chain is overall within one order of magnitude of the cost of just proving the input property. We find the overall level of performance, which we think we could improve further, already rather good, especially considering that a lot of the benchmarks we used are non-trivial.

VI. Related Work

Formally verified model checkers. A natural approach to the certification of verification tools consists in proving the program (here the model checker) correct once and for all. This is possible to a large extent for programs written in programming languages with (largely automated) verification toolsets such as ESC Java 2, Frama-C, VCC, F* *etc.* Proving full functional correctness of a model checker, however, is currently a very challenging job because these tools are often rather complex and tend to evolve quickly with the ongoing advances in the field. When feasible, one great advantage of this approach of course is that the performances of the model checker is minimally impacted by the verification process. One example of this kind of certification effort is the modern SAT solver `versat` which was developed and verified using the programming language Guru [17]. We are, however, not aware of similar results for model checkers.

Another possibility is to prove the underlying algorithms of a model checker correct in a descriptive language of interactive proof assistants such as Coq or Isabelle, and obtain an executable program from these tools through a refinement process or code extraction mechanism. Although the first formal verification of a model checker in Coq for the modal μ-calculus [23] goes back to 1998, only recently have *certified verification tools* started to emerge. Amjad [1] shows how to embed BDD-based symbolic model checking algorithms in the HOL theorem prover so that results are returned as theorems. This approach relies on the correctness of the backend BDD implementation. Esparza *et al.* [10] have fully verified an automata-based model checker for finite state systems with the Isabelle theorem prover. Using successive refinements, they built a correct by construction model checker from high level specifications down to functional (SML) code.

A recent approach for the certification of SAT and SMT solvers [2] consists in having the solver produce a detailed certificate in which each rule is read and verified by a combination of several small certified checkers, written and proved correct in Coq. This approach also allows one to import inside Coq proof terms from these solvers [3].

Certifying model checkers. A number of techniques have been proposed to produce certifying model checkers. Earlier solutions (e.g., [15], [16], [18]) were limited to finite-state systems. The first certifying model checker for infinite-state systems was perhaps the C model checker BLAST [13], which produced certificates for a control flow automaton internally generated from an input C program. BLAST provided proof certificates in the Edinburgh Logical Framework (LF) [12], which limits the scalability of certificate checking when proofs involve reasoning modulo the theory of C's data types.

A more recent certifying model checker is SLAB [9], which produces certificates in the form of inductive verification diagrams to be checked by SMT solvers. We go one step further by relying on SMT solvers that are in turn proof producing. Also, we address the issue of certifying the translation from the input model to the internal representation.

For model checking of parameterized systems, the model checker Cubicle generates certificates as Why3 files that can be independently checked by several SMT solvers and automated theorem provers [8], where trust is claimed through the redundant use of multiple solvers.

VII. Conclusion and Future Work

We have presented a dual technique for generating and checking proof certificates for SMT-based model checkers, and applied it to the model checker Kind 2. Given a Lustre model and one or more invariance properties for it, Kind 2 generates LFSC proofs for the properties it can verify. These proofs have two parts. The first attests that the model and the properties are encoded correctly in Kind 2's internal representation format. It does that by proving the observational equivalence, with respect to the properties, between the internal system and another one produced from the same Lustre input by an independent, third-party tool. The second part attests that the encoded properties are invariants of the internal transition system encoding the Lustre model. Initial certificates, which we call safety certificates, are generated as (possibly combined) k-inductive invariants, and simplified before being verified by the CVC4 SMT solver. The eventual proof certificates, in LFSC format, are assembled from the proofs generated by CVC4 after verifying these safety certificates.

The trusted core of our approach consists in:

1) The LFSC checker (5300 lines of C++ code).
2) The LFSC signatures comprising the overall proof system in LFSC (CVC4's sat.plf, smt.plf, th_base.plf, th_int.plf, th_real.plf and our own kind.plf, for k-induction and safety), for a total of 444 lines of LFSC code.
3) The assumption that Kind 2 and JKind do not have identical defects that could escape the observational equivalence check.

A current but temporary limitation of our certificate generation process is that LFSC proofs may contain an unsound proof rule, trust_f, which derives any formula. This rule is used by the current version of CVC4 to fill in present gaps in its proof generation code. However, it will be progressively phased out as the instrumentation of CVC4 to produce full proofs is completed.

Kind 2 has the ability to do compositional and modular analyses of Lustre models extended with assume-guarantee-style contracts. A possible line of future research is to extend the work described here to apply to such analyses by incorporating their underlying abstraction mechanisms.

Kind 2's proof certificate generation is being leveraged in an ongoing project funded by NASA and the FAA as an innovative way to reduce the cost of tool qualification with respect to DO-178C requirements.

Acknowledgments. We would like to thank Lucas Wagner and Konrad Slind for their feedback on this work and Andrew Gacek for his assistance with JKind.

References

[1] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *TPHOL*, pages 171–187. Springer, 2003.
[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *PSATTT*, 2011.
[3] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *CPP*, pages 135–150. Springer, 2011.
[4] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philos Trans A Math Phys Eng Sci*, 363(1835):2351–2375, 2005.
[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177. Springer, 2011.
[6] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
[7] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In *CAV*, pages 510–517. Springer, 2016.
[8] S. Conchon, A. Mebsout, and F. Zaïdi. Certificates for parameterized model checking. In *FM*, pages 126–142. Springer, June 2015.
[9] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, volume 6015, pages 271–274. Springer, 2010.
[10] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044, pages 463–478. Springer, 2013.
[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.
[13] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526–538. Springer, 2002.
[14] A. Ivrii, A. Gurfinkel, and A. Belov. Small inductive safe invariants. In *FMCAD*, pages 115–122. IEEE, 2014.
[15] O. Kupferman and M. Y. Vardi. From complementation to certification. *Theor. Comput. Sci.*, 345:83–100, November 2005.
[16] K. S. Namjoshi. Certifying model checkers. In *CAV*, pages 2–13. Springer, 2001.
[17] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat: A verified modern SAT solver. In *VMCAI*, volume 7148, pages 363–378. Springer, 2012.
[18] D. Peled and L. Zuck. From model checking to a temporal proof. In *SPIN*, pages 1–14. Springer, 2001.
[19] M. Petit-Doche, N. Breton, R. Courbis, Y. Fonteneau, and M. Güdemann. Formal verification of industrial critical software. In *FMICS*, pages 1–11. Springer, 2015.
[20] Prover Technology. Prover tools. http://www.prover.com/products.
[21] Rockwell Collins. JKind - a Java implementation of the KIND model checker. http://loonwerks.com/tools/jkind.html.
[22] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, London, UK, 2000. Springer.
[23] C. Sprenger. A verified model checker for the modal μ-calculus in coq. In *TACAS*, pages 167–183, London, UK, 1998. Springer.
[24] A. Stump. Proof checking technology for satisfiability modulo theories. *ENTCS*, 228:121–133, 2009.
[25] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *FMSD*, 42(1):91–118, 2013.

Routing under Constraints

Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: alexander.nadel@intel.com

***Abstract*—Routing is an essential stage in physical design, where already placed components are connected by wires. Routing must satisfy various manufacturing requirements, referred to as design rules. We formalize the problem of design-rule-aware routing and introduce a solver, called `DRouter`, for the resulting problem. Plain routing is often modeled as follows: given an undirected weighted graph and a set of m disjoint nets (each net being a set of vertices), a routing is a (minimal) forest of m disjoint trees, where each tree spans a net. `DRouter`'s input comprises a plain routing instance and a bit-vector formula, whose variables include the edges of the graph as Boolean variables (along with other variables). `DRouter` looks for a satisfying assignment to F, such that the satisfied edges comprise a routing. `DRouter` implements an A*-based router inside a SAT solver. It overrides the solver's decision and restart strategies and enhances its learning with routing-aware algorithms. We demonstrate that, on a set of crafted routing instances, `DRouter` has substantially better capacity than either plain reduction to bit-vector reasoning or `Monosat`, a solver that is able to reason about SAT and graph predicates. We show that `DRouter` can route large clips from Intel designs while obeying up to millions of applications of the design rules–a task two industrial routers failed to accomplish.**

I. Introduction

Wire routing (or, simply, *routing)* is an essential stage in the process of the physical design of integrated circuits [17] and printed circuit boards [1]. A router *routes* (that is, connects) components laid out during the placement stage. *Design rules* specify restrictions on the routing process originating in manufacturing requirements.

Plain routing (that is, routing without design rules) is often modeled as the Steiner tree packing problem [1], [4], [8], [9], defined as follows: Let $G = (V, E)$ be a positively weighted simple graph. Let $N_{i \in \{0 \dots m-1\}} \subseteq V$ be m pairwise disjoint non-empty subsets of G's vertices, called the *nets*, where the vertices of each net are called the *terminals*. A *routing* is a forest of *net routings* $E_{i \in \{0 \dots m-1\}} \subseteq E$, such that $(V(E_i), E_i)$ is a tree that spans all N_i's terminals, where all the net routings are pairwise vertex-disjoint and the *optimization requirement* of minimizing the routing's total weight is met. An example is provided in Fig. 1. To solve plain routing, heuristic approaches, relaxing the optimization requirement to some extent, are commonly applied – see [1] for a survey.

In practice, routing must conform to design rules. For example, the *short rule* [18] states that no edge may touch two distinct net routings.

This paper extends the plain routing formulation to model design-rule applications. We let the user provide a SAT or bit-vector instance F along with a plain routing instance, where F's Boolean variables include the edges. The router must satisfy F, while guaranteeing that the satisfied edges comprise a routing. We refer to the resulting problem as *Routing Under Constraints (*RUC*)*.

It is well-known that plain routing (in various flavors) can be reduced to SAT [6], [19]. It has also been observed that design rules can be reduced to SAT [16], [18]. The apparent advantage of any SAT-based router to a heuristic router is that SAT can handle arbitrary constraints (corresponding to applications of arbitrary design rules) efficiently based on its sophisticated conflict analysis. In addition, modeling a new design rule for a heuristic router involves non-trivial modification of the router, whereas in a SAT-based approach any design rule reducible to SAT does not require modification of the router. Furthermore, unlike any heuristic router, a SAT-based router is complete.

The main challenge for any SAT-based approach to routing is scalability. As we shall see, a straightforward reduction of RUC to SAT through bit-vector reasoning does not scale.

To overcome the scalability issue, we designed a RUC solver, called `DRouter`. Essentially, `DRouter` implements a router inside a SAT solver. It overrides the SAT solver's decision and restart strategies with routing-aware strategies and enhances its learning with routing-aware conflict analysis.

The usefulness of applying a graph-aware decision strategy and conflict analysis inside a SAT solver (in contrast to full reduction to bit-vector reasoning) was advocated and highlighted in a recent work on solving the NP-hard problem of finding a bounded-path (that is, a path whose weight falls within a user-given range) in a graph [7]. In [7], the decision strategy replaces the majority of the constraints; it guides the solver towards the solution, while taking additional optimization requirements into account. The decision strategy's role in our work is no less prominent.

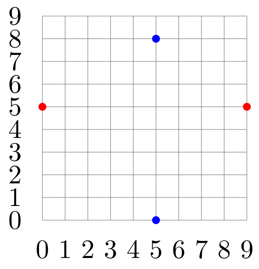
`Monosat` [3] is a recent tool which can reason about graph reachability and bit-vectors. The RUC problem can be easily formulated in `Monosat` language. Let *Pathfinding under Constraints (*PFUC*)* be a restriction of RUC to one 2-terminal net. `DRouter`'s PFUC solving algorithm is conceptually similar to that of `Monosat`. We shall see that `DRouter` is substantially more efficient than `Monosat` for the generic RUC problem. Sect. V provides a detailed comparison between `Monosat` and `DRouter`.

We shall show that `DRouter` can route large clips of Intel design while obeying up to millions of applications of the design rules, whereas two industrial routers failed to do so.

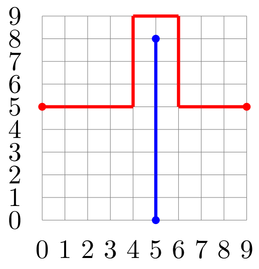
In what follows, Sect. II contains preliminaries. Sect. III introduces our PFUC solving algorithm, called `DPF`. `DRouter`, introduced in Sect. IV, uses `DPF` as the underlying building

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

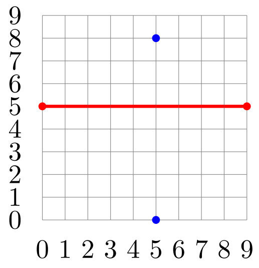
block. Sect.V compares `DRouter` to `Monosat`. Experimental results are presented in Sect. VI. Sect. VII concludes our work.



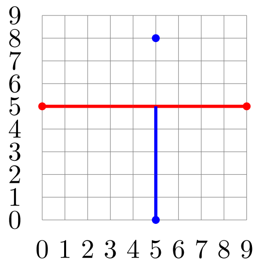
(a) Input



(b) Solution



(c) Conflict in `DRouter`



(d) Conflict in `Monosat`

Fig. 1: Plain routing example on a 10×10 solid grid graph, given two nets of two terminals each $N_0 = \{(0,5),(9,5)\}$ and $N_1 = \{(5,0),(5,8)\}$. Assume the edges' weights are 1.

II. Preliminaries

A. Bit-vector Reasoning and SAT

A bit-vector (BV) solver decides formulas comprising fixed-sized bit-vectors, Boolean variables and a variety of bit-vector and Boolean operators. An eager BV solver works by preprocessing the given BV formula, bit-blasting it to Conjunctive Normal Form (CNF) and solving with a SAT solver. We assume that the reader is familiar with the basics of SAT and eager BV solving. See [10] for a recent overview.

B. Modeling of Routing Under Constraints (RUC)

Our basic RUC modeling extends the plain routing modeling, presented in Sect. I. Let $G = (V, E)$ be a graph, $N_{i\in\{0\dots m-1\}} = \left\{t_0^i, t_1^i \dots t_{|N_i|-1}^i\right\} \subseteq V$ be the pairwise disjoint nets and $F(E \cup U)$ be a bit-vector formula (where U is a set of bit-vector and Boolean variables). Given a model for F, α, let $E^\alpha \subseteq E$ be a subset of edges assigned to 1 in α. The problem of *Routing Under Constraints (*RUC*)* is about finding a model α for F, such that any two terminals of the same net N_i are connected in $(V(E^\alpha), E^\alpha)$ and any two terminals of two different nets are disconnected in $(V(E^\alpha), E^\alpha)$.

Note that we leave out the requirement that the routing solution be a forest of *trees* and also the optimization requirement as to the overall weight. Similarly to heuristic routers, our algorithm will strive to heuristically reduce the overall weight of the satisfied edges.

For convenience, we extend our modeling as follows.

First, we let F use the vertices V (along with the edges) as Boolean variables. Given a RUC model α, an edge or vertex $b \in V \cup E$ is *active* iff $\alpha(b) = 1$.

Second, for every vertex v, we introduce a bit-vector variable $0 \leq \mathit{nid(v)} < m$ of width $\lceil log_2 m - 1\rceil$ to represent the unique net id of active vertices, where the *net id* of net N_i is the index i. For each vertex v, we encode the *net boundary* constraint $0 \leq \mathit{nid(v)} < m$ into F.

Third, we encode the following *edge consistency* constraints into F: for each edge $e = (v, u)$, $e \implies v \wedge u \wedge (\mathit{nid(v)} = \mathit{nid(u)})$.

The short rule, mentioned in Sect. I, can now be encoded as follows: for each (not necessarily active) edge $e = (v, u)$, $v \wedge u \implies (\mathit{nid(v)} = \mathit{nid(u)})$). Note that the short rule does *not* hold for the synthetic example in Fig. 1b.

C. Routing Complexity

Routing is a difficult problem. Plain routing is NP-hard even for one net [11] (in which case it is reduced to the classical problem of finding a Steiner tree). For multiple nets, plain routing is NP-complete even without the optimization requirement [13]. RUC is NExpTime-hard, since BV logic is trivially reducible to RUC, and BV logic is NExpTime-hard [12] (although various subsets of BV logic that are relevant in practice are NP-complete [5]).

D. Reducing RUC to Bit-Vector Reasoning

The following BV encoding for RUC can be applied if the nets are restricted to two terminals only. Assuming the edge consistency constraints are encoded (Sect. II-B), it remains to ensure that for each net i, its pair of terminals t_0^i and t_1^i is connected. That can be done by encoding the following cardinality constraint for each vertex: each terminal has one active neighbor edge and each non-terminal vertex has two active neighbor edges. One can extend the modeling to multi-terminal nets by encoding the construction of directed trees with a terminal sink for each net. We omit further details due to space restrictions. As we shall see, BV encoding does not scale well.

E. A* Algorithm

A* is a commonly used algorithm for finding a path in a graph from a source vertex s to a target vertex t, given an under-estimation, $h(v)$, of the weight from any vertex v to t. If $h(v) = 0$ for every v, A* operates identically to Dijkstra shortest-path algorithm. Having an accurate heuristical estimation helps A* converge faster.

We are interested in graphs having a grid-like structure, that is, graphs, whose vertices represent nodes of a two- or three-dimensional grid. This is because routing in the original physical design problem is carried out in a grid. One example of a grid-like graph is a *solid grid graph*, whose vertices correspond to the points of a two-dimensional grid and whose edges connect any two vertices at distance one (see the example in Fig. 1). Our algorithms apply Manhattan distance as the A* heuristic in a grid-like setting and set $h(v) = 0$ for each vertex v, otherwise.

We need a slightly modified version of A*.

First, the modified A* can receive an optional parameter: a set of *bad* edges and vertices *Bad*, which cannot be used for connecting s to t. The implementation of this feature is straightforward: whenever A* visits an edge or vertex $b \in$ *Bad*, it abandons the exploration of the path containing b.

Second, the modified A* can receive another optional parameter: a set of edges, whose weight should be set to 0 for that particular A* invocation.

Third, in cases where s is not connected to t in G, given a set of bad edges and vertices A* returns a special value $\perp$ and generates a *conflict cut*. Intuitively, a conflict cut is a subset of the bad edges and vertices that prevented A* from connecting s to t. An empty conflict cut would mean that s is disconnected from t in G, independently of the bad set. Below, we provide a more precise definition of the conflict cut.

1) Let *Visited* $\subseteq V \cup E$ be the set of vertices and edges visited by A* during the graph traversal (that is, vertices and edges connected to s, *Bad* included)
2) Let *Unvisited* $\subseteq V \cup E$ be the set of all vertices and edges, unvisited by A*
3) Let the *frontier Frontier* $\subseteq V \cup E$ be a set, including:
    a) All the vertices from *Visited* which have at least one neighbor edge in *Unvisited*, and
    b) All of the edges from *Visited* whose other vertex is in *Unvisited*
4) The *conflict cut* is the set of vertices and edges that belong to both *Bad* and *Frontier*.

III. Pathfinding under Constraints

Let the *Pathfinding under Constraints (*PFUC*)* problem be RUC, restricted to one 2-terminal net. We propose an algorithm for solving PFUC, called `DPF`.

Given a graph $G = (V, E)$, a single net $N = \{s, t\}$ and a BV formula $F(E \cup V \cup U)$, `DPF` should return either a model α for F, such that there exists a path from s to t in $(V(E^{\alpha}), E^{\alpha})$, or UNSAT, if no such model exists. `DPF` is designed to heuristically reduce the overall weight of the satisfied edges.

For the rest of this section, we assume that the constraint $e \implies v \wedge u$ for each edge $e = (v, u)$ is encoded into F. The net boundary and edge consistency constraints from Sect.II-B are unnecessary for pathfinding, since pathfinding involves a single net.

A. `DPF` *Algorithm*

`DPF` is implemented inside an eager BV solver's SAT solver. It overrides the SAT solver's decision strategy with an A*-based algorithm and records additional conflict clauses whenever s becomes disconnected from t because of propagation in F. `DPF` disables the SAT solver's restart strategy.

During `DPF`'s invocation, an edge or vertex b can either be: *a*) *active*–if b is assigned 1, *b*) *inactive*–if b is assigned 0, or *c*) *unassigned*.

`DPF`'s decision strategy tries to connect s to t by activating edges (that is, assigning 1 to edges) in a queue σ, where σ is initialized to the shortest path from s to t using A*. The activated edges of σ are moved from the front of σ to the back of the *actual path* stack π. Before each decision point, the concatenation $\pi \circ \sigma$ comprises a walk from s to t in G, where π's edges are active and σ's edges can be unassigned or active. When the algorithm is finished, π contains a walk from s to t. π might be a walk rather than a path for reasons which will be discussed later. We suggest a simple post-processing algorithm targeting cycle elimination in π in Sect. III-B.

```
 1: class PathFinder
 2:     members:
 3:         vertex $s$;                                        ▷ source vertex
 4:         vertex $t$;                                        ▷ target vertex
 5:         walk $\pi = s \equiv \pi_0\pi_1 \ldots \pi_{|\pi|-1} \equiv f$;
 6:         path $\sigma = f \equiv \sigma_0\sigma_1 \ldots \sigma_{|\sigma|-1} \equiv t$;
 7:     methods:
 8:         Init(vertex $s'$, vertex $t'$, BV Formula $F$) $\rightarrow$ Is Conflict?
 9:             $s := s'$; $t := t'$
10:             $\sigma :=$ A*(s,t)
11:             if A* returned $\perp$ then return $\perp$ else return $\top$
12:         Decide() $\rightarrow$ SAT literal
13:             if $t \in \pi$ then                     ▷ $\pi$ connects $s$ to $t$
14:                 if Unassigned edge $e$ exists then return $\neg e$
15:                 return SAT-Decide()
16:             return $(\sigma_0, \sigma_1)$
17:         Propagate() $\rightarrow$ Is Conflict?
Require: Invoked right after BCP if there was no conflict
18:             while $|\sigma| = 1$ or $(\sigma_0, \sigma_1)$ is active do
19:                 Pop $\sigma_0$ from $\sigma$'s front; push it to $\pi$'s back
20:             if $\sigma$ is not violated then return $\top$
21:             $\sigma$ := A*($\pi_{|\pi|}$,$t$,inactive vertices and edges,active edges)
22:             if A* returned $\perp$ then
23:                 Add a clause comprising the conflict cut
24:                 return $\perp$
25:             while $|\sigma| = 1$ or $(\sigma_0, \sigma_1)$ is active do
26:                 Pop $\sigma_0$ from $\sigma$'s front; push it to $\pi$'s back
27:             return $\top$
28:         Backtrack()
Require: Invoked after completing SAT solver's backtracking
29:             while $\pi$ contains unassigned edges do
30:                 Pop the latest vertex from $\pi$'s back
31:             $\sigma := \{\}$                                     ▷ Clear $\sigma$
```

Fig. 2: `DPF` Solver for Pathfinding under Constraints.

We say that σ is *violated* when one of its vertices or edges becomes inactive.

Consider the class implementing `DPF` in Fig. 2. The members of the class (lines 3– 6) comprise s, t, π and σ. Consider also the `DPF` invocation trace example in Fig. 3.

The method Init (line 8) is invoked before the SAT solver is launched. In Init, `DPF` initializes s and t and then checks whether there exists a path from s to t in G using A*. If no such path exists, the problem is unsatisfiable. Otherwise, the algorithm stores the path in σ. The modified SAT solver is then invoked. Fig. 3a illustrates the situation after initialization for our example.

The method DECIDE (line 12) overrides the SAT solver's decision heuristic. If the target is already reached (that is, $t \in \pi$), DECIDE deactivates (that is, assigns 0) an unassigned edge, if any, to minimize the overall active edge weight. If all the edges are assigned, the algorithm invokes the default SAT decision strategy. If the target is not reached, DECIDE simply returns the first σ's edge (σ_0, σ_1). The methods PROPAGATE and BACKTRACK guarantee that (σ_0, σ_1) is unassigned when DECIDE is invoked.

Consider the method PROPAGATE (line 17). It is invoked right after the SAT solver's BCP (Boolean Constraint Propagation) if BCP did not encounter a conflict. Similarly to BCP, PROPAGATE should return whether it found a conflict ($\top$ and $\perp$, respectively, are returned for "no conflict" and "conflict").

PROPAGATE starts by moving any active edges from σ's front to π's back. Afterward PROPAGATE returns $\top$ if σ is not violated. We call the situation when σ is violated *path violation*. A path violation occurs in Fig. 3b of our example.

In case of path violation, we run A* so as to create a new σ to continue the path π towards t using active and unassigned edges only. We also set the weight of active edges to 0 for A*, in order not to "pay" for a single edge more than once in case A* reuses already assigned edges. This can happen, for example, if there is no path from $\pi_{|\pi|-1}$ to t anymore as is the case in Fig. 3b. This also explains why π might be a walk, rather than a path.

If A* was successful in finding a new σ, PROPAGATE moves any active edges from σ's front to π's back and returns. This is the case in our example. Fig.3c illustrates the situation just after PROPAGATE's completion.

Assume now that A* could not find a new path. This is the case in Fig. 3d continuing our example. In this case the algorithm adds the conflict cut as a conflict clause and returns $\perp$. The conflict cut is a subset of the inactive vertices and edges that blocks any path from s to t. In our example, the conflict cut comprises the following set $\{(2, 0), (3, 1)\}$. The conflict cut, seen as a clause, must always be falsified by the current partial assignment, since all its members are inactive. Hence recording it as a conflict clause triggers SAT solver's conflict analysis.

During conflict analysis, the solver will learn the 1UIP conflict clause and backtrack, so as to have one and only one asserting literal of the conflict clause unassigned. The method BACKTRACK (line 28) is invoked whenever SAT solver's backtracking is completed. The method aligns π with the current partial assignment (by popping all the unassigned edges from π) and clears σ. After backtracking, the solver flips the asserting literal and applies BCP, followed by a PROPAGATE invocation. Note that σ is populated again by PROPAGATE.

In our example, the learned 1UIP conflict clause is $\neg(2, 0) \vee \neg(3, 2)$ (we omit derivation details due to space constraints). The situation after backtracking, propagating and finding a new σ in our example is shown in Fig. 3e. The vertex $(3, 2)$ is rendered inactive because of BCP in the new clause.

This completes the description of our PFUC algorithm. Fig. 3f shows a completed routing for our example.

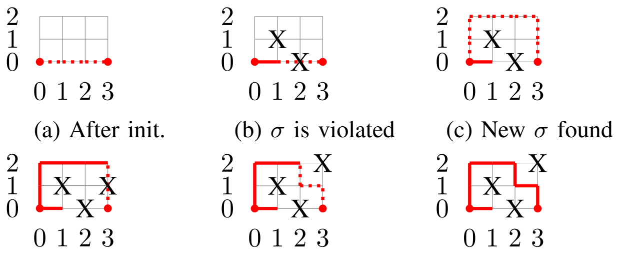
As we mentioned, after completing the routing, the solver deactivates any unassigned edges to reduce the weight and then falls back to the default decision heuristic.

Note that even after the initial routing has been completed, the solver might still backtrack and change the routing. In our example in Fig. 3, replacing the clause $\neg(3, 2) \vee \neg(3, 1)$ by an equivalent set of clauses $\neg(3, 2) \vee \neg(3, 1) \vee x \vee y$, $\neg(3, 2) \vee \neg(3, 1) \vee \neg x \vee y$, $\neg(3, 2) \vee \neg(3, 1) \vee x \vee \neg y$, $\neg(3, 2) \vee \neg(3, 1) \vee \neg x \vee \neg y$, where x, y are auxiliary variables, would cause the solver to generate a "bad" routing through the vertices $(3, 2)$ and $(3, 1)$ which it would only fix later following conflict analysis and backtracking.

B. Optimization with the Decision Strategy

As we have seen, `DPF` applies the following two techniques as part of its decision heuristic to heuristically reduce the routing weight: *a*) using the shortest path A* algorithm, and *b*) deactivating any unassigned edges after the routing is completed.

In addition, one can apply the following post-processing algorithm to eliminate cycles in π (if any) to reduce the total edge weight. The algorithm below reuses the SAT solver instance created by `DPF`. The instance is updated and invoked incrementally. First, assuming `DPF` returned a model α, identify a simple tree in $(E^\alpha, V(E^\alpha))$ and provide its edges as unit clauses to the SAT solver. Second, run a plain SAT solver with the following single modification to the decision heuristic: deactive unassigned edges first.



(a) After init. (b) σ is violated (c) New σ found
(d) A conflict (e) Conflict resolved (f) Routing done

Fig. 3: `DPF` trace example. Assume that $s = (0, 0)$ and $t = (3, 0)$ and that the following CNF is provided: $\neg(1, 0) \vee \neg(2, 0)$, $\neg(1, 0) \vee \neg(1, 1)$, $\neg(3, 2) \vee \neg(3, 1)$. Dotted edges correspond to σ, while bold edges correspond to π. "X" marks inactive vertices.

IV. ROUTING UNDER CONSTRAINTS

This section introduces `DRouter` – our RUC solution. Similarly to `DPF`, `DRouter` is implemented inside a SAT solver. Sect. IV-A below adjusts `DPF` to routing. Our basic `DRouter` algorithm is presented in Sect. IV-B. Sect. IV-C, Sect. IV-D and Sect. IV-E introduce three enhancements to the basic algorithm which are crucial for scalability.

A. Routing-Aware Pathfinding

We need to make simple yet essential modifications to `DPF` to make it routing-aware. Our goal is to be able to apply `DPF` inside `DRouter` to connect terminals within a net.

First, net boundary and edge consistency constraints must be applied (recall Sect. II-B).

Second, the class PATHFINDER should have an additional member *nid*, representing the net id of the routed net, which will be initialized during INIT to a new (third) parameter.

Third, DECIDE will not be invoked after s is connected to t; hence the code in Fig. 3 between lines 13– 15 can be ignored.

Fourth, a crucial modification should be applied to PROPAGATE when a path violation is identified (line 21). When this happens, A* is invoked to find a new path to t. We disallow A* from using the vertices of any net other than the current net (in addition to the disallowed inactive vertices and edges). After this change, line 21 looks as follows:

σ := A*($\pi_{|\pi|}$,t,inactive vertices and edges $\cup$ any active vertex v, such that *nid(v)* $\neq$ *nid*, active edges)

Furthermore, the conflict clause, created at line 23, will contain additional literals as follows: any vertex v of net id *nid(v)* $\neq$ *nid* will contribute to the conflict clause one bit on which the values of *nid(v)* and *nid* differ.

For example, assume that the modified DPF is invoked to connect the two terminals of N_1 after N_0 is routed as shown in Fig. 1c. A* will fail, since all the possible paths are blocked by N_0. The conflict clause will contain the only bit of *nid(v)* for every vertex v of row 5.

B. The Basic Algorithm

We can now present the basic algorithm of DRouter.

A class implementing the algorithm for 2-terminal nets is depicted in Fig. 4 (an extension for multi-terminal nets is proposed later in this section). The class maintains an array of net ids *nids*, the current index to *nids*, and an array of the class PATHFINDER. We assume that PATHFINDER is modified to be routing-aware as explained in Sect. IV-A.

The idea is simply to route the nets one by one using a fresh *pathfinder* (that is, a fresh instance of the class PATHFINDER) for each net.

The method INIT initializes *nids* and then initializes a pathfinder for net id 0.

The method DECIDE simply applies the DECIDE method of the currently routed net, if such exists. If all the nets are routed, DECIDE deactivates unassigned edges, if any, and, otherwise, lets the SAT solver take the decision.

PROPAGATE operates in a loop as long as non-routed nets exist. Within the loop, it tries to propagate in the currently routed net, which might result in a conflict, in which case PROPAGATE returns $\perp$. Otherwise, if the net is not yet routed, the method returns $\top$. If the net is routed, PROPAGATE initializes a pathfinder for the next net, if any, pushes it to the pathfinders vector and continues the main loop.

BACKTRACK backtracks over any fully routed nets, and then it backtracks within the currently routed net (if one exists).

Our algorithm can easily be extended to treat any multi-terminal net by connecting routing previously created for that net to any new terminal until the net is fully routed. A* can be applied, without modifications, for connecting multiple sources to a single target.

```
class DRouter
    members:
        number[] nids;
        number currNidInd;
        PATHFINDER[] pfs;

    methods:
        INIT() → Is Conflict?
Require: SAT solving has not yet started
            nids := $\{0, 1, \ldots, m-1\}$
            PATHFINDER p;
            if P.INIT($t_1^0$,$t_2^0$,0) == $\perp$ then return $\perp$
            Push $p$ to the back of pfs
            currNidInd := 0;
            return $\top$

        DECIDE() → SAT literal
            if currNidInd $\leq m$ then
                $p$ := back of pfs
                return P.DECIDE()
            if Unassigned edge $e$ exists then return $\neg e$
            return SAT-DECIDE()

        PROPAGATE() → Is Conflict?
Require: Invoked right after BCP if there was no conflict
            while currNidInd $< m$ do
                $p$ := back of pfs
                if P.PROPAGATE() == $\perp$ then return $\perp$
                if $p.t \notin p.\pi$ then return $\top$
                currNidInd := currNidInd + 1
                if currNidInd == $m$ then return $\top$
                PATHFINDER p;
                $n$ := nids [currNidInd]
                if P.INIT($t_1^n$,$t_2^n$,n) == $\perp$ then return $\perp$
                Push $p$ to the back of pfs

        BACKTRACK()
Require: Invoked after completing SAT solver's backtracking
            while currNidInd $\geq 0$ do
                $p$ := back of pfs
                P.BACKTRACK()
                if $|p.\pi| > 1$ then return
                Pop from pfs's back
                currNidInd := currNidInd − 1
```

Fig. 4: Basic DRouter for 2-terminal nets.

One can also apply a post-processing algorithm for cycle elimination to reduce the overall solution weight, similarly to Sect. III-B

C. Early Net Conflict Detection

Let *net conflict* be a situation during DRouter invocation when a certain *conflicting net* N_i cannot be routed anymore, since there exists no path in the graph between two of its terminals t_q^i and t_w^i ($q \neq w$).

Our algorithm might encounter a net conflict when A* is applied to route a new terminal of the conflicting net. A net N_j *blocks* the conflicting net, if any vertex v, which belongs to N_j (that is, a vertex v, such that *nid(v)* $= j$), is part of the conflict cut.

A net conflict situation appears in Fig. 1c, where net N_0 blocks the conflicting net N_1.

Our basic algorithm identifies net conflicts only when it gets to routing the conflicting net, but it is desirable to discover and handle such situations earlier. By "handling" we mean recording a clause, which will cause the solver to backtrack and re-route.

We propose the following *early net conflict detection* algorithm. After any net is routed, we check, for every unrouted net, if it can still be routed by applying A* for connecting terminal i to terminal 0, for every $i > 0$. If a conflicting net is discovered, we record a conflict clause comprising the conflict cut found by A*.

To speed things up, one can keep, for each net and each terminal $i > 0$, a pre-routed path β to terminal 0 of that net, and check if β is still not violated before invoking A*. If A* has to be invoked and it finds a path, β can be updated to that path.

D. Net Swapping

Net ordering is crucial for our algorithm. Consider the example in Fig. 1a. If DRouter picks N_1 as the first net, the solution in Fig. 1b will instantly be found without any net conflicts. Picking N_0 as the first net would result in a net conflict, shown in Fig. 1c. The algorithm described so far would have to record conflict clauses to disqualify a variety of paths connecting N_0's terminals until it discovers a path which does not block N_1. Such an approach might cause run-time and/or memory explosion issues in practice.

We propose two solutions for this problem. The first is net swapping, presented next. The second is net restarting, presented in Sect. IV-E.

Net swapping is applied after a net conflict is discovered and the corresponding conflict clause recorded. Assume N_i is the contradicting net. It might be blocked by several previously routed nets. Let N_j be the net routed last out of all nets blocking N_i. In that case, the *nids* array, which defines routing order, should appear as follows: $\{A, j, B, i, C\}$ (where A, B, and C each represent a sequence of net ids). Net swapping backtracks to the decision level just before the algorithm started routing N_j and swaps between the nets N_j and N_i. In addition, it moves N_i to immediately follow N_j. *nids* will look as follows after net swapping: $\{A, i, j, B, C\}$. Hence, after net swapping, the algorithm will attempt to route N_i, followed by N_j.

Net swapping solves the problem in Fig. 1a even if N_0 is picked as the first net simply by swapping the nets after the first net conflict.

Net swapping might not be sufficient for finding a routable net ordering, because it is restricted to two nets only. For example, a problem might occur if the two nets block each other, regardless of the order, because of previously routed nets. In such a case, the algorithm will keep swapping the two nets. The algorithm would still complete because of conflict clause recording, but it might be inefficient. Net restarting, presented next, is another, more global, algorithm for changing the net ordering, based on information derived during net conflict analysis.

E. Net Restarting

Net restarting is the following simple yet effective technique for net conflict-aware net reordering.

We associate a *conflict counter* with each net which is increased whenever the net becomes conflicting in a net conflict. Once the counter reaches a user-given threshold T (10 by default) for some N_i, DRouter restarts and places i before all the other nets in *nids*, so as to start routing N_i right after the restart. The conflict counters for all nets are set to 0 following a net restart.

Assume that N_0 is the first net in our example in Fig. 1a. Applying net restarting alone (without net swapping) will solve the problem after T net conflicts by restarting, placing N_1 before N_0, and routing.

Sect. VI will demonstrate that combining net swapping and net restarting yields the best results in practice.

V. Comparing DRouter to Monosat

Monosat [3] is a recent solver that can reason about a BV formula F and various graph predicates, given one or more graphs sharing edges with F. In particular, Monosat can reason about graph reachability predicates, where a reachability predicate *reach(v,u)* holds iff vertex v is connected to vertex u through active edges (that is, edges assigned 1).

The RUC problem can easily be modeled in Monosat as follows. First, the BV formula F and the graph G are provided as input to Monosat. Second, a reachability predicate *reach*(t_0^k, t_i^k) is created and globally asserted for each terminal t_i^k for $i > 0$ for each net N_k. That guarantees that each net N_k is routed. Third, a reachability predicate *reach*(t_0^k, t_0^l) is created and its *negation* $\neg$*reach*(t_0^k, t_0^l) is globally asserted for each pair of the first terminals t_0^k and t_0^l for each pair of nets N_k and N_l for $k < l$. That guarantees that all the nets are routed disjointly.

Monosat takes advantage of dedicated conflict analysis techniques for reasoning about reachability predicates. It applies the Ramalingam-Reps incremental shortest path algorithm [15] to keep track of the status of reachability predicates. Whenever an asserted predicate *reach(v,u)* is violated, that is, whenever v and u are no longer connected through active edges, Monosat creates a conflict clause comprising the conflict cut, similarly to our DPF algorithm. Whenever an asserted negated predicate $\neg$(*reach(v,u)*) is violated, that is, whenever v and u become connected through active edges, Monosat records a conflict clause comprising the shortest path connecting v to u through active edges.

Monosat can also be configured to apply a dedicated decision heuristic for globally asserted reachablity predicates.[1] Monosat's decision heuristic connects the vertex v to u, for each asserted predicate *reach(v,u)* in the user-given order, by shortest path, using the Ramalingam-Reps algorithm, similarly to our DPF algorithm, the difference being that DPF uses the cheaper A* algorithm lazily, whereas Monosat uses the incremental Ramalingam-Reps eagerly.

Let us compare the functionality of DRouter without net swapping and net restarting to that of Monosat with the decision heuristic on the very simple routing instance in Fig. 1a.

Assume that N_0 is routed first. Both Monosat and DRouter will easily route N_0. The key difference is that DRouter will identify a net conflict immediately after N_0 is routed, since N_0 blocks N_1 (see Fig. 1c), while Monosat will start routing N_1 and discover a conflict only when the nets become connected, as shown in Fig. 1d. Then Monosat will learn a clause consisting of all the active edges in the bottom-left sub-grid $(0,0)-(5,5)$ in Fig. 1d. Monosat will have to create an exponential number of clauses to falsify any single N_0 routing blocking N_1 (including that shown in Fig. 1c), whereas DRouter falsifies any single N_0 routing at once.

Assume now that N_1 is routed first. DRouter solves such a problem instantly without any conflicts. Monosat might still encounter an exponential number of conflicts before finding a solution, since after routing N_1 it will keep trying to route N_0 using its shortest path heuristic right through N_1 routing.

For these reasons, DRouter, even without the advanced techniques of net swapping and net restarting, is expected to be considerably more efficient than Monosat for the RUC problem. Net swapping and net restarting make DRouter substantially more efficient.

All in all, unlike DRouter, Monosat is not *routing*-aware, although it is *reachability*-aware. In addition, Monosat is less optimization-aware than DRouter as Monosat neither tries to deactivate unassigned edges nor has a post-processing optimization loop (recall Sect. III-B).

VI. Experimental Results

In this section, we describe various experiments on crafted and industrial instances. For all the experiments, the run-time is measured in seconds, the memory is measured in Gb, and "TO" stands for time-out.

A. Crafted Instances

This section presents experiments with crafted instances. Detailed results and all the benchmarks are publicly available at [14]. We used Intel® Xeon® CPU E3-1270 v3 machines with 32Gb of memory and 3.50GHz frequency. We set the time-out to 20 min. The following RUC solvers were used: *a*) DR: shortcut for DRouter, *b*) DR-S: DRouter without net swapping, *c*) DR-R: DRouter without net restarting, *d*) DR-SR: DRouter with neither net swapping nor net restarting, *e*) Mn: shortcut for default Monosat (version 1.2.0), *f*) Mn+D: Monosat with the decision heuristic for reachability (*-decide-theories* switch is applied), *g*) BV: Sect. II-D's reduction of RUC to BV and application of Intel's eager BV solver Hazel.

[1]The decision heuristic is not mentioned at all in the conference paper [3] and is only briefly mentioned in the paper's extended version [2]. We are grateful to the first author of [3] for sharing the details in private communication.

Size	First Net	DR	DR-S	DR-SR	DR-R	BV	Mn	Mn+D
10	N_0	0	0	0	0	0	55	TO
1000	N_0	25	31	TO	26	TO	TO	TO
10	N_1	0	0	0	0	0	222	TO
1000	N_1	25	25	25	25	TO	TO	TO

TABLE I: Run-time comparison on several crafted instances.

1) Basic Comparison: The goal of our first experiment is to confirm the conclusion of Sect. V that DRouter should scale much better than Monosat for RUC even on very simple instances without any constraints. To that end, we created two benchmarks comprising the RUC instance in Fig. 1 for the two different possible initial net orderings. We also created two instances for two net orderings for a larger benchmark, structurally similar to that in Fig. 1, comprising a 1000×1000 grid with two nets: $N_0 = \{(0,500),(999,500)\}$ and $N_1 = \{(500,0),(500,998)\}$. The results appear in Table I ("First Net" stands for the first net in the net ordering).

Monosat with the decision heuristic (Mn+D) cannot solve a single instance, whereas DRouter with either net swapping or net restarting (or both) enabled instantly solves both 10×10 instances and easily solves both 1000×1000 instances. This result confirms our analysis in Sect. V.

Interestingly, DRouter without net swapping and net restarting (DR-SR) can easily solve the 10×10 instances, but can solve the 1000×1000 benchmark only when N_0 is routed first, that is, when there are no net conflicts. This is because when N_1 is routed first, there are too many conflict clauses to record for DR-SR.

Default Monosat can solve the 10×10 benchmark, but it is substantially slower than the other solvers. Hence, it comes as no surprise that default Monosat cannot solve the 1000×1000 benchmark.

2) Extended Comparison: We now present experimental results on RUC benchmarks crafted as follows (where $N = 20$):

```
1: for all $M \in \{3,5,7\}$ do
2:     for all $C \in \{0,10,20,30\}$ do
3:         Generate a solid grid graph having $N * M$ rows and columns
4:         Create N 2-terminal nets, with randomly picked terminals
5:         Let $V = (N * M)^2$ be the number of vertices. Generate $C/100 * V$ binary clauses as follows. Pick a random vertex $v = (x, y)$ and another random vertex $u$ sharing either $x$ or $y$ coordinate with $V$. Add the clause $\neg v \vee \neg u$.
```

Note that M regulates the grid size and C regulates the number of generated clauses. We created 10 instances for each $M \times C$ combination.

Consider Table II. DRouter is the only solver able to solve all the instances. Monosat in either mode, BV, and

M	C	DR	DR-S	DR-SR	DR-R	BV	Mn	Mn+D
3	0	10	10	0	0	0	0	0
3	10	10	0	0	10	0	0	0
3	20	10	0	0	0	0	0	0
3	30	10	0	0	0	0	0	0
5	0	10	10	0	0	0	0	0
5	10	10	10	0	0	0	0	0
5	20	10	10	0	0	0	0	0
5	30	10	10	0	10	0	0	0
7	0	10	10	0	0	0	0	0
7	10	10	0	0	0	0	0	0
7	20	10	0	0	0	0	0	0
7	30	10	0	0	0	0	0	0

TABLE II: Comparison of the number of solved crafted instances.

Area in μm^2	Nets	Vertices	Constraints	Time	Memory
24	110	42,456	484,008	25	0.7
24	230	42,456	484,008	391	1
32	352	63,740	667,764	705	2.2
129	788	127,480	2,669,056	14,733	6.5
129	891	127,480	2,669,056	92,950	6.5

TABLE III: `DRouter` performance on industrial instances. "Constraints" represent the number of design rule applications.

`DR-SR` cannot solve a single instance. Some instances are solved solely with net swapping and some others are solved solely with net restarting, but only their combination renders `DRouter` scalable.

B. Industrial Instances

This section shows that `DRouter` can scale to large clips from Intel designs which could not be routed by two modern industrial routers without violating design rules.

Consider Table III. We used Intel® Xeon® CPU E7-4870 machines with 2.40GHz frequency and 528Gb of memory. `DRouter` solves large industrial instances having hundreds of nets and up to millions of design rule applications, where the number of vertices reaches into the hundreds of thousands.

Two industrial routers we tested failed to route these clips. First, a typical heuristic router was only able to find routings that violated some of the rules. Second, an incomplete router based on enumerating some of the potential solutions and then picking an actual solution out of the potential ones using a SAT solver [16], failed to route these clips due to memory-outs (despite the machines' having as much as 528Gb of memory).

These results demonstrate that `DRouter` gives clear added value in industrial settings.

VII. Conclusion

This paper proposed a formal model for the problem of design-rule-aware routing. Our model combines graph theory (for representing the routing problem) and bit-vector logic (for representing applications of the design rules). We introduced a solver for the resulting problem, called `DRouter`. Essentially, `DRouter` implements an A*-based router inside a SAT solver, overriding the solver's decision and restart strategies and enhancing its learning with routing-aware algorithms. We demonstrated that `DRouter` has substantially better capacity than either plain reduction to bit-vector reasoning or the `Monosat` solver. Furthermore, we showed that `DRouter` can route large clips from Intel designs while obeying up to millions of design rule applications–a task two industrial routers failed to accomplish.

VIII. Acknowledgments

We are grateful to Kostas Malinauskas for carrying out the experiments, presented in Sect. VI-B. We thank Suto Gyuszi, Nina Lane and Kostas Malinauskas for many useful discussions. We are grateful to Sam Bayless for his essential help with `Monosat` and to Paul Inbar for editing the paper.

References

[1] N. Abboud, M. Grötschel, and T. Koch. Mathematical methods for physical layout of printed circuit boards: an overview. *OR Spectrum*, 30(3):453–468, 2008.

[2] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. *CoRR*, abs/1406.0043, 2014.

[3] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3702–3709. AAAI Press, 2015.

[4] S. Chopra. Comparison of formulations and a heuristic for packing steiner trees in a graph. *Annals of Operations Research*, 50(1):143–171, 1994.

[5] N. Dershowitz and A. Nadel. Is bit-vector reasoning as hard as nexptime in practice? In *13th International Workshop on Satisfiability Modulo Theories*, 2015.

[6] S. Devadas. Optimal layout via boolean satisfiability. In *1989 IEEE International Conference on Computer-Aided Design, ICCAD 1989, Santa Clara, CA, USA, November 5-9, 1989. Digest of Technical Papers*, pages 294–297. IEEE, 1989.

[7] A. Erez and A. Nadel. Finding bounded path in graph using SMT for automatic clock routing. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2015.

[8] P. J. Esrom. Combinatorial algorithms for integrated circuit layout. *Robotica*, 9(1):118, 1991.

[9] M. Grötschel, A. Martin, and R. Weismantel. The steiner tree packing problem in VLSI design. *Math. Program.*, 77:265–281, 1997.

[10] L. Hadarean. *An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories*. Dissertation, New York University, 2015.

[11] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[12] G. Kovásznai, A. Fröhlich, and A. Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In P. Fontaine and A. Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series*, pages 44–56. EasyChair, 2012.

[13] M. Kramer and J. van Leeuwen. The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. *Advances in computing research*, 2:129–146, 1984.

[14] A. Nadel. Routing under constraints: Benchmarks and detailed results. https://goo.gl/OUXido.

[15] G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[16] N. Ryzhenko and S. Burns. Standard cell routing via boolean satisfiability. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 603–612. ACM, 2012.

[17] N. A. Sherwani. *Algorithms for VLSI physical design automation.* Kluwer, 3 edition, November 1998.

[18] B. Taylor and L. T. Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 344–349. IEEE, 2007.

[19] R. G. Wood and R. A. Rutenbar. FPGA routing and routability estimation via boolean satisfiability. *IEEE Trans. VLSI Syst.*, 6(2):222–231, 1998.

A Consistency Checker for Memory Subsystem Traces

Matthew Naylor, Simon W. Moore, Alan Mujumdar
Computer Laboratory, University of Cambridge, UK
{matthew.naylor, simon.moore, alan.mujumdar}@cl.cam.ac.uk

Abstract—**Verifying the memory subsystem in a modern shared-memory multiprocessor is a big challenge. Optimized implementations are highly sophisticated, yet must provide subtle consistency and liveness guarantees for the correct execution of concurrent programs. We present a tool that supports efficient specification-based testing of the memory subsystem against a range of formally specified consistency models. Our tool operates directly on the memory subsystem interface, promoting a compositional approach to system-on-chip verification, and can be used to search for simple failure cases – assisting rapid debug. It has recently been incorporated into the development flows of two open-source implementations – Berkeley's Rocket Chip (RISC-V) and Cambridge's BERI (MIPS) – where it has uncovered a number of serious bugs.**

I. Introduction

We are interested in verifying that the memory subsystem in a shared-memory multiprocessor implements a well-defined consistency model – a pre-requisite for the correct execution of concurrent programs on such architectures [1]. We take a *specification-based testing approach* inspired by the work of Manovit et al. [2], [3], [4], [5] and their *TSOtool* [6]. TSOtool generates pseudo-random multi-threaded programs, runs them on a multiprocessor, and compares the results against the *Total Store Order* specification (TSO) to reveal potential discrepancies. A key contribution of the work is a state-of-the-art conformance-checking algorithm for TSO that can handle long-running programs – on the order of millions of memory operations and hundreds of cores – *despite this being an NP-complete problem* [7]. TSOtool, and variants of it, have been used with great success at Sun Microsystems [2] and Intel [8].

In this paper, we both build on and deviate from TSOtool in a number of useful ways, as outlined below.

Testing the memory subsystem in isolation Unlike TSOtool, we feed memory requests directly to the memory subsystem using an HDL-level test bench, not via software running on processors connected to the memory subsystem. As a result:

- The memory subsystem can be tested as a reusable component, not constrained to the usage pattern of any particular processor implementation.
- Greater stress can be applied to the memory subsystem directly than may be possible indirectly via software.
- It is faster to simulate the memory subsystem in the absence of processor pipelines, allowing more tests per unit time.
- We avoid the implicit traffic arising from execution of software tests (e.g., fetching instructions, logging test results), allowing simpler failure cases to be found.

	Model	Name & Reference
	SC	Sequential Consistency [10]
$\subset$	TSO	Total Store Order [11]
$\subset$	PSO	Partial Store Order [11]
$\subset$	WMO[1]	Weak Memory Order [11]
$\subset$	POW	POWER model [12]

Fig. 1: Consistency models supported by Axe

More consistency models We can test memory subsystems against a range of consistency models found in modern multiprocessors, not just TSO. For example, we can test Berkeley's Rocket Chip [9], which at the time of writing is intentionally more relaxed than TSO.

Our conformance-checking tool – *Axe* – supports a spectrum of five consistency models shown in Figure 1, each one permitting a subset of the behaviors allowed by the next. In this paper, we focus on support for the SPARC models (SC, TSO, PSO, WMO), which have been sufficient for the memory subsystems we have tested thus far; support for the POWER model (POW) is detailed in the Axe manual [14]. Our checking algorithm for the SPARC models is a generalization of TSOtool's algorithm. Although this generalization leads to a checker with a worse time and space complexity, we show that it still performs very well in practice.

Simpler debugging The TSOtool authors say very little about how best to report violations to the user. Simply indicating that a violation exists is clearly not very helpful when large traces are involved. However, due to the backtracking nature of the checker, it may not be easy to give a concise error message. To address this, we have developed a shrinking procedure that attempts to isolate the smallest subset of a failing trace that still violates the model.

While this procedure works very well for explaining why the model has been violated, it does not always help in understanding what went wrong in the implementation. For this, we exploit one of the great benefits of specification-based testing: we adjust the test-generation method to search for small test-cases that fail.

Open-source tools While TSOTool is a *"proprietary product of Sun Microsystems"* [6], Axe is open-source and freely-available [14], as are the applications of Axe to open-source processors Rocket Chip and BERI [9], [15].

Paper outline We begin by presenting the design and implementation of Axe. This includes the format of memory subsys-

[1]WMO is equivalent to SPARC RMO [11] except that it forbids reordering of loads *to the same address*, making it a subset of POWER [12].

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

tem traces taken as input, the consistency models supported, the checking algorithms we have implemented, performance evaluations of these algorithms, and a tool for shrinking failing traces to reveal minimal violations. After that, we present experiences of using Axe to test the memory subsystem in Berkeley's open-source Rocket Chip [9], including details of test benches developed and the bugs we found. Finally, we compare our approach against litmus testing and with other checking tools reported in the literature. This includes experiences of testing the open-source BERI processor [15], the value of searching for small failure cases, and bugs missed by Axe.

II. Axe consistency checker

Given a *memory subsystem execution trace* containing a set of top-level memory requests and responses (including loads, stores, atomic read-modify-writes, memory barriers, and optional timestamps) initiated by concurrent processor cores (or "hardware threads"), Axe determines whether the trace is valid according to one of the consistency models listed in Figure 1. Unlike some heuristic algorithms, Axe is *complete* in the sense that it will detect *any* violation of the model.

Following Gibbons [7] and Manovit [2], we assume that the address-value pair of every store in a trace is unique, i.e., the same value is never written to the same address more than once. This reduces the amount of nondeterminism in a model, because the store read by any load can be uniquely identified. The restriction is easily met by an automatic test generator, and is justified by the fact that the actual values being stored do not typically affect any interesting hardware behavior. But it does mean that our tool cannot be used for checking memory traces that arise during execution of arbitrary software applications, which are unlikely to meet this restriction.

Another technique for reducing nondeterminism is to modify the hardware to emit extra trace information such as the order in which writes reach a particular internal merge point in the memory subsystem. However, we treat the memory subsystem as a *black box*, and do not inspect or modify its internals in any way: we would like our tool to be as easy as possible to use, i.e., not requiring modifications to the system under test.

A. Syntax of memory traces

Example 1 Here is a simple Axe trace consisting of five operations running on two threads.

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1    @ 100 : 110
1: M[0] == 0    @ 115 :
```

The first number on each line denotes the hardware thread id; `M[`a`]` denotes a memory location with address a; operators == and := denote loads and stores respectively; `sync` denotes a full memory barrier; the optional timestamps beginning with `@` denote the begin and end times at which the request was sent and the response received, respectively.

The textual order of operations with the same thread id is the order in which those operations were submitted to the memory subsystem by that thread. We refer to this order as the *thread-order*. No ordering is implied by the textual order of operations from different threads.

The initial value of every memory location is implicitly zero. For any load of a value other than zero, there must exist a write of that value to the same address in the trace, otherwise the trace is invalid. As explained above, we also require the address-value pair of every write to be unique.

Load operations will typically contain two timestamps, since they involve both a request and a response. Axe currently forbids response timestamps on store operations, making it clear that this information is not used by any of the supported models. All timestamps are completely optional, for a few reasons:

1) Some consistency models are unaffected by timestamps.
2) Timestamps may not be available, depending on how the traces are produced.
3) Example traces are easier to read if only the interesting or relevant timestamp information is supplied.

However, in some consistency models timestamps can affect whether or not a trace is allowed. In the above example, the timestamps indicate that the first load must have finished before the second load begins, implying that the memory subsystem could not have executed the operations out of order. In the SPARC and POWER architectures, a programmer can arrange such a dependency by having the address of the second load be dependent on the result of the first – a so-called *address dependency* [12]. Other kinds of dependency include *data dependencies* (where the value of a store is dependent on the result of a preceding load) and *control dependencies* (where an operation is control-flow dependent on the result of preceding load). These program-level dependencies become observable in the memory trace as end-time-before-begin-time dependencies.

For the SPARC models, Axe considers timestamps to be *local to each thread*, i.e., it does not use timestamps to infer ordering between operations that run on different threads.

There is no explicit support in Axe for *canceled operations*, which often arise in modern CPUs due to speculative execution or exceptions. Traces containing such operations can still be checked by simply replacing them with no-ops. There is also no support for *mixed-width* accesses at present: Axe abstracts over the width of each memory location, and hence the width may vary between traces – but *not within a trace*.

Example 2 Here is another trace, this time containing three operations, the first of which is an atomic read-modify-write.

```
0: <M[0] == 0; M[0] := 1>
1: M[0] := 2
1: M[0] == 1
```

The first line can be read as thread 0 *atomically* reads value 0 from memory location 0 and updates it to value 1. The two memory addresses in an atomic operation must be the same, otherwise the trace is invalid. In the future, it may be desirable to generalize read-modify-write (RMW) to allow any number of operations on any number of addresses, i.e., transactional memory [23].

A common way to express atomic operations in RISC instruction sets is via a pair of *load-linked* and *store-conditional* operations. At the trace level, it is straightforward to convert such a pair into a single read-modify-write:

- If the store-conditional fails, then remove it from the trace and convert the load-linked to a standard load.
- Otherwise, convert both operations to a single read-modify-write operation.

For read-modify-write operations, the response timestamp simply denotes the time at which the read-response is received.

B. Consistency models

We now introduce the supported consistency models by example; a full operational semantics for each model is available in the Axe manual [14]. Lamport's *sequential consistency* [10] is the strongest supported model; it requires that there exists a sequential interleaving of each thread's operations satisfying the trace.

Example 3 (SB) Here is a trace, known as the "store buffer" (SB) trace, that is forbidden by sequential consistency.

```
0: M[1] := 1
0: M[0] == 0
1: M[0] := 1
1: M[1] == 0
```

There are six possible interleavings of each thread's operations and none result in *both* reads returning zero. However, under TSO [11], stores may be buffered locally by a thread, allowing subsequent loads to complete before the buffered stores can be observed globally.

Example 4 (SB+syncs) Under all consistency models, the above behavior can be prevented by inserting a `sync` after the store on each thread; `sync` has the effect of flushing the store buffer of the calling thread. Such memory barriers are necessary to implement Peterson's mutual exclusion algorithm [20], for example.

Example 5 (SB+RMWs) Under TSO, another way to prevent the SB behavior is to replace each write with an atomic RMW, which has the side-effect of flushing the store buffer.

```
0: <M[1] == 0; M[1] := 1>
0: M[0] == 0
1: <M[0] == 0; M[0] := 1>
1: M[1] == 0
```

Example 6 (MP) The following "message passing" trace is forbidden under both sequential consistency and TSO.

```
0: M[0] := 1
0: M[1] := 1
1: M[1] == 1
1: M[0] == 0
```

However, under PSO [11], this is allowed: buffered stores (to different addresses) can be evicted out-of-order. Hence, the second store can be observed globally before the first.

Example 7 (MP+sync) Under PSO, the above behavior can be disallowed by inserting a `sync` between the two stores. However, MP+sync is still allowed by WMO, which permits load buffering as well as store buffering. As a result, the first load may now be buffered and overtaken by the second as they access two different addresses.

Example 8 (MP+syncs) One way to prevent the two loads from being reordered is simply to place another `sync` between them; `sync` waits for all buffered loads to complete.

Example 9 (MP+sync+dep) Another situation in which load reordering is disallowed is when a timestamp dependency forbids it. This trace is disallowed by WMO:

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1  @ 100 : 110
1: M[0] == 0  @ 115
```

Example 10 (LB) The MP+sync example demonstrates reordering of loads, but WMO also allows reordering of a load followed by a store. The following trace is allowed by WMO.

```
0: M[0] == 1
0: M[1] := 1
1: M[1] == 1
1: M[0] := 1
```

Example 11 (LB+syncs & LB+deps) As expected, a `sync` after each load will prevent the behavior. So too will a timestamp dependency between each load and store.

In summary: TSO allows store-load reordering; PSO additionally allows store-store reordering (when the addresses differ); WMO additionally allows load-load and load-store reorderings (when the addresses differ). It is quite easy to see how all these behaviors could arise in the presence of nonblocking L1 caches: any operation that misses in the L1 cache and is buffered may be overtaken by a subsequent operation that hits. Such behavior is important for out-of-order processors, where unnecessary dependencies between operations must be avoided.

The common feature of all these models is the existence (or illusion) of a *single shared memory*: if a write by one thread is observed by another, then it must be observable to all threads. Sometimes known as *multi-copy atomicity* or *global store atomicity*, this property is provided by hardware that implements a single-writer coherence protocol such as MESI.

C. Axiomatic definitions

We now present the axiomatic definitions for the consistency models, upon which the Axe checking algorithm is based. In these definitions, we consider a read-modify-write operation to be both a "load" and a "store".

To begin, it is helpful to distinguish between two different orderings over operations in the trace:

- *Thread Order*: for any given thread, the textual order of operations in the trace issued by that thread.
- *Memory Order*: a total order over all operations.

All valid traces under these models must satisfy the following property (**value axiom**): the value returned by a load from address a equals the value of the latest store (in memory order) from the set $Local \cup Global$ where $Local$ is the set of stores to address a that precede the load in *thread order* and $Global$ is the set of stores to address a that precede the load in *memory order*.

Depending on the model, the following **local axioms** on operations i and j from the same thread must also be satisfied.

SC If i precedes j in thread-order, then i must precede j in memory order.

TSO If i precedes j in thread-order, then i must precede j in memory order when i is a load; or i and j are stores; or i is a `sync` or j is a `sync`.

PSO If i precedes j in thread-order, then i must precede j in memory order when: i is a load; or i and j are stores *to the same address*; or i is a `sync` or j is a `sync`.

WMO If i precedes j in thread-order, then i must precede j in memory order when: i is a load and j accesses the same address; or i and j are stores to the same address; or i is a `sync` or j is a `sync`; or i is a load with end-time t_0 and j has begin-time t_1 and $t_0 < t_1$.

D. Checking algorithm

In this section, we generalize an algorithm for checking traces against the TSO model to support the SC, TSO, PSO *and* WMO models. The central data structure used by this algorithm is the *analysis graph* – in which each node denotes an operation from the trace, and each edge denotes that the source node precedes the destination node in memory order.

Simple algorithm Starting with an empty analysis graph, a simple checking algorithm is as follows.

1) Add each operation in the trace as a node to the analysis graph and add the edges implied by the local axioms defined above. (Redundant edges implied by transitivity need not be added.)
2) Apply the two edge-introduction rules shown in Figure 2 to the graph.
3) Add an edge from each read `M[`x`] == 0` to the first store `M[`x`] :=` v on each thread. This ensures that any read of zero (initial value) from address x must happen before any writes to address x.
4) Apply a standard topological sort procedure to the analysis graph with the following tweak: every time a store operation `M[`x`] :=` v is removed from the graph, add an edge from each load `M[`x`] ==` v to the next unpicked store `M[`x`] :=` w on each thread. This ensures that any read of the current value at address x must happen before any store of another value to address x.
5) If a topological sort can be found (i.e., a total order of operations exists that satisfies the memory order constraints), then the trace is valid; otherwise it is invalid.

The key inefficiency of this algorithm is the nondeterminism present in the topological sort. At any stage, there may exist several store operations that can be removed next. If a bad choice is made, the algorithm must backtrack, because an

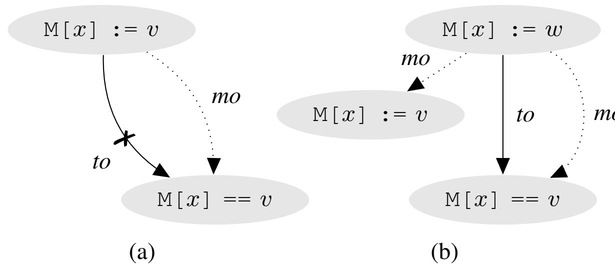


Fig. 2: Edge-introduction rules. Thread-order edges are labelled *to* and memory-order edges *mo*. In (a) the dotted edge is introduced if the solid edge *does not* exist. In (b) the dotted edges are introduced if the solid edge *does* exist and $v \neq w$.

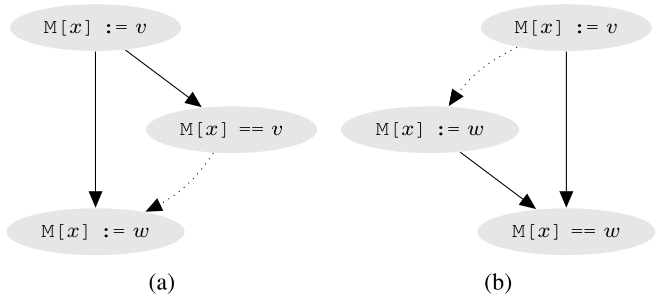


Fig. 3: Edge-inference rules proposed by Manovit [2] (our representation). All edges are memory-order edges. In each case, if the solid edges are known to exist, either directly or by transitivity, and $v \neq w$ then the dotted edge can be inferred.

alternative choice might lead to success. (The order of stores to each address is not known in advance.)

Reducing nondeterminism Manovit proposes the two rules shown in Figure 3 as a way of inferring new edges in the analysis graph, greatly reducing the amount of nondeterminism in the topological sort. Notice that applying these rules can introduce edges that enable the rules to be applied again. Therefore, it is desirable to apply the rules repeatedly until a fixed-point is reached, i.e., until no new edges are inferred.

This leads to two modifications of the simple algorithm above: first, add a new step after step (2) that applies the inference rules until a fixed-point is reached; second, every time a store is removed from the graph in step (4), and new edges are added, reapply the inference rules until a fixed-point is reached.

Reducing rule-application sites Applying the inference rules at all matching sites in the analysis graph would be extremely inefficient and, fortunately, unnecessary. Manovit shows that it is sufficient to apply each rule once for each store s of the form `M[`x`] :=` v with:

- for rule 3a, node `M[`x`] :=` w bound to the earliest store to address x that succeeds s in the analysis graph;
- for rule 3b, node `M[`x`] ==` w bound to the earliest load to address x that succeeds s in the analysis graph.

While there may exist several bindings that satisfy the above constraints (the earliest successor may not be unique in a partial order), the number of application sites to consider is greatly reduced.

Determining the earliest successors The problem now is this: starting from any store operation, how do we efficiently determine the next load and store to the same address in the analysis graph?

To answer this, we maintain two data structures. The first is the mapping $nextLoad(op, t, a)$ that gives the next load (in the analysis graph) to address a on thread t from operation op. (Since loads to the same address on a given thread are totally ordered under all models, this mapping is a function, i.e., unambiguous.) Initially, it is computed by a backward analysis, propagating the next load for each (a, t) pair backwards along the edges of the graph, in reverse topological order. At a fork point, the information at several nodes is merged by taking the minimum load in thread order for each (a, t) pair. When a new edge $i \rightarrow j$ is added to the graph, the $nextLoad$ mapping is updated by applying the same propagation method backwards from node j until no new updates are made.

The second data structure we maintain is the mapping $nextStore$, identical to $nextLoad$ but giving the next store instead of the next load. These two data structures have a number of uses:

1) The inference rules from Figure 3 can be efficiently applied. And when adding an edge, the backward-propagation method used to update the $nextLoad$ and $nextStore$ mappings will naturally visit all the nodes at which the inference rules must be reapplied.
2) The existence of a path from a store to any load or store can be determined in constant-time, avoiding the addition of redundant edges to the graph.
3) Similarly, we can be determine in constant-time whether or not the addition of an edge to the graph will lead to a cycle, allowing immediate failure detection.

Comparison to Manovit's algorithm When specializing the algorithm to the TSO model, it is possible to simplify the $nextLoad$ and $nextStore$ mappings. Instead of mapping each (op, a, t) triple to the next load or next store, it is sufficient to map each (op, t) pair. This is because all loads by the same thread are totally ordered under TSO, as are all stores by the same thread. Once the next load on some thread is determined, the next load to a particular address on that thread can be easily found by looking at the static thread order. Consequently, the size of these data structures reduces from $2 \times N \times A \times T$ for N operations, A addresses, and T threads to $2 \times N \times T$. Not only does this save space, but it makes the backward analysis faster as the amount of information being propagated is smaller. In other words, the efficiency of our checker depends on the number of different address locations used in the trace. This is not the case for Manovit's TSO-only checker.

E. Evaluation

Performance To evaluate the performance of Axe, we have generated a range of traces[2] with various numbers of

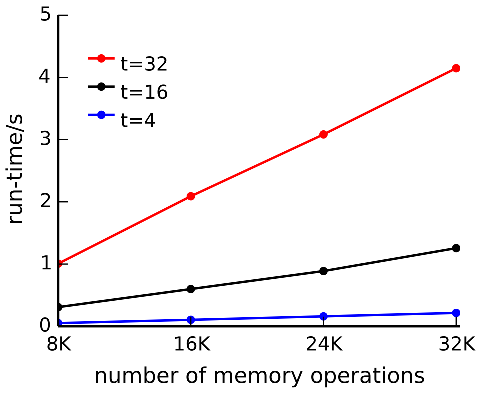


Fig. 4: Performance of the WMO checker

memory operations ($n \in \{8K, 16K, 24K, 32K\}$), threads ($t \in \{4, 16, 32\}$), and addresses ($a \in \{4, 16, 32\}$). For each combination of parameters, we generate 16 traces, giving 576 traces in total. Figure 4 shows how the performance of the WMO checker varies with the number of operations and threads present, averaged over the number of addresses present: in practice, Axe allows rapid checking of large traces which, for a fixed number of addresses and threads, scales linearly with the number of operations.

Correctness Axe has been tested for equivalence against an operational semantics for each model (defined in the Axe manual [14]) and also an axiomatic semantics for each model (defined in §II-C). The test traces include: (1) 199 litmus tests from the PPCMEM distribution [13]; and (2) 200K randomly-generated traces ranging from around 10 to 50 operations in size (distributed with the Axe tool [14]). Axe also gives the expected outcomes for all the traces used in our performance evaluation.

F. Shrinking traces

Given a trace that violates a model, we would like to find the smallest subset of the trace that still violates the model. This is very useful for debugging (§III). Our shrinker works by applying each of the following rewrite rules for $retry$ attempts before moving on to the next rule. Each rule is conditioned on the resulting trace still violating the model.

1) Pick an address and drop all accesses to that address.
2) Drop a random subset ($n\%$) of loads.
3) Drop a random subset ($n\%$) of stores which write a value that is never read.
4) Repeat (3) but for read-modify-write operations.

After that, in reverse trace order, it tries to drop each operation in turn; this is repeated until a fixed-point is reached. For suitable choices of $retry$ and n, the shrinker is both effective and fast, typically yielding fewer than ten operations and taking between a second and a minute for traces between 1K and 32K elements respectively.

III. CASE STUDY: ROCKET CHIP

Rocket Chip is an open-source system-on-chip generator developed at UC Berkeley including support for multiple pro-

[2]Using a model cache implementation with load and store buffering, out-of-order eviction, out-of-order responses, prefetching and invalidation-based coherence. These traces are available at http://dx.doi.org/10.17863/CAM.794

```
1: load-req      0x0000000008 #0 @64
1: store-req  5  0x0000100008 #1 @65
1: store-req  7  0x0000000010 #2 @66
0: store-req  2  0x0000000008 #0 @303
0: load-req      0x0000000008 #1 @304
0: store-req  6  0x0000100008 #2 @305
1: resp       0               #0 @96
0: resp       0               #0 @350
0: resp       2               #1 @351
0: load-req      0x0000000010 #3 @353
1: resp       0               #1 @149
1: load-req      0x0000000108 #3 @152
1: resp       0               #3 @184
0: resp       5               #2 @422
0: resp       0               #3 @424
1: resp       0               #2 @226
```

Fig. 5: A sample trace generated using our extensions to Rocket Chip's GroundTest framework: the first number on each line of the trace is the *thread-id*; #n denotes a *request-id* n; @t denotes a *time* t in clock cycles; hex numbers denote *addresses*; remaining decimal numbers denote *data values* being loaded or stored. This trace contains only *loads*, *stores* and *responses*, but we also support generation of *LR/SC* pairs, *atomic operations*, and *fences*. Notice that the timestamps are not monotonically increasing: in simulation, the Rocket Chip tiles are brought out of reset sequentially.

cessor cores and a cache-coherent shared-memory subsystem. Available cores include implementations of the RISC-V ISA: the in-order Rocket, the out-of-order BOOM, and the Z-scale microcontroller – and (pending release) the Hwacha vector-thread accelerator. Having been taped out 11 times between 2011 and 2015, Rocket Chip is fairly mature – but faces constant change through extensions, redesigns, and refactorings. Rocket Chip is written using the Chisel HDL [16].

The Rocket Chip developers have already recognized the importance of making HDL-level test benches for the memory subsystem: *"In order to test behaviors in our memory hierarchy which are not easy or efficient to test in software, we have designed a set of test circuits called GroundTest"* [9]. GroundTest plugs into the socket given to CPU tiles and generates various kinds of memory traffic directly to the memory subsystem, either via the L1 caches, or directly to the L2, or via DMA.

Rocket Chip is highly parameterized, including the choice of coherence protocol – which by default is MESI, at the time of writing. Since MESI guarantees at most one writer to a cache line at any time, it gives the illusion of a single shared memory – despite the reality of multiple local caches – and is thus expected to conform to one of the SPARC consistency models.

Extending GroundTest We developed a *trace generator* that plugs into the GroundTest framework. Given a random seed, it generates random memory requests from each tile, and emits a trace of events. To illustrate, Figure 5 shows an example of a generated trace.

The number of tiles, requests, and addresses used when generating a trace can all be controlled using compile-time parameters. Ideally though, the number of requests and addresses would be taken as simulation-time parameters, allowing the top-level testing script to gradually increase the sizes of traces in the hope of finding smaller failures first. Unfortunately, Chisel does not yet support a convenient way to read from external sources (such as files or environment variables) during simulation.

Converting traces to Axe format We made a simple script to convert traces emitted by the trace generator into Axe format. For example, given the sample trace from Figure 5, this conversion script yields:

```
# &M[2] == 0x0000000010
# &M[0] == 0x0000000008
# &M[3] == 0x0000000108
# &M[1] == 0x0000100008
1: M[0] == 0 @ 64:96
1: M[1] := 5 @ 65:
1: M[2] := 7 @ 66:
0: M[0] := 2 @ 303:
0: M[0] == 2 @ 304:351
0: M[1] := 6 @ 305:
0: M[2] == 0 @ 353:424
1: M[3] == 0 @ 152:184
```

Notice that lines beginning with # are treated as comments by Axe: we use these comments to record the mapping between physical addresses and addresses used by Axe.

Testing against the SC model We made a script that repeatedly: (1) generates a trace with a random seed; (2) converts the trace to Axe format; and (3) checks the trace against the chosen consistency model. Running this script, we found a 260-element trace that fails to satisfy sequential consistency. Passing this through our shrinking procedure (§II-F), we get:

```
1: M[1] := 185 @ 1921:
1: M[0] := 193 @ 1966:
0: M[0] == 193 @ 2207:2245
0: M[1] := 204 @ 2208:
0: M[1] == 185 @ 2209:2269
```

Similar to the MP example, this trace can be explained either by thread 1's stores being performed out-of-order (PSO) or thread 0's loads being performed out-of-order (WMO).

Testing against the PSO model We also found a 261-element trace that violates PSO, which after shrinking is:

```
0: M[2] == 137 @ 1825:1948
0: M[0] := 154 @ 1886:
1: M[0] == 154 @ 1689:1725
1: M[2] := 137 @ 1690:
```

Similar to the LB example, this trace can be explained by the load and store on thread 0 (or 1) being reordered (WMO).

Coherence bug We observed that a large number of traces satisfy the WMO model, but eventually we hit a 260-element counterexample – which after shrinking is:

```
0: M[2] := 46 @ 497:
1: M[2] == 46 @ 280:513
1: M[2] := 61 @ 729:
1: M[2] == 46 @ 854:979
```

Note that the write of `M[2] := 46` by core 0 is the only write of 46 in the entire trace (the trace generator ensures that all write values are unique). Also, the initial value of each location is 0. Therefore, the write `M[2] := 61` by core 1 has seemingly been dropped. This is a coherence violation and undesirable: if the write of 46 to `M[2]` is interpreted as "core 1, a message is available", then core 1 might end up receiving two messages as it effectively sees the write twice. (It sees 46 once on line 2, then it clears that value with a store on line 3, and finally it sees 46 again on line 4). We reported this issue to the Rocket Chip developers, who identified a race condition in the coherence protocol and fixed it within a few days.

Livelock bug For the above testing we enabled only loads and stores in the trace generator. When we enabled generation of LR/SC pairs, we found a lock-up issue in which a store-conditional would never return under some circumstances. We reported this to the Rocket Chip developers, who diagnosed the problem as a livelock issue in the coherence protocol.

Store-conditional bug With the livelock issue fixed, we found a 228-element counterexample to WMO. After shrinking it is:

```
1: M[3] := 31 @ 340:
0: { M[3] == 31;  M[3] := 178} @ 745:812
0: { M[3] == 178; M[3] := 198} @ 926:955
1: { M[3] == 178; M[3] := 59 } @ 759:761
```

Notice that the read-modify-write by thread 1 atomically changes `M[0]` from 178 to 59. Furthermore, the second read-modify-write by thread 0 atomically changes `M[0]` from 178 to 198. Of course, if these operations really were atomic, this behavior would be impossible. After investigating the raw trace emitted by the generator, we noticed this issue arises when a store-conditional is issued before a load-reserve response is received. We reported this issue to the Rocket Chip developers, who identified it as a bug in which a cache line is not marked as dirty when it should be.

Testing against the WMO model At the time of writing (with loads, stores, LR/SC pairs, atomics, and fences all being generated), Rocket Chip satisfies the WMO model on thousands of large traces, each comprising 64K operations, 16 addresses, and 8 threads.

Liveness A key limitation of the above specification-based testing approach is that it does not check for liveness, e.g., that a store-conditional operation actually succeeds when it should. In response, we added a mode to the trace generator in which it will generate only LR/SC pairs that are expected to succeed. This is possible in Rocket Chip because of the way it implements LR/SC: the L1 cache will hold on to a cache line for a maximum of n cycles after an LR response. Thus, provided the LR and SC are within n cycles of each other, the SC should succeed. In this mode, we observed an LR/SC success rate of 94%. The 6% of failures remain unexplained and we plan to explore this in future work.

IV. Comparisons with related work

A. Litmus testing

Litmus testing is a method of determining whether or not specific memory behaviors are observable in a multiprocessor implementation [17], [18]. Behaviors are captured by litmus tests – small concurrent program-fragments with pre- and post-conditions. To a first approximation, a litmus testing tool works by repeatedly: (1) establishing the test's pre-condition; (2) synchronizing all threads; (3) running the test; and (4) recording the value of the post condition. Slight variations are introduced on each iteration – for example by changing the addresses of the shared variables used, by inserting random delays, or by simply relying on random perturbations due to context switching and other OS activities.

```
{ x=0; 0:r2=x; 1:r2=x; }
   P0             | P1 ;
   ll  r1, 0(r2) | ll  r1, 0(r2) ;
   add r1, r1, 1 | add r1, r1, 1 ;
   sc  r1, 0(r2) | sc  r1, 0(r2) ;
exists (0:r1=0 /\ 1:r1=0)
```

Fig. 6: A litmus test for MIPS in which two threads attempt to concurrently increment a shared variable `x` using load-linked and store-conditional operations. The test looks for the case where *both* store-conditionals fail – a potential liveness bug.

In our efforts to verify the memory subsystem of the BERI multiprocessor [15], [19], we have found litmus testing to be complementary to our Axe-based approach, which does not cover liveness properties. For example, the litmus test shown in Figure 6 caught a serious liveness bug in BERI to which Axe was oblivious. Unlike in Rocket Chip, it is not possible in BERI to capture static conditions under which a store conditional is expected to succeed: concurrent LL/SC accesses to the same address are resolved by a race, and whoever wins invalidates the others. But regardless of who wins, there should exist a winner. The litmus test was able to disprove this by showing a case where all store-conditionals fail. This was due to a bug in which even a failing store-conditional would invalidate the load-linked reservations of other threads.

Litmus testing, as described in [18], is a whole-system approach, covering the processor pipeline, the memory subsystem, and even the compiler. This strength is also a weakness when it comes to modular reasoning and debugging. For example, our test framework for BERI – which employs Axe alongside techniques for finding simple failures [19] – can find a counterexample to sequential consistency containing just five operations. Litmus testing can require hundreds of iterations, with hundreds of memory accesses per iteration, to expose the same behavior. Using the smaller counterexample, it is fat easier to manually trace through the internal hardware state transitions to understand *why* the behavior is occurring.

Another problem with a whole-system approach is that a largely complete and largely working SoC is required before testing can be attempted. The complex software mechanism used to synchronize all threads at the beginning of each litmus test iteration is, on its own, a very demanding test. In contrast, our approach allows incremental development, starting out with plain load and store requests, and later moving to fences and atomics – all without the need for a CPU pipeline, a compiler, or an OS.

Finally, litmus tests consider specific – not arbitrary – sequences of memory operations, and we developed Axe to support full specification-based testing.

B. Intel's checker

We are not the first to generalize TSOtool's checking algorithm to a wider range of consistency models. Intel has incorporated a variant of the algorithm into their MP RIT (multiprocessor random instruction test) framework [8], with support for any consistency model that provides global store atomicity. This means that Intel's algorithm is more general than ours. However, this extra generality comes at a cost, and its benefit is not clear cut.

Cost Intel's algorithm represents the analysis graph as an adjacency matrix and maintains the full transitive closure. Axe exploits the fact that, in WMO, loads to the same address on each thread are totally ordered, as are stores to the same address. This means we don't need to track all successors for each node; we need only track the nearest successor of each node for each (address,thread) pair. The extra cost of Intel's algorithm is apparent in their performance graph: despite parallelizing the checker over multiple cores, a polynomial growth in execution time is observed for traces up to just 8K operations for a fixed 8 threads.

Benefit The benefit of the increased generality over WMO is unclear. Both WMO and Intel's checker require global store atomicity. WMO additionally requires sequential-consistency-per-location, but this property is provided by almost all CPUs [24].

Unlike Axe, Intel's checker is *incomplete*, i.e., there are some model violations that the tool will inherently miss. And as with TSOtool, Intel's checker is not publicly available.

V. Conclusions

We have generalized a state-of-the-art TSO conformance-checking algorithm to support a wider range of consistency models increasingly being found in modern hardware. Although the generalized algorithm has worse time and space complexity – now dependent on the number of distinct memory locations that are accessed – it still performs very well in practice. Using it, we have been able to test the memory subsystem of Berkeley's Rocket Chip, which supports load buffering and out-of-order responses and is therefore intentionally more relaxed than TSO. This testing has uncovered a number of serious memory consistency bugs that have been reported to the Rocket Chip developers in a clear and concise manner using our trace shrinking procedure. In contrast to whole-system approaches, we have focused on testing the memory subsystem as a reusable component that can be understood in isolation and verified incrementally. This not only permits testing at much earlier stage in the development process, but also leads to simpler failure cases. Axe is now part of the standard test infrastructure for both the BERI and Rocket Chip open-source processors.

Acknowledgements Many thanks to Henry Cook, Howard Mao, Peter Neumann, Peter Sewell, Andrew Waterman, Jon Woodruff, and the anonymous reviewers. This work was supported by DARPA/AFRL contracts FA8750-10-C-0237 (CTSRD) and FA8750-11-C-0249 (MRC2), and EPSRC grant EP/K008528/1 (REMS). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Open access Research data supporting this paper can be obtained from `http://dx.doi.org/10.17863/CAM.794`. This includes all the sample traces used to test and evaluate the performance of Axe, and a snapshot of the Axe source code taken in July 2016. However, the latest version of the Axe source code should always be obtained from `https://github.com/CTSRD-CHERI/axe`.

References

[1] S. Adve and K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial*, Computer Journal, volume 29, number 12, pp. 66–76, 1996.

[2] C. Manovit. *Testing memory consistency of shared-memory multiprocessors*, PhD thesis, Stanford University, 2006.

[3] S. Hangal, D. Vahia, C. Manovit, and JY. J. Lu, *TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model*, in ISCA 2004, pp. 114.

[4] C. Manovit and S. Hangal, *Efficient algorithms for verifying memory consistency*, in SPAA 2005, pp. 245–252.

[5] C. Manovit and S. Hangal, *Completely verifying memory consistency of test program executions*, in HPCA 2006, pp. 166–175.

[6] *Homepage of TSOTool*, a program for verifying memory systems using the memory consistency model, http://xenon.stanford.edu/~hangal/tsotool.html.

[7] P. B. Gibbons and E. Korach. *On testing cache-coherent shared memories*, in SPAA 1994, pp. 177-188.

[8] A. Roy, S. Zeisset, C. J. Fleckenstein, J. C. Huang, *Fast and Generalized Polynomial Time Memory Consistency Verification*, CAV 2006, pp. 503.

[9] K. Asanovic et al., *The Rocket Chip Generator*, Technical Report UCB/EECS-2016-17, University of California, Berkeley, 2016.

[10] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, volume 28, number 9, pp. 690–691, 1979.

[11] D. L. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*, 2003.

[12] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. *Understanding POWER Multiprocessors*, PLDI 2011, pp. 175–186.

[13] *Homepage of PPCMEM/ARMMEM*, a tool for exploring the POWER and ARM memory models, https://www.cl.cam.ac.uk/~pes20/ppcmem/.

[14] M. Naylor, S. Moore, and A Mujumdar, *Axe Manual Version 1.4*, https://github.com/CTSRD-CHERI/axe.

[15] *Homepage of the BERI processor (Bluespec Enhanced RISC Instructions)*, http://bericpu.org.

[16] J. Bachrach et al. *Chisel: Constructing Hardware in a Scala Embedded Language*, DAC 2012, pp. 1216–1225.

[17] S. Sarkar, P. Sewell, F.Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M.O. Myreen, and J. Alglave, *The Semantics of x86-CC Multiprocessor Machine Code*, in POPL 2009, pp. 379–391.

[18] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Litmus: Running Tests Against Hardware*, in TACAS 2011, pp. 41–44.

[19] M. Naylor and S. W. Moore, *A Generic Synthesisable Test Bench*, in MEMOCODE 2015, pp. 128–137.

[20] G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Information Processing Letters, vol. 12, no. 3, pp. 115-116, 1981.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*, EECS Department, University of California, Berkeley, May 2014.

[22] W. M. Collier. Reasoning about parallel architectures. Prentice-Hall, Inc., 1992.

[23] C. Manovit, S. Hangal, H. Chafi, A. McDonald, and C. Kozyrakis, K. Olukotun, *Testing Implementations of Transactional Memory*, in PACT 2006, pp. 134–143.

[24] J. Alglave, L. Maranget, and M. Tautschnig, *Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory*, in ACM TOPLAS, volume 36, number 2, 2014.

Hybrid Partial Order Reduction with Under-Approximate Dynamic Points-To and Determinacy Information

Pavel Parízek
Charles University, Faculty of Mathematics and Physics

***Abstract*—Verification techniques for concurrent systems are often based on systematic state space traversal. An important piece of such techniques is partial order reduction (POR). Many algorithms of POR have been already developed, each having specific advantages and drawbacks. For example, fully dynamic POR is very precise but it has to check every pair of visible actions to detect all interferences. Approaches involving static analysis can exploit knowledge about future behavior of program threads, but they have limited precision.**

We present a new hybrid POR algorithm that builds upon (i) dynamic POR and (ii) hybrid field access analysis that combines static analysis with data taken on-the-fly from dynamic program states. The key feature of our algorithm is usage of under-approximate dynamic points-to and determinacy information, which is gradually refined during a run of the state space traversal procedure. Knowledge of dynamic points-to sets for local variables improves precision of the field access analysis. Our experimental results show that the proposed hybrid POR achieves better performance than existing techniques on selected benchmarks, and it enables fast detection of concurrency errors.

I. Introduction

Software systems that involve multiple threads are now ubiquitous, but also prone to errors such as data races. Therefore, efficient methods for automated verification of such systems are important. Verification algorithms based on systematic state space traversal are particularly suited for this purpose, but they do not scale well because of the huge number of possible thread interleavings exhibited by any non-trivial system. An important piece of such algorithms is partial order reduction (POR), which identifies the subset of possible thread interleavings that must be explored to cover all observable behaviors of a given system, and in this way improves the performance and scalability of verification. The goal of POR is: (1) to ensure that, for each set of thread interleavings that differ only in the order of independent actions, at least one interleaving from the set is explored, and (2) to minimize the number of thread interleavings from each set that are explored — that means exactly one thread interleaving in the optimal case. To achieve this goal, POR techniques create non-deterministic thread scheduling choices only at visible actions that represent possible interference between threads.

We distinguish between *visible* actions, which read or modify the global state reachable by multiple threads, and *thread-local* actions. Visible actions are, for example, accesses to fields of heap objects and thread synchronization operations. Only a subset of visible actions is responsible for the actual communication between threads — we call such actions *interfering*. All other actions are *independent*. The state space traveral procedure with POR has to explore all possible interleavings of interfering actions. The main challenge is to identify the interfering actions as precisely as possible.

A prominent example of a POR technique is the dynamic approach by Flanagan and Godefroid [5]. Their algorithm explores individual execution traces (interleavings) one by one using dynamic analysis, and for each trace determines the set of heap objects and fields that were truly accessed by multiple threads. An advantage of dynamic POR is that it recognizes each dynamic heap object, and thus identifies shared memory locations precisely. New thread choices are created retroactively only at accesses to shared locations, and every added choice yields traces that must be explored eventually. On the other hand, a limitation of this approach to dynamic POR is that it has to (i) explore each trace until the end state, (ii) keep track of all accesses to object fields that occurred on the trace, and (iii) check every pair of visible actions to detect all interferences (i.e., to compute the independence relation). This can negatively impact performance especially in the case of programs with large state spaces and long execution traces.

Another viable approach to POR is to use a hybrid analysis of field accesses [12] [13], which consists of two phases — static and dynamic. Static analysis computes only partial data for each program point (i.e., a source code location). Full results are generated and applied on-the-fly during the state space traversal based on the knowledge of dynamic program states. This approach determines an over-approximate set of interfering actions, and it is directly compatible with state matching. On the other hand, it has limited precision because it uses a static pointer analysis that cannot distinguish among dynamic heap objects allocated at the same code location.

We present a new hybrid POR algorithm that builds upon dynamic POR [5] and the hybrid field access analysis [12], combining their advantages and addressing their limitations. The main idea behind the proposed algorithm is the usage of under-approximate dynamic points-to and determinacy information for local variables, which is iteratively refined. We use the definition of determinacy by Schaefer et al. [14], which intuitively says that a given variable is *determinate* at a particular code location if it always has the same value every time program execution reaches the location. An informal

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

overview of the hybrid POR algorithm follows.

The state space traversal procedure augmented with our hybrid POR works in a similar way to dynamic POR. It starts by exploring an arbitrary execution trace, and on-the-fly inserts new thread choices at field accesses that occur on the trace and are deemed to be interfering. Two accesses to the same field f are *interfering* if (i) they are performed by different threads, (ii) they may target the same dynamic heap object, and (iii) at least one of them is a write. Like in the dynamic POR, every new choice corresponds to additional thread interleavings that must be explored eventually. Another important feature of the proposed algorithm is compatibility with state matching, which is needed to support cyclic state spaces.

Hybrid POR recognizes interfering field accesses based on (i) the results of the hybrid field access analysis and (ii) the dynamic points-to and determinacy information. Execution traces are processed by dynamic analysis that tracks field accesses on heap objects. When processing a field access, our hybrid POR algorithm performs the following three steps:

1) It retrieves the dynamic concrete value of the local variable through which the field access was performed, and updates the points-to and determinacy information to reflect the concrete value (i.e., a dynamic heap object).
2) Then it queries the hybrid field access analysis and the dynamic points-to information to find whether there may be a future access to the same field on the same dynamic heap object — in that case, the current field access is marked as interfering with the future one and a new thread choice is created.
3) Finally, for every previous field access on the current trace, our algorithm checks whether the previous access may be interfering with the current field access according to the updated points-to and determinacy information — additional thread choices may be created in this way.

We illustrate the main steps of hybrid POR using the program in Figure 1. It involves two threads that perform field accesses on shared objects. Let us assume that the execution trace read o.f ; write p.g ; read q.h ; write o.f is explored first. Hybrid POR detects interference between the read access to o.f in thread T_1 and the subsequent write in T_2 just before execution of the write, when the dynamic value of the variable o is added to its points-to set and previous field accesses on the trace are inspected. A new thread choice is added at the read access in T_1. The second explored trace is read q.h ; write o.f ; read o.f ; write p.g. In this case, the interference is discovered already before the write to o.f, which precedes all other accesses to o.f on this trace, because the points-to sets of all variables are already non-empty and thus the hybrid analysis can identify the future interfering read access.

The key feature of our algorithm is that the initial under-approximation of points-to and determinacy information is very coarse — every variable of a reference type is assumed to be determinate and to have an empty points-to set. However, as more and more execution traces are explored during the state space traversal, the under-approximation is gradually refined to cover the possible behavior of a program under different schedules. The dynamically computed points-to set and determinacy status for each reference variable enables the hybrid field access analysis to provide precise information about the future behavior of each thread, and that in turn enables the hybrid POR to detect real interference between field accesses in different threads very precisely.

```
   T1                  T2
read o.f ;          read q.h ;
write p.g           write o.f
```

Fig. 1. Example program

A run of the state space traversal procedure terminates when there are no unexplored thread choices and interleavings left. Then, iterative refinement of the points-to and determinacy information must have reached a fixed point, and results of the hybrid analysis together with the dynamic points-to information soundly over-approximate the set of field accesses that may occur during the program execution under any thread schedule. State space traversal with hybrid POR then covers all interleavings of interfering actions. However, in general, termination of the procedure is not guaranteed.

In addition to the limitations of existing approaches mentioned above, our rationale behind the hybrid POR algorithm is based on partially-automated inspection of benchmark programs (Section IV), where we analyzed the determinacy status of variables through which field accesses are performed. We discovered that, for many of the programs, there is a quite high number (over 50%) of cases where such variables are determinate at code locations that correspond to field accesses. Our goal was to exploit this observation to optimize verification of multithreaded programs, and also to enable faster detection of real errors. Results of our experiments show that usage of precise dynamic points-to sets and determinacy information (1) eliminates many redundant thread choices and (2) improves performance especially for large programs.

The rest of the paper is organized as follows. We provide background definitions and an overview of the hybrid field access analysis in Section II. Then, in Section III, we formally define the hybrid POR algorithm and prove its soundness. Section IV contains results of experiments with our implementation and their discussion. We compare the proposed approach with related work in Section V, and then we conclude.

II. Background

Program and state space. A program P consists of threads $t_1, \ldots, t_n$ from the set T, where each thread executes actions from the set A. Each action $a \in A$ corresponds to a program statement. Each state of the program is a snapshot of all variables, heap objects, and threads at some point during its execution. An atomic transition tr between two states is a pair $tr = (t, [a_0, \ldots, a_n])$ of a thread $t \in T$ and a sequence of actions executed by t. The first action a_0 in the sequence is interfering and others must be independent. There can be just one interfering action in any transition because a new thread choice is created when the action to be executed next is interfering. We assume that states are explicitly saved only at transition boundaries, and therefore each visible state s is associated with a choice ch over all threads runnable in s. An execution trace e is a sequence $(t_{i_0}, a_0), \ldots, (t_{i_n}, a_n)$ of thread-action pairs. Each trace represents one thread interleaving. Such definition of execution traces allows us to identify, for each field access action a, the specific thread that executed the action a — this information is needed to detect interference. We also assume that the only source of non-determinism in the program state space are thread scheduling choices at interfering actions. Input data must be specified explicitly in the source code.

Determinacy. In this paper, we extend the definition of determinacy from [14], which applies only to sequential programs, towards multiple threads. A variable v is determinate at a program point p in the context of thread t, if v always has the same value every time (1) execution reaches the point p and (2) the thread t is active. Only the values assigned to v in the scope of thread t are considered when the determinacy status of v with respect to p and t has to be updated.

Hybrid field access analysis. The hybrid analysis of field accesses was introduced by Parízek and Lhoták [12]. It combines static analysis with information taken on-the-fly from dynamic program states. For each dynamic state s reached during the traversal, and for each thread t in s, it computes an over-approximation of the set of object fields possibly accessed by t in the future on any execution path starting in s. Results of the hybrid analysis are computed in two phases.

The first phase involves static analysis, which is run before the state space traversal and computes only partial results. We use a backward flow-sensitive and context-insensitive inter-procedural data-flow analysis. For each point p in the code of thread t, it provides information that cover the future behavior of t only between the point p and return from the method containing p (including nested method calls transitively).

The second phase is performed on-the-fly during the state space traversal. Full results of the hybrid analysis are generated on-demand, every time POR has to decide whether the current field access is interfering with some other action. It is done based on the knowledge of the dynamic call stack of each thread. Let s be the current dynamic state just before execution of a field access. The dynamic call stack of a thread t specifies a sequence $p_0, p_1, \ldots, p_n$ of program points, where $p = p_0$ is the current program counter of thread t (in the top stack frame), and $p_i, i > 0$ is the point from which execution of t would continue after return from the method associated with the previous stack frame. When the hybrid analysis is queried about the current point p of thread t in state s, it takes data computed by the static analysis for each point $p_i, i = 0, \ldots, n$ on the dynamic call stack of t and merges all the data to get the complete result for p in the context of the state s. The result covers the future behavior of t after the point p, and also the behavior of all child threads of t started after p.

A consequence of the usage of dynamic call stack is that results of the whole hybrid analysis are fully context-sensitive and therefore very precise. On the other hand, the results are always valid only for the current dynamic state.

```
1  init : $visited = \varnothing$ ; $pointsto = \varnothing$ ; $determinacy = \varnothing$
2  exploreState($s_0, ch_0$, [ ], $\varnothing$)
3
4  procedure exploreState($s, ch, accs, hbo$)
5    if $s \in visited$ return
6    $visited = visited \cup s$
7    for $t \in$ getRunnableThreads($ch$) do
8      ($s', accs', hbo'$) = executeTransition($s, t, accs, hbo$)
9      $ch'$ = createThreadChoice($s'$)
10     exploreState($s', ch', accs', hbo'$)
11   end for
12
13 procedure executeTransition($s, t_c, accs, hbo$)
14   $a_c$ = getNextAction($t_c$) // must be interfering
15   while $a_c \neq$ null do // not at the end of thread
16     $s$ = executeAction($s, a_c, t_c$)
17     if isErrorState($s$) terminate
18     if isFieldAccess($a_c$) then
19       ($v_c, o_c, f_c, p_c$) = getFieldAccessInfo($a_c, s$)
20       $accs = accs \oplus (a_c, t_c)$
21       extendDynPointstoSet($v_c, o_c, t_c$)
22       inspectPreviousAccesses($s, a_c, t_c, accs, hbo$)
23     end if
24     $hbo$ = updateHappensBeforeOrder($hbo, a_c$)
25     $a_c$ = getNextAction($t_c$)
26     if isInterferingAction($a_c, t_c, s$) break
27   end while
28   return ($s, accs, hbo$)
29
30 procedure isInterferingAction($a_c, t_c, s$)
31   if isFieldAccess($a_c$) then
32     for $t \in$ getOtherThreads($s, t_c$) do
33       if existsFutureInterferingAccess($a_c, t_c, t, s$) then
34         return true
35   // other kinds of actions
36   return false // default
```

Fig. 2. Algorithm for state space traversal with hybrid POR

III. Hybrid POR Algorithm

Figure 2 shows the core of the algorithm for state space traversal combined with hybrid POR. Procedures that detect interfering field accesses are defined in Figure 3. We present a recursive definition of the algorithm because it allows us to explain the key aspects of hybrid POR in a simple and clear way — especially in comparison with an iterative encoding of the algorithm that is more efficient (and therefore used by our implementation) but also more intricate.

Core of the algorithm. The top-level procedure exploreState drives the state space traversal and performs state matching. When this recursive procedure is called for a state s that has not been already visited during the traversal, it retrieves all threads enabled in the choice ch associated with s (line 7) and explores the next transition for each of the threads. Traversal terminates immediately when it reaches an error state.

Three global data structures are used by the algorithm — the set of visited states, the relation *pointsto* that captures dynamic points-to sets for variables, and the relation *determinacy* that maintains the determinacy status of every variable. Information stored in these data structures is preserved across all execution traces. Reading and updating of the relations *pointsto* and *determinacy* are implemented by

```
37 procedure existsFutureInterferingAccess($a_c, t_c, t, s$)
38    $(v_c, o_c, f_c, p_c)$ = getFieldAccessInfo($a_c, s$)
39    for $a_t \in$ getFutureFieldAccesses($t, s$) do
40       if $\neg$(isInterferingAccess($a_c, a_t$) $\wedge$ $t_c \neq t$) continue
41       $(v_t, f_t, p_t)$ = getFieldAccessInfo($a_t$)
42       if isDeterminate($v_t, p_t, t$) then
43          $o_t$ = getSingleDynPointstoValue($v_t, p_t, t$)
44          if $o_c = o_t$ return true
45       else if // indeterminate variable
46          if $o_c \in$ getDynPointstoSet($v_t, p_t, t$) return true
47       end if
48    end for
49    return false
50
51 procedure inspectPreviousAccesses($s, a_c, t_c, accs, hbo$)
52    $(v_c, o_c, f_c, p_c)$ = getFieldAccessInfo($a_c, s$)
53    for $(a_t, t) \in accs$ do // previous accesses
54       if $\neg$(isInterferingAccess($a_c, a_t$) $\wedge$ $t_c \neq t$) continue
55       $(v_t, f_t, p_t)$ = getFieldAccessInfo($a_t$)
56       $pt_t$ = getDynPointstoSet($v_t, p_t, t$)
57       if $o_c \in pt_t$ $\wedge$ $\neg$isOrderedStrictly($hbo, a_t, a_c$) then
58          markInterferingAction($a_t$)
59    end for
```

Fig. 3. Procedures that recognize interfering field accesses

auxiliary procedures isDeterminate, extendDynPointstoSet, getDynPointstoSet, and getSingleDynPointstoValue. Both points-to and determinacy information are always specific to a tuple (v, p, t) of a variable v, a program point p, and a thread t. It means that, like in the case of determinacy, the dynamic points-to set for a variable v is defined only in the context of a specific program point p and thread t.

In addition, our algorithm uses the data structures named $accs$ and hbo, which hold information specific to the currently processed dynamic execution trace. The list $accs$ contains all the field access actions that were performed on the current trace before the current state, and hbo captures the happens-before ordering relation between actions. Both data structures are needed for precise identification of pairs of interfering field accesses, as we explain below in more detail.

The symbol t_c in Figures 2 and 3 represents the currently active thread. Symbols having the subscript c, such as a_c and f_c, refer to information associated with the current thread t_c or with the current field access action. Analogously, symbols having the subscript t refer to information associated with some other thread that is represented by the symbol t.

A run of the algorithm starts with empty determinacy and points-to relations (line 1) in order to satisfy the initial assumption that (i) every variable is determinate and (ii) all possibly concurrent accesses to the same field are performed through variables that have disjoint points-to sets. This initial coarse under-approximation is refined during the state space traversal, and at every moment it reflects all the actions and traces that were explored so far. The hybrid field access analysis depends on the points-to and determinacy information. Both the precision of the analysis and its coverage of possible future behavior of program threads are improved during the run of the algorithm based on the gradually refined under-approximation.

For each executed field access action a_c, the algorithm performs the following four steps (at lines 19-22):

1) Calls the auxiliary procedure getFieldAccessInfo to retrieve information about the field access: the variable v_c through which the access is performed, a dynamic heap object o_c to which v_c points in the current dynamic state s, the field name f_c, and a program point p_c.
2) Updates the list $accs$ of field accesses that were already performed on the current execution trace.
3) Adds the heap object o_c into the points-to set of the variable v_c, and updates the determinacy information for v_c based on the size of its points-to set. The variable v_c remains determinate only if the size is 0 or 1.
4) Inspects all the previous field accesses on the current trace in order to detect additional pairs of interfering actions. We provide more details about this step later.

We use two variants of the function getFieldAccessInfo. The dynamic heap object is returned as an element of the tuple only by the variant that takes the current state s as an argument.

The happens-before ordering relation is updated for each executed action (line 24). It has to reflect also synchronization actions that may block or release some thread.

A transition ends when the next action a_c to be executed in thread t_c is interfering with some other action, because then a new thread choice has to be created. The main part of the corresponding logic is implemented by the procedure isInterferingAction. For every thread other than the current one (t_c), it calls another procedure that looks for interfering future field accesses (line 33). If the action a_c is a field access, and some thread t may in the future access the same field of the same dynamic heap object, then a_c is interfering. Similar checks have to be done for all kinds of actions.

Detection of interfering field accesses. The procedure existsFutureInterferingAccess queries the hybrid field access analysis for the current program point in thread t (line 42), and inspects the results to find whether some of the possible future accesses by thread t may be interfering with the current field access action a_c. For each interfering future access, the algorithm queries the points-to set and determinacy status of the respective variable v_t at the point p_t in thread t. It has to decide whether one of the following two conditions holds.

(A) The variable v_t is determinate, and its points-to set has a single element o_t that is equal to the target object o_c of the currently processed field access a_c.
(B) The variable v_t is not determinate, which means it may point to different objects at distinct execution traces, and some element of the points-to set of v_t is equal to o_c.

If one of the conditions is true then the current field access action a_c in the active thread t_c may really interfere with the future action a_t on some thread interleaving. Note also that the condition A does not hold for all determinate variables, because the points-to set of some variable can be empty as a consequence of the initial under-approximation.

After each update of the dynamic points-to sets (line 21), where the target object o_c of the current access a_c is the newly added element, it is necessary to check all the previous field accesses on the current trace and compare them with a_c, because some new pairs of interfering actions may be discovered. This is done in the procedure $\mathrm{inspectPreviousAccesses}$, using an approach very similar to the original dynamic POR algorithm [5] (which is based on vector clocks). For each possibly intefering previous access a_t on the current trace, the procedure retrieves the dynamic points-to set pt_t for the respective variable v_t and checks presence of o_c in the set. If o_c is in pt_t and there is not a strict happens-before ordering between the field accesses in question, then the action a_t has to be marked as interfering with a_c. The corresponding thread choice will be added later during the state space traversal.

Our hybrid POR algorithm uses the happens-before ordering relation in the same way (and for the same purpose) as the original approach to dynamic POR [5]. That is, to avoid identifying some pairs of field accesses spuriously as interfering, when only a single interleaving of the actions is possible due to thread synchronization. A pair (a_i, a_j) of interfering field accesses, where $i < j$, meaning that a_i precedes a_j on the current trace, is strictly ordered according to the happens-before relation if the following two conditions hold:

1) There is an action a_k, $i < k < j$, that is in the happens-before relation with a_i or with some action following a_i.
2) Both actions a_k and a_j are executed by the same thread that is different from the thread executing a_i.

If both conditions hold, then a_i must happen strictly before a_j with respect to the ordering relation for the current trace. A thread choice at a_i can be soundly avoided because actions a_i and a_j cannot be interleaved the other way in this context.

The main benefit of the dynamic points-to and determinacy information is that hybrid POR can detect interference between field accesses very precisely with respect to (i) possible future behavior of program threads and (ii) previous accesses on the current trace. Usage of the under-approximate dynamic points-to sets enables the hybrid field access analysis to provide much more precise results than with a static pointer analysis.

A. Soundness and Termination

Theorem 1. *The proposed algorithm for state space traversal with hybrid POR terminates either (A) when it reaches an error state or (B) when all possible distinct interleavings of interfering actions in concurrent threads have been explored.*

Proof. The first condition (A) is trivally satisfied by the call of the procedure $\mathrm{isErrorState}$ at line 17 in Figure 2, so we focus on the second condition (B) in our proof. We need to consider only distinct interleavings of interfering actions, because execution traces that differ only in the order of independent actions yield equivalent observable behavior. To satisfy the condition B, our algorithm has to (1) identify all pairs of interfering field accesses, (2) add a new thread choice to every interfering action, and (3) explore all thread choices in the state space. We show in the next few paragraphs that all three tasks are performed in a sound manner by the algorithm.

For each field access a_c, all interfering actions are detected by combination of the hybrid analysis with the inspection of previous accesses. The hybrid analysis alone may fail to detect some future accesses interfering with a_c because of the under-approximate points-to sets. Let a_t be such a future access. Interference between a_c and a_t will be detected when a_t is executed, because the target dynamic object of a_t becomes known at that moment and a_c will then represent a previous access with respect to a_t. A complete dynamic information about every previous action on the current trace is available.

Regarding thread choices at interfering actions, we distinguish two cases. If the algorithm determines that the current field access action a_c in the active thread t_c is interfering with some possible future accesses, then a new choice is created at the current action (line 9 in the procedure $\mathrm{exploreState}$) and its exploration starts immediately. The process is more complicated when some previous field access a_t on the current trace is newly identified as interfering with a_c. A corresponding thread choice will be created and explored later just when the state space traversal procedure backtracks over the action a_t. We omitted the respective statements — for adding new choices retroactively to previous accesses on the current trace — from the pseudo-code in Figure 2, because this aspect of hybrid POR cannot be encoded in the recursive definition of the algorithm in a simple way.

It follows from the discussion above that a new choice is created at a field access action a iff there is some other access interfering with a. For each pair (a_i, a_j) of interfering accesses, existence of the choice guarantees that both interleavings of a_i and a_j will be explored eventually. The state space traversal procedure explores all transitions enabled at each choice ch, and therefore backtracks from a state s associated with ch only when the whole state space fragment with s as the root has been processed. Consequently, no thread interleaving will be omitted during the traversal. $\square$

IV. Evaluation

We implemented our hybrid POR algorithm in Java Pathfinder (JPF) [21], which is a framework for verification and analysis of Java programs. JPF is responsible for traversal of the program state space and for execution of Java bytecode instructions. In order to support decisions about thread choices, we created a non-standard interpreter of bytecode instructions for accesses to object fields. Our interpreter queries results of the hybrid field access analysis and the data structures maintained by the algorithm. We used the WALA library [23] for static analysis and JPF API to retrieve information from the dynamic program states. One custom listener for JPF collects the dynamic points-to sets, and another listener computes the happens-before ordering relation.

The complete source code of our implementation, together with benchmark programs and scripts needed to run all experiments, is available at http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/fmcad16.html.

The goal of our experimental evaluation was to compare the performance and scalability of hybrid POR against selected other approaches to POR. For each approach, we wanted to find how much time it takes to explore the whole state space of individual benchmarks, and how fast it can detect concurrency errors. We consider POR based on heap reachability [4] and the original dynamic POR [5], both combined with stateful traversal of the program state space. While POR based on heap reachability is supported by JPF for a long time, we created our own implementation of dynamic POR. Note that combination of the original dynamic POR with stateful search addresses the first limitation mentioned in Section I, i.e. the need to explore each trace until the end state — details are provided in Section V. We used this variant of dynamic POR in our experiments in order to perform a fair comparison, because other approaches involve state matching too. Moreover, the original dynamic POR [5] (that performs a stateless search) would not scale at all to most of our larger benchmarks, as we found at the initial stage of this research project.

Benchmarks. We performed experiments on 16 multithreaded Java programs, mostly from widely known benchmark suites (Java Grande [19], CTC [18], Inspect [16], and pjbench [22]). Other programs, such as jPapaBench [20] and Simple JBB, were used in our previous work and recent experimental studies. The smallest benchmark in our set is Prod-Cons with 130 lines of source code and 2 threads, while the most complex one is jPapaBench with 4500 lines of code and 7 threads.

Experiments. We evaluated four configurations of POR in our experiments: (1) POR based on heap reachability, (2) POR based on heap reachability together with hybrid analysis of field accesses, (3) dynamic POR with state matching, and (4) hybrid POR. In tables with results, we use a short name ”Heap Reach” for the first configuration in the list and a short name ”HR + fields” for the second configuration (which corresponds to the technique proposed by Parízek and Lhoták in [12]).

For every experiment, we report (1) the number of thread choices created by JPF during the state space traversal, which we use to assess precision of POR techniques, and (2) the total running time of JPF (with static analysis), which indicates performance and scalability of the techniques.

Table I shows results for the first set of experiments, where we configured JPF to explore the whole state space of each benchmark, i.e. we disabled the check for error states. We used the time limit of 12 hours and memory limit of 20 GB.

Error detection performance of the POR techniques is reported in Table II. For the purpose of these experiments, we selected only those benchmarks from our set that already contained some concurrency errors (e.g., race conditions). We used the time limit of 1 hour only in this case.

Discussion. In the case of complete traversal of a program state space (Table I), the results are mixed. Hybrid POR achieves better precision and performance than other approaches for 5 benchmarks out of 15 — Cache4j, Alarm Clock, RAX Extended, Rep Workers, and TSP. The biggest improvement was achieved for Cache4j, where hybrid POR is faster than the second-best configuration ”HR + fields” by the factor of 3.1. Dynamic POR achieves better precision and performance than other techniques for 2 benchmarks, Simple JBB and Linked List. For three benchmarks — CoCoME, Crypt, and SOR — hybrid POR and dynamic POR have the same precision, but dynamic POR yields better performance.

Results are not clearly in favor of one technique in the case of remaining five benchmarks. Hybrid POR is more precise than ”HR + fields” for CRE Demo and Daisy, but it has the same or worse performance. Dynamic POR achieves the best precision for CRE Demo and Elevator, but it is slower than hybrid POR in both cases. On the other hand, hybrid POR is the most precise technique for Prod-Cons, while dynamic POR is the fastest. The results for Elevator also highlight the limitations of dynamic POR that we discussed in Section I — it creates less thread choices than hybrid POR, but it is slower by the factor of 2.2.

All the POR techniques failed on jPapaBench because of: (1) a high number of field accesses at execution traces, (2) the length of transitions (JPF must interpret all instructions), and (3) the size of program states which must be processed by the state matching procedure. Over 1.5 million thread choices were created in the state space of jPapaBench until the timeout. In addition, dynamic POR run out of the time limit also for Daisy, Cache4j, and Rep Workers. The memory limit was sufficiently large for all our experiments. However, especially in the case of Daisy and jPapaBench, memory consumption was quite high and therefore a large part of the running time of JPF was spent by garbage collection.

When considering the search for errors (Table II), state space traversal with hybrid POR is faster than competing techniques for 4 benchmarks out of 7 — Elevator, jPapaBench, Rep Workers, and QSort MT. The biggest improvement by the factor of 5.7 was achieved for jPapaBench, which is the most complex benchmark in our set. Although none of the POR techniques can explore the whole state space of jPapaBench, an error state was reached quite fast with hybrid POR.

Dynamic POR detects an error faster in the case of 3 benchmarks out of 7. For one of them, Alarm Clock, dynamic POR is faster than hybrid POR but it creates more thread choices. It failed to detect any error in jPapaBench before the time limit. In the case of LinkedList, hybrid POR creates much more thread choices because of the under-approximate points-to analysis — more program states and field accesses have to be explored before the points-to analysis is refined enough to identify a data race. Results for other benchmarks nevertheless show that such ”anomaly” is quite rare.

The hybrid POR algorithm has a certain overhead, when compared to dynamic POR, because it runs the static analysis upfront and performs numerous queries of the hybrid analysis results on-the-fly. This overhead is clearly visible on smaller benchmarks, such as Prod-Cons, for which hybrid POR creates less thread choices but its total running time is higher. Just few seconds are taken by the static analysis for each benchmark.

To summarize, we have made the following two main observations based on our experimental results:

- There is not an obvious winner in the comparison between hybrid POR and dynamic POR, as each is better than the other roughly for a half of the benchmarks.

TABLE I
EXPERIMENTAL RESULTS: COMPLETE STATE SPACE TRAVERSAL

	Heap Reach		HR + fields		dynamic POR		hybrid POR	
benchmark	choices	time	choices	time	choices	time	choices	time
CRE Demo	30942	50 s	2476	9 s	2015	11 s	2086	9 s
CoCoME	81150	160 s	23880	59 s	72	3 s	72	5 s
Daisy	28436002	15405 s	6647236	4574 s	-		6028026	7787 s
Crypt	4993	3 s	9	2 s	9	1 s	9	2 s
Elevator	10167560	7617 s	2731316	1954 s	429466	1288 s	461996	585 s
Cache4j	11716552	7336 s	8615847	5613 s	-		1970110	1785 s
Simple JBB	575519	1768 s	277599	959 s	602	31 s	5648	81 s
jPapaBench	-		-		-		-	
Alarm Clock	531463	432 s	141138	117 s	109018	188 s	48166	47 s
Linked List	5919	3 s	1969	5 s	283	1 s	1422	5 s
Prod-Cons	6410	4 s	2532	6 s	592	1 s	356	4 s
RAX Extended	26346	18 s	13864	13 s	11315	125 s	3519	7 s
Rep Workers	9810966	6850 s	1653037	1264 s	-		739418	584 s
SOR	222129	122 s	86193	72 s	135	2 s	135	4 s
TSP	35273	591 s	9285	154 s	101	65 s	86	37 s

TABLE II
EXPERIMENTAL RESULTS: SEARCH FOR CONCURRENCY ERRORS

	Heap Reach		HR + fields		dynamic POR		Hybrid POR	
benchmark	choices	time	choices	time	choices	time	choices	time
Elevator	27053	12 s	9123	7 s	119797	285 s	1156	5 s
jPapaBench	230709	147 s	48337	40 s	-		262	7 s
Alarm Clock	428	1 s	161	3 s	167	1 s	65	3 s
Linked List	1341	1 s	270	3 s	80	1 s	1290	6 s
RAX Extended	1315	1 s	22	2 s	18	1 s	20	3 s
Rep Workers	6685	6 s	1522	5 s	4516	6 s	1054	4 s
QSort MT	3221	2 s	959	3 s	-		274	2 s

- Hybrid POR achieves better performance than purely dynamic POR on benchmarks that have larger state spaces, such as Cache4j and Daisy, and it can successfully verify 3 out of the 4 benchmarks at which dynamic POR fails.

State space traversal with hybrid POR detects errors very fast, and it can also explore all distinct interleavings of interfering actions in a reasonable time. By manual inspection of the execution logs of JPF, we found that the precision achieved by hybrid POR is largely due to the fact that our algorithm maintains the dynamic points-to sets and determinacy information separately for each program point and each thread. Many redundant thread choices are avoided in this way.

Dynamic POR is less precise than hybrid POR for some benchmarks (e.g., Alarm Clock and Prod-Cons) because it can make a redundant thread choice at instruction i that accesses an object o in the following situation: (1) there is an instruction j in another thread that accesses o, (2) j was executed before i, and (3) the object o is not reachable by multiple threads at the time j was executed. In the case of hybrid POR, the hybrid analysis marks the access by j as thread-local, and therefore enables more precise handling of situations like this.

Now we discuss the performance differences between hybrid POR and dynamic POR in either direction. An advantage of hybrid POR is that it needs to check much less pairs of visible field accesses to detect the interfering ones (i.e., to compute the full independence relation). When the hybrid analysis is queried at a dynamic state, it efficiently identifies all future field accesses that cannot interfere with the current action. For those accesses, hybrid POR can safely omit checks of interference also during the inspection of previous actions on the current trace (line 54 in Figure 3). On the other hand, the purely dynamic POR has to consider all the previous accesses. The difference in the number of pairs of visible accesses that must be checked is quite significant for programs with large state spaces and long execution traces. It is mainly for this reason that hybrid POR achieves better performance on larger benchmarks. On the other hand, for some benchmarks, performance of hybrid POR suffers (i) from imprecision of the underlying static analysis and (ii) from the need to refine the under-approximate information in multiple iterations.

V. RELATED WORK

Many approaches to POR have already been developed in the context of model checking and concurrency testing. Notable examples are the original dynamic POR [5] and Cartesian POR [8]. Furthermore, Abdulla et al. [1] recently proposed an optimal algorithm for dynamic POR.

The goal of all POR techniques is to limit the number of thread choices on every execution trace. In addition, most POR algorithms try to minimize the number of transitions to be explored from each thread choice, using the concepts of persistent sets and sleep sets [6]. Our hybrid POR minimizes just the number of thread choices in the state space, i.e. all enabled transitions are explored at each thread choice. Cartesian POR is the most closely related approach in this respect.

The algorithm for dynamic POR by Flanagan and Godefroid [5] works only with stateless model checking. It does not handle cyclic state spaces, and performs redundant computation when re-exploring already visited states. Other researchers designed extensions of this algorithm to address its limitations. Yang et al. [17] combined dynamic POR with stateful search, and Thomson et al. [15] proposed the lazy happens-before relation that enables dynamic POR to avoid redundant exploration of some thread interleavings for programs with coarse-grained locking. We adapted the ideas of Yang et al. [17] in our implementation of dynamic POR in JPF. Upon reaching an already visited state, the algorithm just has to consider field accesses that could occur in the rest of the program execution after the state. The necessary information about possible future field accesses is collected at backtracking steps.

Hybrid POR is directly compatible with state matching. Unlike the approach of Yang et al. [17], it does not have to keep track of field accesses that may occur after the given state. The hybrid field access analysis provides the information about future behavior of each thread.

We are aware of several other techniques involving POR that combine static and dynamic analysis [3] [9]. A common pattern behind them is the computation of an approximate dependency relation (a set of interfering actions) by static analysis, followed by (or interleaved with) the usage of dynamic analysis to improve precision based on information taken from dynamic program states and execution traces. For example, Kusano and Wang [9] proposed a framework that combines dynamic POR with a slicing algorithm in order to focus the search on interfering accesses that may cause assertion violations or deadlocks. The slicing algorithm uses static analysis to identify data dependencies and dynamic analysis to compute a precise aliasing information on-the-fly.

Our approach also follows the recent trend of verification algorithms based on iteratively refined under-approximation that captures (prefixes of) feasible execution traces of a given program. This large group of techniques includes, for example, context-bounded search with iterative increase of the maximal number of preemptions [11], and lazy abstraction with refinement based on interpolants [10]. There are even algorithms, such as UFO [2] and SMASH [7], that combine under-approximation with over-approximation and iteratively refine both abstractions until an error is found or the program is proven safe. The motivation behind such techniques is the detection of real errors in a practical time. When there are sufficient resources, use of the iterative refinement enables the algorithms to gradually increase coverage of the program state space, and to eventually explore all the execution traces.

VI. Conclusion

Our main contribution presented in this paper is the hybrid POR algorithm and its usage in a state space traversal procedure. Hybrid POR combines static analysis with data taken on-the-fly from dynamic program states, with iteratively refined under-approximate dynamic points-to and determinacy information, and also with the happens-before ordering relation.

Results of our experiments show that, for programs with larger state spaces, hybrid POR outperforms all the other approaches that we considered. The ability to look ahead by querying the hybrid field access analysis is the main reason behind good performance of hybrid POR. On the other hand, there is a certain overhead associated with the hybrid field access analysis. Hybrid POR is slower than dynamic POR for small benchmarks due to the overhead, but it still achieves good running times.

In the future, we would like to adapt the lazy happens-before relation [15] in order to improve the precision and performance of hybrid POR even further. We also plan to investigate possible incremental approaches to POR.

Acknowledgements.

This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and Charles University institutional funding SVV-2016-260331.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal Dynamic Partial Order Reduction. Proceedings of POPL 2014, ACM.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. Proceedings of TACAS 2012, LNCS, vol. 7214.
[3] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. Proceedings of ASE 2001, IEEE.
[4] M. Dwyer, J. Hatcliff, Robby, and V. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. Formal Methods in System Design, 25(2-3), 2004.
[5] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. Proceedings of POPL 2005, ACM.
[6] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.
[7] P. Godefroid, A. Nori, S.K. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. Proceedings of POPL 2010, ACM.
[8] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. Proceedings of SPIN 2007, LNCS, vol. 4595.
[9] M. Kusano and C. Wang. Assertion Guided Abstraction: A Cooperative Optimization for Dynamic Partial Order Reduction. Proceedings of ASE 2014, ACM.
[10] K. McMillan. Lazy Abstraction with Interpolants. Proceedings of CAV 2006, LNCS, vol. 4144.
[11] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. Proceedings of PLDI 2007, ACM.
[12] P. Parízek and O. Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. Proceedings of ASE 2011, IEEE.
[13] P. Parízek and O. Lhoták. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. Sci. Comput. Programm., 98, 2015.
[14] M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Dynamic Determinacy Analysis. Proceedings of PLDI 2013, ACM.
[15] P. Thomson and A. Donaldson. The Lazy Happens-Before Relation: Better Partial-Order Reduction for Systematic Concurrency Testing. Proceedings of PPoPP 2015, ACM.
[16] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.
[17] Y. Yang, X. Chen, G. Gopalakrishnan, and R.M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. Proc. of SPIN 2008, LNCS, vol. 5156.
[18] Concurrency Tool Comparison repository, https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison
[19] The Java Grande Forum Benchmark Suite, https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
[20] jPapaBench, http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/
[21] Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf
[22] pjbench: Parallel Java Benchmarks, https://bitbucket.org/pag-lab/pjbench
[23] WALA: T.J. Watson Libraries for Analysis, http://wala.sourceforge.net/

Formal Verification of Division and Square Root Implementations, an Oracle Report

David L. Rager, Jo Ebergen, Dmitry Nadezhin, and Austin Lee
Oracle
Email: {david.rager,jo.ebergen,dmitry.nadezhin,austin.lee}@oracle.com

Cuong Kim Chau and Ben Selfridge
The University of Texas at Austin
Email: {ckcuong,benself}@cs.utexas.edu

***Abstract*—Oracle has developed new implementations for integer division and floating-point division and square root. Our task was to verify the correctness of the new designs by formally proving equivalence between the RTL for these designs and their specifications in the SPARC ISA and in the IEEE 754 Standard on floating-point arithmetic. Performing such verifications involved many steps, which we describe in this paper. The contributions of this paper are two-fold. First, this paper describes Oracle's methodology for abstracting from low-level Verilog to a high-level algorithm using the latest open-source tools. Second, this paper describes the use of interval arithmetic in the error analysis of each algorithm. Our verification efforts proved that the designs had no errors, resulted in various improvements, and reduced the lookup tables by approximately 50% (division) and 75% (square root).**

I. Introduction

Oracle has developed new implementations for integer division and floating-point division and square root. The Oracle implementations are a variant of the Goldschmidt algorithm [1], [2]. Our task was to verify the correctness of these new implementations described in low-level Verilog by showing bit-for-bit equivalence with the specification in the SPARC ISA and the IEEE 754 Standard on floating-point arithmetic. Performing such verifications involved many steps:

- *Parsing and semantics* – Parsing the Verilog and bringing the design into the ACL2 System, a programming language and logic capable of reasoning about the implementation,
- *Algorithm extraction* – Abstracting low-level bit-oriented primitives, (e.g., *nand*s, *nor*s, muxes, etc.) to higher-level data types and mathematical operations like addition and multiplication, representing an algorithm, and finally
- *Algorithm verification* – Proving that the algorithms satisfy the SPARC ISA and IEEE 754 specifications.

This paper describes the techniques we used for performing each of the above steps. Figure 1 shows the three levels of abstraction that we used in our verification steps.

For algorithm extraction, we used the primitives afforded to us by the ACL2 software stack to abstract away the notion of time and think about the circuit as if it were a loop-free combinational circuit.

For verifying the algorithm, we mainly used ACL2's proof engine. The error analysis was the crucial part in this step – we used interval arithmetic to describe the size of the error in the final approximation in terms of errors introduced at each step in the algorithm. We were able to significantly reduce the lookup table sizes using this approach.

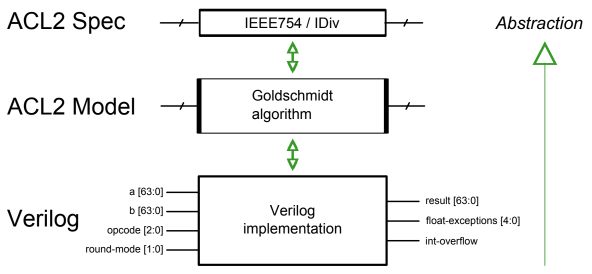


Fig. 1. Abstractions Necessary to Prove IEEE754 Compliance

II. Related Work

In 1998 at AMD, Russinoff was the first to prove that a Goldschmidt implementation of floating-point division and square root satisfies the IEEE 754 Standard using ACL2 [3]. Our work differs from the AMD effort in two ways. First, Russinoff started from a high-level program translated to a circuit implementation, somewhat similar to our algorithm description, instead of the Verilog. Second, our error analysis uses interval arithmetic and is more general, since the same techniques can be applied to many other implementations.

O'Leary et al. were one of the first to verify a complete floating-point unit down to the gate-level using in-house tools [4], a formidable effort. Our effort is similar in goal, but we verify very different division and square-root implementations and use the latest open-source tools.

Centaur developed many of the tools we use, and they have successfully applied them to verify large adder units [5]. We extended Centaur's work by applying their tools to deeper pipelines [6].

III. Tool Choice

We had the following tool requirements for our work. First, we wanted to make as few assumptions as reasonably possible, and we wanted what we did to be mechanically checked. Thus, our first requirement was that the chosen tool needed to be able to parse and reason directly about the Verilog – not just an abstraction of the Verilog that we created and maintained by hand. Second, we required an analytical framework capable of soundly converting logical primitives

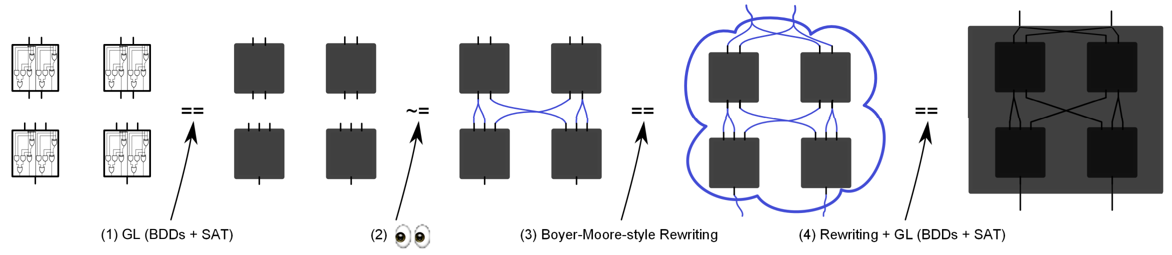

==
~=
==
==
(1) GL (BDDs + SAT)
(2)
(3) Boyer-Moore-style Rewriting
(4) Rewriting + GL (BDDs + SAT)

Fig. 2. Techniques for Circuit Abstraction in ACL2

like *nand*s and *nor*s into higher-level mathematical concepts like M-bit multiplication. Third, the analytical framework needed to be able to formally express the IEEE 754 Standard. Finally, the analytical framework needed to be able to reason about and prove that our implementation of the Goldschmidt algorithm, consisting of operations like M-bit multiplication, indeed satisfies the formalized IEEE 754 Standard.

For our verification effort, we chose the ACL2 programming language, theorem prover, and tool suite [5], [7], because it met each of the above criteria and because of its reputation of scaling to industrial-strength problems.

IV. Parsing and Semantics

As mentioned above, we used the ACL2 System and Libraries [7] to parse the circuit and determine the circuit's semantics. Specifically, we used the VL Parser to parse the Verilog, and we used Esim to determine the semantics of the resulting parse trees. We used the Symbolic Test Vector (STV) framework to initialize values of the circuit and provide the timing abstractions necessary to reason about values that the circuit receives as input and returns as output. By using such an approach, we were able to abstract away the notion of time and think about the circuit as if it were a loop-free combinational circuit. This STV methodology is similar to Symbolic Trajectory Evaluation [8], [9], in that one chooses a time to override a set of wires' values and examines a wire later in the logic path for its symbolic value, expressed in terms of those overridden inputs. Through the STV tools we brought the Verilog design into the logic of the theorem prover as an ACL2 expression on 4-valued Boolean operations.

V. Algorithm Extraction

The Goldschmidt algorithm consists of a repetition of multiplications, additions, and bitwise complements. We wanted to be able to reason in terms of these high-level primitives, as opposed to reasoning about *nand*'s and *nor*'s. As such, we proved that various compositions of low-level operations implement these higher-level mathematical operations. We found GL, a system for reasoning about finite ACL2 objects, to be helpful in many of these proofs. But GL, which uses BDDs and SAT solvers under the hood, could not automatically prove all of the necessary abstractions. For example, GL could not automatically prove that our composition of Booth Encoders and CSA trees correctly implemented multiplication.

Extracting the algorithms from the Verilog involved four types of steps, shown in Figure 2.

1) Initial Abstractions: The first step was to create abstractions for low-level Verilog components, for example a selection of Booth Encoders and CSA trees. Once we were able to formulate an applicable specification, we were able to verify that specification automatically with the assistance of GL and underlying support tools like BDDs and SAT Solvers. We emphasize that the choice of abstraction here can be crucial. For example, a simple carry-save adder with inputs a, b, and c_{in} can be specified by two functions $sum = f(a, b, c_{in})$ and $carry = g(a, b, c_{in})$. For our efforts it turned out to save a lot of work and complexity to specify just the relation that must hold between sum and $carry$: $2 * carry + sum = a + b + c_{in}$.

2) Model the Interconnect: The second step was to model the connections between the black-boxed components in the theorem prover logic. We call these connections the *logical interconnect*. Since we created the logical interconnect by hand, this interconnect was only hypothetically equivalent to the Verilog interconnect. We proved the equivalence between the two interconnects in Step 4. Figure 2 shows the process of creating the logical interconnect by hand in Transition (2).

3) Specify the Larger Component: The third step involved using ACL2's proof engine to show that the logical connections of the intermediate abstractions indeed resulted in a higher-level specification. Industrial circuits tend to have many optimizations and are shared among many implementations, so often a simple specification like *a*b* is insufficient. Moreover, as with Step 1, a well-chosen specification can reduce complexity and save a lot of work. Transition (3) in Figure 2 shows this step.

4) Verify the Interconnect: Finally, we verified that our logical interconnect was the same as the Verilog interconnect. Fortunately, GL is often capable of verifying that the interconnects match in an acceptable amount of time. The new black-boxed specification is shown on the far right of Figure 2.

It is worth noting that we only performed Step 1 when lifting the gates into low-level abstractions. Steps 2-4 were repeated many times, as the black box that one obtains at the end of Step 4 can serve as an input to Step 2 for a higher level of abstraction. This process of repeated abstraction allowed us to go from the low-level gate implementations to a presentation of the algorithm in terms of *, +, bitwise-complements, and other operations performed on M-bit operands.

VI. Algorithm Verification

Another big task was to show that the algorithms met the IEEE 754 Standard and the SPARC ISA specification. For these proofs, we applied some novel techniques to perform the error analysis, the most critical part of the proof. In this section, we explain the basic properties of our IEEE 754 specification and our error analysis. We only discuss the error analysis for floating-point division. Square root and integer division have a similar approach. Among the many other omitted proof obligations are proving that the rounding, exponents, and exceptions were implemented correctly. The complete verification effort did not find any errors and led to several simplifications in the algorithm and implementation.

A. An IEEE 754 Specification

In order to prove compliance with the IEEE 754 Standard we needed to formalize the IEEE 754 Standard in ACL2. Our formalization is written in ACL2 and

- specifies floating-point addition, multiplication, division, square root, and the fused multiply-add operations,
- includes denormals, all special values (+/-0, +/-∞, NaNs), and all exception flags,
- includes four rounding modes: round-to-nearest-even, round-to-positive(or negative)-infinity, and round-to-zero,
- works for any positive number of exponent bits and any positive number of trailing significand bits, and
- builds upon prior work, such as Russinoff's RTL Library [10] and Centaur's Bitops Library [7].

We validated our ACL2 specification against millions of test vectors, consisting of both directed and “random” test vectors.

Our ACL2 formalization begins by converting each floating-point number, except infinities and NaNs, to a rational number. We then perform the math operation exactly using rational numbers, and finally we convert the rational result back to the proper precision using the specified rounding mode.

For most operations this ACL2 formalization was not a problem, since most floating-point operations begin and end with rationals. The result of a square root, however, can be irrational, and ACL2 can only reason directly about rationals. To accommodate this limitation, we defined the square root as the limit of a sequence of rational numbers, where we stop this sequence when rounding the exact square root yields the same result as rounding the last number from the sequence.

B. The Goldschmidt Algorithm

The basic idea behind the Goldschmidt algorithm is as follows. To compute A/B, the algorithm calculates a series of factors T and r_i, for $i \geq 0$, such that $B*T*r_0*r_1*r_2*...$ converges to 1. Then $A*T*r_0*r_1*r_2*...$ converges to the quotient A/B, because

$$\frac{A}{B} = \frac{A*T*r_0*r_1*r_2*..}{B*T*r_0*r_1*r_2*..} \rightarrow \frac{A*T*r_0*r_1*r_2*..}{1}$$

We assume that $A, B \in [1, 2)$. The first factor T comes from a table lookup and is an initial estimate of the reciprocal $1/B$. All entries from the lookup table are in interval $[0.5, 1]$. The other factors r_i are also easily computed by means of a two's complement of the denominator d_i. Table I shows the basic Goldschmidt algorithm for division with loop invariant $n_i/d_i = A/B$. The complete Goldschmidt algorithm continues

TABLE I
Goldschmidt algorithm for division

```
$T = table\_lookup(B);$
$d_0 = B * T;$
$n_0 = A * T;$
$r_0 = 2 - d_0;$
for (int $i = 0$; $i < MAX$; $i$++) {
    $d_{i+1} = d_i * r_i;$
    $n_{i+1} = n_i * r_i;$
    $r_{i+1} = 2 - d_{i+1};$
}
return $n_{MAX}$;
```

after the final multiplication by adding a constant to n_{MAX} and truncating the result to the proper precision yielding the preliminary quotient plus guard bit, say q. Then the exact remainder $A - q * B$ is calculated, and the guard bit, sign of the remainder, and sign of the result determine how to round q. If the result, n_{MAX}, is close enough to the exact quotient, then rounding produces the correct result. In this paper we focus on estimating the error in n_{MAX}. The calculation of the sign of the remainder and rounding is standard.

C. Error Analysis

We express the error in the final result of the Goldschmidt algorithm as a function of the error in the lookup table and other errors introduced in the implementation. First we express the error introduced by just the lookup table. Let the lookup value T be an approximation for $1/B$, where $T = 1/B - u/B$. The value u/B is the absolute error in $1/B$, and u can be seen as the relative error. The relative error u ranges over a small interval around 0, depending on the entry keys and entry values of the lookup table.

Table II gives a sequence of statements from the Goldschmidt algorithm but now in terms of u, the relative error in the table lookup. Using the infinite Taylor series for

TABLE II
Symbolic expressions for $d_i, n_i, r_i, i \geq 0$

statement	**exact symbolic expression**
$d_0 = B * T$	$= 1 - u$
$n_0 = A * T$	$= A * T$
$r_0 = 2 - d_0$	$= 1 + u$
$d_1 = d_0 * r_0$	$= 1 - u^2$
$n_1 = n_0 * r_0$	$= A * T * (1 + u)$
$r_1 = 2 - d_1$	$= 1 + u^2$
$n_2 = n_1 * r_1$	$= A * T * (1 + u + u^2 + u^3)$
..	..

$A/B = A * T/(1 - u)$, we can write the error $n_2 - A/B$ for $|u| < 1$ as

$$n_2 - A/B \quad = \quad A * T * (-u^4 - u^5 - ...)$$

If, for a given ϵ, the lookup table is chosen such that $u \in [-\epsilon, +\epsilon]$, then the error $(n_2 - A/B)$ is in the interval $[\frac{-2\epsilon^m}{(1-|\epsilon|)}, \frac{+2\epsilon^m}{(1-|\epsilon|)}]$ with $m = 4$, where we used $A * T < 2$ for any choice of T. Because the value of m doubles with each iteration of the algorithm, the value of n_i converges rapidly to the quotient A/B.

D. Error Analysis and Finite Hardware Precision

The Goldschmidt algorithm introduces two more errors, besides the error u. First, each multiplication result, except the last, is truncated from $2M$ bits to M bits. Thus, each multiplication introduces an error of $e_i \in [0, 2^{-M})$, where the error e_i can be different in each truncation i. For example, d_0, the result of a multiplication and a truncation, can be expressed as $d_0 = 1 - u + ed_0$. Second, to implement the statement $2 - d_i$, instead of using the two's complement, we take the simpler one's complement. This simplification introduces a fixed error of $eps = 2^{(1-M)}$. Thus r_0 can be expressed as $r_0 = 1 + u - ed_0 - eps$.

Following these changes we can express the final approximation as a polynomial in variables u, A, T, ed_i, en_i, eps for $i \geq 0$. This polynomial can be long, but it can be constructed mechanically. Similarly, we can express the error in the final approximation as a polynomial in the same variables where we bound the infinite sum as before. Because each of these variables ranges over a small interval, we can bound the error with known methods from interval arithmetic [11].

The symbolic expressions for the error in the final approximation and the use of interval arithmetic allowed us to reduce the sizes of the lookup tables. The lookup table for division was reduced by approximately 50%, and the lookup table for square root was reduced by approximately 75%.

To perform and mechanically verify the error analysis, we needed various tools. We first needed a tool to generate the multivariate polynomials for each operation. We also needed a tool to verify the interval arithmetic when given multivariate polynomials and intervals for each of the variables. We first experimented with these tools in Java, where we found the proper entries in the lookup table. We then implemented these tools in an ACL2 library so that we could not only compute the polynomials but also verify our error analysis.

VII. Business Decisions

As a product group we were constrained to a timeline and had to make many assumptions. One decision concerned the initialization of the unit. As is common in industry, Oracle performs a power-on and reset process that sets all registers to zero or one as appropriate. We instead initialized the circuit by flushing the pipeline of the floating-point unit. The primary benefit of this approach is that our simulation is more easily maintained, but a secondary benefit is that our proofs are not relying upon a correct power-on and reset sequence.

We also made a decision to focus on proving the correctness of exactly one floating-point or integer division operation that immediately follows a flush. Verifying the control logic would have significantly increased our work. As such, we rely upon our traditional concrete simulation team to generate enough directed test vectors to demonstrate the correctness of much of the unit's control logic. This allowed us to focus on the data-path, traditionally the riskiest part of floating point operations [12], and meet the desired schedule.

A schedule-critical decision was to perform the algorithm extraction and algorithm verification concurrently. By first specifying the algorithm, we could perform these tasks independently, resolving any discrepancies in the original and final drafts of the algorithm at the end of both processes. This decision approximately halved our total verification time.

VIII. Results and Conclusion

We have proved, under some assumptions, that the Verilog matches bit for bit with our extracted algorithms. We have also proved that the extracted algorithms satisfy the IEEE specifications for floating-point division and square root and the specifications for integer division. Thus, transitively, we know that the Verilog implements these specifications.

Another benefit of our work was the discovery of several optimizations, which were all implemented and proven correct. Our thorough error analysis led to an optimization in the lookup tables for division and square root, yielding a reduction of approximately 75% and 50% in the two lookup tables, respectively. Other optimizations led to simplifications in the hardware and in the proof.

Acknowledgments

We thank Jeff Brooks, Greg Grohoski, Warren Hunt, Matt Kaufmann, Govind Murugan, Chris Olson, and Greg A. Smith for their technical help and support.

References

[1] S. Oberman and M. Flynn, "Division algorithms and implementations," *Computers, IEEE Transactions on*, vol. 46, no. 8, pp. 833–854, Aug 1997.

[2] S. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7^{TM} microprocessor," in *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, 1999, pp. 106–115.

[3] D. Russinoff, "A mechanically checked proof of IEEE compliance of the floating-point multiplication, division, and square root algorithms of the AMD-K7^{TM} processor," *London Mathematics Society Journal of Computation and Mathematics*, no. 1, pp. 148–200, 1998.

[4] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. 3, no. 1, pp. 1–14, 1999.

[5] A. Slobodova, J. Davis, S. Swords, and W. A. Hunt, "A flexible formal verification framework for industrial scale validation," in *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, July 2011, pp. 89–97.

[6] J. Davis and S. Swords, private communication, 05 2016.

[7] "ACL2 System and Books." [Online]. Available: https://github.com/acl2/acl2

[8] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Form. Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, Mar. 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01383966

[9] V. M. A. KiranKumar, A. Gupta, and R. Ghughal, "Symbolic trajectory evaluation: The primary validation vehicle for next generation Intel processor graphics fpu," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*, Oct 2012, pp. 149–156.

[10] D. Russinoff, "A formal theory of RTL and computer arithmetic." [Online]. Available: http://www.russinoff.com/libman/

[11] M. J. C. Ramon E. Moore, R. Baker Kearfott, *Introduction to Interval Analysis*. SIAM Press, 2009.

[12] T. R. Nicely, "Original pentium fdiv flaw e-mail." [Online]. Available: http://www.trnicely.net/pentbug/bugmail1.html

Integrating Proxy Theories and Numeric Model Lifting for Floating-Point Arithmetic

Jaideep Ramachandran*, Thomas Wahl
Northeastern University

Abstract—**Precise reasoning for floating-point arithmetic (FPA) is as critical for accurate software analysis as it is hard to achieve. Several recent approaches reduce solving an FPA formula f to reasoning over a related but easier-to-solve *proxy theory*. The rationale is that a satisfying proxy assignment may directly correspond to a model for f. But what if it doesn't? Prior work deals with this case somewhat crudely, or discards the proxy assignment altogether. In this paper we present an FPA decision framework, parameterized by the choice of proxy theory T, that attempts to *lift* an encountered T model to a numerically close FPA model. Other than assuming some "proximity" of T to FPA, our lifting procedure is T*-agnostic*; it is in fact designed to work independently of how the proxy assignment was obtained. Should the lifting fail, our procedure gradually reduces the gap between the FPA and the proxy interpretations of f. We have instantiated the framework using real arithmetic and reduced-precision FPA as proxy theories, and demonstrate that we can, in many cases, decide f more efficiently than earlier work.**

I. Introduction

Floating point arithmetic, the real-arithmetic (RA) approximation used on most general-purpose computers today, continues to surprise programmers. While computer scientists and mathematicians are aware of the loss of precision such approximation necessarily incurs, many are oblivious to the consequences this can have in programs beyond small inaccuracies in final results. To debug numeric programs effectively before the software is deployed, counterexample-producing analysis tools sensitive to FPA semantics are vital.

The last 5–10 years have seen increased efforts in building floating-point decision procedures. The first such were based on *bit-precise* encodings: floating-point expressions are encoded as propositional [5], [1] or bitvector logic [6] formulas that formalize the prescription of the IEEE 754 floating-point standard [10]. These approaches, while successful, tend to suffer from the size of the encoding that "bit-blasting" entails. Moreover, such very low-level encodings lose the intended numeric proximity of FPA to the real numbers and thus obscure even the simplest identities like $a + b = b + a$.

The other, and more recent, approach to encoding floating-point formulas is to exploit said numeric proximity. The IEEE standard stipulates that a floating-point computation *"shall be performed as if it first produced an intermediate result correct to infinite precision …, and then rounded that intermediate result … to the destination's format"* [10]. Experience has shown that encoding rounding precisely as a mathematical (floating point-free) operation is feasible but expensive [11]. An alternative philosophy is to ignore the rounding altogether, solve the formula interpreted over the *proxy theory* of real arithmetic using off-the-shelf RA solvers, and check any obtained models for satisfaction of the formula under FPA semantics. This idea has been used successfully to detect floating-point exceptions in `C` programs [2].

In line with recent work on floating-point model construction [12], we present in this paper a method that extends the paradigm of reasoning about FPA via some proxy theory T to a (complete, in principle) decision method. Our method begins by abstracting the given floating-point formula f into the proxy theory T to obtain a formula f_T of the same propositional structure but with T constraints that are assumed to be easier to decide. It then tries to find a T-model σ_T of f_T. If successful, we cast σ_T to a "nearby" FPA assignment σ (how exactly this is done depends on T). We now determine whether $\sigma \models f$ according to FPA semantics; if yes, a model for f has been found. If $\sigma \not\models f$, previous methods disagree widely on how to proceed. In [2], where $T =$ RA, this case is treated as a failure. In [12], where T is reduced-precision FPA, the authors attempt to reconstruct full-precision FPA models simply by initializing the unused bit positions with zeros.

In this paper we propose a *numeric model lifting* procedure that exerts much more fine-grained control over how a potential model for f is obtained. We aim to find this model in the close vicinity of (the non-satisfying) σ. To this end, our procedure heuristically determines a subset O of f's variables such that modifying the assignment to these variables slightly has a chance to make σ satisfying. We now partially instantiate f, namely by the assignment σ restricted to the variables *outside* of O, to obtain a formula f'. This reduces the original decision problem for f to a decision problem for f' over only $|O|$ variables.

Our method now goes a significant step further: instead of solving f' from scratch, we use a strategy reminiscent of lazy SMT solving: assignment σ_T, satisfying the T abstraction f_T of f, gives rise to a Boolean model for the *propositional skeleton* of f_T. Since f_T is designed to have the same propositional structure as f (and hence as f'), we can reuse this skeleton assignment, and simply solve a conjunction Δ of FPA constraints: for each constraint in f', we require its truth value to be the same as that σ_T has assigned to the corresponding T constraint in f_T.

We summarize the point of constructing Δ. First, if Δ is satisfiable, via some assignment ε, then so is f; a satisfying assignment is given by updating σ's assignment to O-variables using ε. Second, Δ is a conjunction of FPA constraints (no

*Email: jaideep@ccs.neu.edu. Supported by NSF grant CCF-1218075.

propositional structure), and contains only $|O|$ variables. In our experiments, we found that choosing a *single* variable in O often suffices. In that case we have reduced f drastically, to a *univariate conjunction of constraints*. This reduced problem can now be given to an FPA solver such as MATHSAT [6], with largely increased prospects for a speedy decision.

If the lifting step does not succeed, or f_T is unsatisfiable to begin with, our procedure refines f_T, in a manner that depends on the choice of T. The step-wise refinement often turns unsatisfiable abstractions f_T into satisfiable ones. A classical example are formulas debunking "false identities", like $(x + y) + z > x + (y + z)$, which is unsatisfiable in RA, but becomes satisfiable after a *one-step* refinement; Sect. II illustrates this in detail. If, for each intermediate abstraction f_T, a T model cannot be found or the subsequent lifting fails, the iterative process eventually refines f_T to f; the search for models in the proxy theory was in vain. In the spirit of [12], our method is intended for fast model construction.

We have experimented with two proxy theories in this paper: real arithmetic and reduced-precision floating-point (Sect. V). Both are often easier to solve than FPA [2], [12].

We finally note that special floating-point values like infinities and NaNs will occur in the assignment σ_T only if the proxy theory T is "aware" of such values (e.g. RA is not). Since the model lifting process presented in this paper is designed to be T-agnostic, we mostly avoid discussing special values. Our implementation currently enforces their absence in σ_T for proxy theories that have them. Incorporating special values fully into our framework is left for future work.

II. A MOTIVATING EXAMPLE

Our approach deals with floating-point formulas of propositional structure in a way that is reminiscent of lazy SMT solving; we present the details of this in Sect. IV. In the present section we focus on the theory-specific (numeric) aspects. Consider therefore the atomic floating-point formula

$$f \quad :: \quad (a_1 \oplus a_2) \oplus a_3 \;>\; a_1 \oplus (a_2 \oplus a_3) \;, \tag{1}$$

where $\oplus$ denotes floating-point addition. To keep the presentation succinct in this section, we assume single-precision and *round-to-negative* as rounding mode.[1] We will demonstrate how our proposed framework processes this formula using real arithmetic as proxy theory ($T = \text{RA}$).

Motivated by the success of earlier work in finding floating-point models by searching in the reals instead [2], we express this formula in the logic of real arithmetic to obtain $f_T :: (a_1 + a_2) + a_3 > a_1 + (a_2 + a_3)$, and give it to an SMT solver. The solver responds that f_T is unsatisfiable.

With the determination to construct a model in mind, our technique mistrusts the UNSAT result and proceeds by increasing the precision of the abstraction. Fortunately, we can perform this refinement in a lazy manner, by interpreting *parts* of f in floating-point, others in real arithmetic. Suppose we decide that the top-level $+$ of the right-hand side expression in f_T is to be interpreted in (refined to) FPA. This turns f_T into the formula

$$f'_T \quad :: \quad (a_1 + a_2) + a_3 \;>\; a_1 \oplus (a_2 + a_3) \;. \tag{2}$$

The domain of all variables remains the real numbers. This is a formula in *Mixed Real-FPA* (MRFPA); details of its semantics are given in Sect. V.

Why is this refinement useful? The answer is that chances of finding a model for f by examining f'_T are higher than doing so by examining f_T, since the semantics of MRFPA will ensure that the $\oplus$ in f'_T implements floating-point addition (although its operands are reals; details in Sect. V). In addition, the cost of examining f'_T is only moderately higher than that of examining f_T, and hopefully lower than that for f. To analyze f'_T we need solver support for MRFPA, which is given by (an extension of) the tool REALIZER [11, details in Sect. V].

Giving f'_T to the extension of REALIZER, we obtain—for the first time—a satisfying assignment σ_T, namely

$$\sigma_T \quad :: \quad a_1 = a_2 \approx 1.1755 \cdot 10^{-38} \;, \quad a_3 \approx 1.9722 \cdot 10^{-31} \;.$$

The left hand side term of f'_T evaluates slightly larger than the right hand side. We now project these real numbers to single-precision floating-point, which is done simply by rounding. We then apply the resulting assignment, call it σ, to the floating-point formula f. Unfortunately, σ does **not** satisfy f: the left-hand and right-hand side sums turn out to be the same.

Instead of immediately refining f'_T further, our method does not give up the hope that a model for f can be found in a neighborhood of σ. We therefore now try to "nudge" this assignment so that it satisfies f. The plan is simple: we pick **one** of the a_i variables to modify—say our choice is a_3—while leaving all others constant. We then build a new, *univariate* formula Δ as follows:

$$\Delta \quad :: \quad (\overline{a_1} \oplus \overline{a_2}) \oplus a_3 \;>\; \overline{a_1} \oplus (\overline{a_2} \oplus a_3) \;,$$

where, for $i = 1, 2$, $\overline{a_i} := \sigma(a_i)$. By design of our method, if Δ is satisfiable, say via ε, then so is f, and we obtain a satisfying assignment for f from σ by changing the value assigned to a_3 using ε. The key is that Δ is simpler than f: it contains only one free variable (a_3). We have reduced the original floating-point decision problem to a much simpler one such that any model for the simpler problem gives rise to a satisfying assignment for f.

Finishing up our example: applying the solver MATHSAT [6] to Δ we learn that increasing a_3 by $1.1755 \cdot 10^{-38}$ leads to a satisfying assignment for f: the left sum is now larger.

Recent work uses reduced-precision FPA as proxy theory [12] and attempts to "patch" a proxy assignment (like σ_T) to a satisfying floating-point one using syntactic means: by padding the lower-precision bitvector assignment with 0s or 1s. This initially fails and requires more refinement iterations, ultimately entailing higher cost, as our experiments will show. In contrast, our method takes the numeric circumstances into account, as reflected in formula Δ. As a result, a satisfying assignment for Δ *guarantees* the existence of a model for f.

[1]Using the more common mode *round-to-nearest-even* (RNE), the example works as well but requires more refinement steps. Our experiments use RNE.

III. Deciding FPA using a Proxy Theory and Model Lifting

We describe in this section our procedure for deciding a floating-point formula f; see Fig. 1. In addition to f, the (implicit) input to the procedure includes floating-point specifics like the format parameters for range and precision, as well as settings like the rounding mode, which we assume to apply across the entire formula. If f is determined to be satisfiable, the algorithm returns a satisfying assignment σ.

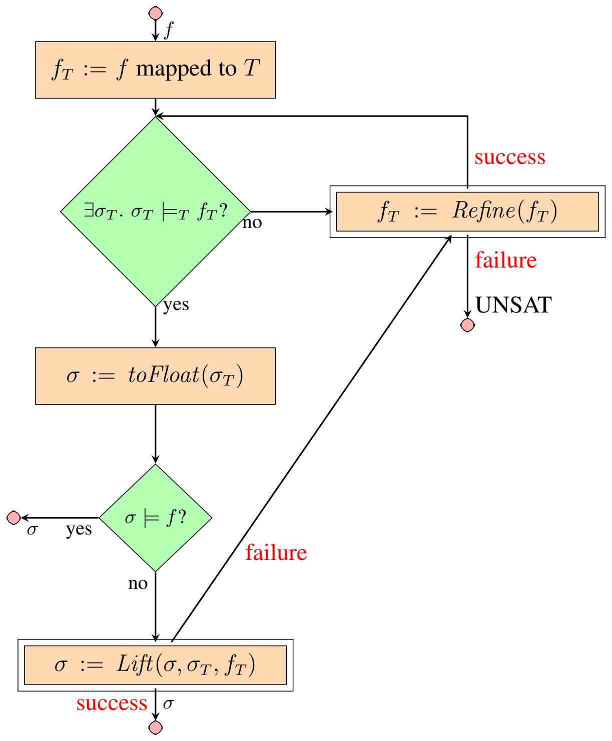

f
$f_T := f$ mapped to T
$\exists\sigma_T.\ \sigma_T \models_T f_T$?
no
yes
success
$f_T := \mathit{Refine}(f_T)$
failure
UNSAT
$\sigma := \mathit{toFloat}(\sigma_T)$
$\sigma \models f$?
yes
σ
no
failure
$\sigma := \mathit{Lift}(\sigma, \sigma_T, f_T)$
success
σ

Fig. 1: Deciding FPA formula f via proxy theory T

The procedure begins by mapping f to a formula over the proxy theory T; we can view the resulting f_T as an abstraction of f. (In general, though, it is neither an over- nor an underapproximation of f; this happens to be immaterial for our procedure.) How this map is defined is clearly T-specific. However, we require that it maintains the propositional structure (skeleton) of f and applies only to its (atomic) theory constraints. For example, in the case of reduced-precision FPA as the proxy theory, the mapping simply changes the floating-point format parameters that come with f. In the case T = RA (real arithmetic), the mapping causes all arithmetic function symbols and constants to be interpreted over the reals. We give more details on these specific proxy theories in Sect. V.

Given formula f_T in theory T, the procedure now repeatedly tries to find a model for f_T and to "lift" that to a model for f. If no model for f_T can be found, or the lifting fails, it refines f_T so as to narrow the semantic gap to the input formula f. This is reflected in Fig. 1 as follows. The procedure first performs the satisfiability check $\exists\sigma_T.\ \sigma_T \models_T f_T$ (where $\models_T$ is the satisfaction relation in T). If the check fails, f_T is refined; more on that below. If a satisfying assignment σ_T is found, we call a procedure *toFloat* that casts the T-assignment σ_T to a "nearby" floating-point assignment σ. This step is T-dependent and may for instance involve rounding (when T is "more precise" than standard FPA) or fresh bit initialization (when T is "less precise" than standard FPA).

We now ask whether σ is indeed a satisfying floating-point assignment for f. This amounts to plugging in the values given by σ, and evaluating the grounded formula f. Unless the satisfiability of f depends on floating-point peculiarities such as the lack of associativity (example in Sect. II), the query $\sigma \models f$ may well succeed ($\models$ is the satisfaction relation in FPA); this was observed in [2] for a large fraction of their (floating-point exception) benchmarks. In that case the procedure terminates, returning σ.

A negative result to the query $\sigma \models f$ is interpreted by the procedure to mean that the floating-point solution to f that we are suspecting in the vicinity of σ_T cannot be obtained simply by rounding or syntactic initialization. We therefore launch a more aggressive subroutine *Lift* that tries to modify the values assigned by σ to certain variables of f to force σ to be satisfying. This routine is described in Sect. IV. Note that, while *toFloat* casts a T-assignment to floating point, *Lift* maps one floating-point assignment to another.

The *Lift* procedure is designed such that, if there exists a satisfying floating-point assignment in the vicinity of σ, *Lift* will eventually find it, given enough time. In this case, *Lift* in Fig. 1 returns the new σ; the procedure terminates. Lifting can fail because f is unsatisfiable, or because the lifting timed out. The latter indicates that assignment σ is not a good starting point for finding a model for f. We increase the precision of the abstraction, by refining f_T.

The *Refine* step fails when, upon invocation, f_T is "equivalent" to f in a sense that depends on the abstraction map. For the case of reduced-precision FPA as proxy theory, this simply means the floating-point format of f_T equals that of f. In that case, the satisfiability check $\exists\sigma_T.\ \sigma_T \models_T f_T$ in the previous iteration was actually a satisfiability check for f. Since that did not succeed, f is unsatisfiable.

Correctness. Termination of the framework can be enforced with some "cooperation" from T: we assume T to be chosen such that mapping f to T, and the calls $\mathit{toFloat}(\sigma_T)$ and $\mathit{Refine}(f_T)$ are straightforward and "fast". The test $\sigma \models f$? is trivial. The decision problem $\exists\sigma_T.\ \sigma_T \models_T f_T$? may not terminate, e.g. due to undecidability of T. We solve this problem by enforcing a timeout for this step. The call $\mathit{Lift}(\sigma, \sigma_T, f_T)$ is discussed in detail in Sect. IV. As we shall see, it introduces no potential for nontermination.

Finally, the framework itself (Fig. 1) contains a loop. We require of the proxy theory that it permits gradual refinement of T formulas to floating-point formulas. How this is done exactly depends on T and is discussed, for two instances, in Sect. V. With these provisions, instances of the framework in Fig. 1 are terminating. The framework is also easily seen to be sound for SAT and UNSAT outcomes; we omit the details.

IV. Model Refinement: From Proxy Models to FPA Models

We revisit Fig. 1. Suppose formula f_T (the current abstraction of floating-point formula f) is satisfiable and gives rise to a model σ_T. Suppose further that the cast operation $\mathit{toFloat}(\sigma_T)$ yields a non-satisfying assignment σ for f. This means that σ assigns to at least one FPA constraint in f a different Boolean value than σ_T does to the corresponding T theory constraint in f_T. The goal of the model refinement procedure *Lift* is to reconcile this difference, thereby lifting assignment σ to a proper model for f.

The basic idea is as follows. Given that $\sigma_T \models_T f_T$, there exists an assignment to the variables in the *Boolean skeleton* of f_T that makes this propositional skeleton formula true. Since, by construction, f and f_T have the same Boolean skeleton (this is required of the abstraction; see Sect. III), the goal is to modify the assignment to the floating-point variables in f such that the corresponding Boolean skeleton assignment coincides with that induced by σ_T. If we succeed, f is satisfied. This turns the original FPA formula f into a structurally simple conjunction of FPA theory constraints, since the "target" Boolean value for these constraints is determined via σ_T.

We point out parallels of this reduction of formula structure to lazy SMT solving: there, a formula f over a background theory $\mathcal{T}$ is solved by first applying a $\mathcal{T}$*-oblivious* propositional SAT solver to f's Boolean skeleton. A solution gives rise to a conjunction of $\mathcal{T}$ constraints a model for which is a model for f. In our work we do not use a SAT solver — it is too weak for our purposes: we seek a (proxy) theory assignment that makes the skeleton true **and** gives hope that a satisfying FPA assignment can be found nearby.

Notation. Let V be the set of arithmetic variables in f, and let P be the set of propositional variables in the *Boolean skeleton* of f. We can think of variables $p \in P$ as pointers to the FPA theory constraints of f. Recall that the abstractions f_T produced during the main procedure in Fig. 1 all maintain the skeleton of f. Hence, there exists a function γ that takes $p \in P$ and f or f_T as input and returns the theory constraint of f or f_T pointed to by p. Consider this example for $T = \mathrm{RA}$:

$$\begin{aligned} f &= x \oplus y > 10 \lor x \ominus y < 7 \\ f_T &= x + y > 10 \lor x - y < 7 \ . \end{aligned}$$

Choosing $p_1 \lor p_2$ as the skeleton, we then have $V = \{x, y\}$, $P = \{p_1, p_2\}$, and

$$\begin{array}{ll} \gamma(p_1, f\ ) = (x \oplus y > 10), & \gamma(p_2, f\ ) = (x \ominus y < 7) \\ \gamma(p_1, f_T) = (x + y > 10), & \gamma(p_2, f_T) = (x - y < 7) \quad . \end{array}$$

We finally use *Eval* to denote a function that takes a formula φ and an assignment $\mathcal{A}$ to all variables in φ and returns the Boolean value of φ under $\mathcal{A}$, respecting the semantics of φ.

Our lifting procedure is shown in Alg. 1. The algorithm scheme receives the FPA assignment σ that fails to satisfy f, the T assignment σ_T that does satisfy f_T, and formula f_T. (The algorithm also has access to the [unchanged] formula f.) We begin by selecting all *invertible* constraints *Inv* in f (via pointers p): those for which assignments σ_T and σ disagree in the Boolean value assigned in f_T and f, resp.

In Line 2 we select a set O of *offset variables*: floating-point variables whose assignment we plan to modify to make σ satisfying. An upper bound on O is that each $v \in O$ must be contained in at least one invertible constraint. More details on the selection are given in **Implementation** below. In Line 3 we build the set P_O of (pointers to) constraints that contain at least one O-variable: these are the constraints whose truth value may be affected when assignments to O-variables are modified.

Line 4 modifies f to f' by instantiating every non-offset variable $v \in V \setminus O$ by its literal floating-point assignment $\sigma(v)$. Finally, in Line 5 we construct a constraint Δ that realizes the above basic idea: for each theory constraint $\gamma(p, f')$ in f' with at least one O-variable, enumerated via pointers $p \in P_O$, we require that it be assigned the truth value $\mathit{Eval}(\gamma(p, f_T), \sigma_T)$ (a constant) given by σ_T to the corresponding theory constraint $\gamma(p, f_T)$ in f_T. The right conjunct of formula Δ restricts the assignment to O-variables v to some interval around $\sigma(v)$; terms $v.l$ and $v.r$ are floating-point literals (see **Interval constraints** below).

Intuitively, constraint Δ is satisfiable whenever there is a satisfying assignment to f in some small neighborhood of $\sigma = \mathit{toFloat}(\sigma_T)$. We are hopeful this is the case, since σ_T satisfies f_T. Hence, if there exists a satisfying assignment ε to Δ, we modify σ by updating, using ε, the values assigned to O-variables. If ε does not exist, the lifting fails.

Implementation. Our lifting procedure has reduced the floating-point decision problem for f to that for Δ. Is the reduced problem simple enough that we can solve it using an off-the-shelf FPA decision procedure? Formula Δ is a conjunction of constraints — no propositional reasoning is required to decide it. The free variables in Δ are precisely the offset variables. The choice of set O thus critically influences the variable complexity of Δ; we use heuristics to keep it small. If v occurs in expensive constraints in f, such as in high-degree polynomials or other non-linear terms, the variable ranks low in our selection heuristics. For example, if f contains the quadratic form $x \otimes x \ \oplus \ x \otimes y$, our heuristic chooses $O = \{y\}$; Line 4 in Alg. 1 turns the entire term into the *univariate, linear* floating-point term

$$\sigma(x) \otimes \sigma(x) \ \oplus \ \sigma(x) \otimes y \ .$$

Interval constraints. Gradient analysis of f in a neighborhood of assignment σ may reveal that a variable v needs to be increased, say. In this case, we use a lower bound $v.l = \sigma(v)$ in the range constraint $v.l \le v \le v.r$ in Line 5. For example, for $f = x \oplus y > 4.0 \land x \ominus y < 2.0$ with $\sigma = \{x = 3.0, y = 1.0\}$, gradient analysis reveals that y needs to be increased; we set $y.l = 1.0$. In the absence of such information, we choose interval $[v.l, v.r]$ to be symmetric around $\sigma(v)$, of a width that is a small fraction of $|\sigma(v)|$.

Algorithm 1 (Scheme) Lifting σ to FPA model

```
Input: $\sigma$: FPA assignment (falsifying $f$), $\sigma_T$: $T$ assignment (satisfying $f_T$), $f_T$: abstract formula
1:  $Inv := \{p \in P \mid Eval(\gamma(p, f_T), \sigma_T) \neq Eval(\gamma(p, f), \sigma)\}$    ▷ invertible constraints
2:  select subset $O$ of $\bigcup_{i \in Inv} Vars(\gamma(i, f))$    ▷ offset variables
3:  $P_O := \{p \in P \mid Vars(\gamma(p, f)) \cap O \neq \emptyset\}$    ▷ $O$-affected constraints
4:  $f' := f|_{v \to \sigma(v) \mid v \in V \setminus O}$    ▷ partially instantiated formula
5:  $\Delta := \bigwedge_{p \in P_O} \gamma(p, f') = Eval(\gamma(p, f_T), \sigma_T) \;\wedge\; \bigwedge_{v \in O} v.l \leq v \leq v.r$
6:  if $\exists \varepsilon.\ \varepsilon \models \Delta$ then
7:      for each $v \in O$
8:          $\sigma(v) := \varepsilon(v)$
9:      return $\sigma$
10: else
11:     return failure
```

V. Proxy Theories for Floating-Point Arithmetic

We have instantiated our framework with two proxy theories at opposite ends of the precision spectrum: reduced-precision FPA and real arithmetic (which can, somewhat awkwardly, be viewed as an infinite-precision "approximation" of FPA). The former is a fairly obvious candidate: an FPA formula is abstracted by interpreting it over a floating-point format with smaller precision and/or range. Reduced-precision FPA is almost invariably easier to solve. Models can be cast to original-precision FPA by initializing the fresh bits to 0. Step-wise refinement consists of gradually increasing the precision (in our work: across the entire formula; more sophisticated schemes are possible). T is therefore actually the family of FPA theories parameterized by precision/range. Such proxy theories have been used before [12, without numeric lifting]. We discuss instead a less obvious choice for T in this section.

Real Arithmetic as Proxy Theory

As suggested in Sect. II and reported earlier [2], real arithmetic (RA) is suitable for an approximate interpretation of a floating-point formula f: many formulas are easier to decide over the reals, since the complexity of rounding is avoided. A satisfying real assignment can easily be cast to a floating-point assignment via rounding. To enable *step-wise refinement* of the RA-interpretation of f back to FPA, however, we need a proxy theory that can express combinations of real and floating-point terms, such as $a_1 \oplus (a_2 + a_3)$ (Sect. II).

Our proxy theory therefore is actually not real arithmetic, but an extension that we call *Mixed Real-Floating-Point Arithmetic* (MRFPA) and define as follows. Let $\mathbb{R}$ be the set of real numbers, and $\mathbb{F}$ be the numbers in $\mathbb{R}$ representable in floating-point over some fixed precision and range (these parameters are constant in this section). Let $rd\colon \mathbb{R} \to \mathbb{F}$ be the function that implements the given rounding mode, and let $Var_{\mathbb{R}}$ and $Var_{\mathbb{F}}$ be a set of real and floating-point variables, resp.

The syntax of MRFPA formulas f is as follows.

$$\begin{array}{rcl} f & :: & t_{\mathbb{R}}\,\theta_{\mathbb{R}}\,t_{\mathbb{R}} \mid t_{\mathbb{F}}\,\theta_{\mathbb{F}}\,t_{\mathbb{F}} \mid \neg f \mid f \vee f \\ \theta_{\mathbb{R}} & :: & < \mid = \\ \theta_{\mathbb{F}} & :: & <_{\mathbb{F}} \mid =_{\mathbb{F}} \\ \alpha_{\mathbb{R}} & :: & + \mid \times \mid / \\ \alpha_{\mathbb{F}} & :: & \oplus \mid \otimes \mid \oslash \\ \alpha_{\mathbb{M}} & :: & +_{\mathbb{M}} \mid \times_{\mathbb{M}} \mid /_{\mathbb{M}} \\ t_{\mathbb{R}} & :: & c \in \mathbb{R} \mid v \in Var_{\mathbb{R}} \mid (t_{\mathbb{R}}\,\alpha_{\mathbb{R}}\,t_{\mathbb{R}}) \mid t_{\mathbb{F}} \\ t_{\mathbb{F}} & :: & c \in \mathbb{F} \mid v \in Var_{\mathbb{F}} \mid (t_{\mathbb{F}}\,\alpha_{\mathbb{F}}\,t_{\mathbb{F}}) \mid (t_{\mathbb{R}}\,\alpha_{\mathbb{M}}\,t_{\mathbb{R}}) \end{array} \tag{3}$$

Intuitively, MRFPA formulas are built over $\mathbb{F}$ terms $t_{\mathbb{F}}$, which evaluate to elements of $\mathbb{F}$, and $\mathbb{R}$ terms $t_{\mathbb{R}}$, which more generally evaluate to elements of the superset $\mathbb{R}$. $\mathbb{R}$ terms are formed using real operators $\alpha_{\mathbb{R}}$. $\mathbb{F}$ terms are formed using floating-point operators $\alpha_{\mathbb{F}}$ or *mixed* operators $\alpha_{\mathbb{M}}$. Operators $\alpha_{\mathbb{M}}$ can take operands that are floating-point representable, and those that are not. There are no mixed comparison operators $<_{\mathbb{M}} \mid =_{\mathbb{M}}$, as they are identical to the real operators $< \mid =$.

The semantics of MRFPA formulas is defined recursively via an overloaded evaluation function $[\![\cdot]\!]$ that maps $\mathbb{R}$ terms to elements of $\mathbb{R}$, $\mathbb{F}$ terms to elements of $\mathbb{F}$, and formulas to a Boolean value, as follows. Let $A_{\mathbb{R}}\colon Var_{\mathbb{R}} \to \mathbb{R}$ be an $\mathbb{R}$ assignment to variables in $Var_{\mathbb{R}}$, and $A_{\mathbb{F}}\colon Var_{\mathbb{F}} \to \mathbb{F}$ be an $\mathbb{F}$ assignment to variables in $Var_{\mathbb{F}}$. The semantics of terms is as follows: $[\![c]\!] = c$ for constants $c \in \mathbb{R} \cup \mathbb{F}$, $[\![v]\!] = A_{\mathbb{R}}(v)$ for $v \in Var_{\mathbb{R}}$ and $[\![v]\!] = A_{\mathbb{F}}(v)$ for $v \in Var_{\mathbb{F}}$, and

$$\begin{array}{rcl} [\![t1_{\mathbb{R}}\,\alpha_{\mathbb{R}}\,t2_{\mathbb{R}}]\!] & = & [\![t1_{\mathbb{R}}]\!]\,\alpha_{\mathbb{R}}\,[\![t2_{\mathbb{R}}]\!] \\ [\![t1_{\mathbb{F}}\,\alpha_{\mathbb{F}}\,t2_{\mathbb{F}}]\!] & = & [\![t1_{\mathbb{F}}]\!]\,\alpha_{\mathbb{F}}\,[\![t2_{\mathbb{F}}]\!] \\ [\![t1_{\mathbb{R}}\,\alpha_{\mathbb{M}}\,t2_{\mathbb{R}}]\!] & = & rd([\![t1_{\mathbb{R}}]\!]\,[\![\alpha_{\mathbb{M}}]\!]\,[\![t2_{\mathbb{R}}]\!]) \end{array}$$

where $[\![+_{\mathbb{M}}]\!] = \oplus$, $[\![\times_{\mathbb{M}}]\!] = \otimes$, etc. Operators $\alpha_{\mathbb{M}}$ differ from the corresponding real operators $\alpha_{\mathbb{R}}$ in that they round the result. They also differ from the corresponding floating-point operators $\alpha_{\mathbb{F}}$: the latter take only $\mathbb{F}$ terms as inputs.

The semantics of an MRFPA formula f is then as follows:

$$\begin{array}{rclcrcl} [\![t1_{\mathbb{R}}\,\theta_{\mathbb{R}}\,t2_{\mathbb{R}}]\!] & = & [\![t1_{\mathbb{R}}]\!]\,\theta_{\mathbb{R}}\,[\![t2_{\mathbb{R}}]\!] & \quad & [\![\neg f]\!] & = & \neg[\![f]\!] \\ [\![t1_{\mathbb{F}}\,\theta_{\mathbb{F}}\,t2_{\mathbb{F}}]\!] & = & [\![t1_{\mathbb{F}}]\!]\,\theta_{\mathbb{F}}\,[\![t2_{\mathbb{F}}]\!] & & [\![f1 \vee f2]\!] & = & [\![f1]\!] \vee [\![f2]\!] \end{array}$$

Our definition of MRFPA ignores numeric anomalies such as infinities and NaNs; see discussion in Sect. I.

The use of MRFPA as proxy theory requires specific solver support, such as obtained by extending the tool Realizer [11]. The tool translates floating-point formulas into numerically equivalent formulas over mixed real-integer arithmetic (RIA): it replaces $x \oplus y$ by $rd(x+y)$, where rd encodes rounding as a RIA operation involving floor and ceiling functions. Our (straightforward) extension permits MRFPA as input, not just floating-point formulas.

In practice, deciding real-integer arithmetic is costly and in fact undecidable in the non-linear case. We have therefore experimented with MRFPA as proxy theory only for linear formulas; the prospects for extending this to richer classes are discussed in Sect. VIII.

VI. Experimental Evaluation

The techniques described in this paper have been implemented in our tool MOLLY (roughly, "Model Lifter"), both with reduced-precision FPA as proxy theory (called "RPFPA" in the sequel), and with real arithmetic as proxy.

Tool set-up. For our RPFPA experiments, we used MATHSAT [6, v5.3.8] to obtain proxy models and also to solve the constraint during the lifting of proxy models to FPA models. For lifting, MOLLY picks one variable at a time; currently in an arbitrary way. The formula refinement process increases the number of bits in the exponent by 1, and in the mantissa by 3.

We compare against MATHSAT and against the technique presented in [12, called "Approx" there and in Table I]. We used MATHSAT with the options `input=smt2` , `-model` , `-theory.eq_propagation=false` and `-theory.fp.bit_blast_mode=1` both when used inside our tool and also when used stand-alone for the comparison. For comparison with "Approx", we used our own tool MOLLY but with model lifting *turned off*: our routine *toFloat* then exactly implements the "padding" used in [12]. Not using their implementation allows us to exactly assess the contribution of the lifting.

All evaluations were performed on a machine with Intel(R) Core (TM) i7-4770 3.40GHz CPU, having 8 GB RAM and running x86_64 Ubuntu 14.04 LTS. An overall timeout (TO) of 20 min was used for each benchmark for every tool.

Benchmarks. We evaluated our technique primarily on two benchmark sets. The first benchmark set, named "I. Non-linear benchmarks from [4]" in Table I, contains a mix of 213 formulas from prior published work [4]. Since we currently do not support casts, we ignored them and interpreted all operations as being for the same (single) precision. We also disallowed special floating-point values in the solution by adding the SMT-LIB assertion *fp.isNormal* for every variable.

The second set of benchmarks, named "II. False Identity benchmarks" in Table I, were created by us and are available for download here. These are formulas of the form $E - \widehat{E} > \epsilon$ along with range constraints on the input variables; the expression $\widehat{E}$ is obtained from E using a real-arithmetic rewrite rule, i.e. $\widehat{E}$ is mathematically equivalent to E. Some of the simpler polynomials, for instance, involve factors, e.g. comparing deviation of $x^3 - y^3$ from the product of its factors $(x-y)$ and $(x^2 - xy + y^2)$, for a specific ordering of operations. We have formulas for such comparisons for a variety of polynomials, ranging from Horner scheme evaluations to power series expansions for the *sine* function. Such decision problems are relevant for optimizing compilers since a rewrite based on an equivalence in real arithmetic is often unsafe in FPA. These benchmarks are all satisfiable and values of ϵ were chosen such that MATHSAT solves each of these in less than 5 min.

Results. Running MOLLY on the first set of non-linear benchmarks confirmed the results reported in [12]: solving a simpler reduced precision approximation, often with the initial reduced precision of 3 bits for each of mantissa and exponent, suffices to solve a significant number of the satisfiable constraints. There were only some opportunities for numeric model lifting; the results on all those 22 benchmarks are reported in the set "I. Non-linear benchmarks from [4]" in Table I (1–22). A majority of these benchmarks turned out to be satisfiable and for the rest the satsifiability status is still unknown. To evaluate effectiveness of model lifting, a liberal timeout of 12 min was set for the numeric model lifting step. From Table II, MATHSAT solves one benchmark more than MOLLY, which in turn solves one more than "Approx". The average solving time per solved benchmark for MOLLY (219s) is greater than that for "Approx" (127s) but lesser than that for MATHSAT (443s). For the set of "False Identity benchmarks" in Table I (23–37), we used a timeout of 3 min per iteration for the reduced precision solving and a timeout of 1 min for the model lifting stage. MATHSAT and MOLLY solve all the 15 benchmarks, with MOLLY taking the least average time per benchmark (86s), closely followed by "Approx" (89s), which timed out on two.

Real arithmetic as proxy theory

We also evaluated MOLLY on a set of constraints consisting of *linear* formulas that involve checking non-associativity of FPA operations. We assumed single-precision FPA, with *round-to-nearest-even* rounding mode for FPA operations. MOLLY uses our real arithmetic abstraction detailed in Sect. V. For solving MRFPA formulas, we extended the tool REALIZER [11], which previously accepted pure FPA formulas as input, to also accept MRFPA formulas that are generated in the first formula refinement. In this case, the refinement step marks real arithmetic operators in some parts of the formula as FPA operators.

In Table III, *#Vars* indicates the size of the formula, e.g. for *#Vars*=5, the decision problem is

$$(((a_1+a_2)+(a_3+a_4))+a_5) > ((((a_1+a_2)+a_3)+a_4)+a_5).$$

MOLLY outperforms MATHSAT and is also seen to scale well. In each case, after a few iterations, our model lifting technique succeeded in transforming a real arithmetic assignment into a satisfying floating-point assignment. Based on a simple analysis of the behavior of the expressions constituting the formula in the neighborhood of the approximate assignment, a single variable was chosen to invert the result of the comparison. As before, we used our tool with model lifting disabled to mimic the tool "Approx" from [12]. Here we also ran the actual tool from [12]: it performed many more iterations and eventually timed out on each instance.

	MOLLY			APPROX [12]		MATHSAT
Problem	*It*	*Lifted?*	*Time (s)*	*It*	*Time (s)*	*Time (s)*
I. Non-linear benchmarks from [4]						
1	1	✓	7.8	2	5.0	344.0
2	1	✓	15.8	2	12.3	986.5
3	2	×	60.1	2	45.6	995.9
4	-	-	**TO**	-	**TO**	977.6
5	-	-	**TO**	-	**TO**	983.6
6	-	-	**TO**	-	**TO**	977.1
7	-	-	**TO**	-	**TO**	983.5
8	-	-	**TO**	-	**TO**	**TO**
9	8	×	337.1	8	330.8	**TO**
10	-	-	**TO**	-	**TO**	**TO**
11	1	✓	3.2	2	0.3	61.8
12		×	680.5	2	0.3	**TO**
13	7	✓	863.3	-	**TO**	**TO**
14	-	-	**TO**	-	**TO**	**TO**
15	-	-	**TO**	-	**TO**	**TO**
16	8	×	484.7	8	116.6	46.7
17	8	×	350.3	8	322.2	47.0
18	2	✓	4.9	6	29.4	46.8
19	2	✓	22.1	3	32.5	47.2
20	1	✓	3.3	2	6.3	46.5
21	2	✓	263.4	3	599.9	46.8
22	3	✓	39.1	4	118.8	65.7
II. False Identity benchmarks						
23	3	✓	148.6	8	163.7	60.5
24	2	✓	64.6	8	137.9	108.4
25	8	×	162.7	8	137.2	108.4
26	1	✓	0.9	8	137.2	108.2
27	8	×	278.2	8	162.8	47.7
28	1	✓	12.4	8	123.1	51.8
29	4	×	70.2	4	9.8	112.4
30	2	✓	62.6	8	108.5	108.7
31	3	✓	144.5	8	172.4	122.5
32	3	✓	157.2	-	**TO**	133.6
33	1	✓	1.1	4	0.6	133.6
34	4	✓	181.4	-	**TO**	605.4
35	1	✓	2.1	8	7.7	596.5
36	1	×	0.1	1	0.1	0.3
37	3	×	0.5	3	0.5	0.3

TABLE I: Numeric model lifting on non-linear problems.
"It." = # of iterations; *Lifted?* = ✓ if final satisfying assignment obtained via model lifting, otherwise (via $toFloat$) = ×

Table I, Table II and Table III indicate MOLLY is efficient on benchmarks that require staying close to the original precision to find satisfying assignments. Numeric model lifting then closes the gap between the abstract but imprecise (with respect to FPA) solutions and genuine floating-point arithmetic.

VII. RELATED WORK

The idea of using real arithmetic to solve floating-point constraints approximately has been implemented before [2]. The earlier approach uses this real arithmetic approximation only once for a formula and is hence incomplete, for instance, a formula that is unsatisfiable in the reals but satisfiable in floating-point can not be handled. In contrast, as shown in Sect. VI, we can handle such an input formula by refining the formula iteratively when the answer obtained in an iteration is not a correct answer to the original formula.

The above mentioned earlier work aims to detect exceptions in floating-point programs, by encoding, in real arithmetic, path conditions of programs as well as exceptional conditions like underflow, overflow, division by zero and certain invalid operations involving NaN. This approximation, ignoring rounding entirely, was sufficient to detect several exceptions, primarily of the underflow and overflow types, in a publicly available library. However, we eventually encode rounding for every operation as per the IEEE 754 standard, as we intend our procedure to be used to uncover bugs due to rounding, for instance, in floating-point comparisons in control flow conditions.

A framework for using abstractions that are neither under approximations nor over approximations of the original formulas was proposed recently [12]. These approximations are refined iteratively as necessary. The authors instantiated this framework for floating-point arithmetic using lower precision floating-point numbers. We extend this idea using numeric model lifting techniques.

In the above work, the authors mention very simple heuristics, like padding the solutions with 0s, for lifting a satisfying assignment from a lower (s, e) to one for the actual problem, but these are unlikely to succeed for many cases, especially in the context of detecting anomalies due to floating-point peculiarities or when the approximate assignment contains

		MOLLY	"APPROX" [12]	MATHSAT
I	# Solved	14	13	15
	Total Time(s)	3067	1650	6656
	Avg. Time(s)	219	127	443
	# TO	8	9	7
II	# Solved	15	13	15
	Total Time(s)	1287	1161	2237
	Avg. Time(s)	86	89	149
	# TO	0	2	0

TABLE II: Statistics for data from Table I. Total Time is the sum of solving times for the solved instances

	MOLLY			APPROX [12]		MATHSAT
#Vars	*It*	*Lifted?*	*Time (s)*	*It*	*Time (s)*	*Time (s)*
35	6	✓	30.5	15	153	81.6
40	3	✓	11.9	7	34	278.2
45	8	✓	448.6	33	**TO**	457.1
50	5	✓	25.1	20	344	164.5
55	5	✓	28.3	16	210	754.8
60	3	✓	17.2	34	**TO**	**TO**
65	7	✓	42.0	11	88	**TO**

TABLE III: Demonstrating numeric model lifting with Real arithmetic proxy theory on FPA-specific problems

non-integral values.

In the decision procedure world, the tools Z3 and MATHSAT have support for floating-point arithmetic, primarily based on bit vector reasoning and bit-blasting. With increasing size and complexity of FPA constraints, the resulting propositional encoding becomes very large, which is problematic especially if the input formula itself is large, or when the formula has non-linear arithmetic operations. An attempt was made to alleviate this problem by applying a combination of under- and over-approximations to the same formula [5].

Goubault and Putot [8] present abstract domains and methods to bound the difference between floating-point and real-arithmetic interpretations of the program, and these have been incorporated into FLUCTUAT [7], and can be used for test-case generation. Abstract interpretation and interval arithmetic techniques provide clear efficiency benefits over model exploration approaches such as ours, and feature a high level of automation. They have been successfully applied in industrial contexts. On the other hand, they are approximate and may not suffice when accurate analysis is paramount. This is reflected especially in the potential for spurious assignments.

Various formalizations and libraries for FPA have been developed in the domain of theorem proving [9]. More recently, these provers have been used to certify programs [3]. The use of such tools requires expert skills to provide hints to steer the theorem prover towards the goal. In contrast, model exploration approaches such as ours aim at principally push-button techniques.

VIII. CONCLUSIONS

We have presented a framework for building solvers for floating-point decision problems, by reducing them to decisions in some proxy theory T. The assumptions are that (i) T models are often close to FPA models, and (ii) T formulas are on average easier to decide than FPA formulas. Examples of suitable proxy theories include reduced-precision FPA and real arithmetic. Previous work embeds such reductions into a CEGAR loop [12]. Our framework extends it by a *numeric model refinement* procedure, which tries to lift T models to FPA models. The procedure determines, using a floating-point solver, how much certain variables need to be adjusted away from the T model, to compensate for the difference between T and FPA. We derive a new formula with a simpler structure and fewer free variables, and whose satisfiability immediately gives rise to an FPA model. Experimental results indicate our technique can find satisfying assignments efficiently.

Future work. We plan to extend our work in two main directions. One is the use of *approximate* numeric techniques, rather than (exact) decision procedures, to solve formulas in the proxy theory T: thanks to model lifting, a precise solution in T is not required for the first green box in Fig. 1. This relaxation opens up a host of other and potentially very scalable techniques especially for complex non-linear input constraints, including for the case of real arithmetic as proxy theory, for which we currently have limited support for non-linear formulas. The other direction is to improve our strategy for dealing with unsatisfiable formulas, rather than just "waiting" for the refinement to revert f_T back to f; the latter causes all model finding efforts to be wasted.

REFERENCES

[1] CBMC. http://www.cprover.org/cbmc/, accessed: 2015-03-23
[2] Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 549–560. POPL '13, ACM, New York, NY, USA (2013)
[3] Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in coq. In: 20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011. pp. 243–252 (2011)
[4] Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design 45(2), 213–245 (2014)
[5] Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD. pp. 69–76 (2009)
[6] Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS. LNCS, vol. 7795. Springer (2013)
[7] Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings. pp. 53–69 (2009)
[8] Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. pp. 232–247 (2011)
[9] Harrison, J.: A machine-checked theory of floating point arithmetic. In: Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings. pp. 113–130 (1999)
[10] Institute of Electrical and Electronics Engineers (IEEE): 754-2008 — IEEE standard for floating-point arithmetic. IEEE pp. 1–58 (2008)
[11] Leeser, M., Mukherjee, S., Ramachandran, J., Wahl, T.: Make it real: Effective floating-point reasoning via exact arithmetic. In: DATE. pp. 1–4 (2014)
[12] Zeljic, A., Wintersteiger, C.M., Rümmer, P.: Approximations for model construction. In: IJCAR. pp. 344–359 (2014)

Trustworthy Specifications of ARM® v8-A and v8-M System Level Architecture

Alastair Reid
Research, ARM Ltd.
first.last@arm.com

***Abstract*—Processor specifications are of critical importance for verifying programs, compilers, operating systems/hypervisors, and, of course, for verifying microprocessors themselves. But to be useful, the *scope* of these specifications must be sufficient for the task, the specification must be *applicable* to processors of interest and the specification must be *trustworthy*.**

This paper describes a 5 year project to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate ARM's conventional architecture documentation. We have developed executable specifications of both ARM's A-class and M-class processor architectures that are complete enough and trustworthy enough that we have used them to formally verify ARM processors using bounded model checking. In particular, our specifications include the semantics of the most security sensitive parts of the processor: the memory and register protection mechanisms and the exception mechanisms that trigger transitions between different modes. Most importantly, we have applied a diverse set of methods including ARM's internal processor test suites to improve our trust in the specification using many other expressions of the architectural specification such as ARM's simulators, testsuites and processors to defend against common-mode failure. In the process, we have also found bugs in all those artifacts: testing specifications is very much a two-way street.

While there have been previous specifications of ARM processors, their *scope* has excluded the system architecture, their *applicability* has excluded newer processors and M-class, and their *trustworthiness* has not been established as thoroughly.

Our focus has been on enabling the formal verification of ARM processors but, recognising the value of this specification for verifying software, we are currently preparing a public release of the machine-readable specification.

I. Introduction

Recent years have seen an increasing focus on verification of machine-code programs [1], compilers [2], operating system kernels [3], hypervisors [4] and processors [5]. These activities rely on having correct specifications of the meaning of machine-code and one of the first steps in such verification efforts is creating a specification of the computer architecture of interest.

Three key properties of a processor specification are its *scope*, its *applicability* and its *trustworthiness*.

The *scope* of a specification is the set of features that one can reason about. For example, a certified compiler such as CompCert [2] only requires a specification of those instructions that the compiler could generate. But in order to reason about arbitrary user-mode binaries, one would need a specification of the entire instruction set. And to reason about Operating System code, the scope of the specification is dramatically increased and includes a specification of instructions for changing execution mode (e.g., entering/leaving supervisor mode), interrupt handling mechanisms, page faults, mechanisms for changing memory protection, etc. To date, all formal specifications of the ARM architecture have been targetted at reasoning about user-mode programs and have not included a specification of these system-level features.

The *applicability* of a processor specification is whether the specification applies to the target processor. Most changes to architecture specifications are backward compatible extensions and so most proofs about code for one architecture version are valid when executing that code on a processor implementing a later architecture version. But architecture revisions also remove instructions, add restrictions or change functionality so proofs based on the ARMv6 specification (1996) or the ARMv7-A specification (2007) are not necessarily sound for ARMv8-A (2013). This is especially true for ARM's Microcontroller architecture which has a completely different exception model from ARM's mainstream architecture.

The *trustworthiness* of a processor specification is whether the specification can be trusted to reflect the behaviour of all processors implementing the specification. The ARMv7 HOL specification of Fox and Myreen [1] is noteworthy for the degree of testing performed: systematically testing all user-mode, integer instructions against three actual processors. This is a critical step and must be repeated against as many expressions of the architecture as possible (processors, implementations, testsuites, etc.) and must be used to test the full scope of the specification.

The effort required to create a specification increases with the desired scope, applicability and trustworthiness of the specification. Worse, since ARM regularly releases extensions and corrections to the architecture, the challenge of retaining applicability to current processors is more of a continuous process rather than a one-off sprint. Our solution to this problem has been to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate conventional documentation.

This paper describes our work over the last 5 years on transforming the ARM processor specifications from documents intended for human consumption into trustworthy machine-readable specifications.

Creating this specification required understanding and cod-

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

ifying the precise meaning of various notations used in the documentation; inferring the lexical, syntax, type rules and semantics from examples in the documentation; making the specification conform to these rules; filling gaps in the original specification; and creating a frontend and several backends to allow the specification to be executed.

Using ARM's specifications directly addresses the issues of *scope* and *applicability* but the resulting formal part of the specification is just one part of the whole specification and, like any large specification, may contain bugs wrt the informal parts of the specification or with the architects' informal intent. To address the issue of *trust*, we have used a diverse set of testing methodologies to compare against as many different expressions of the specification as possible: testsuites, simulators and processors. We have simulated billions of instructions and used bounded model checking to compare the RTL of five ARM processors currently in development against the specification [6]. Bugs found in the process have been fixed in the master copy of the specification from which ARM's architecture specification documents are generated. This process has the effect of distilling more of the architectural intent into the formal part of ARM's official specification.

The structure of this paper is summarized in Figure 1 which gives an overview of the specifications, tools, verification IP, and testing we created or used in the process of this project. Section II gives a brief overview of the structure and content of the different ARM Architectures. Sections III and IV describe the steps we took to convert ARM's existing informal documentation into machine-readable, executable, trustworthy specifications of the ARM-v8A and ARM-v8-M architectures; Section V discusses related work; and Section VI concludes.

This paper deals with the Instruction Set Architecture (ISA), Exceptions, Memory Protection/Translation and Security. It does not deal with multiprocessor features and, in particular, the Memory Ordering Model [3], [7], [8]. And it does not deal with debug or performance monitoring features.

II. ARM Specifications

ARM Architecture specifications have two main sections: Application Level Architecture and System Level Architecture.

The Application Level Architecture (aka the Instruction Set Architecture or ISA) consists of all instructions and all user-mode registers (the integer and floating point register files, condition flags, stack pointer and program counter). ISA specifications consist of instruction encodings, matching rules to match encodings to opcodes and the semantics of instruction execution.

The System Level Architecture defines Memory Translation and Protection, Synchronous Exceptions (e.g., page faults and system traps), Asynchronous Exceptions (e.g., interrupts), Security (e.g., register banking and access protection of registers), and System Registers and System Operations (which are used to control and read the status of all the system-level features), In other words, the facilities needed to support Operating Systems, Hypervisors and Secure Monitors.

The ARM architecture comprises three main processor classes: "A-class" processors support Applications (characterized by having an operating system that uses address translation to provide virtual memory); "R-class" processors support Real-Time systems that cannot handle the timing variability associated with virtual memory and use memory protection instead; and "M-class" microcontrollers are optimized for programming interrupt-driven systems in the C language. The A-class specification consists of two parts: *AArch32* supports 32-bit programs and is generally backward compatible with ARM's traditional architecture; and *AArch64* which supports 64-bit programs.

The A- and R-class architecture [9] share the same ISA and exception model but have different memory protection/translation models. The M-class architecture [10] has a subset of the A-class ISA but has significant differences from A-class at both the Application Level and System Level.

A. ISA Differences between A/R- and M-class

The M-class architecture only supports the Thumb® (aka "T32") variable-length instruction encodings whereas the A/R-class architecture also supports the A32 and A64 encodings.

Much more significantly though, the specifications identify certain instruction encodings as UNPREDICTABLE for which a processor is free to do anything that can be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and that does not halt or hang the processor or parts of the system.

In the M-class architecture, many of the instruction encodings which access the stack pointer (R13) or the program counter (R15) are UNPREDICTABLE but the same encodings are well defined in the A/R-class architecture. This is a significant difference — it would be unsound to use the A-class specification to reason about Thumb machine code intended for an M-class processor.

More broadly, when performing formal verification, it is essential to ensure that the specification version being used matches the architecture version supported on the target processor because later specifications are *almost* but *not entirely* backward compatible. This is obvious but easily overlooked.

B. System Differences between A-, R- and M-class

The R/M-class architectures support memory protection based on setting attributes and protection for a small number of contiguous memory regions whereas the A-class architecture supports both address translation and memory protection for a large number of memory pages.

M-class processors automatically save the callee-save registers on the stack on taking an exception whereas A/R processors require registers to be saved in software. This allows M-class processors to respond more quickly to interrupts and also allows exception handlers to be written in plain C with no assembly language or special calling conventions. This has a large impact on the architecture specification since it introduces many corner cases associated with the effect of triggering memory faults while saving or restoring registers.

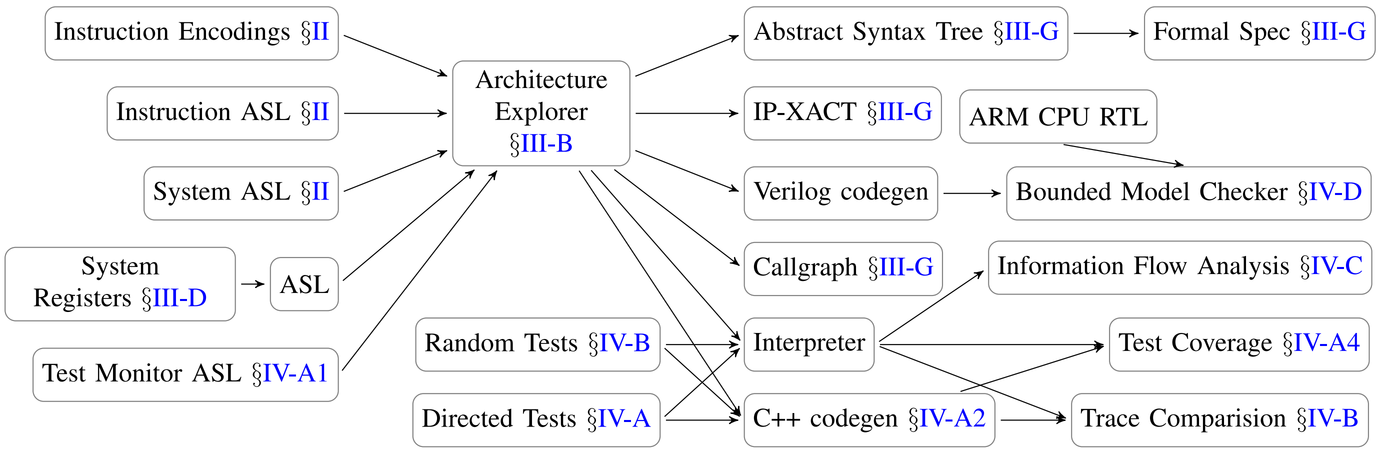


Fig. 1: Overview of specifications, tools, verification IP and testing. This flow was applied separately to the v8-A specification and to the v8-M specification. Section numbers indicate which section primarily discusses each aspect.

M-class processors have an orthogonal set of 8 execution states composed of combinations of three properties: privileged/unprivileged, secure/non-secure and handler/thread. A/R-class processors have a more traditional set of nested execution states EL0, EL1 (supervisor), EL2 (virtualization) and EL3 (secure monitor) with increasing levels of privilege at each level.

A consequence of these differences is that the M-class system specification is completely different from the A/R-class system specification.

III. Executable Specifications

We faced five major challenges in turning ARM's documentation-based specification into an executable specification: (1) Scale: ARM specifications are very large; (2) Informality: ARM specifications are written in "pseudocode"; (3) Gaps: key parts of the specification only existed in natural language specification; (4) System Register Specifications; and (5) Implementation Defined Behaviour.

A. ARM Specifications Are Large

One of the main challenges in creating machine-readable specifications of the ARM Architecture is the scale of the problem. The A and M-class architectures together consist of over 6,000 pages of documentation, 1,570 instruction encodings, over 50,000 lines of pseudocode, over 4,500 system register fields grouped into 772 system register, and 112 system operations. To this specification that ARM publishes, we added an additional 8,190 lines of support pseudocode which were required to make the execution executable. (A more detailed breakdown of the size of the specification is given in table 2a and table 2b.)

B. Pseudocode

A secondary challenge in creating a machine readable specification was that the bulk of the specification is written in what the ARM documentation refers to as "pseudocode". For example, the T32 CMP instruction is specified with the following encoding diagram and pseudocode in the v8-A architecture. (The same instruction is UNPREDICTABLE in v8-M if "m == 13".)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
CONDITIONAL
    n = UInt(Rn); m = UInt(Rm);
    (shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
    if n == 15 || m == 15 then UNPREDICTABLE;
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

Fortunately for us, this "pseudocode" was fairly complete and it appeared possible to implement a conventional parser, typechecker and interpreter for pseudocode (a tool we call "Architecture Explorer"). Through a process of experimentation, discussion and negotiation with the architecture designers, we were able to infer consistent indentation rules, precedence rules, a type-system and semantics and to clean up the specifications to use the resulting simpler, more consistent language that is now internally referred to as ARM Specification Language (ASL).

At a high level, ASL is an indentation-sensitive, imperative, strongly typed, first-order language with dependent types (to reason about length of bit vectors), type inference, exceptions, enumerations, arrays, records, no pointers. Unusually for an otherwise simple language, ASL allows overloading of array syntax for function calls: the use of "R[m]" and "R[n]" on lines 4 and 5 of the example above are both function calls. This syntactic sugar provides an initial impression that registers (and memory) are simple arrays, while allowing one to dig deeper and understand register banking, virtual memory, etc. We refer readers to Fox and Myreen [1] or to ARM's specification [9, Appendix G] for a more detailed description of ASL.

The initial cleanup of syntax and type errors resulted in changes to approximately 12% of the lines of code but,

	ARMv8-A				ARMv8-M	
	AArch32	AArch64	Shared	Support	Spec	Support
Instrs.	18318	5757			4998	
Integer	23		352		246	
Float Point			1179		953	76
Exceptions	1474	1611	235		781	
Registers	310	446	398		2011	461
Memory	1584	1169	393		369	481
Debug	675	537	1103			
Instr. Fetch				199	367	128
Test Monitor	-	-	-	1323	-	1893
Misc.	1647	1137	2984	1678	415	1434
Total	24315	10657	5489	3200	9898	4990

(a) Size of ASL specification (lines of code)

	v8-A	v8-M
Registers	586	186
Fields	3951	622
Constant	985	177
Reserved	940	208
Impl. Defined	70	10
Passive	1888	165
Active	68	62
Operations	112	10

(b) Size of System Register specification

Fig. 2: Size of ARM Specifications

since ARM specifications are extensively reviewed before release, these were all fairly low-grade errors: they confused automatic tools but few were likely to confuse a human reader. The process of cleaning up the specification also uncovered a number of instances of "implement by comment" where comments were used instead of pseudocode: these parts had to be rewritten before the code could be executed. These simple comments often turned out to be surprisingly complicated and the process of writing code would identify corner cases or the need to modify other parts of the specification.

C. Gaps in the specification

Some parts of the architecture were only defined in English and the information to implement them was typically scattered throughout the documentation. An example is the specification of the "top-level" step of fetching an instruction, decoding and executing it, and incrementing the program counter was not written in ASL and the description was scattered across the specification document. The exact specification of this step took some time to develop as it includes details like dealing with page faults that occur during instruction fetch, not incrementing the PC after a branch instruction or exception, conditional execution of instructions and its interaction with UNDEFINED encodings, and testing for pending interrupts.

D. System Register Specification

The major negative surprise of this project was how hard it was to specify something as apparently simple as a register.

The A-class architecture specification comprises 586 system registers which are used to read the status of and to control the behaviour of the processor (such as whether the MMU or cache is turned on) and to perform operations such as flushing the cache or invalidating the TLB. The main properties of these registers are captured in the architecture specification by tables specifying the opcode to access each register, its name, size (32/64-bits) whether it is read-only and the reset value of the register. For each register, there is a description consisting of a register diagram which identifies the name and extent of any used bits in the register. And each such field of contiguous bits has a natural language specification.

The challenge in creating a machine-readable specification for system registers is that different fields within the register can behave in several different ways. After some experimentation we settled on identifying five major types of field.
i) *Constant fields* have an architecture defined value and cannot be changed.
ii) *Reserved fields* are not used in the current version of the architecture but could be assigned a meaning in future versions of the architecture. These are like constant fields but, to maintain forward compatibility, software should not assume that the field is constant and should avoid changing the value of that field.
iii) *Implementation Defined fields* have an implementation defined value that programs may read to determine whether the processor has some ISA or system level feature.
iv) *Passive fields* behave like a global variable and simply store the value last written to the field. The value written often has a significant effect such as enabling address translation but this effect is completely captured by the ASL functions implementing the affected behaviour.
v) *Active fields* do not behave like a global variable: reading the field may not see the last value written to the field; writing to the field may be disabled by the value of some other register; etc. These are used for everything from system timer registers (which decrement every cycle) to allowing a hypervisor to intercept interrupts targetted at the guest operating system.

Fields that are Constant, Reserved, Implementation Defined or Passive are easy to describe completely and are described in a simple table-based format but 68 of the fields of system registers are Active fields whose behaviour can only be captured by writing ASL getter and setter functions to implement the natural language specification. The process of implementing registers with active fields proved to be quite error prone as the behaviour of the fields was rather subtle.

It was also hard to find the correct design point. We chose to identify just 5 classes of field but we could have identified further common patterns within the Active class. For example, there are some pairs of registers that have complementary effects such as enabling and disabling exceptions. If this pattern is a one-off, it is probably best described as an Active register but if the pattern occurs in several pairs of registers, then the argument for recognizing it as a new class of field becomes stronger. As the number of tools using the system register specification grows, we expect that we will identify a number of patterns that are useful to recognise explicitly because that enables tools to make more use of the specification without having to embed the ASL parser/interpreter.

One significant aspect of system registers not yet captured in the executable specification is what Lustig et al. [8] call a *memory transistency model* which captures places where the specification allows reordering of writes to system registers with respect to other instructions and requires insertion of instruction barrier instructions (ISB) to restrict.

E. Implementation Defined Behaviour

The specification allows for some implementation defined behaviour such as whether a particular feature is implemented or the number of memory protection regions supported. This behaviour is often specified by "stub functions" returning booleans or an enumerated value and with a natural language definition. We had to implement these stub functions before we could execute the specification. In most cases, these feature test functions could be implemented by testing a corresponding implementation defined field.

F. Executable Specification

After creating all the tooling, bugfixes, etc. described above, there were some further steps required to make the specification executable so that it could be tested. We had to add additional infrastructure such as generating decode trees for a set of encodings to identify which instruction to execute; ELF readers to load test programs into memory; a physical memory implementation which allocates pages of memory on demand. and breakpoint and trace facilities to use when debugging.

We also introduced a continuous integration flow where every specification change runs regression tests. This was critical for confining new code to the ASL subset of pseudocode.

G. Machine Readable Specifications

Our primary goal in doing the above was not to make the specification executable but, rather, to improve its quality so that the specification is useful to many potential users. To support these uses, we generate a variety of machine-readable outputs.
i) *IP-XACT* is a standard XML-based format for describing registers in a chip [11]. It is used by debuggers needing to view or change the value of a register.
ii) *Callgraph summaries* are convenient summaries of the function calls and variable accesses performed by each instruction and function in the specification. One use of these summaries is in generating a summary of the list of exceptions that an instruction can raise — for inclusion in documentation.
iii) *Abstract Syntax Trees* are a complete dump of Architecture Explorer's internal representation after typechecking. We have provided these to the University of Cambridge REMS group who are in the process of transforming them into a form suitable for formal verification of machine-code programs.

IV. Trustworthy Specifications

ARM spends considerable effort on reviewing specifications. It also benefits from feedback from users of the specifications: processor designers, verification engineers, implementers of simulators, compiler writers, etc. Nevertheless, the sheer size of the specification made it unlikely that the specifications are bug-free. This was especially true of the relatively fresh v8-M specification since it had not yet had the benefit of feedback from users of the specification.

This Section describes the steps we have taken to test the v8-A and v8-M specifications using testsuites, random instruction sequences, information flow analysis and using bounded model checking to compare against the Verilog implementation of processors. One of the recurring themes of this project was that this testing process improves the specification and our trust in the specification — but it also improves the tools, verification IP, etc. that is being used to test the specification which creates a virtuous cycle of improving any other uses of those tools and artifacts.

A. Using ARM Processor testsuites

ARM performs extensive testing of its processors and simulators (it is estimated that more than 80% of the engineering effort of designing a new processor is spent on testing the processor). One part of this testing process is use of ARM's Architecture Validition Suite (AVS) which consists of programs that test the architectural conformance of individual instructions, memory protection, exception handling and all other aspects of the architecture. Excluding multiprocessor and debug tests, the AArch64 AVS consists of over 11,000 test programs with a combined runtime of over 2.5 billion instructions; the M-class AVS consists of over 3,500 test programs with a combined runtime of over 250 million instructions. Almost all of these tests were considered to be free of assumptions about instruction timing or implementation defined behaviour. (ARM has a large number of other tests which were less appropriate to run because they are aimed at testing micro-architectural performance optimizations in particular processors.)

Using ARM's official Architecture Validition Suite has some significant advantages: the suite is very thorough, checks many corner cases, and has good control and data coverage of the architecture; the suite is self-checking: each test prints "PASSED" or "FAILED" when it runs; and, since the purpose of the tests is to test processors, it was possible to compare the behaviour against actual processors for additional confidence. The primary disadvantage of using the AVS was that the tests are "bare metal" tests that exercise the System Level Architecture and require a large test harness to run.

As we started using Architecture Explorer to develop new architecture extensions (such as the new security features of v8-M), we encountered a chicken-and-egg problem: the AVS is extended with new tests only once the architecture specification is available but we were still writing the specification. Worse, v8-M is not entirely backward compatible with the previous architecture version so we could not even run the old tests. This led us to use a hybrid approach: we temporarily created a modified specification supporting the old memory protection design so that we could use the old tests; and we created a temporary test suite to test the new security features of v8-M (see Section IV-C) before the official test suite was developed. Once updated AVS tests became available, we switched to using the official test suite.

1) Programmable Monitor and Stimulus Generator: Part of the development of every ARM processor is creating a test harness which allows the AVS to be run. This test harness consists of a programmable monitor and stimulus generator that allows programs to monitor their own behaviour at a very low-level. The test monitor design dates back to the earliest days of ARM and each successive architecture extension typically adds new test features.

The monitor consists of 177 memory mapped registers of which 45 are Active. The main features of the test monitor are

(i) Console FIFO for writing ASCII text to log file.

(ii) Memory attribute monitors which record the attributes of memory accesses in a given range of addresses. This allows test programs to verify that the MMU/MPU is correctly associating attributes such as cacheability of an access with each address. These checkers are repeated for each bus interface.

(iii) Memory abort generators to trigger a bus fault response if the processor accesses a specified range of addresses.

(iv) Interrupt generators to test triggering, prioritization and nesting of interrupts.

(v) Reset generators to schedule resets.

2) Optimizing the simulator: During this testing process, we slowly built our capability from being able to execute one instruction to being able to execute most usermode instructions, to being able to execute entire tests and then entire testsuites. As we did so, we were increasingly limited by the performance of our interpreter which initially ran at a few hundred instructions per second. Over time, we have optimized this in a variety of ways increasing performance to 5kHz (v8-A) and 50kHz (v8-M). The main optimizations applied are: (i) Memoizing a few critical functions associated with the current configuration or execution state (this has not been yet been applied to v8-A); (ii) Implementing a few critical arithmetic functions as builtin primitives even if they can be defined in ASL; (iii) Creating a C++ code generator and runtime (including ELF reader, etc.).

3) Testing the specification: One of the issues found while testing the specification initially manifested as a failing AVS test. On closer inspection, we found a mismatch between the English text and the pseudocode and that the test had originally followed the pseudocode and ARM's reference simulator followed the English text. This mismatch had been "fixed" by changing the test to match the simulator. Consulting the architects, we learned that the pseudocode was correct and the English text was wrong and so the English text, the test and the simulator were fixed to match the architects' intent.

The pass rate of our specifications on the AVS is summarized in Table I. We have achieved a 100% pass rate for the v8-A and v8-M ISA tests and for the v8-M System tests. For the v8-A System tests, there remain some failing tests in areas related to interprocessing (switching between 32-bit and 64-bit modes) and prioritization of multiple exceptions within the same instruction. These results omit debug and multiprocessor tests which are just under 50% of the total number of tests.

	ARMv8-A	ARMv8-M
ISA		
Integer	100%	100%
Floating Point	100%	100%
SIMD	100%	100%
System		
Exceptions	100%	100%
Memory	99%	100%
Interprocessing	98%	-

TABLE I: Pass rate for AVS testsuite

4) Testing the testsuite: Testing the specification with a testsuite has the side-effect of testing the testsuite. We found two classes of problems in the process of diagnosing test failures. The first is that a test may depend on some property not guaranteed by the architecture but which had been true in every tested processor. For example, a test might check that a reserved field of a register is always zero and will then fail on later versions of the architecture. Secondly, many of the M-class AVS tests depended on UNPREDICTABLE behaviour but this had not been observed before because, in practice, UNPREDICTABLE behaviour can depend on the particular pipeline state when an instruction runs.

To improve testing of the AVS, we extended the interpreter to collect line coverage information as it executes. A rare example of a coverage hole we found was in a floating point test which tested with inputs that produced the result $+0.0$ but did not test with inputs that produced the result -0.0 — with the result that one of the branches associated with rounding was not being exercised. The AVS development team now routinely measure the architectural coverage of testsuites.

B. Random Instruction Sequence Testing

Random Instruction Sequence (RIS) testing is a complementary technique to the directed testing of using hand-written tests based on generating random sequences of instructions. ARM's RIS tool [12] uses templates that specify the desired distribution of instructions, the likelihood of reuse of a given register, etc. Automatically generating random tests is different from hand-writing tests because it requires an accurate simulator to define the correct behaviour of a test. Also, because RIS generates random sequences of instructions, it is necessary to run the same test on multiple systems (processors, simulators or the specification) and compare execution traces. So at least two models are needed to develop RIS tests.

We were able to use the executable specification as part of the process for testing new RIS tests by extending the simulator to generate a trace and extending the existing trace comparision script to accept those traces. This process was especially useful for the v8-M specification because the v8-M support in ARM's reference simulator was new and had not been fully debugged. Using RIS to test the simulator against the executable specification was an effective way of testing the RIS tests, the simulator and the specification.

This process was able to uncover subtle errors in the specification. For example, v8-M's new security features splits some of the system registers into two banked registers –a non-secure register and a secure register– and the appropriate register is automatically accessed depending on the current security mode. But instructions that switch between secure and non-secure registers start in one mode and end in a different mode and the normally convenient automatic banking mechanism obscures exactly which of the two registers is being accessed. RIS testing found an error in the specification of the Test Target (TT) instruction which queries the security state and access permissions of a memory location.

C. Information Flow Analysis for v8-M

The most significant new feature of the v8-M microcontroller specification is a set of security extensions to enable secure Internet of Things applications.

To improve confidence in both the extensions and in the way they were expressed in the ASL specification, we modified the interpreter to generate dynamic dataflow graphs on which we could perform information flow analyses. Most of the analyses performed can be characterized as a non-interference property: ensuring that non-secure modes cannot see secure data and that non-secure data can only influence secure code in safe ways.

An example scenario tested in this process involved information leaks via interrupts. Interrupts automatically save integer registers on the stack of the interrupted code and zero the integer registers but, in order to keep interrupt latency low, floating point registers are lazily saved on the stack only when/if the interrupt handler uses a floating point instruction. We wanted to ensure that lazy FP state preservation did not introduce security holes. We wrote tests that iterated over all combinations of initial mode, final mode, whether FP registers had been modified and scanned the dynamic dataflow graph for information leaks.

This form of testing caught two classes of bugs. First, it caught bugs in how the architecture specification implemented the architectural intent — resulting in fixes to how the specification was written. Second, and more importantly, it caught bugs in the architectural intent by identifying potential security attacks that had not been considered before.

D. Bounded Model Checking of Processors

We have been using both the v8-A and the v8-M architecture specifications to perform bounded model checking of pipelines for processors currently under development at ARM [6]. This has primarily focused on verifying the ISA-implementation parts of the processor, not the memory system, security mechanisms or exception support. This process has been very effective at detecting bugs in various stages of processor development. But, besides verifying processors, it has another important side-effect of performing a very thorough check that the architecture specification and our tooling agrees with how the processor implementors interpret the specification. We found no errors in the published part of the specification in this process but we did find a rather subtle bug in our understanding of conditional UNDEFINED encodings and UNPREDICTABLE encodings.

The M-class specification requires that conditional execution of an UNDEFINED instruction behaves as a no-op if the condition does not hold and we had assumed that the same was true for UNPREDICTABLE instructions. During verification of a processor, the model checker detected an apparent bug that involved a conditional UNPREDICTABLE encoding but, through discussion between the processor designers and the architects, we learned that there had been a recent clarification of the architecture which said that conditional UNPREDICTABLE encodings are UNPREDICTABLE even if the condition does not hold.

This error in our interpretation of the specification had not been detected by testing because it is very, very hard to construct useful tests of the UNPREDICTABLE instructions because they are almost entirely unconstrained and can branch, change registers, trigger exceptions, etc.

E. Summary

Large specifications are as likely to contain errors as large programs so we have used many different approaches to test the specifications. In the process, we realized that although ARM publishes an official specification, the full requirements are really distributed around many different places in the company: the AVS suite, the reference simulator ARM uses for processor verification, and the processor implementations. The act of testing all these different instantiations of the specification against each other has the effect of centralizing this specification in a single location.

V. Related Work

The most closely related work is that of Goel et al. [13] who have created an executable specification of many key parts of the x86-64 ISA and system architecture including paging, segmentation and both user/supervisor levels. Their model has been verified against real processors using the Pin binary instrumentation tool and they have added a syscall emulation layer to let them run real programs including (amusingly) a SAT solver. This is a monumental piece of work that sets the standard against which other architecture specifications should be judged. Despite the similarities, our different project priorities have led to many differences: (1) They have a specification of user and supervisor levels, we also have a specification of hypervisor and secure monitor levels. (2) They have used their specification to formally verify *software* using theorem proving, we have used our specification to formally verify *hardware* using bounded model checking. (3) They have implemented syscall emulation to let them use user-level programs as tests, we have implemented a test monitor and debugged the EL2/EL3 levels to allow us to run ARM's Architecture Conformance Suite which explores the dark corners of the architecture by running bare-metal programs. (4) They have focussed on modelling the x86-64 64-bit ISA, we have modelled the A64, A32 and T32 ISAs. (5) They have consulted processor designers to understand Intel's architecture specification document, we have had all our bugfixes and clarifications reviewed by ARM's architects and incorporated into ARM's official architecture specification document.

The most closely related ARM specifications are the Fox/Myreen ARM v7-A ISA specification in HOL [1] and Flur et al.'s ISA and concurrency specification in Sail [3] both of which were tested against actual processors using random and directed tests (8400 tests in Flur et al., 281,307 tests in Fox/Myreen). In addition to user-mode instructions, our specification covers both the ARMv8-M architecture and the larger ARMv8-A architecture, includes floating point, Advanced-SIMD and the System Level Architecture. We have tested the entire specification in multiple ways and with a larger range of values and simulated more than 2.5 billion instructions in the process. And we have used a model checker to compare the ISA specification against actual implementations for all instructions, all execution modes, all integer inputs and a subset of floating point inputs [6].

Shi [14] extracted the ISA pseudocode from ARM's v6 Architecture Reference Manual, automatically translated the code to Coq and used that to verify that the ARM model in the SimSoC simulator written in C faithfully implemented the Coq specification. This is an impressive piece of work, and it would be interesting to repeat their work using our new, more trustworthy specification or to extend their proof to cover the system level architecture.

The other major ARM ISA specification that we are aware of is embedded in the CompCert compiler and is used in the proof that the compiler faithfully translates the input C program to ARM assembly code. This specification is limited to a subset of the user-mode ARMv6 specification and there is no published statement of how it was validated.

Hunt created a specification of the FM8501 processor [5] and used it to formally verify the processor. The process of formal verification greatly increases the trust we can place in the corresponding parts of the specification because it ensures that all the corner cases in both the processor and the specification have been explored.

More broadly, anyone wrestling with a large specification is obligated to find ways to verify that the formal specification captures the (informal) requirements.

VI. Conclusions

Historically, ARM's specification efforts have focused on a single set of products: the ARM Architecture Reference Manuals [9], [10]. However, there are many more potential uses of the specification if the specification is delivered in a flexible, machine-readable format – for example, formal verification of hardware and software, tools that manipulate instruction encodings, debug tools, creating hardware verification tests. Traditionally, all these other users manually transcribe parts of the specification into some other notation: HOL, C, Verilog, spreadsheets, etc. This process is laborious and error-prone but, worse, it is fragmented: bugfixes or clarification found by one group are not necessarily propagated to other groups or to the master specification. Our primary goal in this project was to enable formal verification of ARM processors against the specification. But, by supporting as many of these uses as possible, we created a virtuous cycle where bugfixes or improvements were incorporated into the central specification so that all users benefit from bugfixes as well as to amortize the development effort across many uses.

This paper describes the steps required to create trustworthy specifications of the full v8-M and v8-A architectures including the instruction set architecture, memory protection and translation, exceptions and system registers. While checking that a formal specification captures the architects' informal intent is an unending process, we believe that our specification is the most trustworthy and complete system specification of any mainstream processor architecture.

We are currently working with Cambridge University on a public release of our specification suited to verification of machine code programs.

References

[1] A. C. J. Fox and M. O. Myreen, “A trustworthy monadic formalization of the ARMv7 instruction set architecture,” in *Proc. Interactive Theorem Proving ITP 2010*, ser. LNCS, vol. 6172. Springer, 2010, pp. 243–258.

[2] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[3] S. Flur *et al.*, “Modelling the ARMv8 architecture, operationally: concurrency and ISA,” in *Proc. Principles of Programming Languages, POPL 2016*, 2016, pp. 608–621.

[4] M. Dam, R. Guanciale, and H. Nemati, “Machine code verification of a tiny ARM hypervisor,” in *Proc. Workshop on Trustworthy Embedded Devices*, ser. TrustED '13. ACM, 2013, pp. 3–12.

[5] W. A. Hunt, “FM8501: A verified microprocessor,” ser. LNCS, vol. 795. Springer, 1994.

[6] A. Reid *et al.*, “End-to-end verification of ARM® processors with ISA-Formal,” in *Proc. Computer Aided Verification (CAV)*, ser. LNCS, vol. 9780. Springer-Verlag, 2016, pp. 42–58.

[7] J. Alglave, L. Maranget, and M. Tautschnig, “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.

[8] D. Lustig *et al.*, “Coatcheck: Verifying memory ordering at the hardware-OS interface,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2016, pp. 233–247.

[9] ARM Ltd, *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd, 2013.

[10] ——, *ARM v7-M Architecture Reference Manual*. ARM Ltd, 2006.

[11] IEEE, “IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows,” *IEEE Standard 1685-2014*, 2014.

[12] B. Greene and M. McDaniel, *The Cortex-A15 Verification Story*. http://www.testandverification.com/downloads/DVClub-Jan-2012/Cortex-A15-Verification-Story-DVclub-final.pdf, 2011.

[13] S. Goel *et al.*, “Simulation and formal verification of x86 machine-code programs that make system calls,” in *Formal Methods in Computer-Aided Design, FMCAD*, 2014, pp. 91–98.

[14] X. Shi, “Certification of an instruction set simulator,” Ph.D. dissertation, University of Grenoble, July 2013.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Equivalence Checking using Gröbner Bases

Amr Sayed-Ahmed[1] Daniel Große[1,2] Mathias Soeken[3] Rolf Drechsler[1,2]
[1]Faculty of Mathematics and Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany [3]Integrated Systems Laboratory (LSI), EPFL, Switzerland
{asahmed,grosse,drechsle}@informatik.uni-bremen.de mathias.soeken@epfl.ch

***Abstract*—Motivated by the recent success of the algebraic computation technique in formal verification of large and optimized gate-level multipliers, this paper proposes algebraic equivalence checking for handling circuits that contain both complex arithmetic components as well as control logic. These circuits pose major challenges for existing proof techniques. The basic idea of Algebraic Combinational Equivalence Checking (ACEC) is to model the two compared circuits in form of Gröbner bases and combine them into a single algebraic model. It generates bit and word relationship candidates between the internal variables of the two circuits and tests their membership in the combined model. Since the membership testing does not scale for the described setting, we propose reverse engineering to extract arithmetic components and to abstract them to canonical representations. Further we propose arithmetic sweeping which utilizes the abstracted components to find and prove internal equivalences between both circuits.**

We demonstrate the applicability of ACEC for checking the equivalence of a floating point multiplier (including full IEEE-754 rounding scheme) against several optimized and diversified implementations.

***Index Terms*—Formal Verification, Equivalence Checking, Gröbner Bases, Reverse Engineering, Floating-Point Multiplier.**

I. Introduction

Arithmetic circuits are typically difficult instances for classical Boolean reasoning approaches that are based on, e.g., *Binary Decision Diagrams* (BDDs) or *Boolean Satisfiability* (SAT), as they suffer from exponential worst-case complexity. Boolean reasoning based on Gröbner bases (available with algebraic computation packages) offers a robust mechanism that verifies arithmetic circuits at gate-level (see, e.g., [1], [2]) and their power has recently been demonstrated in formally verifying large and optimized gate-level multipliers [3]. However, circuits that also contain control logic pose a major difficulty for algebraic computation based reasoning techniques and no satisfactory solution has yet been presented. In this paper, we show techniques that allow to reason over circuits which combine data-path and control logic using symbolic computation reasoning. To the best of our knowledge, this is the first full automated technique that formally verifies binary floating-point circuits without any kind of case splitting or other manual effort.

So far, verification using algebraic computation models the circuit under verification as polynomials $G = \{g_1, \dots, g_k\}$ and tests the membership of the specification polynomial p_{spec} in G. The polynomials in G contain *internal variables* for all gates in the circuit, whereas p_{spec} is expressed only in terms of the primary inputs and primary outputs (n input bits and m output bits in total). p_{spec} can be viewed as a map over the finite integer space, i.e., $p_{\text{spec}} : \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^m}$. Membership testing is performed by reducing (dividing) p_{spec} wrt. G. If this reduction completes with no remainder, the circuit fulfills the specification. During the reduction, it is possible that the intermediate polynomials blow up which can be eluded by applying intermediate rewriting strategies (see, e.g., [3]).

Motivated by the fundamental problem that not every circuit specification p_{spec} can be represented in a canonical and abstract form over $\mathbb{Z}_{2^n}$, we are interested in *equivalence checking*, i.e., we want to prove the functional equivalence of two circuits in the absence of a specification. This can be done as follows: Assume the two circuits checked for equivalence represent the functions $f_1(x_1, \dots, x_n) = (y_1, \dots, y_m)$ and $f_2(x_1, \dots, x_n) = (z_1, \dots, z_m)$ and are given as two sets of polynomials G_1 and G_2. Then we divide each polynomial $z_j - y_j$ for $1 \leq j \leq m$—which formulates the equivalence of each output bit—by polynomials from the combined model $G = G_1 \cup G_2$. This naïve method does not scale since during the reduction the internal variables in the polynomials in G cause for a tremendous overhead which can only be resolved when the primary input variables x_i appear in the polynomials.

This problem can be circumvented if one knows internal equivalences in the two circuits which allows to put internal variables into relation. Conceptually, this is similar to SAT sweeping and as a consequence G is simplified. This ultimately avoids a blow-up of the polynomials during reduction. The difficulty is finding internal equivalences. To solve this problem we propose reverse engineering techniques: First, expected arithmetic word-level components such as multipliers and adders are detected in the circuit using structural signatures. Then, the proposed arithmetic sweeping uses the I/O boundaries of detected word-level components to prove internal equivalences and to prevent division blow-ups.

To further reduce verification runtime during the divisions we propose decomposition and a general reduction rule that allow more compact representations and semi-canonical representations for different implementations of the same function.

The result is a new *Algebraic Combinational Equivalence Checking* (ACEC) technique which is based on Gröbner bases. In contrast to classical combinational equivalence checking [4], [5], it can check the equivalence of two circuits which contain different architectures of arithmetic units, e.g multipliers and adders, as well as control logic parts. Our experimental evaluation demonstrates the applicability of our algebraic equivalence checking approach on several optimized floating-point multipliers which cannot be verified by other proof techniques.

II. Preliminaries

Using concepts from algebraic geometry and symbolic computation, we model the given combinational circuits with

This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative, by H2020-ERC-2014-ADG 669354 CyberCare, and by the German Academic Exchange Service (DAAD). Also, we thank OneSpin Solutions for making their EC tool available to us.

a set of multivariate polynomials based on Gröbner bases and we formulate the equivalence checking problem as testing the membership of some relationships between these circuits using the ideal membership concept. In the following, we define common notations of these algebraic concepts based on [6]. Then, we present the Gröbner bases modeling and the ideal membership testing algorithm.

A. Notation and Definitions

The ring of integers modulo 2 ($\mathbb{Z}_2$) is called a *Boolean ring*. As is shown in [7], [8], theory of Gröbner bases can be applied on Boolean rings, it is referred as a *Boolean Gröbner bases*. For a Boolean polynomial ring $\mathbb{Z}_2(x_1,\ldots,x_n) = \mathbb{Z}_2[x_1,\ldots,x_n]/\langle -x_1^2+x_1,\cdots,-x_n^2+x_n\rangle$ of n Boolean variables, the polynomials $\langle -x_i^2+x_i\rangle$ are added to the polynomial ring $\mathbb{Z}_2[x_1,\ldots,x_n]$ to keep the variables x_i in the Boolean domain. A *monomial* $M = x_1^{\alpha_1}\cdots x_n^{\alpha_n}$ is the power product over the variables $x_1,\ldots,x_n$. As for Boolean variables $x_i^2 = x_i$, the powers α_i are always equal to one. A *polynomial* $p = c_1M_1+\cdots+c_tM_t$ is a finite sum of terms, where each term is the product of an coefficient c_i and a monomial M_i. The monomials of a polynomial are ordered according to a *monomial ordering* '$>$', such that $M_1 > \cdots > M_t$, the *leading term* of the polynomial is $\mathrm{lt}(p) = c_1M_1$, the *leading monomial* is $\mathrm{lm}(p) = M_1$, and the *leading coefficient* is $\mathrm{lc}(p) = c_1$. We denote $\mathrm{tail}(p) = p - \mathrm{lt}(p) = c_2M_2+\cdots+c_tM_t$.

In this work, the monomial order follows the reverse topological order of the variables of the modeled circuit and the coefficient $c_i \in \mathbb{Z}$ for all $i \neq 1$, where the leading coefficient $\mathrm{lc}(p) = c_1 \in \{-1,1\}$. The coefficients c_i are not limited to $\{0,1\}$ as in Galios Field $\mathbb{GF}_2$, they could be arbitrary as shown in [7] or integers as in this work.

For a set of polynomials $P = \{p_1,\ldots,p_s\} \in \mathbb{Z}_2(x_1,\ldots,x_n)$, an *affine variety* $V(p_1,\ldots,p_s)$ is the set of all solutions of the polynomial equations $p_1(x_1,\ldots,x_n) = \cdots = p_s(x_1,\ldots,x_n) = 0$. The affine variety depends not just on the given set of polynomials, but rather on the ideal generated by the polynomials. An *ideal* $I = \langle P\rangle = \{\sum_{i=1}^{s} h_i\cdot p_i : h_i \in \mathbb{Z}_2[x_1,\ldots,x_n]\}$ is generated by this set of polynomials P, and we call P the bases (generators) of the ideal I. The ideal I may have many other bases. The bases are different representations of the set of polynomials P. One of these bases is called *Gröbner bases* $G = \{g_1,\ldots,g_{\hat{s}}\}$, for which $V(G) = V(I)$. Gröbner bases reveal the properties of the ideal that allow to solve the ideal membership testing problem in an algorithmic fashion.

Definition 1: A *polynomial division* of two polynomials p and g denoted as $p \xrightarrow{g}_+ r$ is performed as $r = p - \frac{cM}{\mathrm{lt}(g)}g$. If a non-zero term cM of p is divisible by the leading term of g, then p reduces to r modulo g. Similarly, p can be reduced (divided) wrt. a set of polynomials P to obtain a remainder r, denoted $p \xrightarrow{P}_+ r$, such that no term in r is divisible by the leading term of any polynomial in P.

Definition 2: A polynomial reduction method named *S-polynomial* of polynomials p and g in a polynomial set P, is the combination $\mathrm{Spoly}(p,g) = \frac{L}{\mathrm{lt}(p)}p - \frac{L}{\mathrm{lt}(g)}g$, where L is the least common multiple $\mathrm{LCM}(\mathrm{lm}(p),\mathrm{lm}(g))$.

To compute the Gröbner bases $G = \{g_1,...,g_{\hat{s}}\}$ for an ideal $I\langle p_1,\ldots,p_s\rangle$, *Buchberger's algorithm* constructs G in a finite the number of steps by applying $\mathrm{Spoly}(p,g) \xrightarrow{G}_+ r$ in every step. Gröbner bases are computed if all $\mathrm{Spoly}(p,g) \xrightarrow{G}_+ 0$.

Lemma 1: Given a finite set $G \in \mathbb{Z}_2(x_1,\ldots,x_n)$, suppose that we have $p,g \in G$ such that $\mathrm{LCM}(\mathrm{lm}(p),\mathrm{lm}(g)) = \mathrm{lm}(p)\cdot\mathrm{lm}(g)$. In other words, the leading monomials of p and g are relatively prime. Then $\mathrm{Spoly}(p,g) \xrightarrow{G}_+ 0$ [6].

According to Lemma 1, a given polynomial set is a Gröbner basis, if the leading monomials of all polynomials in the set are relatively prime. By combining this lemma with the affine variety concept of an ideal, we define the Gröbner bases of an ideal as follows:

Definition 3: A finite subset $G = \{g_1,\ldots,g_{\hat{s}}\}$ wrt. a monomial order of an ideal I is said to be a Gröbner basis of I if $V(G) = V(I)$ and all leading monomials in G are relatively prime.

A given ideal may have different Gröbner bases, where one basis can be reduced to other bases by eliminating (substituting) some of ideal variables based on the *Elimination Theorem* [6], in the following, this process is named model rewriting. These bases can be reduced again to a canonical representation of the ideal that is called *reduced Gröbner basis*.

Definition 4: A reduced Gröbner basis for a polynomial ideal I is a Gröbner basis G for I, such that for all $g_i \in G$, no term in g_i is divisible by the leading term $\mathrm{lt}(g_j)$ for all $i \neq j$.

Lemma 2: Let $I \neq 0$ be a polynomial ideal. Then, for a given monomial ordering $>$, I has a unique reduced Gröbner basis [6].

We utilize the uniqueness property of the reduced Gröbner basis for canonical polynomial abstraction in Section IV.

The *Ideal Membership Testing* (IMT) decides whether a given polynomial p lies in the Gröbner basis ideal $G = \{g_1,\ldots,g_{\hat{s}}\}$. It applies a division algorithm to check that the remainder r on dividing p by G is equal to zero. The division is denoted $p \xrightarrow{G}_+ r$.

B. Modeling a Circuit as Gröbner Basis

Logic gates are modeled by polynomials and signals as Boolean variables. The polynomials of basic Boolean gates are

$$\begin{aligned} z = \neg a &\implies g := -z + 1 - a \\ z = a \wedge b &\implies g := -z + ab \\ z = a \vee b &\implies g := -z + a + b - ab \\ z = a \oplus b &\implies g := -z + a + b - 2ab. \end{aligned}$$

Each logic gate is modeled in a way that the gate output variable z is described in terms of the gate input variables a, b. The polynomial $x^2 - x$ is added to the model for each variable to enforce the Boolean domain. In practice, the ideal polynomials $\langle -x^2 + x\rangle$ are replaced by reducing x^k to x every time its degree becomes greater than one during any computational step. For example, the monomial $x_1^2x_2^3x_3$ is equal to $x_1x_2x_3$ in the Boolean domain.

By ordering each variable of the model according to its reverse topological level in the circuit, the generated polynomials satisfy Definition 3 by construction. Every polynomial is of the form $p_i := x_i + \mathrm{tail}(p_i)$, where x_i is the gate's output variable and $\mathrm{tail}(p_i)$ are terms consisting of the gate's input variables, describing the function implemented by the gate. According to this polynomial form, all leading monomials of the model are relatively prime.

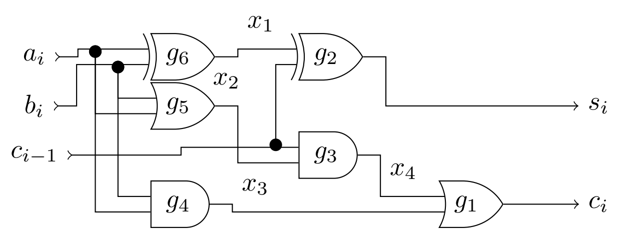


Fig. 1. A simple full adder.

Example 1: Consider the full adder circuit implementing the function $s_i + 2c_i = a_i + b_i + c_{i-1}$ shown in Fig. 1. Its algebraic model is

$$\begin{array}{ll} g_1 := -c_i - x_4x_3 + x_4 + x_3 & g_2 := -s_i - 2x_1c_{i-1} + x_1 + c_{i-1} \\ g_3 := -x_4 + x_2c_{i-1} & g_4 := -x_3 + a_ib_i \\ g_5 := -x_2 - a_ib_i + a_i + b_i & g_6 := -x_1 - 2a_ib_i + a_i + b_i \end{array}$$

The specification polynomial[1] is $p_{\mathrm{r}} := -2c_i - s_i + c_{i-1} + b_i + a_i$. Ordering the polynomial variables in the reverse topological order of the circuit yields $c_i > s_i > x_4 > x_3 > x_2 > x_1 > c_{i-1} > b_i > a_i$. Following this order, the leading monomials of all polynomials will be relatively prime. E.g., the leading monomial of g_1 is c_i, and it is relative prime to all other leading monomials. According to Definition 3, the extracted algebraic model is therefore a Gröbner basis.

Modeling the circuit directly as Gröbner basis polynomials avoids Buchberger's algorithm and makes it computationally feasible to apply the membership testing.

C. Ideal Membership Testing

Given a specification (or relationship) polynomial p_{r} and a circuit model in form of a Gröbner basis G, p_{r} is divided in every iteration by some polynomial $g \in G$ (see Definition 1). The polynomial division can be seen as substituting the variables in p_{r} with the corresponding tail terms of the respective polynomials in G. For example, given $p_{\mathrm{r}} := x_4x_3 + x_1$ and a polynomial $g := -x_4 + x_2x_1$, then $\mathrm{r} = p_{\mathrm{r}} - \frac{x_4x_3}{-x_4}g = x_3x_2x_1 + x_1$, where the polynomial division substitutes x_4 in p_{r} with x_2x_1. The division (substitution) iterations are executed according to a certain order, the *substitution order*. As in [1], [9], the substitution ordering follows the reverse topological order of the circuit variables.

Following Example 1, the extracted algebraic model is a Gröbner basis, therefore the ideal membership testing of p_r can be applied. The substitution order will follow the reverse topological order of the circuit:

$p_{\mathrm{r}} \xrightarrow{g_1} -s_i + 2x_4x_3 - 2x_4 - 2x_3 + c_{i-1} + b_i + a_i$
$\xrightarrow{g_2} 2x_4x_3 - 2x_4 - 2x_3 + 2x_1c_{i-1} - x_1 + b_i + a_i$
$\xrightarrow{g_3} 2x_3x_2c_{i-1} - 2x_3 - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 + b_i + a_i$
$\xrightarrow{g_4} 2x_2c_{i-1}b_ia_i - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 - 2b_ia_i + b_i + a_i$
$\xrightarrow{g_5} 2x_1c_{i-1} - x_1 + 4c_{i-1}b_ia_i - 2c_{i-1}a_i - 2c_{i-1}b_i - 2a_ib_i + b_i + a_i \xrightarrow{g_6} 0$ Since the final division result is 0, p_{r} has been proven.

III. Algebraic Combinational Equivalence Checking

This section introduces the proposed algebraic combinational equivalence checking approach. Given two circuits C_1 and C_2 that represent the functions $f_1(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ and $f_2(x_1, \ldots, x_n) = (z_1, \ldots, z_m)$, respectively, our aim is to show the equivalence of C_1 and C_2, i.e., $(y_1, \ldots, y_m) = (z_1, \ldots, z_m)$ for all $x_1, \ldots, x_n$. We propose to solve this problem using symbolic computation. Since the specification of C_1 and C_2 may be unknown or since it may not be expressible in a canonical and an abstract form over $\mathbb{Z}_{2^n}$, we cannot use previous work [1]–[3] that performs ideal membership testing with respect to a given specification.

Instead we propose to represent C_1 and C_2 as polynomial sets G_1 and G_2 and combine them into a single model $G = G_1 \cup G_2$. We then formulate the problem as testing the membership of relations between variables in C_1 and C_2 wrt. G. An obvious choice for such a relation is the equivalence of output signals $y_i = z_i$ which can be expressed in a polynomial as $y_i - z_i = 0$. However, reducing such a polynomial wrt. G causes a tremendous overhead since the substitution of all the internal variables in G_1 and G_2 will blow up the sizes of the polynomials in G.

To overcome this problem we suggest to find internal equivalences, i.e., polynomials that express equivalence of two internal signals in G_1 and G_2. Reducing these polynomials wrt. G causes a smaller overhead and simplifies G. This technique is similar to SAT sweeping in combinational equivalence checking [4] and we call it *arithmetic sweeping* in the following. Arithmetic sweeping works as follows: for each internal variable v_1 in G_1 we search for an equivalent variable v_2 in G_2, i.e., v_1 and v_2 represent the same function wrt. to the primary inputs. We call such a pair (v_1, v_2) *bit equivalence* and are able to substitute v_2 by v_1 in all polynomials. For some internal variables we will not be able to prove equivalence to another variable. These variables are eliminated by substitution with proved bit equivalent variables of their transitive fan-in.

However, performing arithmetic sweeping on the overall combined model G is not scalable. First, the number of candidates for bit equivalences is too large, and second, checking a pair of variables for equivalence that have a large transitive fan-in may be too difficult. To circumvent this problem, we first apply *reverse engineering* for two main goals i) extracting and abstracting arithmetic word-level components to canonical polynomials; ii) partitioning the circuits G_1 and G_2 into smaller parts. The algorithm works as follows: First, we try to find an instance of an arithmetic word-level component both in G_1 and G_2 and abstract them to canonical polynomials. If this is successful, we obtain an input boundary and an output boundary for the component in G_1 and G_2. The pairs of input boundaries and output boundaries are candidates for *word equivalences*. Having them, we perform arithmetic sweeping only in the transitive fan-in of the input boundaries. If this ultimately proves that the input boundaries are equivalent and we have proven that *abstracted polynomials* of the two arithmetic components found by reverse engineering are equivalent, we can merge the transitive fan-in of the output boundaries from G, making the model significantly smaller.

Details on the algorithms of our ACEC are explained in the remaining sections. In Section IV we show how to find arithmetic word-level components using reverse engineering. This also partitions the circuits into smaller parts based on the word-level components and the respective transitive fan-in of it. Both results are the input for the arithmetic sweeping. Section V explains arithmetic sweeping to find (and prove) internal equivalences in the transitive fan-ins of the detected

[1]Please note that later in the paper we use polynomials which relate different bit or word variables, so we call them relationship polynomials.

components' input boundaries. Finally in Section VI, we offer efficient polynomial representation based on functional decomposition and a new general reduction rule to speed up the different division steps.

IV. Reverse Engineering of Data-path Units

Key in ACEC is to find arithmetic components using reverse engineering in order to reduce the model size in which arithmetic sweeping is performed. Reverse engineering needs to find equivalent components and abstract them to canonical polynomials over integer field. The propagation of carry bits between internal nets of data-path units is one of the main properties that helps to locate such units. In the proposed reverse engineering algorithm we exploit this property to extract data-path units from the combined model G. According to our observation, these carry bits are modeled as shared monomials between polynomials of G. Continuing with Example 1, the simple full adder can be modeled by two polynomials $g_1 : -s_i+4c_{i-1}b_ia_i-2c_{i-1}b_i-2c_{i-1}a_i-2b_ia_i+c_{i-1}+b_i+a_i$ and $g_2 : -c_i - 2c_{i-1}b_ia_i + c_{i-1}b_i + c_{i-1}a_i + b_ia_i$. The shared monomials $c_{i-1}b_ia_i$, $c_{i-1}b_i$, $c_{i-1}a_i$, and b_ia_i model the internal carry propagations of the full adder. The terms of these shared monomials have another property. Their coefficients have different signs, and they are multiples of each other. We call terms with these properties *carry terms*.

To reveal carry terms, a rewriting scheme based on the rewriting principles of [3] is proposed in Section IV-A. Note that in the full adder example, carry terms are not visible in the original model of the full adder. They can be only revealed when the model is rewritten to two polynomials.

After rewriting the model, the reverse engineering algorithm builds an *adder network* for every group of polynomials that share carry terms. For each adder network which models a data-path unit *one* canonical polynomial using *Gaussian elimination* algorithm is derived, see Section IV-B.

A. Model Rewriting

The model rewriting schemes of [3] have shown an ability to reveal vanishing monomials (monomials that always evaluate to zero) as well as common monomials between the polynomials model of a multiplier circuit. This revealing ability empowers our reverse engineering algorithm to build adder networks for different architectures of large scale multipliers and adders. The proposed rewriting schemes combine the knowledge of the circuit gates with the algebraic model. The first scheme *XOR rewriting* rewrites the model using the S-polynomial method such that the model depends only on inputs and output variables of XOR gates whereas all other variables are substituted. The second *common rewriting* scheme rewrites the model obtained from XOR rewriting such that the model depends only on variables that are used in more than one polynomial.

Applying these schemes on a control logic circuitry causes a blow-up in the number of model terms since control logic usually does not contain XOR gates which yields to substitutions for large the number of the control logic variables. To take advantage of these schemes for circuits which contain data-path and control logic, we distinguish the control logic part of a circuit by its multiplexers (MUXes) and disallow XOR rewriting and common rewriting from substituting input and output variables of MUXes. This guarantees that both schemes will be applied only on the data-path logic. The polynomials of the rewritten model describe functions of XORs, MUXes, and the cone of gates which are bounded by inputs and outputs of XORs and MUXes.

B. Abstracting Data-path Units to Canonical Polynomial

After rewriting G the algorithm builds different adder networks from polynomials that share carry terms. It then groups polynomials of G. A new polynomial joins a group, if it shares a carry term with other polynomials in the group. For example, the polynomials g_0 and g_1 are in the same group, if one of them has the term $-x_0x_1$ and the second has the term $2x_0x_1$. Groups of each extracted adder network are handled as independent algebraic ideal. It is abstracted to one canonical polynomial using *Gaussian elimination*.

Example 2: To illustrate the proposed approach, consider the model of a 3-bit ripple carry adder implementing the function $\sum_{i=0}^{2} 2^i s_i = \sum_{i=0}^{2} 2^i(a_i + b_i)$.

$s_3 = c_2 \quad \Longrightarrow \quad g_1 := -s_3 + c_2$

$c_2 = (a_2 \wedge b_2) \vee (a_2 \wedge c_1) \vee (b_2 \wedge c_1) \quad \Longrightarrow$

$g_2 := -c_2 \boxed{-2c_1b_2a_2 + c_1b_2 + c_1a_2 + b_2a_2}$

$s_2 = a_2 \oplus b_2 \oplus c_1 \quad \Longrightarrow$

$g_3 := -s_2 \boxed{+4c_1b_2a_2 - 2c_1b_2 - 2c_1a_2 - 2b_2a_2} + c_1 + b_2 + a_2$

$c_1 = (a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0) \quad \Longrightarrow$

$g_4 := -c_1 \boxed{-2c_0b_1a_1 + c_0b_1 + c_0a_1 + b_1a_1}$

$s_1 = a_1 \oplus b_1 \oplus c_0 \quad \Longrightarrow$

$g_5 := -s_1 \boxed{+4c_0b_1a_1 - 2c_0b_1 - 2c_0a_1 - 2b_1a_1} + c_0 + b_1 + a_1$

$c_0 = a_0 \wedge b_0 \quad \Longrightarrow \quad g_6 := -c_0 \boxed{+b_0a_0}$

$s_0 = a_0 \oplus b_0 \quad \Longrightarrow \quad g_7 := -s_0 \boxed{-2b_0a_0} + b_0 + a_0$

Rewriting the model yields that polynomials g_2, g_3 have common non-linear monomials (colored green/dashed box in the example). The similar structural property can be seen for equally colored terms of the polynomials g_4, g_5 and polynomials g_6, g_7, respectively. To cancel the carry terms between g_3 and g_2, Gaussian elimination is applied. It multiples g_2 by 2 and adds it to g_3. The result is the polynomial $h_1 := -2c_2 - s_2 \ + c_1 + b_2 + a_2$ which represents a full adder function. Applying the same step on other related polynomials yields another two full adders $h_2 := -2c_1 - s_1 \ + c_0 + b_1 + a_1$ and $h_3 := -2c_0 - s_0 \ + b_0 + a_0$. Applying Gaussian elimination again on the three full adder polynomials to cancel shared terms and achieve a reduced Gröbner basis, will multiply h_1 by 2 and adds to h_2. The result will be $h_4 := -4c_2 - 2s_2 - s_1 + 2b_2 + 2a_2 + c_0 + b_1 + a_1$. Finally, the reduced Gröbner basis polynomial $h_5 := -8c_2 - 4s_2 - 2s_1 - s_0 + 4b_1 + 4a_1 + 2b_1 + 2a_1 + b_0 + a_0$ is derived by multiplying h_4 by 2 and adding it to h_3 for canceling the shared monomial c_0.

Lemma 3: Let $G_r = g_1, \cdots, g_t$ denote the generated Gröbner basis by Gaussian elimination wrt. a unique monomial order $>$. As G_r contains the one and only polynomial g_1, then g_1 is the unique canonical representation of the function f implemented by the adder network ideal.

Proof: Based on Lemma 2, for every ideal there is a unique reduced Gröbner basis. Since the adder network ideal G_r, which has been generated by Gaussian elimination, has only one polynomial g_1, no term in g_1 is divisible by the leading term of any other polynomial in G_r. Therefore Definition 4 of reduced Gröbner basis holds for G_r and g_1 is a canonical abstracted representation of the function f implemented by the adder network ideal.

Please also note that to avoid the blow-up in the number of terms during Gaussian elimination of large scale multipliers, as illustrated in [1], [9], the elimination order must follow the reverse topological order of the circuit variables.

In addition to abstracting data-path units, the reverse engineering algorithm determines their inputs and outputs boundaries. This works as follows: The algorithm extracts this information from the original polynomials (ideal) of the adder network. It uses a property of a Gröbner bases model that a variable of a leading monomial of a polynomial is the output variable of this polynomial. Based on this property, for the ideal of an adder network, output variables of polynomials that are not used as inputs for other polynomials in the ideal are identified as the output variables of this adder network. Finally, an output word for each abstracted polynomial can be derived.

V. Arithmetic Sweeping

Arithmetic sweeping aims to find internal equivalences which avoids prohibitive run time during the polynomial division. Of course when having identified candidates for internal equivalence, it is still necessary to prove their equivalence (which is also done using the same division algorithm for the relationship polynomials of the candidates). Hence, to gain an overall benefit we need i) promising candidates and ii) moderate runtimes for the equivalence proofs. Our proposed arithmetic sweeping reaches both goals as follows.

For i), the reverse engineering step provides arithmetic components. From this we generate promising candidates based on the I/O boundaries of these components. The algorithm uses the I/O boundaries to partition the variables of the combined model G into groups. Simulation deduces *word equivalence* (wE; for details see below) candidates between outputs of the arithmetic components. For every nominated wE the partitioning of model variables is performed by classifying two groups of variables. One for the transitive fan-ins variables of the input boundaries of wE and the other are internal variables of the two related arithmetic components. Deducing only internal *bit equivalences* (bE; see below) between variables in the same group increases the potential of equivalence.

For ii), the equivalence proofs become feasible for several reasons. Arithmetic sweeping generates two types of relationships which are *bit equivalence* (bE) pair and *word equivalence* (wR) pair. bE describes the equivalence of a pair of variables (v_i,v_j) and is formulated by the polynomial $g := -v_i + v_j$. The word equivalence (wE) polynomial is formulated as $g := B - \hat{B}$ for the word pair candidate (B,$\hat{B}$), where $B = 2^{n-1}b_{n-1} + \cdots + b_0$ and $\hat{B} = 2^{n-1}\hat{b}_{n-1} + \cdots + \hat{b}_0$. For each arithmetic component we have determined an *abstract* canonical polynomial in the reverse engineering step. The major advantage over SAT sweeping is that the proof for the internal equivalences is performed by dividing wE polynomials wrt. the abstracted polynomials. For doing this, a new word model G_W is created and the abstracted polynomials are added to it as follows: For every abstracted polynomial, an integer word variable B is created and a polynomial $-B + f(a_1, \cdots, a_m)$ is added to the word model. The polynomial $-B + 2^{n-1}b_{n-1} + \cdots + b_0$ is used to interpret the equivalence between two output words B and $\hat{B}$, as shown in Lemma 4 of Section V-B. To summarize, dividing wE wrt. abstracted polynomials has a major influence on the performance of the technique—it avoids the exhaustive cost of searching for equivalences between internal variables of the data-path units which usually have a large the number of non-equivalent variables in their transitive fan-ins.

A. Generating Relationship Polynomials

The choice of relationship candidates is always the main problem of different equivalence checking techniques. ACEC draws on the simulation approach of [10] and the extracted data-path polynomials to deduce bit and word relationships. Four steps are performed to generate relationship polynomials i) nominating wE polynomials, ii) classifying the model variables to groups, iii) generating bE polynomials, and finally iv) sorting wE and bE polynomials in a relationship list.

Based on a fixed size of global simulation over the primary inputs of G, word relationships between the output words of the data-path polynomials are deduced. Two words build a wE polynomial, if their integer values are equal under all simulated assignments.

The approach classifies the variables of G to groups according to wE polynomials. One wE polynomial categorizes two groups, the first consists of all transitive fan-in variables of the polynomial; and the second contains internal variables which are bounded by outputs and inputs variables of wE.

Example 3: To illustrate this idea, consider a model which has four extracted data-path units (DPU_1, DPU_2, DPU_3, and DPU_4), as shown in Fig. 2. The simulation nominates two wEs, one relates the output word of DPU_1 and DPU_2, the other one is between DPU_3 and DPU_4. The approach classifies the model variables into 5 groups i) a group for transitive fan-in variables of DPU_1 and DPU_2, ii) a group which contains internal variables of DPU_1 and DPU_2, iii) transitive fan-in variables of the wE between DPU_3 and DPU_4, iv) their internal variables, and v) the remaining variables of C_1 and C_2 which are not classified in groups.

Classified groups of G and global simulation are used to determine for every model variable v_i a set of variables ϕ_i. We have $v_j \in \phi_i$, if Boolean values of v_i and v_j are the same under each of input assignments; and therefore v_i and v_j belong to the same classified group. Finally, bE polynomials between v_i and other variables of ϕ_i are generated. We call them *bE polynomials* of the variable v_i.

After classifying model variables and generating wE and bE relationships, these nominated relationships are sorted topologically wrt. the circuit and their leading variables. The sorting procedure aims to test a wE polynomial after testing all bE polynomials of variables in its transitive fan-in group. First, the wE polynomials are sorted topologically. Next, the procedure iterates over the wE polynomials for inserting in the list for every wE i) bE polynomials of variables in the transitive fan-in group of this wE, ii) the wE polynomial itself, then iii) bE polynomials of variables in its internal group. Finally, the bE polynomials of remaining variables that are not included in groups that are related to wEs, are inserted in the end of the list.

B. Testing Membership of Internal Relations

During the testing of internal relationships, the approach calls the IMT algorithm to divide every polynomial p_r from the relationship list wrt. G or G_W, if p_r is a bE polynomial, the division is done wrt. G, otherwise is performed wrt. G_W. Based on the remainder result of dividing p_r, the approach

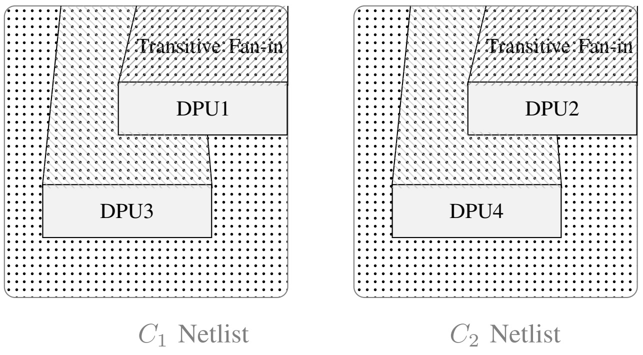


Fig. 2. Schematic of a combined model including word relationships

eliminates or merges variables of p_r from the models G and G_W.

The merging decision is taken, in the case that the remainder result of dividing p_r is equal to zero. The approach merges every two variables of p_r which are derived to be functionally equivalent to one variable. In case that p_r is a wE polynomial, equivalence is derived based on the following lemma.

Lemma 4: Given the equivalent of two integer words $A = 2^{n-1}a_{n-1} + \cdots + a_0$ and $B = 2^{n-1}b_{n-1} + \cdots + b_0$. If A and B have the same number system and the number system is not redundant, then the bit variables a_i and b_i which have same weights (coefficients) are equivalent.

Merging results are reducing the number of polynomials in G, these merged variables are considered new primary inputs, therefore polynomials of their transitive fan-in variables are removed from G. Continuing with the previous model example, after deriving the equivalences between corresponding output variables of DPU_1 and DPU_2, merging equivalent variables will produce a compact version of G. Polynomials that model DPU_1 and DPU_2 are removed, in addition to those which model their transitive fan-in variables. In order to avoid redundant divisions, the remaining bE polynomials which test the already merged variables will be removed from the relationship list.

A variable of the model that has no functional equivalences is eliminated by substituting it with the leading terms of its polynomials which are functions in proved bit equivalent variables. The elimination decision will be taken for variables v_i of p_r. If the remainder of dividing p_r is not equal to zero, and there are no more untested bE or wE polynomials in the list which are related to v_i. These eliminations facilitate the division process of next relationships. It increases the number of shared input variables of polynomials of G which simplifies the division process of p_r wrt. G. For example, dividing $p_r : -v_i + v_j$ wrt. a model that has polynomials $g_1 : -v_i + x_1x_2 + x_3$ and $g_2 : -v_j + x_1x_2 + x_3$ will be simplified to a subtraction operation. The remainder of the division will be $x_1x_2 + x_3 - x_1x_2 - x_3 = 0$.

VI. Efficient Polynomial Representation

The polynomial is the heart of the algebraic computation technique. An efficient representation of a polynomial has a major impact on the performance of any algebraic algorithm. To circumvent a blow-up in the number of polynomial terms for representing different Boolean functions, we propose i) a decomposition method which reduces the number of terms in polynomials significantly for some Boolean functions, and ii) a general reduction rule to cancel redundant terms of the polynomial. The decomposition method and the reduction rule offer semi-canonical representations which simplify the division of the IMT algorithm.

A. Different Decompositions

Inspired by decomposition types of decision diagrams [11], we enhance representations of polynomials by considering two decomposition types which are:

$$\begin{array}{ll} f = f_{|x=0} + x(f_{|x=1} - f_{|x=0}) & \text{positive Davio (pD)} \\ f = f_{|x=1} + (1-x)(f_{|x=0} - f_{|x=1}) & \text{negative Davio (nD),} \end{array}$$

where x denotes a Boolean variable, the functions are combined with addition, subtraction, and multiplication operations.

Our observation is that polynomials have been typically represented by pD decomposition. This obstructs a compact representation for some Boolean functions like a chain of OR gates. For example, consider a 4-input OR function $f(x_0, x_1, x_2, x_3)$, its polynomial representation that follows only pD will be $f = x_0 + x_1 + x_2 + x_3 - x_0x_1 - x_0x_2 - x_0x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$. By decomposing f using nD for all of its variables, it will be $f = 1 - \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$, where $\bar{x}_i = 1 - x_i$. For the n-bit OR function, a polynomial which follows pD consists of $2^n - 1$ terms, while nD polynomial has only two terms. Representing a Boolean function with less the number of terms has a major influence on reducing the number of addition, subtraction, and multiplication operations, therefore it enhances significantly the performance of any symbolic computation algorithm. For applying these decompositions, we add to the model negation version $\bar{v}_i$ for every variable v_i in the model, in addition to the polynomials $g := -\bar{v}_i - v_i + 1$.

As known from the field of decision diagrams, the choice of the type of the decomposition and the order of the variables plays a key role for the size of the diagram. In this work, we fix the order of the variables to the reverse topological order and we propose an approach to determine the *Decomposition Type* (DT) of each variable. As the main goal of applying different decompositions is reducing the number of polynomials terms, the decision of DT is taken based on this factor and the structure of the circuit.

For this purpose, we modify the modeling way of the circuit that is explained in Section II-B as follows:

$$\begin{aligned} z = \neg a &\implies g := -z + \bar{a} \\ z = a \wedge b &\implies g := -z + ab \\ z = a \vee b &\implies g := -z + 1 - \bar{a}\bar{b} \\ z = a \oplus b &\implies g := -z + a + b - 2ab, \end{aligned}$$

such that the DT of input variables of inverters and OR gates is nD, for AND and XOR gates, it is pD. As shown in this modeling, one variable may have more than one DT in the model. During model rewriting, see Subsection IV-A, a polynomial g is rewritten by substituting one of its variables v_i, as result of this step, another variable v_j in g may have different decomposition types – the variable v_j and its negation $\bar{v}_j$ are within the same polynomial g. In this case, we unify the DT of v_j based on the one which achieves the higher reduction on number of terms in g. In case of n variables with different DTs within same polynomial, the possible combinations of DTs for these variables are 2^n. For example, consider a polynomial with two variables v_1 and v_2, the possible decomposition combinations will be (v_1, v_2), $(v_1, \bar{v_2})$, $(\bar{v_1}, v_2)$, or $(\bar{v_1}, \bar{v_2})$. Trying all combinations to find the best representation leads to a prohibitive run time because of calling the decomposition algorithm 2^n times. To bypass this problem, our approach takes the decomposition decision of every variable independently from others. This restriction on choosing DTs accelerates significantly the run time of the proposed approach to find compact representations for polynomials. In this work, we have used an implemented decomposition algorithm which designed for decision diagrams [12]. For this, we have implemented a two directions parser. It parses a function from a polynomial to a K*BMD and vice versa.

B. General Reduction Rule

A key observation in [3] is the significance of applying a logic reduction rule to cancel redundant terms and avoid blow-up of these terms during the division algorithm. The rule exploits that $(a \oplus b) \cdot (a \wedge b) = 0$ for all a and b and therefore can be used to remove terms from polynomials. If, e.g, $f = a \oplus b$ and $h = a \wedge b$, any term containing both f and h can be removed. We propose to generalize this rule.

Let X be a set of variables and let f and h be two Boolean functions over the variables $X_1 \subseteq X$ and $X_2 \subseteq X$, respectively, with $X_1 \cap X_2 \neq \emptyset$. If there exists exactly one assignment to h such that it evaluates to true, it may be possible that g simplifies to a constant value when assigning the common variables according to that assignment. To illustrate the concept consider a multiplexer function $f(a, b, c) = ac - bc + b$ and $h(a, b) = ab$. Clearly $h = 1$, only if $a = b = 1$, and $f(1, 1, c) = 1$. Therefore, we conclude that $fh = h$ and we can simply polynomials accordingly. For a polynomial $g := -v_1v_2fhv_3 + v_1v_2hv_3$, by applying this rule on the monomial $v_1v_2fhv_3$, it simplifies to $v_1v_2hv_3$ and the polynomial g will be evaluated to $g = -v_1v_2hv_3 + v_1v_2hv_3 = 0$. This reduction rule is called *one assignment rule*.

An approach to apply the one assignment rule is as follows:

1) Searching for monomials in the algebraic model that have two variables of functions f and h which shared some of their inputs, such that the function h has one satisfiable assignment.
2) Reducing f after assigning values to shared inputs which evaluate h to one.
3) If f is equal to zero or one, then rewriting the monomial by substituting f with its value.

VII. Experimental Evaluation

ACEC is implemented in C++. We compared it to the equivalence checkers of ABC [13] tool and a commercial tool (OneSpin EC-360). The experiments were carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux.

We applied ACEC to the problem of verifying a floating-point (FP) multiplier. It computes the operation $P = A \times B$ for two FP operands $A = (-1)^{s_a} \times 2^{e_a} \times f_a$ and $B = (-1)^{s_b} \times 2^{e_b} \times f_b$. s_a denotes the sign, e_a the exponent, and f_a the significand including the implicit bit of the operand A (similarly for B and P). The operation can be defined as $s_p = s_a \oplus s_b$ and $2^{e_p} \times f_p = RND(2^{e_a+e_b} \times f_a \times f_b)$. RND is the round and normalize function according to the IEEE standard for floating-point arithmetic (IEEE Std 754-2008).

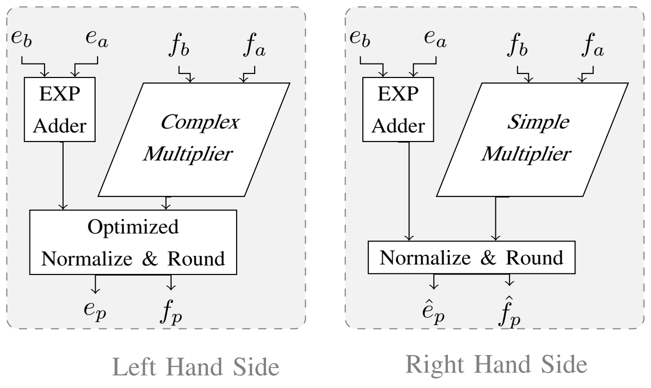


Fig. 3. Compared FP Multiplier Circuits

We have scaled and modified the structure of the FP multiplier unit of the open cores design module DOUBLE-FPU [14] for building dissimilar FP instances. As shown in Fig. 3, the compared circuits have different multiplier architectures and their control logic units are optimized distinctively. The multiplier units are generated using the online tool *Arithmetic Module Generator* [15]. These generated circuits[2] were synthesized from Verilog to gate level netlists using *Yosys* [16].

The multiplier architectures are categorized according to 1) the type of the partial products generator, 2) the partial products accumulator, and 3) the last stage adder. In our experiments, we use a partial products generator, namely *simple partial products* (SP). The types of partial products accumulators are *array* (AR), (4,2) *compressor tree* (CT), and *wallace tree* (WT). The last stage adder are *ripple carry adder* (RC), *carry look-ahead adder* (CL), and *brent-kung adder* (BK).

In Table I, we demonstrate the runtimes of checking the equivalences of divergent FP multipliers against the same circuit reference. The reference consists of a simple multiplier (SP-AR-RC) and unoptimized normalize round unit. While the compared circuits contain complex multipliers and round units which are optimized using the Yosys option *share*[3]. The first column of Table I shows the type of the multiplier architecture. The second and the third columns give the number of bits of an FP operand of the circuit in addition to the size of its significand and its exponent according to the IEEE standard. The next three columns provide the runtimes. The timeout (TO in the table) is set to 24 hours. The experimental results clearly demonstrate the advantage of ACEC in verifying circuits that include data-path and control logic. While other equivalence checking tools can verify the correctness up to 16 bits, we are able to verify the correctness of a single precision binary floating-point multiplier (32 bit).

Table II shows some statistics about the algorithms of ACEC for checking the equivalence of the FP multiplier instances that contain the multiplier architecture SP-WT-CH. For the reverse engineering algorithm, it shows the runtime of rewriting the combined model G; the runtime of extracting and abstracting data-path units; and the number of the extracted units. These results show that the reverse engineering algorithm extracts

[2]The benchmarks, binary of our tool, and log files are available at http://www.informatik.uni-bremen.de/agra/eng/asc.php

[3]It merges shareable resources into a single resource. A SAT solver is used to determine if two resources are shareable

TABLE I
RUNTIMES FOR CHECKING FP MULTIPLIERS EQUIVALENCES

Multiplier Architecture	FP operand # bits	Significand/Exponent # bits	Commercial (h:m:s)	ABC (h:m:s)	ACEC (h:m:s)
SP-CT-BK	16	13/3	00:08:50	TO	00:01:42
SP-WT-CH	16	13/3	00:09:08	TO	00:01:44
SP-CT-BK	24	21/3	TO	TO	00:17:49
SP-WT-CH	24	21/3	TO	TO	00:25:58
SP-CT-BK	32	25/7	TO	TO	02:24:01
SP-WT-CH	32	25/7	TO	TO	03:41:43

TABLE II
STATISTICS OF ACEC FOR EQUIVALENCE CHECKING OF FP MULTIPLIERS

# bits	ACEC Algorithms		
	Reverse Engineering		
	Model Rewriting (h:m:s)	Extract & Abstract (h:m:s)	# Data-path Units
16	00:00:46	00:00:23	21
24	00:11:56	00:10:04	23
32	00:32:50	02:10:30	23
	Arithmetic Sweeping		
	# Variables of G	# Proved Equivalences	Runtime (h:m:s)
16	1888	401	00:00:27
24	4440	666	00:03:36
32	5889	854	00:58:04
	Efficient Polynomial Representation		
	Decomposition		Logic Reduction
	# Reduced Terms	# Eff./Total Calls	# Canceled Terms
16	2400	514/3477	2916
24	9732	1013/8684	17153
32	16317	1345/12477	36390

more candidates for data-path units than the expected number. For two combined FP multipliers, six data-path units should be extracted, two significand multipliers, two exponent adders, and two incrementers in the rounding stages. Also, the results show that most of the run-time of ACEC is spent in reverse engineering (on average about 65%).

For arithmetic sweeping, Table II gives total the number of variables of the combined model; the number of proved equivalences between variables of the two compared circuits; and the time spent by the sweeping algorithm. The results demonstrate that variables of G which have functional similarities between each other account for less than 45% of the total the number of variables. Further, the table shows the number of saved terms by the decomposition of polynomials; the number of effective (Eff.) calls for the decomposition algorithm wrt. the total calls for the algorithm (effective calls are those which save terms of polynomials), and the number of canceled terms by the reduction rule.

VIII. RELATED WORK

One noteworthy challenge is developing a fully automated technique which proves that a floating-point design is in consistence with the IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008). Theorem provers have been applied extensively to verify the properties of floating-point designs. Although a lot of automation has been added and floating-point libraries have been created to avoid repetition of proofs, theorem proving methodology still requires an enormous amount of manual effort, expert knowledge, and high understanding of the design [17]. The paper by Jacobi [18] is the most automated work up to today, however, it skips the hardest part to verify, the multiplier.

As mentioned already in the paper all the existing works (e.g. [1]–[3], [9]) using Gröbner bases for circuit verification only target pure arithmetic components w/o control logic.

The recently proposed reverse engineering algorithms [19], [20] for the extraction of arithmetic word level components from a gate-level netlists are not applicable to designs with a non-arithmetic combinational logic attached to the output.

IX. CONCLUSION

In this paper we have presented a new algebraic equivalence checking technique for checking the equivalence of circuits that combine data-path and control logic. The technique utilizes a new reverse engineering algorithm to extract and abstract arithmetic components from the combined model of the Gröbner bases representation of the compared circuits. Based on input and output boundaries of the abstracted components the proposed arithmetic sweeping deduces less and promising candidates for bit and word equivalences between the compared circuits. The technique circumvents the blow-up in the number of terms of polynomials during the utilized algorithms by offering different types of decompositions for polynomials and using an efficient reduction rule. Experimental results demonstrated the efficiency of our technique for the equivalence checking of large floating-point multipliers which cannot be verified with existing Boolean combinational equivalence checking techniques.

For future work we want to investigate canonization for control logic as well as managing the membership testing for non-equivalent circuits.

REFERENCES

[1] F. Farahmandi and B. Alizadeh, “Gröbner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction,” *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[2] T. Pruss, P. Kalla, and F. Enescu, “Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields,” *TCAD*, vol. 35, no. 7, pp. 1206–1218, 2016.

[3] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, “Formal verification of integer multipliers by combining Gröbner basis with logic reduction,” in *DATE*, 2016, pp. 1048–1053.

[4] A. Kühlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[5] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, “Improvements to combinational equivalence checking,” in *ICCAD*, 2006, pp. 836–843.

[6] D. Cox, J. Little, and D. O’Shea, *Ideals, Varieties, and Algorithms*. Springer, 1997.

[7] Y. Sato, S. Inoue, A. Suzuki, K. Nabeshima, and K. Sakai, “Boolean Gröbner bases,” *Journal of symbolic computation*, vol. 46, no. 5, pp. 622–632, 2011.

[8] A. Nagai and S. Inoue, “An implementation method of boolean Gröbner bases and comprehensive boolean Gröbner bases on general computer algebra systems,” in *International Congress on Mathematical Software*, 2014, pp. 531–536.

[9] M. Ciesielski, C. Yu, D. Liu, and W. Brown, “Verification of gate-level arithmetic circuits by function extraction,” in *DAC*, 2015, pp. 52:1–52:6.

[10] E. Goldberg, M. Prasad, and R. Brayton, “Using SAT for combinational equivalence checking,” in *DATE*, 2001, pp. 114–121.

[11] R. Drechsler and D. Sieling, “Binary decision diagrams in theory and practice,” *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, 2001.

[12] R. Drechsler, M. Herbstritt, and B. Becker, “Grouping heuristics for word-level decision diagrams,” in *ISCAS*, vol. 1. IEEE, 1999, pp. 411–414.

[13] R. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *CAV*, 2010, pp. 24–40.

[14] D. Lundgren, “Double precision floating point core verilog,” available at http://opencores.org/project,double_fpu, 2016.

[15] “Arithmetic module generator based on ACG,” available at http://www.aoki.ecei.tohoku.ac.jp/arith/, 2016.

[16] C. Wolf, “Yosys open synthesis suite,” available at http://www.clifford.at/yosys/, 2016.

[17] A. Slobodová, *Challenges for formal verification in industrial setting*. Springer Berlin Heidelberg, 2007.

[18] C. Jacobi, k. Weber, V. Paruthi, and J.Baumgartner, “Automatic formal verification of fused-multiply-add FPUs,” in *DATE*, 2005, pp. 1298–1303.

[19] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, “Simulation graphs for reverse engineering,” in *FMCAD*, 2015, pp. 152–159.

[20] C. Yu and M. Ciesielski, “Automatic word-level abstraction of datapath,” in *ISCAS*, 2016, pp. 1–6.

Accurate ICP-based Floating-Point Reasoning

Karsten Scheibler*, Felix Neubauer*, Ahmed Mahdi†, Martin Fränzle†,
Tino Teige‡, Tom Bienmüller‡, Detlef Fehrer§, Bernd Becker*
*Chair of Computer Architecture, University of Freiburg, Germany
†Research Group Hybrid Systems, Carl von Ossietzky University of Oldenburg, Germany
‡BTC Embedded Systems AG, Oldenburg, Germany
§Sick AG, Waldkirch, Germany

***Abstract*—In scientific and technical software, floating-point arithmetic is often used to approximate arithmetic on physical quantities natively modeled as reals. Checking properties for such programs (e.g. proving unreachability of code fragments) requires accurate reasoning over floating-point arithmetic.**

Currently, most of the SMT-solvers addressing this problem class rely on bit-blasting. Recently, methods based on reasoning in interval lattices have been lifted from the reals (where they traditionally have been successful) to the floating-point numbers. The approach presented in this paper follows the latter line of interval-based reasoning, but extends it by including bitwise integer operations and cast operations between integer and floating-point arithmetic. Such operations have hitherto been omitted, as they tend to define sets not concisely representable in interval lattices, and were consequently considered the domain of bit-blasting approaches. By adding them to interval-based reasoning, the full range of basic data types and operations of C programs is supported. Furthermore, we propose techniques in order to mitigate the problem of aliasing during interval reasoning.

The experimental results confirm the efficacy of the proposed techniques. Our approach outperforms solvers relying on bit-blasting as well as the existing interval-based SMT-solver.

***Index Terms*—SMT, floating-point, dead-code detection**

I. Introduction

Already with the advent of digital computing in the 1940s, the need arose for encoding and manipulating real-valued quantities by sufficiently densely spaced discrete approximations and their pertinent operations. The two immediate suggestions, namely to employ equidistant quantization and integer encodings (today known as fixed-point representation) or alternatively to provide wider value ranges by scaling a fixed-point mantissa with a variable integer exponent (called floating-point representation) have since remained the dominant technical solutions – while any alternatives, like exact handling of fractions, stay properly confined to niche applications only. Despite being conceived and implemented as early as Zuse's Z3 computer completed in 1941, floating-point numbers are still the method of choice for representing real-valued quantities in scientific and technical computing.

Given that in most applications floating-point numbers are just substitutes for the physical entities they are meant to represent (e. g. signal values), there is a long-standing debate in the formal verification community as to:

1) whether automatic analysis tools should either manipulate abstract algorithms over the reals neglecting their implementation details (a stance usually taken in the hybrid systems community), or
2) evaluate properties of the actually implemented algorithms including the peculiarities of their machine data types (a position frequently encountered in the program analysis community), or
3) even combine the first with the second in a variety of forms, e. g. in models of embedded feedback control treating environmental entities as reals while manipulating their computational images as floats.

From our perspective, there is no definite answer to these issues, as the adequacy of selecting a certain variant of arithmetic clearly depends on the context, which may cover parts of both: the physical environment as well as the embedded program. In our satisfiability-modulo-theory (SMT) solver iSAT3, we have consequently embedded support for the reals and for various float and integer formats, plus the necessary casts in between for being able to relate them within logical expressions.

The interesting fact about such a combination bridging mathematical and computational arithmetic theories is that it crosses the border between two fundamentally different implementation paradigms in SMT solving. On the one hand, floating-point reasoning has mostly been implemented in SMT-solvers via so-called bit-blasting (see e. g. [1]) – reducing constraint problems over the floats to propositional SAT and its corresponding solvers. On the other hand, the reals are not immediately amenable to such bit-blasting – which has led to set-based methods for handling their models in constraint solving. Various connections between the two worlds have been identified and exploited over the years, like

- DPLL(T) and CDCL(T) algorithms for SMT solving [2],
- "logical arithmetic" [3] and interval constraint propagation [4] traversing interval lattices over the reals,
- the iSAT [5] and ACDCL [6] algorithms unifying CDCL-style SAT-solving with constraint propagation in lattices.

Despite this, the basic approaches have until recently remained dichotomous in the SMT community: set-based reasoning is for the reals and bit-blasting for the floats. Only very recently Brain et al. [7] have demonstrated that set-based reasoning exploiting the lattice structure of floating-point intervals in an ACDCL (abstract conflict-driven clause learning) scheme can efficiently solve floating-point constraint systems. The price for avoiding bit-blasting, however, was a confinement to "usual" arithmetic operations, like addition, multiplication, etc. – thereby avoiding support of bitwise operations and casts between types, which tend to define sets not concisely representable in interval lattices. As such operations are regularly encountered in actual programs, this particular approach does not yet pose a threat to the predominance of bit-blasting approaches for the analysis of floating-point constraints derived from program analysis.

We address that issue by providing an SMT-solver that (1) scales well on floating-point dominated arithmetic constraint problems, (2) supports all kinds of type casts and bitwise operations usually encountered in imperative programs, and (3) permits to reason about machine data types as well as real numbers, as necessary for the analysis of actual embedded control. Therefore, this paper presents the extensions we added to iSAT3 in order to obtain an arithmetic SMT-solver that exploits abstract, set-based

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

reasoning – yet reconciles it with support for the full range of basic data types and operations of, e. g., C programs.

The experimental results obtained on various industrial benchmarks confirm the efficacy of the proposed techniques. Especially on unsatisfiable constraint instances (which are relevant for our application domain of reliably detecting subtly data-dependent unreachability of code fragments in embedded C code) our approach consistently outperforms current solvers relying on bit-blasting as well as the existing ACDCL-based floating-point SMT-solver.

Structure of the paper.: After introducting the preliminaries and iSAT3 in Section II, we describe our approach in Section III. Experimental results are discussed in Section IV, before we conclude the paper in Section V.

II. iSAT3

In the following we will denote Boolean variables with b, integer-valued variables with i and real-valued variables with r. Furthermore, literals associated to i are denoted with li (and with lr in case of r).

A. Preliminaries

We provide a short introduction to SAT and related techniques (for a more detailed survey refer to [8]) as it will be helpful for understanding the basic concepts in iSAT3. Given a propositional formula and asking the question whether there exists an assignment to its variables rendering the formula true is also known as the *satisfiability problem* (SAT). Programs for solving this kind of problem are called SAT-solvers.

Most modern SAT-solvers do not operate on arbitrary Boolean formulas – instead they require a *conjunctive normal form* (CNF). The Tseitin-transformation [9] can be used to rewrite a given Boolean formula into CNF. A CNF consists of a conjunction of clauses with each clause being a disjunction of literals and a literal being a Boolean variable b or its negation $\neg b$.

In the *conflict-driven clause learning* (CDCL) [10] scheme the search process for a satisfying assignment consists of three alternating steps: propagation of implied assignments, deciding a variable and resolving a conflict.

Propagation: One core component of a SAT-solver is the *Boolean constraint propagation* (BCP) [11] which is used to detect implied assignments. Everytime a clause with n literals contains $n-1$ literals being already assigned to false (a so-called *unit-clause*), the remaining literal has to be true in order to retain a chance to satisfy the formula. An implication queue is used for keeping track of all variables which were assigned recently and thus might have created new unit-clauses. BCP is applied until either no unit-clauses are left or a *conflicting clause* (a clause with all literals being false) was derived.

Decisions: If BCP finished without deriving a conflicting clause, a decision is made. This is done by assigning a currently unassigned variable with true or false. Every assignment to a variable (either due to propagation or due to a decision) is stored in the so-called implication graph.

Conflict resolution: In case a conflicting clause was derived, it will be the starting point of the conflict analysis. The implication graph is traversed backwards according to the *first unique implication-point* (1UIP) scheme in order to construct a *conflict clause* which explains the current conflict. Furthermore, non-chronological backtracking is performed until the created conflict clause becomes unit. Adding this clause to the CNF prevents the SAT-solver to visit this conflict again and thus prunes the search space.

In *bounded model checking* (BMC) [12] a Boolean formula F is used to encode a transition system along with a property to be checked. F contains symbolic representations of the initial state(s) I, the transition relation T and the negated property $\neg P$. F is satisfiable if and only if the underlying state transition system allows a finite sequence of transitions violating P. BMC is able to prove the absence of such sequences up to a user-defined depth k. In order to prove that P holds for all k, Craig interpolation can be applied.

For propositional formulas A and B with $A \Rightarrow B$, a Craig interpolant C is an overapproximation of A which still implies B: $A \Rightarrow C \wedge C \Rightarrow B$. Furthermore, C only contains variables which occur in A and B. Craig interpolants can be computed with a SAT-solver by exploiting the resolution proof which is created during solving the unsatisfiable formula $A \wedge \neg B$. In the context of BMC, Craig interpolation is used to find a set of states S being invariant regarding T and overapproximating the reachable states [13].

B. The SMT-Solver iSAT3

SAT-modulo-theories (SMT) aims at solving Boolean combinations of *theory atoms*, e. g.

$$(r_1 + r_2 + r_3 < 7) \wedge ((r_1 \geq r_2) \vee (\sin(r_1) \cdot r_3 < 10))$$

In classical SMT a given formula is split into a set of theory atoms and a Boolean skeleton which abstracts the truth-values of the theory atoms with Boolean literals. Therefore, a SAT-solver and a separate theory solver are employed during the solving process. This scheme is also abbreviated as DPLL(T) or CDCL(T) – with T being the theory used within the atoms.

In contrast to CDCL(T), there is no such separation between the SAT and the theory part in the iSAT algorithm [5], [14] – instead *interval constraint propagation* (ICP, see e. g. [4]) is tightly integrated into the CDCL framework in order to reason about the theory atoms directly, yielding a unified lattice-based view now known as abstract conflict-driven clause learning (ACDCL) [6].

In this paper we build on iSAT3 [15], [16] – which is the third implementation of the iSAT algorithm. The iSAT algorithm and all its implementations were originally developed for the verification of hybrid systems. Therefore, they aim at solving SMT formulas containing Boolean, integer- and real-valued variables. The theory atoms may contain linear and non-linear arithmetic involving transcendental functions, e. g. $(i_1 + i_2 = i_3)$, $(|r_1 - r_2| < \min(r_1, r_2))$ or $(\sqrt[3]{r_3} + \sin r_4 < e^{r_5})$.

The iSAT algorithm maps each integer- and real-valued variable to an interval and expects for those variables initial intervals as part of the given formula. In the context of hybrid systems, variables usually encode physical quantities like temperature and velocity and have therefore natural bounds. During the search process these intervals will be narrowed with ICP in order to find a solution. Since each interval bound is represented as a floating-point number, outward rounding is applied in order to get safe interval enclosures. This ensures that ICP only cuts off definitive non-solutions. But the iSAT algorithm might not find a conclusive answer in all situations – as equations like $r_1 = r_2 \cdot r_3$ can only be satisfied by point intervals in general. For continuous domains, ICP cannot guarantee to reach such point intervals. Nonetheless, if the iSAT algorithm classifies an SMT formula F as satisfiable (or unsatisfiable) then F is indeed satisfiable (or unsatisfiable). In the remaining cases, the algorithm terminates with a *candidate solution*. Because it relies on machine data types with a finite precision, the iSAT algorithm always terminates. But for performance reasons we omit newly deduced bounds if the difference to the current bound is below the *minimum progress*. Furthermore, interval splits are only performed if an interval has a larger width than the *minimal splitting width*.

Before solving, a Tseitin-like transformation is applied to the given SMT formula in order to obtain a conjunctive form by introducing new auxiliary variables for sub-expressions. In particular, each theory atom is decomposed into a *simple bound literal* and a set of *primitive constraints*. **Example ①**: the formula

$$(r_1 + r_2 + r_3 < 7) \wedge (r_1 \geq r_2)$$

would be rewritten as follows by introducing the new auxiliary variables b_{h1} (for the Tseitin-encoding of the $\wedge$-operator) and r_{h1}, r_{h2}, r_{h3} (for the decomposed theory atoms):

$$\begin{gathered}(lr_{h2}^{\prec 7} \vee \neg b_{h1}) \;\wedge\; (lr_{h3}^{\succeq 0} \vee \neg b_{h1}) \;\wedge \\ (\neg lr_{h2}^{\prec 7} \vee \neg lr_{h3}^{\succeq 0} \vee b_{h1}) \;\wedge\; (b_{h1}) \;\wedge \\ (r_{h1} = r_1 + r_2) \;\wedge\; (r_{h2} = r_{h1} + r_3) \;\wedge\; (r_{h3} = r_1 - r_2)\end{gathered}$$

Each simple bound literal imposes a lower or an upper bound on the associated integer- or real-valued variable. Within simple bound literals we denote the relational operators less-than, less-equal, greater-equal and greater-than with $\prec, \preceq, \succeq$ and $\succ$[1]. A simple bound literal could be strict, (e. g. $lr^{\prec 4}$) or non-strict, (e. g. $lr^{\preceq 4}$)[2]. As example ① shows, sometimes it is necessary to rewrite theory atoms in order to obtain a simple bound literal for them (e. g. $(r_1 \geq r_2)$ was rewritten to $(r_1 - r_2 \geq 0)$). A primitive constraint contains up to three variables besides a unary or binary operator. Having a fixed set of primitive constraints makes it easier to apply ICP later on.

The search process in iSAT3 is similar to CDCL: it consists of alternating propagation and decision phases interspersed with the resolution of conflicts – but additionally, the first two phases are extended with ICP and interval splits.

Propagation: Similar to a SAT-solver, we use an implication queue to keep track of changed variables. Besides Boolean variables, this queue might also contain integer- or real-valued variables. BCP handles all unit-clauses which contain changed Boolean literals and simple bound literals. Additionally, every unassigned simple bound literal will be evaluated during BCP whether it is already implied by another simple bound literal associated to the same integer- or real-valued variable. **Example ②**: if the simple bound literal $lr^{\preceq 7}$ is already false and $lr^{\preceq 5}$ is unassigned, the evaluation of $lr^{\preceq 5}$ will result in the generation of the *implication clause* $(\neg lr^{\preceq 5} \vee lr^{\preceq 7})$ which is attached to the implication graph. This clause is unit and will imply $\neg lr^{\preceq 5}$ immediately.

Furthermore, ICP-contractors are applied to all those primitive constraints which contain changed integer- or real-valued variables. Each deduction performed by ICP generates a new clause (consisting of simple bound literals) which contains the reasons of the deduction as well as the newly deduced stronger bound. These clauses are attached to the implication graph. **Example ③**: for the primitive constraint $(r_{h1} = r_1 + r_2)$ with $r_{h1} \in [1, 9], r_1 \in [1, 3]$ and $r_2 \in [4, 10]$ the new lower bound for r_{h1} can be deduced because of the current lower bounds of r_1 and r_2: $(\neg lr_1^{\succeq 1} \vee \neg lr_2^{\succeq 4} \vee lr_{h1}^{\succeq 5})$.

We use ICP solely to deduce new bounds – in our implementation ICP does not check whether an empty interval was derived. Instead, this check is performed on the Boolean level by evaluating the newly created simple bound literal l which is unassigned at the moment of its creation. If needed, an implication clause is created in order to assign l. This might cause an ICP-generated clause to become conflicting and triggering the conflict resolution.

The propagation phase ends when either the implication queue is empty or a conflicting clause was derived.

Decisions: If no conflicting clause was derived, a decision is made. Besides deciding existing Boolean literals and simple bound literals, an integer- or real-valued variable could be subject of a decision as well. This is done by splitting its interval and mapping the split-value to a new simple bound literal which is then decided.

Conflict resolution: In case a conflicting clause was derived, iSAT3 operates like a SAT-solver – because all clauses in the implication graph only contain assigned literals. At this point it does not matter whether they are Boolean or simple bound literals. In contrast to classical SMT-solving which only learns inconsistent combinations of theory atoms, iSAT3 introduces new simple bound literals during the search process and therefore refines the Boolean skeleton with interval boxes which do not contain solutions.

In fact iSAT3 represents a given SMT formula completely as a CNF with these three kinds of clauses:

1) the clauses in the Boolean skeleton: cl_bs
2) the implication clauses: cl_impl
3) the clauses generated by ICP-contractors: cl_arith

Regarding example ①, the Boolean skeleton cl_bs is encoded in the first four clauses. The truth-values of the decomposed theory atoms are represented by the simple bound literals $lr_{h2}^{\prec 7}$ and $lr_{h3}^{\succeq 0}$. The three primitive constraints can be also seen as place-holders for all the cl_arith clauses which can be generated with the according ICP-contractors.

While the clauses in cl_bs are generated before the solving process starts, the clauses in cl_impl and cl_arith are generated lazily on-the-fly during solving. Therefore, iSAT3 performs Craig interpolation as in the propositional case. The resolution proof (used for creating the Craig interpolant) is build from these clauses:

$$\begin{gathered}cl\_bs_A \wedge cl\_impl_A \wedge cl\_arith_A \wedge \\ cl\_bs_{\neg B} \wedge cl\_impl_{\neg B} \wedge cl\_arith_{\neg B}\end{gathered}$$

III. Extending iSAT3 with FP Reasoning

In the following we will denote floating-point variables with f and literals associated to f with lf. Furthermore, we denote the floating-point comparisons less-than, less-equal, equal, greater-equal and greater-than with $<_F, \leq_F, =_F, \geq_F$ and $>_F$. Note that for floating-point operands $(f_1 <_F f_2)$ is not equivalent to $\neg(f_1 \geq_F f_2)$ – if one of the operands is a NaN both comparisons will be false (the same applies to $(f_1 \leq_F f_2)$ and $\neg(f_1 >_F f_2)$). Further note that there is no $(f_1 \neq_F f_2)$ – this comparison is interpreted as $\neg(f_1 =_F f_2)$. Additionally, we denote the relational operators less-than, less-equal, greater-equal and greater-than for simple bound literals with $\prec_F, \preceq_F, \succeq_F$ and $\succ_F$

In this paper we focus on floating-point values (i. e. normal and sub-normal numbers, -0, +0, *not-a-number* (NaN), -inf and +inf) with radix 2 according to the IEEE-754 standard [17]. In

[1]While there is no semantic difference between $(<, \leq, \geq, >)$ and $(\prec, \preceq, \succeq, \succ)$ in the context of integer and real numbers, there will be a subtle difference between the floating-point comparison operators and its relational operators for simple bound literals regarding the signed zeros.

[2]In fact strict bounds are only needed for real-valued variables in order to represent the negation of a simple bound literal properly: $\neg lr^{\preceq 4} \Leftrightarrow lr^{\succ 4}$. This is not needed for integer-valued variables, because: $\neg li^{\preceq 4} \Leftrightarrow li^{\succeq 5}$.

the following we describe the extensions added to iSAT3 in order to support accurate floating-point reasoning (see Section III-A) as well as ICP-based reasoning for bitwise integer operations (see Section III-C). But besides adaptions specific for floating-point variables, we also propose two techniques which mitigate the problem of *aliasing* during ICP in general (i. e. one variable occurs more than once in a theory atom causing coarser intervals to be deduced by ICP, see Sections III-B and III-E).

Already having intervals whose endpoints are represented with floating-point numbers, makes it look very organic to extend iSAT3 in order to allow accurate reasoning over floating-point arithmetic. As a design decision we wanted to keep the ICP-contractors close to their current functionality. Therefore, we use a separate encoding for the floating-point value NaN and introduce a special NaN-literal for every floating-point variable. This allows us to handle the NaN-related propagations completely with BCP outside of the ICP-contractors (see Section III-A1).

There are further changes in iSAT3 regarding floating-point variables: (1) an initial interval is no longer required, (2) we use a deduction limit (see Section III-E) instead of the minimum progress, and (3) we do not apply the minimum splitting width in order to split always down to point intervals. As we only have non-strict bounds in floating-point context, this ensures that the solver terminates with a conclusive answer in all cases. The authors of [7] also offer a decision procedure for floating-point logic as an instance of abstract CDCL (ACDCL). Besides using ICP, their approach has further similarities with the original iSAT algorithm [5], [14], e. g. the reasoning over simple bounds.

A. Accurate Floating-Point Reasoning

1) Adapting the Boolean Encoding: Floating-point arithmetic introduces some special values (e. g. -inf and +inf) in order to accomodate the finite amount of representable numbers. We represent all subnormal numbers, both infinities and the signed zeros as simple bound literals – in contrast to the floating-point comparisons, we distinguish the signed zeros with $-0 \prec_F +0$ in the context of simple bound literals[3]. As mentioned, NaN is handled separately. This requires an adaption in the Boolean encoding: a floating-point variable could be either represented as an interval (like it is already done with real arithmetic) – or it could be NaN. For this reason we introduce a NaN-literal lf^{NaN} for every floating-point variable f. If lf^{NaN} is true then f is NaN – regardless of other assigned simple bound literals lf. If lf^{NaN} is false the simple bound literals lf become relevant.

In each clause in cl_impl and cl_arith we add lf^{NaN} once whenever a simple bound literal for f occurs. When considering an implication clause similar to example ② but now for a floating-point variable f it would look like this:

$$(lf^{NaN} \vee \neg lf^{\preceq_F 5} \vee lf^{\preceq_F 7})$$

In the same way, when considering a clause created by an ICP-contractor similar to example ③ but now with the floating-point variables f_{h1}, f_1 and f_2, it would look like this:

$$(lf_1^{NaN} \vee lf_2^{NaN} \vee lf_{h1}^{NaN} \vee \neg lf_1^{\succeq_F 1} \vee \neg lf_2^{\succeq_F 4} \vee lf_{h1}^{\succeq_F 5})$$

Obviously, these clauses are satisfied whenever a contained NaN-literal is true – rendering the truth-values of the remaining literals in the clause irrelevant.

[3]All simple bound literals are ordered according to the `totalOrder` predicate from [17] – in contrast to `totalOrder`, we keep NaN out of our ordering and treat it as unordered.

While we add the NaN-literals directly to the clauses in cl_impl and cl_arith, we use a different approach for the Boolean skeleton cl_bs. Here, we embed the NaN-literals before applying the Tseitin-like transformation. Furthermore, we ensure that floating-point comparisons with one of the signed zeros are properly handled.

With a slight abuse of notation let fs_1 and fs_2 be arbitrary non-constant floating-point sub-expressions not containing comparison operators and let lfs_1 and lfs_2 be the literals associated to the auxiliary variables representing fs_1 and fs_2. Additionally, let $\circ_{\hat{F}} \in \{<_{\hat{F}}, \leq_{\hat{F}}, =_{\hat{F}}, \geq_{\hat{F}}, >_{\hat{F}}\}$ be floating-point comparison operators which are semantically equivalent to $\circ_F \in \{<_F, \leq_F, =_F, \geq_F, >_F\}$. The syntactic distinction is made, because we will apply rewrite rules on the theory atoms and want to keep track whether the NaN-cases are already handled for a comparison operator. Let c be a floating-point value and let $prev(c)$ be the directly neighboring floating-point value which is smaller than c. We apply the following rewrite rules to the theory atoms:

1) $(c <_F fs_1) \rightsquigarrow (fs_1 >_F c)$ and
   $(c \leq_F fs_1) \rightsquigarrow (fs_1 \geq_F c)$ and
   $(c \geq_F fs_1) \rightsquigarrow (fs_1 \leq_F c)$ and
   $(c >_F fs_1) \rightsquigarrow (fs_1 <_F c)$ and
   $(c =_F fs_1) \rightsquigarrow (fs_1 =_F c)$
2) $(fs_1 \circ_F c) \rightsquigarrow \neg lfs_1^{NaN} \wedge (fs_1 \circ_{\hat{F}} c)$
3) with $\bullet_F \in \{\leq_F, =_F, \geq_F\}$:
   $$\begin{aligned}(fs_1 \bullet_F fs_2) \rightsquigarrow\ & \neg lfs_1^{NaN} \wedge \neg lfs_2^{NaN} \wedge \\ & ((fs_1 - fs_2 \bullet_{\hat{F}} +0) \vee \\ & (lfs_1^{\preceq_F -inf} \wedge lfs_2^{\preceq_F -inf}) \vee \\ & (lfs_1^{\succeq_F +inf} \wedge lfs_2^{\succeq_F +inf}))\end{aligned}$$
4) with $\bullet_F \in \{<_F, >_F\}$:
   $$\begin{aligned}(fs_1 \bullet_F fs_2) \rightsquigarrow\ & \neg lfs_1^{NaN} \wedge \neg lfs_2^{NaN} \wedge \\ & (fs_1 - fs_2 \bullet_{\hat{F}} +0) \wedge \\ & \neg(lfs_1^{\preceq_F -inf} \wedge lfs_2^{\preceq_F -inf}) \wedge \\ & \neg(lfs_1^{\succeq_F +inf} \wedge lfs_2^{\succeq_F +inf})\end{aligned}$$
5) $(fs_1 =_{\hat{F}} c) \rightsquigarrow (fs_1 \geq_{\hat{F}} c) \wedge (fs_1 \leq_{\hat{F}} c)$
6) $(fs_1 \geq_{\hat{F}} c) \rightsquigarrow \neg(fs_1 <_{\hat{F}} c)$ and
   $(fs_1 >_{\hat{F}} c) \rightsquigarrow \neg(fs_1 \leq_{\hat{F}} c)$
7) $(fs_1 <_{\hat{F}} +0) \rightsquigarrow (fs_1 \leq_{\hat{F}} prev(-0))$
8) $(fs_1 \leq_{\hat{F}} -0) \rightsquigarrow (fs_1 \leq_{\hat{F}} +0)$.
9) $(fs_1 <_{\hat{F}} c) \rightsquigarrow (fs_1 \leq_{\hat{F}} prev(c))$
   If $c \in \{-inf, NaN\}$ the theory atom will be replaced with the Boolean constant false.

We apply these rules until no further rewritings are possible. If more than one rule could be applied at the same time, the more specific rule is applied first (e. g. rule 7 instead of 9). Note that after the rewriting all remaining theory atoms look like this: $(fs \leq_{\hat{F}} c)$[4]. This allows a direct translation into a simple bound literal: $lfs^{\preceq_F c}$. Afterwards the Tseitin-like transformation is applied to the rewritten formula in order to obtain cl_bs together with the set of primitive constraints.

The rules 3 and 4 are needed in order to allow the creation of simple bound literals which can be used within cl_bs. While the floating-point comparisons allow two infinities with the same sign as its operands, the floating-point subtraction will return a NaN. Therefore, these cases are handled explicitly outside of the subtraction in both rules.

With these rewrite rules the theory atom $(fs =_F +0)$ would be rewritten as follows:

[4]Simple bound literals for integer variables have no strict bounds, e. g. $\neg li^{\preceq 4} \Leftrightarrow li^{\succeq 5}$. We apply the same idea to simple bound literals for floating-point variables, e. g. $\neg lf^{\preceq_F \texttt{prev(-0)}} \Leftrightarrow f^{\succeq_F -0}$.

$\rightsquigarrow$ (rule 2) $\neg lfs^{NaN} \wedge (fs =_{\hat{F}} +0)$
$\rightsquigarrow$ (rule 5) $\neg lfs^{NaN} \wedge (fs \geq_{\hat{F}} +0) \wedge (fs \leq_{\hat{F}} +0)$
$\rightsquigarrow$ (rule 6) $\neg lfs^{NaN} \wedge \neg(fs <_{\hat{F}} +0) \wedge (fs \leq_{\hat{F}} +0)$
$\rightsquigarrow$ (rule 7) $\neg lfs^{NaN} \wedge \neg(fs \leq_{\hat{F}} prev(-0)) \wedge (fs \leq_{\hat{F}} +0)$

This means the floating-point comparison with +0 will be finally translated to simple bound literals which enclose -0 and +0.

Furthermore, there is an additional operator which mimics the semantics of an assignment in C. For this operator, it is required to distinguish all possible floating-point values – signed zeros and NaN as well. The rewriting rules for this operator have similarities to the rules above – but due to lack of space we omit these rules here.

We decided to encode NaN separately in order to perform the NaN-propagation outside the ICP-contractors. Therefore, we encode the propagation eagerly into clauses prior solving. E. g. for the primitive constraint $(f_{h1} = f_1 + f_2)$ we would generate these clauses:

$$(\neg lf_1^{NaN} \vee lf_{h1}^{NaN}) \ \wedge \ (\neg lf_2^{NaN} \vee lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee \neg lf_1^{\preceq_F -inf} \vee \neg lf_2^{\succeq_F +inf} \vee lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee \neg lf_1^{\succeq_F +inf} \vee \neg lf_2^{\preceq_F -inf} \vee lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee lf_1^{\preceq_F -inf} \vee lf_1^{\succeq_F +inf} \vee \neg lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee lf_2^{\preceq_F -inf} \vee lf_2^{\succeq_F +inf} \vee \neg lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee lf_1^{\preceq_F -inf} \vee lf_2^{\preceq_F -inf} \vee \neg lf_{h1}^{NaN}) \wedge$$
$$(lf_1^{NaN} \vee lf_2^{NaN} \vee lf_1^{\succeq_F +inf} \vee lf_2^{\succeq_F +inf} \vee \neg lf_{h1}^{NaN})$$

The clauses encode the following propagations:

- f_{h1} is NaN when f_1 or f_2 is NaN
- f_{h1} is NaN when f_1 and f_2 are infinities with opposite signs
- f_{h1} is not NaN when f_1 and f_2 are not NaN and f_1 is never -inf or +inf
- f_{h1} is not NaN when f_1 and f_2 are not NaN and f_2 is never -inf or +inf
- f_{h1} is not NaN when f_1 and f_2 are not NaN and f_1 and f_2 are never -inf
- f_{h1} is not NaN when f_1 and f_2 are not NaN and f_1 and f_2 are never +inf

Similar clauses exist for the remaining primitive constraints.

2) ICP-Contractors for Floating-Point: The way of doing ICP for floating-point arithmetic is very similar to doing floating-point approximated ICP for real arithmetic – especially the forward deduction only differs in how the floating-point rounding-mode is set. E. g. when deducing for r_{h1} in the primitive constraint $(r_{h1} = r_1 + r_2)$, the floating-point addition of the interval endpoints of r_1 and r_2 is executed with downward rounding for the lower bound and upward rounding for the upper bound. When looking at a similar primitive constraint with floating-point variables $(f_{h1} = f_1 + f_2)$, we calculate both bounds with the floating-point rounding-mode which is relevant for the originating theory atom – as all our benchmarks assume round-to-nearest with tie-to-even, we use this rounding-mode as our default. The backward deduction relies on ideas presented in [18].

B. Adapted Decision Heuristics

As mentioned above, floating-point ICP-contractors generate clauses containing NaN-literals. These clauses are only unit when the NaN-literals are false. Therefore, each floating-point ICP-contractor is only called if all variables in the primitive constraint under consideration have their NaN-literals assigned to false. For this reason, we introduce a queue of literals containing *preferred decisions* (prefdec). Everytime iSAT3 wants to make a decision, it will first check whether there are still unassigned literals in the prefdec-queue. If this is the case these literals are decided first. We put all NaN-literals in the prefdec-queue and let the solver assign them to false whenever possible.

Furthermore, we adapt the decision heuristics regarding interval splits. Every k-th split we pick a random integer or floating-point variable which is not an auxiliary variable and try to assign a point interval to it. On the one hand this might result in weaker conflict clauses (because in worst case only a point interval is excluded) – but on the other hand it helps to reduce the detrimental aliasing effect. We determined experimentally $k = 4$ to be a good trade-off.

C. ICP-based Reasoning for Bitwise Integer Operations

One of our use-cases is dead-code detection in C programs. These programs can contain a mix of floating-point arithmetic, integer arithmetic and bitwise integer operations. Thus, adding support for floating-point arithmetic is not enough – we have to include support for bitwise integer operations as well. Therefore, we add ICP-contractors for the following bitwise integer operations: BW-NOT, BW-AND, BW-OR, BW-XOR, BW-LSHIFT and BW-RSHIFT. Both BW-SHIFT-operations are more or less special forms of integer multiplication and division. The first four operations expect besides its operand(s) a bitwidth as additional constant argument. Here, the basic idea for the according ICP-contractors is to quickly get a safe overapproximation of the interval which encloses all possible results. E. g. for a BW-AND with positive operand intervals an overapproximation for the resulting upper bound is the minimum of the upper bounds of the two operands. Similarly, for a BW-OR with positive operand intervals the resulting upper bound is not larger than the sum of both upper bounds of the operands. These overapproximations can be refined when both operand intervals have a common bit prefix. A detailed overview is given in [19].

Furthermore, we add two ICP-contractors for integer casts: SCAST (for casting the operand interval to a signed interval) and UCAST (for casting the operand interval to an unsigned interval). Both contractors are more or less modulo operations according to a given constant bitwidth. With these cast-operators we are able to mimic the semantic of C integer operations, e. g. assignments between integers with different signedness or bitwidths. Additionally, ICP-contractors for casts between floating-point and integer variables are provided as well.

D. Relaxed Reasons for Multiplication and Division

When constructing the reasons of the performed ICP deductions for a primitive constraint, there is sometimes some degree of freedom. Up to now iSAT3 used a subset of the current bounds of the variables occuring in the primitive constraint as reasons, e. g. when calculating a new lower bound $(i_{h1} \succeq c_1)$ in the primitive constraint $(i_{h1} = i_1 + i_2)$ then the lower bounds $(i_1 \succeq c_2)$ and $(i_1 \succeq c_3)$ are the reason for this deduction. For addition and subtraction, there is no choice, but this is not the case for multiplication and division. E. g. assume the upper bound $(i_{h2} \preceq 1000000)$ is calculated in the primitive constraint $(i_{h2} = i_3 \cdot i_4)$ with $i_3 \in [-1000, 1000]$ and $i_4 \in [900, 1000]$. Up to now the following reasons were used: $(i_3 \preceq 1000)$, $(i_4 \succeq 900)$ and $(i_4 \preceq 1000)$. But instead of $(i_4 \succeq 900)$ the simple bound $(i_4 \succeq 0)$ would be sufficient for this deduction. Such relaxations of the reasons are now used. To some extent this is a local version of the conflict generalization idea presented in [7].

E. Global-ICP

Up to now, ICP is applied to each primitive constraint individually. But sometimes a more global view is needed in order to recognize conflicts – especially if some kind of aliasing occurs in the set of primitive constraints. Lets explain the idea of global-ICP with the help of an example:

$$\ldots \wedge (b_1 \rightarrow (i_1 = i_2)) \wedge (i_1 \neq scast(ite(b_2, i_2, 0), 32))$$

If b_1 and b_2 are both true, the following constraint has to be solved: $(i_2 \neq scast(i_2, 32))$ – which is unsatisfiable if $i_2 \in [-2^{31}, 2^{31} - 1]$.

ICP is able to recognize this, but it would take millions of deductions – because it starts to shrink the intervals of i_1 and i_2 only by the smallest possible bound improvement. This results in a cyclic deduction chain which continously deduces slightly better bounds for i_1 and i_2.

Such cyclic deduction chains are recognized in iSAT3 by applying a limit to the number of deductions performed per variable within one call of the deduction function. We set this limit to 64. If a variable v exceeds this limit, no further deductions are made for v within the current call. Furthermore, global-ICP will analyze the implication graph. First, all primitive constraints are collected which contributed to new bounds for v. Because the deduction chain is cyclic, we expect to see the same primitive constraints over and over again.

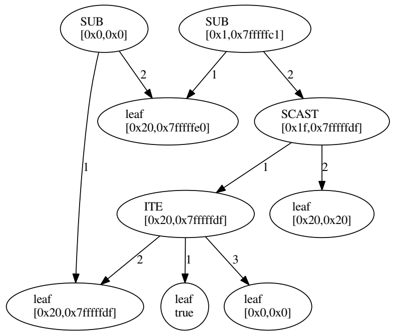


Fig. 1. The primitive constraints from the example involved in the cyclic deduction chain (before simplification).

Regarding the example, assume during the search process b_1 and b_2 were both set to true and decisions as well as propagations lead to $i_1, i_2 \subseteq [-2^{31}, 2^{31} - 1]$. This triggers a cyclic deduction chain. In Figure 1 the involved primitive constraints and its intervals are shown (the number on each edge represents the argument order – note that the second edge of the ITE-node is in fact its first operand).

There are some simplifications possible, e.g. the ITE-node is obsolete, because the Boolean selector variable b_2 is currently true. We can remove the ITE-node if we keep b_2 as a side-condition for a possible conflicting clause. Furthermore, the SUB-node with the interval $[0, 0]$ could be removed – because $(i_1 - i_2 = 0) \Leftrightarrow (i_1 = i_2)$. We do this by replacing the first argument of the SUB-node with its second argument[5]. The remaining primitive constraints are shown in Figure 2.

Now it is checked whether both children of the remaining SUB-node are syntactically equivalent. In this example this is the case, because the SCAST-node is obsolete regarding the current

[5]For real and integer numbers this holds in all cases. For floating-point arithmetic, there is a special case: the signed zeros. For all other floating-point values f with $(-inf <_F f <_F +inf) \wedge f \notin \{-0, +0, NaN\}$ this replacement is valid. Therefore, it is ensured that only floating-point intervals which do not contain -0 or +0 are considered in such cases.

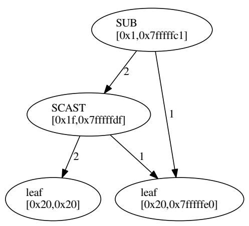


Fig. 2. The primitive constraints from the example involved in the cyclic deduction chain (after simplification).

interval of its first operand and the bitwidth contained in its constant second argument. A SUB-node with two syntactically equivalent children and an interval not containing 0 is conflicting. Therefore, a conflicting clause is created which contains the side conditions of the removed primitive constraints, the intervals of the remaining non-constant leaf nodes and the negated literal representing the lower bound of the SUB-node. This clause is then used as the starting point of the conflict analysis in order to obtain a conflict clause.

$$\left(\neg b_2 \vee \neg li_{h1}^{\succeq 0} \vee \neg li_{h1}^{\preceq 0} \vee \neg li_{1}^{\succeq \texttt{0x20}} \vee \neg li_{1}^{\preceq \texttt{0x7fffffe0}} \vee \neg li_{h2}^{\succeq 1}\right)$$

In such situations global-ICP can help to considerably prune the search space. It could be also applied in the context of Craig interpolation – as long as all involved primitive constraints are either completely from the A- or B-part of the formula. The idea is similar to the idea of creating clauses with the ICP-contractors for the primitive constraints. Every primitive constraint is marked with A (or B) – the created clause is thus a consequence from an A-fact (or B-fact). Therefore, it inherits the A-mark (or B-mark). Because it is ensured that all primitive constraints in global-ICP are either marked A (or B), the created conflicting clause is a consequence from A-facts (or B-facts) and therefore inherits the A-mark (or B-mark) as well. In our experiments, we observed that only in few cases a cyclic deduction chain contains primitive constraints coming from A and B, so global-ICP has been applied in most of the cases.

Our current implementation only checks for constellations similar to the example (including constellations with floating-point variables and certain floating-point cast-operators). This catches many cases, but certainly not all causes of a cyclic deduction chain.

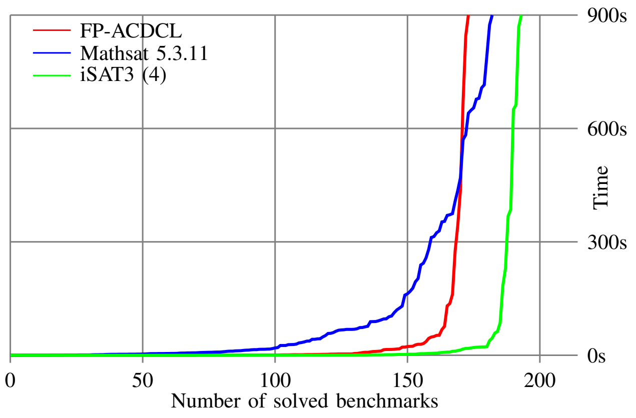
IV. Experimental Results

We performed two groups of experiments and evaluated four versions of iSAT3 in order to give an overview of the efficacy of global-ICP (see Section III-E), intermediate point splits (*psplits*, see Section III-B) and the bound relaxation (*brelax*, see Section III-D). The four versions of iSAT3 are:

(1) without brelax, psplits and global-ICP
(2) with brelax, but without psplits and global-ICP
(3) with brelax and psplits, but without global-ICP
(4) with brelax, psplits and global-ICP

A. Experiment 1

For the first experiment, we converted the 213 benchmarks provided in [7] into the iSAT3 input language. We did a comparison with FP-ACDCL [7] (using the best setting regarding



Solver	S+U	SAT	UNSAT	TO	MO
FP-ACDCL	173	97	76	40	0
Mathsat 5.3.11	182	101	81	23	8
iSAT3 (1)	164	90	74	47	2
iSAT3 (2)	166	91	75	45	2
iSAT3 (3)	186	111	75	27	0
iSAT3 (4)	193	111	82	20	0

Fig. 3. Comparison between FP-ACDCL, Mathsat and iSAT3 over a set of 213 benchmarks from [7].

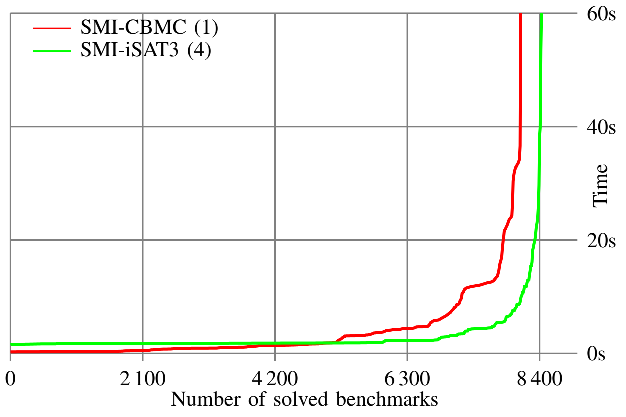
conflict generalization) and Mathsat5 5.3.11[6] [1] which relies on bit-blasting. The experiments were performed on an Intel Core i7-2600 with 3.4 GHz under Ubuntu 12.04 (32 bit). Per benchmark a time limit of 900 seconds and a memory limit of 2 GB was used. The results are shown in Figure 3. The diagram depicts the number of solved benchmarks together with the run time needed per benchmark. In the table, the columns SAT and UNSAT contain the number of solved satisfiable and unsatisfiable instances (the sum of both is shown in column S+U). The columns TO and MO contain the number of benchmarks which could not be solved within the given time limit (timeout) or within the given memory limit (memout).

All proposed techniques help to improve the number of solved benchmarks, e. g. brelax allows to solve two more satisfiable instances. With psplits and global-ICP we have two approaches mitigating the aliasing effect in ICP. While psplits helps to find a solution faster and therefore increases the number of solved satisfiable instances, global-ICP helps to prune the search space and increases the number of unsatisfiable instances.

In total iSAT3 (4) solved 193 benchmarks – this is an improvement of 20 benchmarks compared to FP-ACDCL and 11 benchmarks compared to Mathsat5. The picture is similar when comparing the number of solved satisfiable and unsatisfiable instances separately. In both cases iSAT3 (4) shows the best results. Furthermore, it can be observed that the ICP-based solvers FP-ACDCL and iSAT3 have lower memory requirements compared to Mathsat5. With its bit-blasting technique, Mathsat5 ran out of memory in 8 cases.

The diagram in Figure 3 reveals that also regarding runtime iSAT3 outperforms the other two solvers, e. g. when counting the number of instances which were solved within 2 seconds per instance, then Mathsat5 solved 36, FP-ACDCL 117 and iSAT3 (4) 148 instances.

[6]We used the Mathsat5 settings from [7], because they increase the number of solved instances compared to the default settings. Please note that Mathsat 5.2.6 is able to solve the benchmarks from [7] directly, while Mathsat 5.3.11 does not support the older syntax anymore. Therefore, we used the newer syntax provided in http://www.cs.nyu.edu/∼barrett/smtlib/QF_FP_Hierarchy.zip. Nonetheless, the results of Mathsat 5.2.6 and 5.3.11 are roughly the same.



Solver	S+U	SAT	U51	U∞	TO	MO
SMI-CBMC (0)	8166	7557	609	0	612	0
SMI-iSAT3 (4)*	8439	7373	1066	0	339	0
SMI-CBMC (1)	8099	7424	44	631	679	0
SMI-iSAT3 (1)	7647	6671	153	823	1131	0
SMI-iSAT3 (2)	7650	6673	154	823	1128	0
SMI-iSAT3 (3)	8169	7192	156	821	609	0
SMI-iSAT3 (4)	8430	7427	172	831	348	0

Fig. 4. Comparison between SMI-CBMC and SMI-iSAT3 over a set of 8778 SMI benchmarks. The table contains the results in two operational modes: BMC-only (lines 1-2), with k-induction and Craig interpolation (lines 3-7).

B. Experiment 2

In the second experiment we relied on a set of benchmarks which originate from TargetLink[7]-generated production C code (compiled from Simulink-Stateflow models) containing floating-point arithmetic and integer arithmetic as well as casts between these two domains. Each single benchmark represents a goal defined by a structural code coverage metrics like MC/DC. Starting with the original C code (annotated with coverage goals), the internal algorithms of the industrial-strength embedded test-vector generation tool BTC Embedded*Tester*®[8] preprocess and translate fragments of the code into an intermediate language called SMI (which is a C-like programming language). More precisely, the annotated C code is first translated into SMI and incrementally transformed into a simpler canonical form, e. g. by data type flattening and loop unrolling. The final SMI contains a global feedback loop where each loop execution corresponds to an execution step of the function call under test.

In our experiments, we used the version of SMI-CBMC which is part of BTC Embedded*Tester*® 3.4 (July 2014). SMI-CBMC relies on the bit-blasting-based solver CBMC 4.8[9] as its backend and supports *incremental* BMC, i. e. it incrementally unwinds and solves the step function calls of the global feedback loop [20]. We used SMI-CBMC with two different settings:

(0) pure, incremental BMC for efficiently finding test vectors for coverage goals.
(1) k-induction for detecting unreachable goals (dead-code). Due to technical reasons the current version of SMI-CBMC does not support incremental solving in this mode.

Similar to SMI-CBMC, SMI-iSAT3 reads a SMI file, translates it into the iSAT3 input language and calls iSAT3 in order to solve

[7]http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm
[8]http://www.btc-es.de/
[9]http://www.cprover.org/svn/cbmc/releases/cbmc-4.8-incremental/

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

the translated file. SMI-iSAT3 is a prototypical implementation which uses a bunch of wrapper scripts and runs in a Cygwin environment. Furthermore, every satisfying assignment found by iSAT3 was successfully verified by a SMI-simulator. All experiments were performed on an Intel Core i7-2600 with 3.4 GHz and 8 GB RAM under Windows 7 (64 bit). We set a time limit of 60 seconds per benchmark. Setting a memory limit within the Cygwin environment did not work properly. Therefore, we omitted such a limit.

The benchmarks are handled as BMC problems. We limited the number of BMC unwindings to 51. The results are shown in Figure 4. The diagram depicts the number of solved benchmarks together with the run time needed per benchmark. In the table, the column SAT contains the number of solved satisfiable instances while the columns U51 and U∞ show the number of unsatisfiable instances (U51: unsatisfiable until depth 51, U∞: unsatisfiable for all depths proved with k-induction or Craig interpolation). The column S+U contains the accumulated number of solved benchmarks (SAT + U51 + U∞) while TO and MO show the number of aborted benchmarks due to timeout and memout.

The first two lines of the table in Figure 4 compare both solvers in a BMC-only setting. SMI-CBMC (0) finds 184 more satisfiable instances compared to SMI-iSAT3 (4)*. This number indicates that there are some satisfiable benchmarks which are still demanding for our ICP-based approach (e. g. due to syntactic constellations in cyclic deduction chains which are currently not recognized by global-ICP). Therefore, the number of solved satisfiable instances is nearly the same for SMI-iSAT3 (4)* and SMI-iSAT3 (4). On the other hand SMI-iSAT3 (4)* solves 457 more U51 instances compared to SMI-CBMC (0). This shows that SMI-CBMC (0) suffers from its bit-blasting technique when it is required to unwind the transition relation many times.

While CBMC uses k-induction in order to detect dead-code, iSAT3 uses Craig interpolation. The last five lines of the table in Figure 4 contain the results for this setting. Similar to the results in Figure 3, all proposed techniques increase the number of solved instances of SMI-iSAT3. For the SMI-benchmarks global-ICP boosts the performance for satisfiable as well as unsatisfiable instances. When counting the number of solved satisfiable instances, SMI-iSAT3 (4) and SMI-CBMC (1) perform equally well. In the context of dead-code detection the number of solved U∞ instances is of special interest. Here, SMI-iSAT3 (4) proves for 831 instances that the code fragment of interest is unreachable – this is an improvement of 30% compared to the 631 instances found by SMI-CBMC (1). Also the number of solved U51 instances is higher in SMI-iSAT3 (4): 172 compared to 44. Regarding the accumulated number of solved instances SMI-iSAT3 (4) solves 331 instances more than SMI-CBMC (1).

The diagram in Figure 4 shows that SMI-iSAT3 (4) has some sort of initial runtime-offset. This is due to the prototypical integration in the Cygwin environment. But even with this penalty SMI-iSAT3 (4) outperforms SMI-CBMC (1) in terms of runtime, e. g. when counting the number of instances which were solved within 3 seconds per instance, then SMI-CBMC (1) solved 5305 instances while SMI-iSAT3 (4) finished 6960.

V. Conclusion and Future Work

iSAT3 aims at solving Boolean combinations of theory atoms containing linear and non-linear arithmetic as well as transcendental functions. In this paper we presented an elegant extension of iSAT3 providing accurate reasoning for floating-point arithmetic. Furthermore, in order to support the full range of basic C data types and operations, we added ICP-contractors for bitwise integer operations as well. With global-ICP and the presented adaptions in the decision heuristics, we were able to mitigate the problem of aliasing in ICP. The results confirm the efficacy of our proposed techniques. Compared to solvers based on bit-blasting (Mathsat5 and SMI-CBMC) and compared to another ICP-based solver (FP-ACDCL) iSAT3 solved the highest number of benchmarks.

Future Work: The set of primitive constraints collected by global-ICP is small in many cases. Although we could extend the current approach in order to check for other syntactic constellations as well, this is unlikely to catch all cases. A symbiotic combination with bit-blasting could be promising, i. e. applying bit-blasting locally in the sense of an additional theory solver to this small set of primitive constraints.

Acknowledgment

This work has been partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (DFG, SFB/TR 14 AVACS) and by the Cluster of Excellence BrainLinks-BrainTools (DFG, grant number EXC 1086)

References

[1] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *TACAS 2013*.
[2] S. A. Wolfman and D. S. Weld, "Combining linear programming and satisfiability solving for resource planning," *Knowledge Eng. Review*, vol. 16, no. 1, pp. 85–99, 2001.
[3] J. G. Cleary, "Logical arithmetic," *Future Computing Systems*, vol. 2, no. 2, pp. 125–149, 1987.
[4] F. Benhamou and L. Granvilliers, "Continuous and Interval Constraints," in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, 2006, pp. 571–603.
[5] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *Journal on Satisfiability, Boolean Modeling, and Computation*, vol. 1, no. 3-4, pp. 209–236, 2007.
[6] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening, "An abstract interpretation of DPLL(T)," in *VMCAI 2013*.
[7] 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.
[8] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.
[9] G. S. Tseitin, "On the complexity of derivations in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, A. Slisenko, Ed., 1968.
[10] J. P. M. Silva and K. A. Sakallah, "Grasp - a new search algorithm for satisfiability," in *ICCAD*, 1996, pp. 220–227.
[11] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.
[12] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.
[13] K. L. McMillan, "Interpolation and sat-based model checking," in *CAV 2003*.
[14] C. Herde, "Efficient solving of large arithmetic constraint systems with complex boolean structure: proof engines for the analysis of hybrid discrete-continuous systems," Ph.D. dissertation, 2011.
[15] K. Scheibler, S. Kupferschmid, and B. Becker, "Recent improvements in the SMT solver iSAT," in *MBMV 2013*.
[16] K. Scheibler and B. Becker, "Using interval constraint propagation for pseudo-boolean constraint solving," in *FMCAD 2014*.
[17] M. Colishaw, "IEEE standard for floating-point arithmetic," pp. 1132–1138, 2008.
[18] C. Michel, "Exact projection functions for floating point number constraints," in *AI&M 2002*.
[19] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, "Extending iSAT3 with ICP-Contractors for Bitwise Integer Operations," SFB/TR 14 AVACS, Reports of SFB/TR 14 AVACS 116, 2016, iSSN: 1860-9821, http://www.avacs.org.
[20] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Successful use of incremental BMC in the automotive industry," in *FMICS 2015*.

SWAPPER: A Framework for Automatic Generation of Formula Simplifiers based on Conditional Rewrite Rules

Rohit Singh and Armando Solar-Lezama
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {rohitsingh,asolar}@csail.mit.edu

***Abstract*—This paper addresses the problem of creating simplifiers for logic formulas based on conditional term rewriting. In particular, the paper focuses on a program synthesis application where formula simplifications have been shown to have a significant impact. We show that by combining machine learning techniques with constraint-based synthesis, it is possible to synthesize a formula simplifier fully automatically from a corpus of representative problems, making it possible to create formula simplifiers tailored to specific problem domains. We demonstrate the benefits of our approach for synthesis benchmarks from the SyGuS competition and automated grading.**

I. Introduction

Formula simplification plays a key role in SMT solvers and solver-based tools. SMT solvers, for example, often use local rewrite rules to reduce the size and complexity of the problem before it is solved through some combination of abstraction refinement and theory reasoning [1], [2]. Moreover, many applications that rely on solvers often implement their own formula simplification layer to rewrite formulas before passing them to the solver [3], [4], [5].

One important motivation for tools to implement their own simplification layer is that formulas generated by a particular tool often exhibit patterns that can be exploited by a custom formula simplifier but which would not be worthwhile to exploit in a general solver. Unfortunately, writing a simplifier by hand is challenging, not only because it is difficult to come up with the simplifications and their efficient implementation, but also because some simplifications can actually make a problem harder to solve by the underlying solver. This means that producing a simplifier that actually improves solver performance often requires significant empirical analysis.

In this paper, we present SWAPPER, a framework for automatically generating a formula simplifier from a corpus of benchmark problems. The input to SWAPPER is a corpus of formulas from problems in a particular domain. Given this corpus, SWAPPER generates a formula simplifier tailored to the common recurring patterns in this corpus and empirically tuned to ensure that it actually improves solver performance.

SWAPPER operates in four phases. In the first phase (1), the system uses representative sampling to identify common repeating sub-terms in the different formulas in the corpus. In the second phase (2), these repeating sub-terms are passed to the rule synthesizer which generates conditional simplifications that can be applied to these sub-terms when certain local conditions are satisfied. These conditional simplifications are the simplification rules which in the third phase (3) must be compiled to actual C++ code that will implement these simplifications. In the fourth phase (4), SWAPPER uses auto-tuning to evaluate combinations of rules based on their empirical performance on a subset of the corpus (Training set) in an effort to identify a subset of the rules that maximizes solver performance.

In this paper, we focus SWAPPER on formulas generated by the SKETCH synthesis solver [6]. We choose SKETCH solver because it is already very efficient, and it has been shown to be faster than the most popular SMT solvers on the formulas that arise from encoding synthesis problems [7]. A major part of this performance comes from carefully tuned formula rewriting, so improving the performance of this solver is an ambitious target. The SKETCH synthesizer has been applied to a number of distinct domains which include storyboard programming [8], query extraction [9], automated grading (Autograder) [10], sketching Java programs [11], SyGus competition benchmarks [12], synthesizing optimal CNF encodings [13] and programming of line-rate switches [14]. Crucially for our purposes, we have available to us large numbers of benchmark problems that are clearly identified as coming from some of these different domains. For example, SKETCH has over a thousand benchmarks for Autograder problems obtained from student submissions to introductory programming assignments on the edX platform. For this paper, we will focus on two important domains: Autograder and SyGus competition benchmarks and present a small case study with the CNF (SAT) encodings benchmarks.

This paper makes the following key contributions:

1) We demonstrate how to automate the process of generating conditional rewrite rules specific to the common recurring patterns in formulas from a given domain.
2) We demonstrate the use of autotuning to select an optimal subset of rules and generate an efficient simplifier.
3) We evaluate our approach on multiple domains from the SKETCH synthesizer and show that the generated simplifiers reduce the synthesis times of SKETCH by 15%-60% relative to the existing SKETCH with its hand-crafted simplifier.

II. Overview of SWAPPER

SWAPPER takes as input a corpus of problems (formulas) $\{P\} \subset \mathcal{D}$ that are assumed to be from the same domain $\mathcal{D}$ and therefore share certain structural features—what it means for problems to come from a given domain is left ambiguous,

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

but the assumption underlying our work is that problems that come from the same domain will have similar structure and will respond similarly to simplifications. Given $\{P\}$, the goal of SWAPPER is to produce a formula rewriter such that for any $P \in \mathcal{D}$ from the domain, it rewrites P to $P' = rewrite(P)$, where P' is logically equivalent to P but easier to solve.

The rewriters produced by SWAPPER are term rewriters that work by making local substitutions when a known pattern is encountered in a context satisfying certain constraints. For example, the rewrite rule below indicates that when the guard predicate (*pred*) $b < d$ is satisfied, one can locally substitute the pattern on the left hand side (LHS): $or(lt(a,b), lt(a,d))$ with the smaller pattern on the right hand side (RHS).

$$or(lt(a,b), lt(a,d)) \xrightarrow{b<d} lt(a,d)$$

SWAPPER's approach is to automatically generate large collections of such rules and then compile them to an efficient rewriter for the entire formula. Making this approach work requires addressing four key challenges: (1) Choosing promising LHS patterns (2) Finding the best rewrite rule for a given LHS (3) Generating an efficient implementation that applies these rules and (4) Making sure that the rules actually improve the performance of the solver. In the rest of this section, we outline how each of the components of SWAPPER address these challenges.

A. Pattern Finding

The first phase of SWAPPER addresses the problem of identifying promising LHS patterns for rewrite rules. In order to do this, the pattern finder takes as input a corpus of formulas $\{P\}$ and uses a representative sampling scheme (explained in Sec. III) to find frequently recurring patterns in the formulas from the corpus. The idea is that rewrite rules that target patterns that occur frequently in the corpus are more likely to have a high impact in the complexity of the overall formula.

In addition to identifying high frequency patterns, this phase also identifies properties that tend to hold when those patterns occur. Specifically, we assume that the rewriter has access to a function $static(a)$ that for any sub-term a of a larger formula P, can determine an over-approximation of the range of values that a can take when the free variables in P are assigned values from their respective ranges. In the rewriter that is currently part of the SKETCH solver, for example, this function is implemented by performing abstract interpretation over the formula.

For a given pattern, the pattern finder keeps a set of contexts, where each context corresponds to the information captured by $static$ on an occurrence of that pattern in the corpus. For example, in the case of the rewrite rule above, this phase may discover that the pattern $or(lt(a,b), lt(a,d))$ is very common, so finding a simplification rule for this pattern would be advantageous. Pattern finding may also discover that this pattern often occurs in a context where a rewriter can prove that $static(b) = (-\infty, 0]$ and $static(d) = (0, \infty)$. Using such context information, SWAPPER can learn rules that make more aggressive simplifications based on the strong assumptions.

Note that this problem is similar in essence to the Motif discovery problem [15], famous for its application in DNA fingerprinting [16]. However, existing techniques used for solving Motif discovery problem are not directly usable in SWAPPER because they lead to loss of information required for the next phase in SWAPPER (See Sec. VIII for more details).

B. Rule Synthesis

Once SWAPPER identifies promising LHS patterns for rules, together with properties of their free variables that can be assumed to hold in the contexts where these LHS patterns appear (e.g. $b \leq 0$ and $d > 0$ in the example above), the next challenge is to synthesize the rewrite rules.

A *conditional rewrite rule* has the form:

$$LHS(x) \xrightarrow{pred(x)} RHS(x),$$

where x is a vector of variables, LHS and RHS are expressions that include variables in x as free variables and *pred* is a guard predicate defined over the same free variables and drawn from a restricted grammar. The triple must satisfy the following constraint: $\forall x.\ pred(x) \implies (LHS(x) = RHS(x))$

The goal of rule synthesis is therefore twofold: (1) to synthesize predicates $pred(x)$ that can be expected to hold on at least one context identified by pattern finding for the LHS pattern, and (2) to synthesize for each of these candidate predicates, an optimal RHS for which the constraint above holds. At this stage, optimality is defined simply in terms of the size of the RHS, since it is difficult to predict the effect that a transformation will have on solution time. As we will see later, optimality in terms of size does not guarantee optimality in terms of solution time but at this stage in the process our goal is simply to identify potentially good rewrite rules. We formulate this as a Syntax-Guided Synthesis Problem [12] and solve it using the SKETCH synthesis tool [6]. The details of how the rules are synthesized are given in Sec. IV.

C. Code generation

The next challenge for SWAPPER is to generate efficient C++ code for pattern matching and replacement. To achieve this, SWAPPER builds upon the ideas of term rewrite systems like Stratego/XT [17] and GrGEN.NET [18], and performs some optimizations detailed in Sec. V. The role of this phase is similar to what the Alive system [19] does when generating peephole optimizers for LLVM.

One important optimization at this stage is rule generalization. For example, pattern finding may have discovered that the pattern $or(lt(plus(x,y), b), lt(a,d))$ was frequent, and the rule synthesis phase may have discovered the rewrite rule:

$$or(lt(plus(x,y), b), lt(a,d)) \xrightarrow{b<d} lt(plus(x,y), d)$$

Rule generalization, would identify that $plus(x,y)$ is unchanged by the rewrite rule, so the rule could be made more general by replacing $plus(x,y)$ in both patterns with a free variable to arrive at the rule shown earlier. SWAPPER needs to verify that the generalization preserves the correctness of the rule in order to avoid generating incorrect transformations. It is important to note that rule generalization doesn't affect predicates because those have already been minimized during the Rule Synthesis step.

D. Autotuning

There are two motivations for autotuning the set of obtained rewrite rules: (1) there is a trade-off between the strength of the predicate and the reduction that can be achieved by a rule: rules with weak predicates are easier to match than rules with strong predicates, but rules with strong predicates can offer more aggressive simplification, and (2) the rules that give the most aggressive size reduction are not necessarily the best ones; for example, a rule may replace a very large LHS pattern with a small RHS but in doing so it may prevent other rules from being applied, resulting in a formula that is larger than the formula obtained without the rewriting.

For these reasons, writing optimal simplifiers based on rewrite rules is a challenging task even for human experts, which motivates our approach of using synthesis and empirical autotuning methods to automatically discover optimal sets of conditional rewrite rules. To this end SWAPPER uses OpenTuner [20], a machine learning based off-the-shelf autotuner and provides an optimization function that emits the actual performance of the solver. This phase is comparable to algorithm configuration [21], [22], which has been used for tuning parameters for SAT solvers [23]. But, unlike algorithm configuration, the optimization function in SWAPPER is based on choices of subsets and permutations of rewrite rules, and is provided to OpenTuner [20] as a black-box.

We now describe each of these phases in more detail.

III. PATTERN FINDING: SAMPLING BASED CLUSTERING

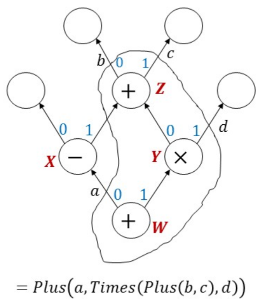
We first describe a few key features of the SKETCH synthesis system [6] which will serve both as a component and as a target for SWAPPER.

SKETCH synthesis system: SKETCH is an open source system for synthesis from partial programs. A partial program (also called a sketch) is a program with holes, where the potential code to complete the holes is drawn from a finite set of candidates, often defined as a set of expressions or a grammar. Given a partial program with a set of assertions, the SKETCH synthesizer finds a completion for the holes that satisfies the assertions for all inputs from a given input space. SKETCH uses symbolic execution to derive a predicate $P(x,c)$ that encodes the requirement that given a choice c for how to complete the program, the program should be correct under all inputs x i.e. $\exists c \forall x P(x,c)$.

SKETCH , like most solvers, represents formulas as Directed Acyclic Graphs (DAGs) in order to exploit sharing of subterms within a formula. The formulas in SKETCH involve boolean combinations of formulas involving the Theory of Arrays and Non-linear Integer Arithmetic. Because SKETCH has to solve an exists-forall ($\exists\forall$) problem, the formulas distinguish between existentially and universally quantified variables: *inputs* and *controls* respectively. The formula simplification pass that is the subject of this paper is applied to this predicate $P(x,c)$ as it is constructed and before the predicate is solved in an abstraction refinement loop based on counterexample guided inductive synthesis (CEGIS) [6].

Probabilistic Pattern Sampling: A pattern in the context of SWAPPER is an expression tree that has free variables as leafs. Our goal for the pattern finding phase is to generate a representative sample of patterns S from a corpus of DAGs $\{P\}$. In order to formalize the notion of a representative sample of patterns, we first need to define a few terms.

Consider a formula P represented as a DAG. We can define a *rooted sub-graph* of P as a sub-graph of P such that all its nodes can be reached from a selected root node in the sub-graph. A rooted sub-graph can be mapped to a pattern, where the edges at the boundary of the sub-graph correspond to the free variables in the pattern. This relationship is illustrated in Fig. 1, where we see a graph for a problem P where a rooted sub-graph has been selected, and we see the pattern that corresponds to that sub-graph. Note that a single formula P may have many sub-graphs that all correspond to the same pattern.



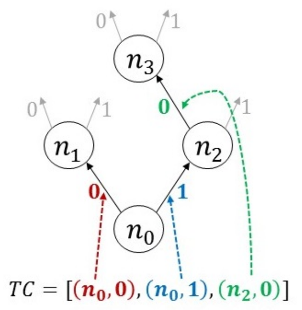


Fig. 1. Pattern from a rooted sub-graph Fig. 2. Example Tree Construction

Given a constant K, let the set $Sub_K(P)$ be the set of all rooted sub-graphs of size K of P. Given these definitions, we are now ready to state the problem of representative sampling patterns from a corpus.

Definition 1: *Representative pattern sampling* Given a corpus of problems $\{P_i\}$ and a size K, a pattern sampling approach is said to be representative if it is equivalent to sampling uniformly from the set $\bigcup_i Sub_K(P_i)$, and then mapping each of the resulting sub-graphs to their corresponding pattern.

The key problem is then how to uniformly sample the space of rooted sub-graphs in a collection of formulas. In order to describe the algorithm for this, we first build the notion of a *Tree Construction*. The formal definition is given below, but intuitively, a *Tree Construction* (TC) is a recipe for generating a tree.

Definition 2: A TC for a tree of size $K \geq 2$ and arity $\delta \geq 1$ is a list of $K-1$ pairs $[(s_i, t_i)]_{i=0}^{i=K-2}$, where each pair represents an edge that is being added to the tree. Each edge is identified by its source node s_i (which should already be in the tree) and by the index $t_i < \delta$ of the edge that is added to that source node. A TC cannot have repeated edges, so each edge adds a new node to the tree. Therefore, if n_0 is the original root node in the tree, and n_i is the node added by the i^{th} edge, then $s_i \in \{n_0, n_1, \ldots, n_i\}$ for all $i \leq K-2$.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

We use *Tree* (τ) to represent the tree constructed from a TC τ in this manner.

For example, if we assume binary trees ($\delta = 2$), the following would be a valid tree construction of size 4: $\tau = [(n_0, 0), (n_0, 1), (n_2, 0)]$, and would construct the tree *Tree* (τ) as shown in Fig. 2.

Assuming trees of degree δ, it is relatively easy to uniformly sample the space of Tree Constructions for trees of size K. The idea is to keep track of the boundary of the tree (all the possible edges that have not been expanded) and to grow the tree by sampling uniformly at random from this boundary. The exact algorithm is shown below.

Algorithm 1: Uniform Sampling for Tree Constructions

```
input : $K \geq 2$ : Size of the TC to be found
        $\delta \geq 1$ : Bound on number of parents of any node
        $\{n_0, n_1, ..., n_{K-1}\}$ : Set of $K$ node symbols
output: $\tau$ : A Tree Construction of size $K$
1 $\tau \leftarrow List()$
2 $\Lambda \leftarrow List()$                 ▷ maintains adjacent/boundary edges
3 foreach $i \in [0, 1, \ldots, K-2]$ do
4     foreach $j \in [0, 1, \ldots, \delta - 1]$ do
5         $\Lambda$.append$\big((n_i, j)\big)$          ▷ adds $\delta$ edges to boundary
6     $(s, t) \leftarrow$ sample $(\Lambda)$        ▷ boundary $|\Lambda| = \big((i+1)\delta\big) - i$
7     $\tau$.append$\big((s, t)\big)$
8     $\Lambda$.remove$\big((s, t)\big)$          ▷ removes an edge from boundary
```

It is easy to see that any TC of size K will be sampled by Algorithm 1 with a probability of $\Pi_{0 \leq i < K-1} \frac{1}{((i+1)\delta) - i}$ because $|\Lambda| = \big((i+1)\delta\big) - i$ at the i^{th} step and we sample uniformly from Λ for each $0 \leq i < K - 1$. This probability is independent of the TC being considered and hence, every TC is equally likely to be sampled by Algorithm 1.

Now, we are going to define an algorithm for representative pattern sampling that uses our ability to uniformly sample from the space of TCs (denoted by **TC**). The strategy will be as follows, given a corpus of formulas $\{P_i\}$, we are going to define a subset of the product space $\bigcup_i \text{nodes}\big(P_i\big) \times \textbf{TC}$, which we will call *Canonical*, we then define a mapping μ from *Canonical* to $\bigcup_i Sub_K(P_i)$, and we are going to show that mapping is one-to-one and onto. Finally, we will use rejection sampling [24] to sample from *Canonical* uniformly and then apply μ to in turn, sample uniformly from $\bigcup_i Sub_K(P_i)$. In the rest of the section, we define the *Canonical* set, the function μ and show that it is bijective.

Definition 3: *Canonical* set: Given a corpus of formulas $\{P_i\}$, we define *Canonical* $\subset \bigcup_i \text{nodes}\big(P_i\big) \times \textbf{TC}$ as the set of tuple-pairs $\big(\eta, \tau\big)$ with $\eta \in \text{nodes}\,(P_i)$ for some i such that:

1) It is possible to follow the TC τ at node η and construct a *rooted sub-graph* Q of P_i rooted at η i.e. there is a graph homomorphism h : *Tree* $(\tau) \mapsto P_i$ such that $h(n_0) = \eta$ and Q is the rooted sub-graph of P_i formed by the nodes corresponding to the image of h in P_i.
2) The ordering of nodes in TC τ is the same as the (unique) Breadth First Search (BFS) ordering of nodes in Q.

For example, In Fig. 1, for $(\eta, \tau) = (W, [(n_0, 1), (n_1, 0)])$ by following TC τ we obtain the homomorphism h that maps $h(n_0) = W, h(n_1) = Y, h(n_2) = Z$ and the rooted sub-graph Q corresponds to the enclosed region with nodes W, Y, Z. Also, the ordering of nodes given by τ matches the BFS ordering of nodes induced by h in Q, hence, $(\eta, \tau) \in$ *Canonical*. Note that $(W, [(n_0, 1), (n_0, 0)]) \notin$ *Canonical* because it induces the ordering W, Y, Z which is not in the BFS order W, X, Y.

Given the way we defined *Canonical*, constructing the mapping μ : *Canonical* $\mapsto \bigcup_i Sub_K(P_i)$ is straightforward: $\mu\big(\,(\eta, \tau)\,\big) = Q$ where Q is the rooted subgraph obtained by following TC τ starting at node η (Def. 3). To show that μ is onto, we consider a rooted subgraph S of P_i for some i, since any node of a rooted subgraph can be reached from the root η_S, we can construct the BFS tree BFS(S) of S and the corresponding TC τ_S that is the recipe for constructing BFS(S) so that $\mu\,((\eta_S, \tau_S)) = S$. To show that μ is one-to-one, we observe that for two $(\eta_1, \tau_1), (\eta_2, \tau_2)$ to map to the same rooted subgraph S, the corresponding trees should be the same (the BFS tree of S) and the ordering of nodes in τ_1, τ_2 should be the same as well (corresponding to BFS order of S), which would mean $\tau_1 = \tau_2$ and $\eta_1 = \eta_2$.

Clustering patterns: While grouping patterns together into clusters, SWAPPER considers the following: **(1) Pattern Expression**: SWAPPER builds a string of the expression represented by the pattern. The free variables in this pattern are numbered in the BFS order and the operands for commutative operations are ordered lexicographically to group together patterns when they are equivalent because of commutativity. **(2)** $static$ **function values from the benchmark formulas** : The range of values inferred by the solver (See **II-A**) for each free variable in the pattern are collected and represented as a mapping of names of the free variables to their ranges. Note that the same pattern can occur with different $static$ function values, these values are appended to one *Pattern Expression*.

Stopping Criterion: SWAPPER samples until the total number of patterns with probability of occurrence greater than a threshold ϵ converges i.e. the next M samples do not change the number of such patterns. Both ϵ and M are inputs to SWAPPER. In our experiments, we started with $M = 10,000$ and $\epsilon = 0.05$ and then increased M and decreased ϵ gradually in steps of $10,000$ and 0.01 respectively, and, sampled again until we didn't find any new patterns ($M = 50,000$ and $\epsilon = 0.02$). SWAPPER samples patterns of sizes $2, 3, 4, ...$ and stops after sampling patterns of size 7.

IV. Rule Synthesis: Syntax-Guided Synthesis

In this phase, SWAPPER finds corresponding rewrite rules for the set of patterns $\{Q\}$ obtained from the Pattern Finding phase. For each pattern $Q(x)$, the Pattern Finding phase also collects a set of $static$ range values (**II-A**) that can be valid for the free variables x in that pattern when it occurs as a rooted subgraph in the benchmark DAGs. We use the notation $assume(x)$ to represent a predicate over the free variables x

that evaluates to true when each free variable is in the range values given by $static$ function.

A. Problem Formulation

SWAPPER needs to find correct rewrite rules (**II-B**) for a given LHS pattern. Additionally, we want to avoid rules with predicates that will never hold in practice, so we focus on rules with predicates that are implied by the assumptions given by $static$ function obtained from the Pattern Finding Phase.

Problem 1: Given a pattern $LHS(x)$, predicate $assume(x)$ discovered by the solver for a given occurrence of the LHS pattern, and grammars for $pred(x)$ and $RHS(x)$, find suitable candidates for $pred(x)$ and $RHS(x)$ which satisfy the following constraints:

1) $\forall x : assume(x) \implies pred(x)$
2) $\forall x :\ if\ (pred(x))\ then\ (LHS(x) == RHS(x))$
3) size(RHS) < size(LHS), where size(Q) is the number of nodes in the pattern Q
4) $pred(x)$ is the weakest predicate (most permissive) in the predicate grammar that satisfies previous constraints

The space of predicates: For *pred* SWAPPER employs a simple Boolean expression generator that considers conjunctions of equalities and inequalities among variables: $pred(x) \rightarrow boolExpr(x)$ and $boolExpr(x)$ | $boolExpr(x)$, $boolExpr(x) \rightarrow x_i$ binop x_j | unop x_i where binop $\in \{<, >, =, \neq, \leq, \geq\}$, unop $\in \{\epsilon, \neg\}$.

These predicates are inspired by existing predicates present in the rules in the Sketch's hand-crafted simplifier and are easier to check statically than more complicated predicates.

Grammar for RHS: The template for RHS simulates the computation of a function using temporary variables. This computation can be naturally interpreted as a pattern. Essential grammar for the generator for RHS is shown here:

$$\begin{array}{lll} RHS(x) \equiv & let & t_1 = \textbf{simpleOp}(x); \\ & & \ldots \\ & & t_k = \textbf{simpleOp}(x, t_1, ..., t_{k-1}); \\ & in\ t_k & \end{array}$$

where **simpleOp** represents a single operation node (e.g. AND, PLUS etc) with its operands being selected from the arguments. For example, the expression $(a + b) \times c$ can be represented as: let $t_1 = Plus(a, b); t_2 = Times(t_1, c);$ in t_2. We put a strict upper bound on k as the number of nodes in the LHS for which the RHS is being searched for.

B. Correctness Constraint

Setting apart constraint 4), Problem 1 can be formulated as a classic syntax-guided synthesis [25] problem:

$$\begin{aligned} \exists c_p c_r\ \forall x\quad & \big(assume(x) \implies pred(x, c_p)\big) \\ & \wedge pred(x, c_p) \implies \big(LHS(x) = RHS(x, c_r)\big) \end{aligned}$$

where c_p and c_r are the choices the solver needs to make to get a concrete $pred(x)$ and $RHS(x)$. Specifically, these are the choices of: (i) when to expand the grammar or when to use a terminal, and (ii) which subset of inputs to choose for a particular operation as operands. We will enforce the constraint 4) on top of solutions to the synthesis problem.

We explored two different techniques for synthesizing such rewrite rules: (1) Symbolic SKETCH based synthesis of rules, and (2) Enumerative search with heuristics. SWAPPER uses a hybrid approach to get the best of the both aforementioned techniques: scalability and exhaustiveness. We describe the hybrid technique briefly.

C. Hybrid Enumerative/SKETCH-based synthesis

SWAPPER breaks the synthesis problem into two parts:

(1) Constraints and optimizations on predicates: SWAPPER uses the enumerative approach, generates all possible candidate predicates from the specification grammar and checks for their validity based on collected assumptions $assume(x)$. It prunes the space of predicates by handling symmetries and avoiding extra work based on the result of the underlying synthesis problem (explained below). For example, if $x = (a, b)$, $assume(x) = (1 < a < 10) \wedge (b = 0)$ then some of the valid predicates will be $\{a > b, \neg b, a \geq b, a \neq b\}$.

(2) Synthesis of RHS: SWAPPER hard-codes a predicate and realizes the RHS synthesis problem in SKETCH using the **generator** and **minimize** features ([26] [27]) of the SKETCH language. In SKETCH, **generator**s are used to define the template for a grammar as a recursive program (e.g. RHS), and **minimize** keyword is used to find the smallest possible value of a computed variable in SKETCH language (e.g. size of $RHS(x)$ for a fixed $pred(x)$).

Optimization on the space of predicates: SWAPPER computes the relationship between predicates based on whether one implies the other or not for all values of the free variables e.g. $(a < b)$ implies $a \neq b$ but doesn't imply $a > b$. SWAPPER iteratively finds RHS for the least applicable predicates at any given stage. When there is no possible simplification rule for a least applicable predicate then SWAPPER can prune out all predicates implied by it because there cannot exist a rule with a more applicable predicate. This helps SWAPPER reduce overall time for the rule synthesis step.

This hybrid technique has the benefit of being able to exhaustively search for rules of big sizes while making the core synthesis problem faster (fewer constraints) and highly parallelizable (multiple SKETCH instances with different predicates are run in parallel). Note that SKETCH does synthesis of rules guaranteeing their correctness for large but bounded values of $\forall$ quantified variables (inputs), hence, we fully verify the generated rules with z3 [1] as well by expressing them as SMT constraints before using them for code generation.

V. EFFICIENT SIMPLIFIER CODE GENERATION

The code generation phase in SWAPPER implements two important optimizations. The first is *rule generalization.* As described earlier in **II-C**, the goal of rule generalization is to make the rule more applicable by eliminating redundant nodes from the LHS and RHS patterns. The second optimization is to reduce the cost of pattern matching by identifying common substructures in different patterns and avoiding redundant checks for those patterns, similar to how a compiler for a functional language would optimize pattern matching [28].

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Additionally, the code generator will ensure that the pattern matching code can identify DAGs that are equivalent to a given LHS because of commutativity. Overall, the generated code is more efficient than the hand-written optimizer because the automatic code generator can optimize pattern matching without regard to the impact of optimization on the readability of the generated code.

VI. Auto-tuning Rules

SWAPPER uses OpenTuner [20] to auto-tune the set of rules according to a performance metric (based on time, memory, size of DAGs etc). OpenTuner uses an ensemble of disparate search techniques and quickly builds a model for the behavior of the optimization function treating it as a black box.

Optimization Problem Setup: SWAPPER specifies the set of all rules to the tuner and creates the following two configuration parameters: (1) A permutation parameter: for deciding the order in which the rules will be checked. (2) Total number of rules to be used.

The optimization function (**fopt**) takes as input a set of rules and returns a real number. This number corresponds to performance improvement of SKETCH on the benchmarks after rewriting them using the generated simplifier (Sec. V). The auto-tuner tries to maximize this reward by trying out various subsets and orderings of rules provided to it as input while learning a model of dependence of **fopt** on the rules.

VII. Experiments

In order to test the effectiveness of our system, we focus on three questions: (1) Can SWAPPER generate good simplifiers in reasonable amounts of time and with low cost of computational power? (2) How do the simplifiers generated by SWAPPER affect SMT solving performance of SKETCH relative to the hand written simplifier in SKETCH? (3) How domain specific are the simplifiers generated by SWAPPER?

For evaluation of SWAPPER on SKETCH domains, we compared the following three simplifiers:

1) Hand-crafted: This is the default simplifier in SKETCH that has been built over a span of eight years. It comprises of simplifications based on (a) rewrite rules that can be expressed in our framework (**II-B**) (b) constant propagation (c) structure hashing [29] and (d) a few other complex simplifications that cannot be expressed in our framework
2) Baseline: This disables the rewrite rules that can be expressed in our framework from the Hand-crafted simplifier but applies the rest of the simplifications (b)-(d).
3) Auto-generated: This incorporates the SWAPPER's auto-generated rewrite rules on top of the Baseline simplifier.

Now, we elaborate on the details of the experiments.

A. Domains and Benchmarks:

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366

We investigated benchmarks from two domains of SKETCH applications. Sygus corresponds to the SyGus competition benchmarks translated from SyGus format to Sketch specification [12] and AutoGrader ones are obtained from student's assignment submissions in the introduction to programming online edX course [10]. For each of these domains we picked suitable candidates for SWAPPER's application by (1) eliminating those benchmarks which did not have more than 5000 terms in the formula represented by their DAGs and those which took less than 5 seconds to solve - so that there's enough patterns and opportunity for improvement (2) removing those which took more than 5 minutes to solve - this was done to keep SWAPPER's running time reasonable because we need to run each benchmark multiple times during auto-tuning phase. Using these cutoffs, the total number of usable benchmarks for AutoGrader domain were reduced from 2404 to 45 and for Sygus from 309 to 22.

B. Synthesis Time and Costs are Realistic:

To generate a simplifier, SWAPPER employed a private cluster running Openstack as the infrastructure for parallelized computations with parallelisms of 20-40 on two virtual machines emulating 24 cores, 32GB RAM of processing power each. A worst case estimate of the cost of computation done by SWAPPER based on our experiments using the Amazon Web Services [30] estimator is presented below. SWAPPER can be used to automatically synthesize a simplifier for a very reasonable cost (less than \$50)

Domain	Pattern Finding	Rule Synthesis	Auto-Tuning	Total Time (hours)	Cost
AutoGrader	3 hours	1 hour $\times 5$	0.08×150	20	\$22
Sygus	2 hours	1 hour $\times 5$	0.1×150	22	\$24

C. SWAPPER Performance

To test the performance of SWAPPER on SKETCH benchmarks from a particular domain, we divided the corpus into three disjoint sets randomly (SEARCH, TRAIN, TEST). The SEARCH set was used to find patterns in the domain and TRAIN set was used in the auto-tuning phase for evaluation. And finally, TEST set was used to empirically confirm that the generated simplifier is indeed optimal for the domain. Moreover, we used 2-fold cross validation to ensure that there was no over-fitting on TRAIN set. We achieved this by exchanging TRAIN and TEST sets and auto-tuning with the TEST set instead of the TRAIN set. We obtained similar performing simplifiers as a result and verified that there was no over-fitting.

We implemented the evaluation of benchmarks in SWAPPER as a python script that takes a set of DAGs as input, runs SKETCH on each of them multiple times (set to 5 in our experiments) and obtain the quartile values(3 points that cut data into 4 equal parts including the median) for time taken. In the graphs presenting SKETCH solving times, we show the upper and lower quartiles around the median with dotted or shaded lines of the same color as the thick line depicting the median time. Also, note that we will not consider simplification time in these experiments because of it being a one-time negligible (a fraction of a second) time-step as compared to further SKETCH solving.

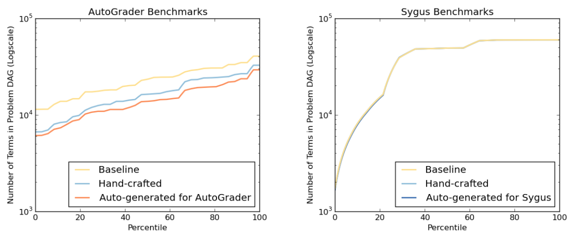


Fig. 3. Change in sizes with different simplifiers

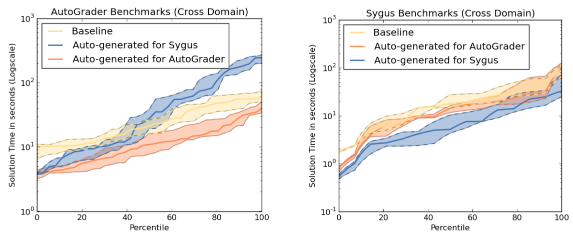


Fig. 5. Domain specificity of Auto-generated simplifiers: Time distribution

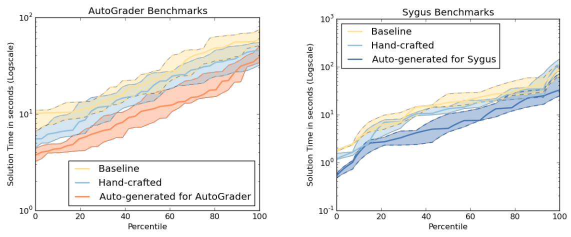


Fig. 4. Median running time percentiles with Quartile confidence intervals

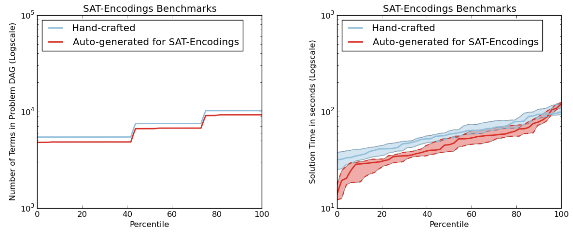


Fig. 6. SAT-Encodings domain case study

We obtained 301 rules for AutoGrader domain and 105 rules for Sygus domain. The optimal simplifier for AutoGrader used 135 of the rules and the one for Sygus used 65 rules.

Benefits over the existing simplifier in SKETCH: Auto-generated simplifier reduced the size of the problem DAGs by 13.8% (AutoGrader) and 1.1%(Sygus) on average as compared to the size of DAGs obtained after running Hand-crafted simplifier (Figure 3). On DAGs obtained after using Auto-generated simplifier on average, SKETCH solver performed better than on those obtained by using Hand-crafted simplifier (Figure 4): (1) Auto-generated simplifier made SKETCH run faster on 80% of the AutoGrader benchmarks and 90% of the Sygus benchmarks (2) The average times taken by SKETCH to solve a benchmark simplified using Auto-generated simplifier were 13s (AutoGrader) and 8s (Sygus) as compared to 20s and 21s respectively for the Hand-crafted simplifier. Figures 3 and 4 show distribution of sizes and times for SKETCH solving after applying all three simplifiers with percentiles on the x-axis. It clearly shows the consistent improvement in performance by applying the Auto-generated simplifier. Note that Sygus benchmarks are written at a level of abstraction that is very close to the DAGs in Sketch and hence there aren't many opportunities for size reduction for these problems.

We found that there are two reasons why the auto-generated rules improved upon the hand-crafted rules: (1) The synthesizer discovered rules with large LHS patterns that were not present in the hand-crafted optimizer. (2) The autotuner was able to discover that some rules caused a performance degradation even when they reduced the formula size.

Benefits over the unoptimized version of SKETCH: Auto-generated simplifier reduced the size of the problem DAGs by 38.6% (AutoGrader) and 1.6%(Sygus) on average (Figure 3). Application of Auto-generated simplifier results into huge improvements in running times for SKETCH solver on both AutoGrader and Sygus benchmarks as compared to application of Baseline: the average time of solving a benchmark was reduced from 27.5s (AutoGrader) and 22s (Sygus) to 13s and 8s respectively (Figure 4).

Domain specificity: We took the Auto-generated simplifier obtained from one domain and used it to simplify benchmarks from the other domain and then ran SKETCH on the simplified benchmarks. Application of Auto-generated simplifier obtained from Sygus increased the SKETCH running times drastically on a few AutoGrader benchmarks when compared to the application of Baseline simplifier (Figure 5), and, resulted into SKETCH running slower than after application of Auto-generated simplifier obtained from AutoGrader domain. Application of Auto-generated simplifier generated for Auto-Grader domain reduced the running times of SKETCH solver on average as compared to the Baseline on Sygus benchmarks but the times were still far away from the performance gains obtained by application of Auto-generated simplifier generated for the Sygus domain (Figure 5). This validates our hypothesis of these generated simplifiers to be very domain specific.

D. SAT Encodings Domain We performed an additional case study using SWAPPER on problems generated during synthesizing optimal CNF (SAT) encodings [13]. We used a subset of 70 benchmarks with solution times between 30s and 100s and divided it randomly into SEARCH, TRAIN, TEST sets with $21, 22, 27$ benchmarks respectively. We compare the Hand-crafted simplifier against the Auto-generated simplifier for this domain in Fig. 6. SWAPPER generated 117 rules and, on average, the SKETCH solving time reduced from 58.8s to 51.1s, the DAG sizes were reduced by 11%.

VIII. RELATED WORK

A pre-processing step in constraint solvers and solver-based tools (like Z3, Boolector [2], SKETCH etc) is an essential one and term rewriting has been extensively used as a part this pre-processing step [31], [4], [5], [3]. These pre-processing steps are very important and can have a significant impact on performance.

Each part of our framework solves an independent problem and is different from the state of the art, specialized for our purposes. A recent paper introducing Alive [19], a domain specific language for specifying, verifying, and compiling peephole optimizations in LLVM is the closest to our framework as a whole. Their rewrite rules are guarded by a predicate, they use static analyses to find the validity of those guards, they verify the rules and then compile them to efficient C++ code for rewriting LLVM code: all similar to our phases. However, their system is targeted towards the compilers community and relies upon the developers to discover and specify rewrite rules. Our work is targeted towards the solver community and automatically synthesizes the rewrite rules from benchmark problems of a given domain.

In the context of Motif discovery problem [15] (finding recurrent sub-graphs) recently we have seen some attempts to use machine learning [32] and distributed algorithms [33] to compute the Motifs as quickly as possible. Our DAGs, on the other hand, have labeled nodes and our motifs have to account for symmetries due to commutative nodes, which makes direct translation to Motif discovery problem more difficult.

In the superoptimization community, people explore all possible equivalent programs and find the most optimal one. One could view SWAPPER as a superoptimizer for formula simplifiers. Superoptimizing an individual formula will be too expensive, but [34] came up with the idea of packaging the superoptimization into multiple rewrite rules similar to what we are doing here except in the context of programs. Although it looks similar in spirit to our work, there are a few differences. Most importantly, [34] uses enumeration of potential candidates for optimized instruction sequences and then checks if it is indeed most optimal. Whereas, we use a hybrid approach that primarily relies on constraint based synthesis for generating the rules, which offers a possibility of specifying a structured grammar for the functions.

The third phase in SWAPPER automatically generates simplifier's code is similar to a term or graph rewrite system like Stratego/XT [17] or GrGEN.NET [18]. They offer declarative languages for graph modeling, pattern matching, and rewriting. Both the tools generate efficient code for program/graph transformation based on rule control logic provided by the user. We build upon their ideas and develop our own compiler because we already had an existing framework for simplification (SKETCHSimplifier). Our strategy is comparable with LALR parser generation [35] where the next look-ahead symbol helps decide which rule to use.

Acknowledgment: We thank the DARPA MUSE grant FA8750-14-2-0242 for supporting this project.

REFERENCES

[1] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," ser. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 337–340.
[2] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," ser. TACAS '09, 2009, pp. 174–177.
[3] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *PLDI '09*, 2009, pp. 363–374.
[4] A. Cheung, A. Solar-Lezama, and S. Madden, "Partial replay of long-running applications," ser. ESEC/FSE '11. ACM, 2011, pp. 135–145.
[5] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Conference OSDI '08*.
[6] A. Solar-Lezama, "The sketching approach to program synthesis," in *APLAS*, 2009, pp. 4–13.
[7] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama, "Type assisted synthesis of recursive transformers on algebraic data types," *CoRR*, vol. abs/1507.05527, 2015.
[8] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," 2011.
[9] A. Cheung, A. Solar-Lezama, and S. Madden, "Inferring sql queries using program synthesis," *CoRR*, vol. abs/1208.2013, 2012.
[10] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, pp. 15–26, Jun. 2013.
[11] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama, "Jsketch: sketching for java," E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015.
[12] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Results and analysis of sygus-comp'15," 2015, pp. 3–26.
[13] J. P. Inala, R. Singh, and A. Solar-Lezama, "Synthesis of domain specific CNF encoders for bit-vector solvers," in *SAT 2016*, 2016.
[14] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown, "Packet transactions: A programming model for data-plane algorithms at hardware speed," *CoRR*, vol. abs/1512.05023, 2015.
[15] G. K. K. Sandve and F. Drabløs, "A survey of motif discovery methods in an integrated framework." *Biology direct*, Apr. 2006.
[16] N. C. Jones and P. A. Pevzner, *An Introduction to Bioinformatics Algorithms (CMB)*. MIT Press, Aug. 2004.
[17] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 52–70, 2008.
[18] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski, "GrGen: A Fast SPO-Based Graph Rewriting Tool," pp. 383 – 397, 2006.
[19] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," *SIGPLAN Not.*, vol. 50, no. 6, pp. 22–32, Jun. 2015.
[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe, "Opentuner: an extensible framework for program autotuning," in *PACT '14*.
[21] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Int. Res.*, 2009.
[22] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *CP '09*.
[23] F. Hutter, M. T. Lindauer, A. Balint, S. Bayless, H. H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *CoRR*, vol. abs/1505.01221, 2015.
[24] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering*, 3rd ed. Pearson/Prentice Hall, 2008.
[25] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa *et al.*, "Syntax-guided synthesis." IEEE, 2013, pp. 1–8.
[26] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, Berkeley, CA, USA, 2008, aAI3353225.
[27] R. Singh, R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama, "Modular synthesis of sketches using models," in *VMCAI 2014*.
[28] M. Pettersson, "A term pattern-match compiler inspired by finite automata theory," in *CC' 92*. Springer-Verlag, 1992, pp. 258–270.
[29] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis." ACM, 2006.
[30] "Amazon Web Services," Online. [Online]. Available: http://aws.amazon.com/
[31] N. B. Leonardo de Moura, "Smt: Techniques, hurdles, applications," *SAT/SMT Summer School, MIT*, 2011. [Online]. Available: http://research.microsoft.com/en-us/um/people/leonardo/mit2011.pdf
[32] M. A. Kon, Y. Fan, D. Holloway, and C. DeLisi, "Svmotif: A machine learning motif algorithm," ser. ICMLA '07, pp. 573–580.
[33] Y. Liu, B. Schmidt, and D. L. Maskell, "An ultrafast scalable many-core motif discovery algorithm for multiple gpus," ser. IPDPSW '11.
[34] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," ser. ASPLOS XII. ACM, 2006, pp. 394–403.
[35] R. N. Horspool, "Incremental generation of lr parsers," *Computer languages*, vol. 15, pp. 205–233, 1989.

Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions

Ermenegildo Tomasco*, Truc L. Nguyen*, Omar Inverso†, Bernd Fischer‡, Salvatore La Torre§ and Gennaro Parlato*
*{et1m11,tnl2g10,gennaro}@ecs.soton.ac.uk, Electronics and Computer Science, University of Southampton, UK
†omar.inverso@gssi.infn.it, Gran Sasso Science Institute, L'Aquila, Italy
‡bfischer@cs.sun.ac.za, Division of Computer Science, Stellenbosch University, South Africa
§slatorre@unisa.it, Dipartimento di Informatica, Università degli Studi di Salerno, Italy

Abstract—**Lazy sequentialization is one of the most effective approaches for the bounded verification of concurrent programs. Existing tools assume sequential consistency (SC), thus the feasibility of lazy sequentializations for weak memory models (WMMs) remains untested. Here, we describe the first lazy sequentialization approach for the total store order (TSO) and partial store order (PSO) memory models. We replace all shared memory accesses with operations on a shared memory abstraction (SMA), an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. We give efficient SMA implementations for TSO and PSO that are based on temporal circular doubly-linked lists, a new data structure that allows an efficient simulation of the store buffers. We show experimentally, both on the SV-COMP concurrency benchmarks and a real world instance, that this approach works well in combination with lazy sequentialization on top of bounded model checking.**

I. Introduction

Testing remains the most widely used approach to find program errors. It is useful when a high percentage of the selected executions lead to a violation of the program specification [26]. However, testing-only approaches such as stress testing remain highly ineffective for concurrency errors that manifest themselves rarely and are difficult to reproduce and repair [26]. Such "Heisenbugs" have become more prevalent on modern hardware architectures that use weak memory models (WMMs), because WMMs introduce additional non-determinism that remains outside the control of the testing environment. Consequently, testing needs to be complemented by automated verification techniques that can handle concurrency (and the non-determinism it introduces) symbolically.

One of these techniques is SAT/SMT-based *bounded model checking* (BMC), which has been used successfully to discover subtle errors in sequential software, even at large scale [16], [23]. Sequential BMC tools can be extended symbolically to the concurrent case by conjoining the formula representing the effect of each individual thread in isolation with a second formula representing the possible interferences caused by concurrent accesses to the shared memory [25], [4]. Since this second formula effectively includes an axiomatization of the underlying memory model, this approach can in principle work for both sequential consistency (SC) and different WMMs.

Partially supported by EPSRC grant no. EP/M008991/1.

However, embodying a memory model at the formula level requires extensive (and non-reusable) modifications of the underlying sequential BMC tool, and can affect scalability since the resulting expressions are large and complex.

An alternative approach is *sequentialization*, which translates concurrent programs into sequential programs with data non-determinism that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during the analysis. This allows the reuse of existing sequential BMC tools. Eager sequentializations [18], [27] guess the different values of the shared memory before the verification and then simulate (under this guess) each thread in turn. They can thus explore infeasible computations that need to be pruned away afterwards. Lazy sequentializations [17] instead guess the context switch points and compute the memory. They thus only explore feasible computations and can be used as basis of very effective verification tools, such as Lazy-CSeq [14]. This is shown by Lazy-CSeq's top rankings in recent software verification competitions but also borne out in practice: using Lazy-CSeq we discovered in 30 minutes a bug in the safestack benchmark [28], while all other approaches, including testing, failed [26]. However, to the best of our knowledge, lazy sequentializations have been developed only for SC, and not for any of the WMMs that are prevalent in modern computer architectures.

In this paper, as a first contribution, we therefore develop the first lazy sequentialization for multi-threaded programs for the total store order (TSO) [24] and partial store order (PSO) memory models. More specifically, we replace all accesses to shared memory items (i.e., reads from and writes to shared memory locations, and synchronization primitives like lock and unlock) by explicit calls to *API operations* over a *shared memory abstraction* (SMA, see Section II). For example, if $\mathtt{x}$ and $\mathtt{y}$ are two shared scalar variables then the statement $\mathtt{x} = \mathtt{y} + \mathtt{x} + 3$ is translated into $\mathtt{write}(\mathtt{x}, \mathtt{read}(\mathtt{y}) + \mathtt{read}(\mathtt{x}) + 3)$. The SMA can be seen as an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. This isolates the WMM from the remaining concurrency aspects, and allows us to reuse existing (lazy) sequentialization techniques and tools for SC. Our approach bears some similarity to the axiomatic representation of memory models [25], [4] but the fundamental difference is

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

that we work at the code level—in effect, we apply the very idea of sequentialization to WMMs themselves.

The TSO and PSO implementations of the SMA we describe here are the second contribution of the paper. They are carefully designed to optimize some parameters that lead, in combination with a lazy sequentialization targeting BMC tools, to efficient SAT/SMT encodings. Sections III and IV describe an efficient implementation of memory ADT for TSO, while Section V extends this to PSO. Section VI describes a first experimental evaluation of our approach. In particular, we compare our prototype implementation to CBMC (which implements TSO at the formula level) [4] and Niddhug [1] (which combines stateless model checking and dynamic partial order reduction). The experiments show that our approach delivers a comparable performance on simple benchmarks, but outperforms both CBMC and Nidhugg on the more complex safestack problem. It also shows that the number of timestamps (i.e., writes to the store buffers) required to expose TSO and PSO bugs is generally small. This implementation constitutes the third contribution of our paper.

II. Shared Memory Abstractions

We consider multi-threaded programs with C-like syntax including pointer arithmetics, dynamic memory allocation, and POSIX threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. We assume that all shared memory and synchronization operations are performed through an abstract data type that we call *shared memory abstraction* (SMA). This form can be achieved through a source-to-source transformation, if necessary. Introducing the SMA provides a separation of concerns between the shared memory and the control-flow related aspects of concurrent programs, such that the verifier design can focus on each of these aspects in isolation.

SMA API: Let us assume that shared scalar variables and threads are identified by unsigned integers. The full set of SMA API routines is:

- `init()` initializes the internal data structures of the SMA and the shared variables.
- `read(v,t)` and `read_addr(a,t)` return the value of the read issued by thread `t` of the shared variable `v` and the address `a`, respectively.
- `write(v,val,t)` and `write_addr(a,val,t)` capture the write by thread `t` of the value `val` into the shared variable `v` and the address `a`, respectively.
- `addr(v)` returns the address of the shared variable `v`.
- `malloc(expr)` dynamically allocates a number of shared memory locations given by the value of the expression `expr` and returns the base address.
- `lock(m,t)` and `unlock(m,t)` are the standard thread synchronization operations to acquire and release, respectively, a mutex `m` for thread `t`; if `m` is already acquired, the `lock` operation is blocking for `t`, i.e., `t` is suspended until `m` is released and then acquired.
- `fence(t)` flushes the store buffer of thread `t` and updates the shared memory accordingly.

SMA implementations: The semantics of concurrent programs, and in particular the concurrency aspects, can vary with the underlying memory model. For a program P, we can capture such a semantics by plugging a corresponding SMA implementation into P and thus interpreting the resulting program according to the standard interleaving semantics that assumes atomicity and sequential consistency of the memory operations. An SMA implementation consists of variables and data structures to capture the shared memory and functions that manipulate them, as listed in the API. Thus, we can model it as a transition system in the usual way.

III. Designing a TSO Shared Memory Abstraction

Here, we recall the TSO memory model and introduce two SMA implementations that capture it. We start with a reference implementation called TSO-SMA that represents the standard TSO semantics [24] directly but leads to complex formulas when used in a BMC-based sequentialization tool chain. We therefore introduce a new representation where the *per thread* store buffers are replaced by *per variable* write lists. We show how this, together with an indexing scheme, allows us to perform the shared memory updates implicitly, and in fact even to entirely remove any explicit representation of the shared memory. This reduces the size of the formulas for two reasons. First, BMC tools perform *function inlining* (i.e., replace each function call with the actual function code), so removing the updates, which can happen at any time and thus need to be inlined at every visible operation [17], reduces the size of the program and thus the formula. Second, because we remove the memory we do not need (propositional) variables to represent each memory write; we can instead reuse those used to represent the variable write lists. We describe an efficient shared memory abstraction eTSO-SMA that is based on this representation and is equivalent to TSO-SMA. In this section we give both SMA implementations at an abstract level; in the next section, we give the details of eTSO-SMA including concrete data structures and code for the API operations, and argue the equivalence of eTSO-SMA and TSO-SMA.

Total Store Ordering: TSO is a *relaxed-consistency* shared memory model where, different from SC, the ordering of write and read operations performed by different threads on the same variable can be altered. The behaviour of the TSO memory model can be described with respect to the architecture shown in Figure 1 [19].

Each thread t is equipped with a *store buffer* where the write operations performed by t are temporarily cached according to a FIFO policy. The effects of a cached write are visible only to the thread that has performed it. A read by a thread t of a variable y (resp. location at the address p) retrieves the value from the shared memory unless there is a cached write to y (resp. to the location at p) that is pending in its store buffer; in that case, the value of the most recent write in t's store buffer is returned. When a write is moved out of the store buffer the shared memory is updated accordingly. Memory *updates* can occur nondeterministically at any time in the computation provided that there are writes cached in some

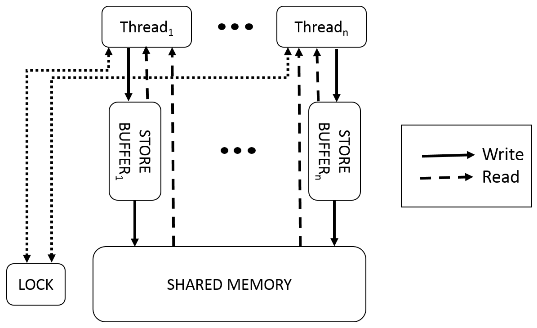


Fig. 1. TSO architecture.

store buffer. Memory *fences* simply flush the store buffer of the thread executing them. A lock can be acquired only if the store buffer of the acquiring thread is empty, i.e., does not contain pending writes.

TSO-SMA: The reference implementation TSO-SMA directly simulates the behavior of the TSO memory model according to the architecture shown in Figure 1. We use an array-based queue of bounded size for each thread store buffer, and a copy of the shared variables of the initial program to store one configuration of the shared memory.

Here, the simulation of each write operation results in a small formula: we just need to encode a single element write into the queue. However, read operations lead to larger and more complex formulas. The main source of complexity is in the number of steps required to determine the most recently performed write operation still cached in the queue. All these steps need to be encoded in the formula. Similarly, a flush operation involves every element of a store buffer which again need to be encoded in the formula. The formula for simulating shared memory updates is even bigger. In fact, even though each single update requires only a constant number of steps (to dequeue the write and then modify the content of a memory location), memory updates can occur nondeterministically at each step and in the limit the writes from all store buffers can be passed into the shared memory. Therefore, memory updates require a formula of size proportional to the sum of the maximum number of elements that can be stored in each (bounded) queue. Since these updates need to be simulated at each shared memory access, this leads to a considerable blow-up of the formula size for the whole sequentialized program, rendering this direct implementation hopelessly inefficient.

Timestamping writes: The handling of memory updates can be improved by performing the dequeueing operations implicitly. For this, we keep track of the (discrete) time in the executions with a global variable `clock` and add a *timestamp* to each write that is enqueued in a store buffer. These timestamps represent the future time at which a cached write will be used to update the shared memory. Since timestamps refer to future events and writes from different buffers can occur in any order (because the memory updates are nondeterministic), timestamps must be guessed. To ensure consistency with the TSO semantics, we must enforce that the timestamps of successive writes in the same store buffer

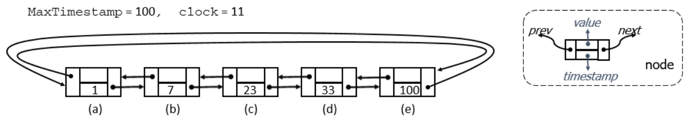


Fig. 2. T-CDLL example.

follow a non-decreasing order. Further, a timestamp must be assigned a value that is at least the current `clock` value.

By adopting this time-stamping schema we can keep the content of a store buffer up-to-date without actually dequeuing the writes when a memory update occurs. In fact, we can just increase the value of `clock`, and all writes that have an *expired* timestamp (i.e., a timestamp that is less or equal to the value of `clock`) can be treated as removed from their respective store buffers.

eTSO-SMA: We now describe an efficient implementation where each SMA operation can be encoded with a formula of constant size and where there is no need to explicitly encode memory updates. For ease of presentation, let us assume that the original program uses only shared scalar variables and that memory locations are never accessed using their addresses. The two main ingredients of this implementation are: (1) the combined use of timestamps and the variable `clock` as above and (2) a re-arrangement of the writes of the store buffers into lists per shared variables. We refer to each such list as a *variable write list* (vw-list, for short). For each shared variable x, we denote with Q_x its associated vw-list. vw-lists contain writes as pairs (val, ts) where val is the written value and ts is the associated timestamp. A write is *expired* if its timestamp is less than or equal to the current value of `clock`.

In addition to the writes of x that are currently cached in store buffers, Q_x also contains the last write of x that has been used in a shared memory update. This gives the current value of x in the shared memory, and is the only expired write in Q_x. Thus, as sketched above, we do not need additional variables to track the shared memory. Further, we keep Q_x ordered by non-decreasing timestamps and in case of writes with the same timestamp, in the order of insertion in the list. Hence, the current valuation in the shared memory of each variable is easy to retrieve from the front of the list.

Temporal Circular Doubly Linked List: To implement the vw-lists efficiently, we introduce *temporal circular doubly linked lists* (T-CDLL, for short), which are circular doubly linked lists where:

1) nodes are of the form shown in Figure 2; fields $prev$ and $next$ contain respectively the link to the predecessor and the successor in the list as usual, and the fields $value$ and $timestamp$ contain the write;
2) there is a unique *sentinel node*; it does not correspond to an actual write and its timestamp is maximal;
3) the sequence of nodes from the successor of the sentinel node through the sentinel node, via the $next$ link, is ordered by non-decreasing timestamps;
4) the *head* of the list is defined as the only node that (i) contains an expired write and (ii) whose successor

contains a non-expired write; it is uniquely determined by `clock`.

An example of T-CDLL is given in Figure 2. There the sentinel node is (e), the head node is (b) for `clock` value 11 and would change to (c) as soon as `clock` reaches 23.

Note that the portion of T-CDLL from the head through the sentinel node ensures the properties stated for the vw-lists, and thus we can easily use this portion to store each such list. Moreover, a T-CDLL also manages the list of available nodes in the remaining part: when the value of `clock` is increased so that the current head node expires, this becomes available and its successor becomes the new head. However, the node remains linked in the list; hence, all nodes between the sentinel and the head constitute a "free list" for further writes. Caching a write only requires checking that the successor of the sentinel is expired (but not the head of the list), and if so, overwriting the values and re-linking the node.

For example, in the T-CDLL of Figure 2, only node (a) is available. By updating `clock` to 30, the head becomes (c) and (b) becomes available but remains linked to the list of available nodes that starts from (a). If we then try to cache a write with timestamp 40 in the vw-list, we can take (a), unlink it from the T-CDLL and then link it back between (d) and (e) with value and timestamp updated according to the write.

The T-CDLL for a vw-list is initialized with a fixed number of nodes that stay unchanged along the computation (i.e., the size of lists we can encode is bounded). The initial value for the timestamps is 0 except for the sentinel node, whose timestamp is set to the maximum allowed timestamp. In the next section we show how to implement T-CDLLs efficiently by using four parallel arrays, one for each node field.

IV. Implementation of eTSO-SMA

The code of eTSO-SMA comprises a *module* whose *abstract view* is essentially the API given in Section II and whose *implementation view* (i.e., the complete data type declaration along with the actual code implementing the API operations) will be discussed in this section.

Memory bounds: We assume that during the program execution all threads can access the memory, which consists of a finite sequence of locations of the same size. Each location has its own memory address which corresponds to its position in the memory. Shared variables are allocated to distinct memory locations. However, eTSO-SMA does not capture the entire memory explicitly but only tracks a bounded number of memory locations.

We use several parameters to bound our analysis and in particular the memory representation. `T` denotes the maximum thread identifier used in the program, `N` the number of nodes in each T-CDLL, `V` the maximum number of memory locations tracked along any execution (including the locations of the shared variables), and `K` the maximum timestamp.

We assume that each shared variable has an integer identifier in the range $[0, \mathtt{V}-1]$. Further, we can have an additional number of memory locations to track that can be accessed only through their memory addresses.

Auxiliary Data Structures: We use the following global auxiliary variables of type integer:

- `clock` keeps track of the number of writes performed by all threads; it is initialized to 0 and bounded by `K`.
- `address[v]` contains the physical memory address of `v`, for any `v`$\in [0, \mathtt{V}-1]$; the `init` function initializes it with distinct nondeterministic values; the values do not change during the program simulation;
- `value[v][node]` and `tstamp[v][node]` store the value and timestamp of each write associated with each location.
- `max_tstamp[t]` stores for each thread the largest timestamp of any executed write;
- `prev[v][node]` and `next[v][node]` store the link to the previous and next node in the T-CDLL for each location.
- `last_value[v][t]` and `last_tstamp[v][t]` store the last value written and the timestamp of the last write performed by each thread for each location.

The nodes of the T-CDLL corresponding to variable `v` are kept in `prev[v][i]`, `value[v][i]`, `tstamp[v][i]` and `next[v][i]` for `i`$\in [0, \mathtt{N}-1]$.

Malloc and init: During its execution a thread can require a block of n consecutive locations by invoking $\mathtt{malloc}(n)$, which returns the address of the first location of a newly allocated heap block. Memory addresses can be used to access this shared memory. Let p be a shared pointer variable and `x` be a local variable. A location with address i is *pointed to* by p if the value of p is i. Then, `*p = x` copies the value of `x` into the location pointed to by p, and statement `x = *p` copies the value of the location pointed to by p into `x`. Note that we do not represent the heap memory explicitly as we only track some of its locations. Concerning to `malloc` we maintain a bounded sequence, say of fixed size m, where each element represents a memory block. In particular, for each block we store its base address, its size, and whether it has been allocated. This sequence is implemented using arrays. The `init` function initializes each of these blocks with a nondeterministic base address and a nondeterministic size, making sure that block do not overlap. These values do not change during program executions.

Clock update: `clock_update` is a service routine that updates the variable `clock` with a nondeterministic value picked in the range from its current value to its allowed maximum value `K` (see Figure 3, lines 11-15). As a consequence of such an update, some of the writes in the T-CDLLs may expire, thus modifying some of the head nodes and therefore, the valuation of variables in the shared memory and the configuration of the T-CDLLs.

We stress that this is a very convenient way to implement the shared memory updates since we achieve this without altering the underlying data structures. Moreover, it is correct w.r.t. the TSO semantics since the writes flow into the shared memory by increasing values of the timestamps, and by the ordering we ensure on the timestamps, this enforces a correct simulation of the TSO semantics on the memory updates.

```
 1: static uint clock;
 2: static uint address[V];
 3: static  int value[V][N];
 4: static uint tstamp[V][N+1];
 5: static uint prev[V][N+1];
 6: static uint next[V][N+1];
 7: static uint last_value[V][T];
 8: static uint last_tstamp[V][T];
 9: static uint max_last_tstamp[T];
10:
11: void clock_update( ){
12:    int tmp;
13:    assume( tmp <= K && tmp >= clock );
14:    clock = tmp;
15: }
16:
17: void write(uint v,int val,uint t){
18:   clock_update();
19:      // remove expired node from list
20:   uint node = next[v][N];
21:   assume(tstamp[v][next[v][node]]<= clock);
22:   next[v][N] = next[v][node];
23:   prev[v][next[v][N]] = N;
24:      // select position in the list for insertion
25:   uint succ = nondet();
26:   assume( succ<=N && tstamp[v][succ]>clock );
27:   uint pred = prev[v][succ];
28:      // guess suitable timestamp
29:   uint ts=nondet();
30:   assume( ts >= clock
31:           && ts >= max_last_tstamp[t]
32:           && ts >= tstamp[v][pred]
33:           && ts <  tstamp[v][succ] );
34:      // insert node at selected position
35:   value[v][node] = val;
36:   tstamp[v][t] = ts;
37:   next[v][node] = succ;
38:   prev[v][node] = pred;
39:   next[v][pred] = node;
40:   prev[v][succ] = node;
41:      // update auxiliary data
42:   max_last_tstamp[t] = ts;
43:   last_tstamp[v][t] = ts;
44:   last_value[v][t] = val;
45: }
46:
47: void write_to_address(uint a,int val,uint t){
48:      // select identifier of the memory address
49:   int v = nondet();
50:   assume( v < V );
51:   assume( address[v] == a );
52:   write( v, val, t );
53: }
54:
55: int read( uint v, uint t ) {
56:    clock_update();
57:      // retrieve the value from thread store-buffer
58:   if ( last_tstamp[v][t] > clock )
59:       return last_value[v][t];
60:      // retrieve the value from shared memory
61:   int node = nondet();
62:   assume( node < N &&
63:           tstamp[v][node] <= clock &&
64:           tstamp[v][next[v][node]]>clock);
65:   return value[v][node];
66: }
67:
68: int read_from_address( uint a, uint t ) {
69:      // selecting the id for the memory address
70:   int v = nondet();
71:   assume( v < V );
72:   assume( address[v] == a );
73:   return( read(v,t) );
74: }
75:
76: void fence( uint t ){
77:   clock_update();
78:      // make all thread's write expired
79:   if ( clock < max_tstamp[t] )
80:       clock = max_tstamp[t];
81: }
```

Fig. 3. Code stubs for eTSO-SMA.

Write operation: We recall that function `write` takes as input a variable identifier `v`, a value `val`, and a thread identifier `t`. `write` first updates the clock value by invoking `clock_update` and then simulates the write operation by changing the state of the memory representation by adding a new write in the T-CDLL associated with `v`. The code for this function is given in Figure 3 at lines 17-45.

Take an available node from the T-CDLL of the variable identifier (lines 20-23). Variable `node` is set to a position of the array encoding the T-CDLL for `v` that corresponds to the successor of its sentinel node. From the property of part 3 of the definition of T-CDLLs (that we maintain as an invariant in our implementation), this is the node with the smallest timestamp in the list. We are going to use this node to cache the write. The `assume` statement at line 21 ensures that the successor of `node` is also expired. Otherwise, `node` would be the head node and thus there are no available nodes, therefore the computation must be blocked. We then remove the node from the list by appropriately setting the fields `next` and `prev` of the successor and the predecessor of `node`, respectively (lines 22-23).

Select position for insertion (lines 25-27). We nondeterministically guess a node `succ` that is the candidate to be the successor of `node` in the list (after insertion). We make sure that `succ` encodes a non-expired write by checking that its timestamp is greater than the current value of `clock`. Note that, a node with this feature always exists in since the timestamp of the sentinel node is `K`. We then set the local variable `pred` to the predecessor of `succ`. Node `pred` will be the predecessor of `node` after its reinsertion in the list.

Guess a suitable timestamp for the new write (lines 29-33). First, we guess a value (line 29), then we check that it is not smaller than the current `clock` value (line 30), and the last timestamp used for the same thread (line 31). This last check guarantees that the writes from the same thread update the shared memory in the FIFO order. The last two constraints at lines 32-33 guarantee the T-CDLL invariant that nodes must be in a non-decreasing order (part 3 of T-CDLL definition). Note that we allow the timestamp of the new write to be the same as that of the previous write in the list (which corresponds to a situation where they are passed to the shared memory in the same update). However, the inequality is strict w.r.t. the next write in the list (line 33). In this way we guarantee that a write cannot overtake another write from the same thread that is already cached in the store buffer with the same timestamp (which would violate the TSO semantics).

Node insertion at selected position (lines 35-40). We can now insert the new write into the T-DCLL. We first set its value, then its selected timestamp, and finally we insert `node` between `prec` and `succ` (lines 37-40). Clearly, the resulting list is still a T-DCLL.

Update auxiliary data (lines 42-44). We update the auxiliary variables to ensure that their invariants are maintained.

Write-to-address operation: The first step of function `write_to_address` is to select the identifier corresponding to address, if any, and then call `write`.

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

Read operations: The function `read` takes as input a variable identifier `v` and a thread identifier `t`, and returns the value of `v` retrieved from the current state of the memory system. The implementation of `read` is shown in Figure 3. It first updates the clock. Then, it checks whether the last performed write by `t` on `v` has not expired yet. This condition ensures that if there is still a pending write in `t`'s store buffer, then the value of the latest such write is returned, as per the TSO semantics. Otherwise, it returns the value of the latest expired write in the T-CDLL of `v` (lines 61-65), which is always guaranteed to exist by the invariant property of T-CDLLs, and as previously argued, it corresponds to the valuation of `v` in the shared memory. When a read is performed using a memory address (lines 68-74), we first retrieve the location identifier, say `v`, corresponding to the memory address `a` and then return the value returned by calling `read` on `v`.

Fence operation: To flush the store-buffer of a thread it is sufficient to mark all its writes as expired. Thus, function `fence` shown in Figure 3 at lines 76-81 first updates the clock to the current time and then sets again the clock to the value of the maximum ticket issued by thread `t` in case it results greater than the current clock.

Correctness: We have already observed the main properties concerning the correctness of our implementation. A formal proof of this can be given by showing that the transition system T that captures eTSO-SMA is equivalent to the transition system $T_{\mathtt{TSO}}$ that captures TSO-SMA (and thus the semantics of the TSO memory model) in the sense that they can simulate each other behaviors going through equivalent configurations. The most complicated case of the proof is for the `write` function; we sketch this one here. From the observations in the write-operations section above, we have that after the execution of `write` the corresponding T-CDLL is correctly updated. Also the current write is added with a timestamp that is not smaller than the timestamp of the last previously cached write from the same thread (line 31), and in case the two timestamps are equal and the two writes are on the same variable, line 33 guarantees that we keep the same order as in the store buffer. This implies that if we start from equivalent configurations of T and $T_{\mathtt{TSO}}$, then the configurations of T and $T_{\mathtt{TSO}}$ resulting after the invocation of `write` are also equivalent. Therefore, we get:

Theorem 1. *eTSO-SMA and TSO-SMA are equivalent.*

V. Extension to the PSO memory model

We recall that the semantics of PSO is the same as for TSO except that each thread is endowed with a store buffer for each shared memory location. To handle PSO we just need to modify the implementation of eTSO-SMA with the following changes in the write function from Fig. 3. In the write of a location `v` by a thread `t` we do the following:

- the guessed timestamp `ts` must be not lower than the timestamp of the last write of `t` on `v` (according to the PSO semantics, a write by a thread t of a variable following a previous write by t can overtake it, but cannot overtake a previous write of the same variable); line 31 of Fig. 3 must be replaced with
```
        && ts >= last_tstamp[v][t]
```
- `ts` must be the last timestamp of `t` if it is greater than the current one; line 42 of Fig. 3 must be replaced with
```
    max_last_tstamp[t] =
         (ts<=max_last_tstamp[t])?
                max_last_tstamp[t]:ts;
```

Denoting with ePSO-SMA our prototype tool obtained from eTSO-SMA by the above changes and with PSO-SMA the reference implementation for PSO (obtained similarly to TSO-SMA for TSO), we get:

Theorem 2. *ePSO-SMA and PSO-SMA are equivalent.*

VI. Experimental Evaluation

Prototype implementation: We implemented our approach for C programs with POSIX threads in a prototype tool called LazySMA.[1] It is based on the open-source CSeq framework [13], [11] which allows the development of sequentializations following a modular approach: tools are built as pipelines of source-to-source transformations where the result of the last transformation is fed into a sequential analysis backend. For our prototype we implemented a new transformation that replaces each memory access by the corresponding operation from the API. In order to combine this new transformation with Lazy-CSeq [14], [15] we needed to "inject" the new memory management layer into a few locations where memory and concurrency handling overlap. For example, we changed the original `lock` and `unlock` simulation procedures of Lazy-CSeq to use the barriers for memory synchronization required by the weaker memory model.

Experimental set-up: We have compared our prototype implementation (with CBMC v5.3 as sequential verification backend) against two tools with built-in support for analysis under WMMs: CBMC [12], a mature bounded model checker tool for C/C++ programs, and Nidhugg [1], a bug-finding tool that combines stateless model checking with dynamic partial order reduction on relaxed memory executions.

We ran the experiments on a dedicated machine with a Xeon W3520 2.6GHz processor and 12GB of physical memory running 64-bit linux 3.0.6. We set a 10GB memory limit and a 600s timeout for the simple benchmarks and timeout of 14,400s for the safestack example. For each tool and benchmark, we set the parameters to the minimum value needed to expose the error.

Standard benchmarks: We first evaluated our approach over a set of benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The results are summarized in Table I. The unwind parameter was used by all the three tools considered in the comparison. The sum of naddr and nmalloc gives the parameter `V` from Section IV. The parameter bitwidth gives the size of integers (in bits) used in the sequential analysis and the parameter rounds is the number of rounds used by Lazy-CSeq.

[1]`http://users.ecs.soton.ac.uk/gp4/cseq/fmcad16.zip`

TABLE I
ANALYSIS RUNTIMES UNDER TSO AND PSO

		parameters							TSO runtime (*s*)			PSO runtime (*s*)		
	bug?	unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	LazySMA	CBMC	NIDHUGG	LazySMA	CBMC	NIDHUGG
dekker	●	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	●	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	●	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	●	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	●	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	●	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	●	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	●	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

The first block contains results for the classical mutual exclusions algorithms (`dekker`, `lamport`, `peterson`, `szymanski`). These implementations are correct under SC but not under TSO or PSO. All tools find the errors, but because of their small size, Nidhugg outperforms both CBMC and our prototype (which incurs a constant overhead for the sequentialization process) on these programs.

The second block of the table contains variations of the fibonacci-benchmark, in which two worker threads concurrently increase two shared counters, and a main thread checks whether any the two counters can reach a defined value. A full exploration of the thread interleavings is required to identify the error (or show its absence) in this program. Techniques such as partial-order reduction do not apply, and several tools struggle to analyze it. We have included both the safe and unsafe versions. Here, Nidhugg is generally slower than both CBMC and our prototype tool, and fails to terminate on the safe version. Our prototype beats CBMC on the safe cases, but is slower on the unsafe ones.

The next two benchmarks originate from industrial code: `parker` models a semaphore-like synchronization class that breaks under TSO [1] (and thus also under PSO), and `stack` which was taken from SV-COMP [7].

The last two lines show the average values for 5803 litmus tests for WMMs; note that we ran these under TSO and under PSO. For TSO, both our prototype and CBMC successfully identified the 277 test cases containing a reachable error, while Nidhugg failed to find one of them. For PSO, Nidhugg and IMU-CSeq both find 968 unsafe instances, while CBMC claims that there are 971 unsafe instances but this includes three spurious counterexamples. The performance gap between CBMC and our tool could be reduced with a more efficient implementation, as our prototype transforms each file nearly 20 times, each time requiring parsing and unparsing.

Safestack: We have conducted further experiments on a real world benchmark, Safestack [10], which is a lock-free stack implementation designed for weak-memory models. It contains a rare bug that is hard to find with automatic bug-finding techniques already under SC (including random testing, Nidhugg, CIVL [31], CBMC and other approaches based on BMC) [26]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq. It requires a minimum of 4 threads, 3 loop unwindings, and 4 rounds of computation to expose a bug caused by two of these threads simultaneously modifying the element at the first position of the array implementing the queue. This shows that the error is actually quite deep, which explains why other approaches based on explicit handling of interleaving fail.

TABLE II
ANALYSIS RUNTIMES FOR SAFESTACK UNDER TSO AND PSO

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
K	N	rounds	Time	Mem.	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

Safestack is written in C++. We manually translated it into C, providing simulation functions for the C11 atomic functions used in the test. We experimented with this C version, with different bounds for the queue size of each memory address, and the maximal timestamp along any bounded computation. Table II summarises these experiments. Note that we only used three bits to represent integers during the analysis. We then checked whether counterexamples found also hold for full 32-bits integers, by running Lazy-CSeq over the exact schedule extracted from the counterexample. A "Yes" entry in the CEX? column means that the counterxample holds, thus there are no spurious counterexamples (due to overflow).

Because SC and TSO coincide if maxclock is set to 1, the first four lines indirectly show the overhead paid for our TSO encoding. Since the SC analysis using Lazy-CSeq (not shown here) requires approximately 3 minutes, the TSO encoding itself thus introduces an approximately 3x-4x overhead. The last two lines show that we can still find the error under "proper" TSO. It also shows that the weaker memory model reduces to 3 (from 4) the number of rounds required to expose this error; however, the analysis time grows noticeably, by almost an order of magnitude. Finally, increasing maxclock (for fixed values of qsize and rounds) shows that the analysis explores more reorderings of reads over writes (witnessed by the increased memory consumption).

VII. Related Work

The transformation of concurrent programs under TSO into equivalent programs under SC is intrinsic in the architecture from Figure 1. However, the explicit modeling of the store buffers in the resulting program introduces a substantial overhead for standard SC verification tools.

In [5], the authors replace the store buffers with $O(k)$ local variables per thread, where k is the number of context-switches for each thread that is allowed in the analysis. The main intuition there is: when a write operation occurs, a future context number is guessed with the meaning that the write will be visible to the other threads at that context. This is similar to our guess of a future timestamp in the sense that it implicitly gives a total ordering of the shared memory updates, but in our setting this is completely unrelated to the thread context at which the memory update will occur.

In [29], the authors replace the store buffers by embedding each of them symbolically in the thread locations. Their translation goes through the construction of the corresponding transitions systems that seems appropriate for a backend as SPIN but in our experiments works poorly with BMC since it introduces a lot of redundancy in the constructed formulas.

Another main difference of our approach with both [5] and [29] is that we rearrange the contents of store buffers *per variable* instead of *per thread* and entirely maintain them in T-CDLLs of bounded size. This, along with the strong invariant properties of T-CDLLs, results in smaller formula encodings computed by the BMC backend tools (see Section III).

Other recent work concerning the verification of concurrent programs under WMMs is [1], [2], [3], [4], [6], [8], [9], [30]. The most related to ours is [3] where the authors give a general reduction technique to SC by augmenting the programs with arrays to simulate the caching and buffering due to the WMMs and use it in combination with CBMC. In [4], CBMC is enhanced with a reduction based on partial orders.

We have designed our translation to target BMC backends and used the tool Lazy-CSeq [14], [13] for our experiments. Lazy-CSeq implements an efficient lazy sequentialization that works exceptionally well with BMC backends and has won the SV-COMP twice [7]. It performs a bounded context-switching analysis [21] and has been recently extended to unbounded programs [20]. The idea of sequentialization was originally proposed in [22] but became popular with the first scheme for an arbitrary but bounded number of context switches [18].

VIII. Conclusions

In this paper we have described a new approach to the verification of concurrent programs under WMMs. We have introduced an abstract data type that factors out the semantics of the memory model, allowing us to reuse tools designed for the analysis of concurrent programs under SC. We have given an efficient implementation of the ADT that works well in combination with Lazy-CSeq. We have demonstrated the effectiveness of this approach for finding bugs under TSO and PSO: our prototype tool is competitive with existing tools on standard benchmarks used in the literature; it also works for more complex benchmarks that are, to the best of our knowledge, out of reach for existing bug-finding approaches. We have developed our approach for TSO, and extended it to PSO simply by organizing the cached writes per variable and thread, and not just per variable. We believe that our approach can be extended to further WMMs.

References

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, pp. 353–367, 2015.

[2] T. Abe and T. Maeda. A general model checking framework for various memory consistency models. In *PDP*, pp. 332–341, 2014.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pp. 512–532, 2013.

[4] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pp. 141–157, 2013.

[5] M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, pp. 99–115, 2011.

[6] M. F. Atig, A. Bouajjani, and G. Parlato. Context-bounded analysis of TSO systems. In *FPS*, pp. 21–38, 2014.

[7] D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, pp. 401–416, 2015.

[8] A. Bouajjani, G. Calin, E. Derevenetc, and R. Meyer. Lazy TSO reachability. In *FASE*, pp. 267–282, 2015.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PDLI*, pp. 12–21, 2007.

[10] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dep. Sec. Comput.*, (6):368–379, 2013.

[11] B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE*, pp. 710–713, 2013.

[12] A. Horn and D. Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. In *FORTE*, pp. 19–34, 2015.

[13] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *ASE*, pp. 807–812, 2015.

[14] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV*, pp. 585–602, 2014.

[15] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In *TACAS*, pp. 398–401, 2014.

[16] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In *MEMICS*, pp. 30–39, 2014.

[17] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, pp. 477–492, 2009.

[18] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Meth. in Sys. Des.*, (1):73–97, 2009.

[19] A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In *ASPLOS*, pp. 45–58, 2015.

[20] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *ATVA*, 2016. To appear. http://eprints.soton.ac.uk/397033/.

[21] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pp. 93–107, 2005.

[22] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pp. 14–24, 2004.

[23] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Successful use of incremental BMC in the automotive industry. In *FMICS*, pp. 62–77, 2015.

[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, (7):89–97, 2010.

[25] N. Sinha and C. Wang. On interference abstractions. In *POPL*, pp. 423–434, 2011.

[26] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: an empirical study. In *PPoPP*, pp. 15–28, 2014.

[27] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, pp. 551–565, 2015.

[28] D. Vyukov. Bug with a context switch bound 5, 2010.

[29] H. Wehrheim and O. Travkin. TSO to SC via symbolic execution. In *HVC*, pp. 104–119, 2015.

[30] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, pp. 250–259, 2015.

[31] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: formal verification of parallel programs. In *ASE*, pp. 830–835, 2015.

Combining Requirement Mining, Software Model Checking and Simulation-Based Verification for Industrial Automotive Systems

Tomoya Yamaguchi and Tomoyuki Kaga
TOYOTA MOTOR CORPORATION
{tomoya_yamaguchi,tomoyuki_kaga}@mail.toyota.co.jp

Alexandre Donzé and Sanjit A. Seshia
University of California, Berkeley
{donze,sseshia}@berkeley.edu

***Abstract*—The verification and validation of industrial closed-loop automotive systems still remains a major challenge. The overall goal is to verify properties of the closed-loop combination of control software and physical plant. While current software model-checking techniques can be applied on a software component of the system, the end result is not very useful unless the interactions with the physical plant and other software components are captured. To this end, we present an industrial case study in which we combine requirement mining, software model-checking, and simulation-based verification to find issues in industrial automotive systems. Our methodology combines the the scalability of simulation-based verification of hybrid systems with the effectiveness of software model-checking at the unit level. We presents two case studies: one on a publicly available Abstract Fuel Control System benchmark and another on an actual production SiLS (Software in the Loop Simulator) benchmark. Together these case studies demonstrate the practicality of the proposed methodology.**

I. Introduction

In recent years, functional requirements for automotive control systems have become far more sophisticated, leading to the development of more complex and larger scale control software. This in turn has increased the importance of verification and validation (V&V) processes in the automotive industry since software can affect the integrity of the automotive system as a whole.

The industry authors of this paper have been part of a team which, for more than a decade, has attempted to use verification techniques such as a model checking [1] on production automotive systems. The strong guarantees provided by model checking make it attractive for the automotive industry. Unfortunately, even with impressive tools, these attempts have proved to be time-consuming, generally requiring considerable person-hours and expertise to be applied, with little or no conclusive results and many false alarms. A major factor is that most tools can handle only small, unit-level components, whereas to be truly useful, one needs to map an issue found at the unit level to a system-level issue that an engineer can confirm.

In order to apply model-checking at the software component-level and deduce results at the system-level, one has to make the right assumptions on the interfaces of modules (pre- and post-conditions). To this end, this paper proposes to leverage recently-developed simulation-based verification techniques for cyber-physical systems (e.g. [2], [3]) that can be used for falsifying temporal logic properties as well as to mine specifications from simulation traces [4]. Such methods have proven to scale well and able to provide useful information about cyber-physical systems of industrial size and complexity. We show how requirement mining, simulation-based verification, and software model checking can be combined to (1) obtain more precise pre-conditions for software modules in order to reduce the number of false-positives from model checkers, and (2) to guide the search for concretizing probable issues at the system levels when they do exist. We present results on a case study of an Abstract Fuel Control System benchmark [5] as well as on an actual production powertrain design in a SiLS (Software in the Loop Simulator) setting. We show that the resulting V&V methodology is more scalable than software verification and provides better guarantees than simulation-based verification.

II. Overview of Our Approach

In order to address the V&V problem identified in the preceding section, we identified two tasks which can help:

- Finding good pre-conditions for unit level software components, which characterize the states they can reach in the closed-loop system.
- Mapping counterexamples found at the unit level to "system-level" counterexamples, i.e., concretizing the unit-level counterexample on the closed-loop system.

Right now, the first item is performed manually. The second item is also performed manually, but only incompletely — the unit level counterexample is validated but typically not extended to a system level counterexample. In our experience, *finding good pre-conditions* takes up to 20% of total model checking person-hours and *validating a unit-level counterexample* takes 50% of total person-hours [6] [12].

We therefore propose a methodology that combines simulation-based verification of the closed-loop system with software model checking at the unit level. Software model checking is exhaustive, and simulation-based verification is scalable to the system level: therefore, their combination allows us to find corner-case issues in the code that generate counterexamples at the system level.

More specifically, this methodology combines requirement mining, software model checking and simulation-based verification in a complementary fashion. The key steps in this methodology are as follows (see flowchart in Fig. 1):

1. *Pre-condition (range) mining:* Using a system for mining requirements of closed-loop cyber-physical systems [4], we generate pre-conditions for a software component in terms

ISBN: 978-0-9835678-6-8. Copyright owned jointly by the authors and FMCAD, Inc.

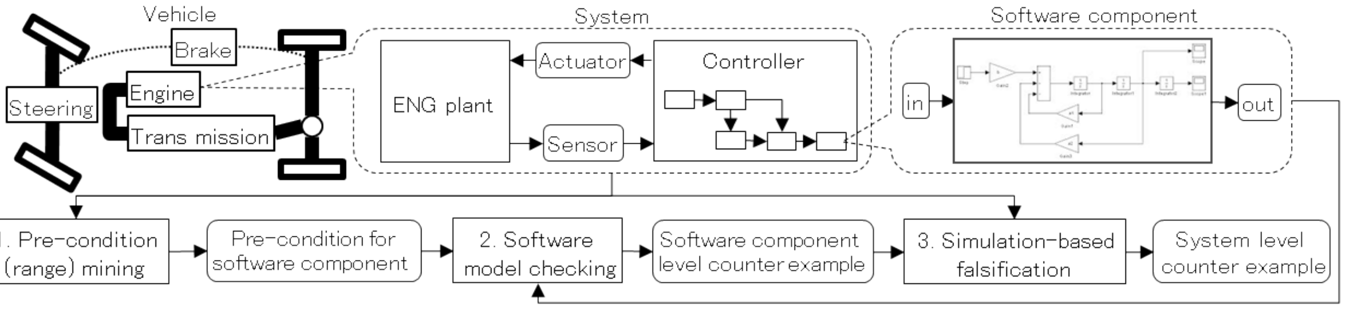

Vehicle
Brake
Steering
Engine
Trans mission
System
ENG plant
Actuator
Controller
Sensor
Software component
in
out
1. Pre-condition (range) mining
Pre-condition for software component
2. Software model checking
Software component level counter example
3. Simulation-based falsification
System level counter example

Fig. 1. Composition of vehicle system & proposed method

of ranges of values that selected interface variables must always lie in.
In general, the requirements are specified in signal temporal logic [7] [12]. For the purpose of this step, the STL specification is parametrized, and has the syntactic form below:

$$\begin{aligned} \boldsymbol{x} &= (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \\ \pi_{min} &= (\pi_{min\ 1}, \ldots, \pi_{min\ n}) \\ \pi_{max} &= (\pi_{max\ 1}, \ldots, \pi_{max\ n}) \\ \Box & \left(\bigwedge_{i=1}^{n} ((\pi_{min\ i} \leq \boldsymbol{x}_i) \wedge (\boldsymbol{x}_i \leq \pi_{max\ i})) \right) \end{aligned} \tag{1}$$

where $\boldsymbol{x}$ are input variables of the target software component, and π_{min} and π_{max} are parameters to be mined. (Our technical report[12] shows how the tool operates.)

2. *Software model-checking:* Given the generated preconditions, we run a software model checker to check assertions or post-conditions for a unit-level software component. If any unit-level counterexamples are found, then we go to the next step. Otherwise, we can mark this component as verified.
We use off-the-shelf software verifiers. For the case studies, we used Simulink Design Verifier (SLDV, [8]) and the C Bounded Model Checker (CBMC, [9]). (Our technical report [12] includes background on these tools. If the software verifier generates a counterexample, then we map this counterexample back to an assignment to the input variables, and denote this assignment by the vector $\widehat{\boldsymbol{x}}$.
3. *Simulation-based Falsification:* Given a counterexample for a unit-level software component, we try to extend it to a system-level counterexample by the use of simulation-based verification [2], [3]. The use of these tools requires the unit-level counterexample to be encoded into a suitable property expressed in signal temporal logic [7]. If the tool succeeds in finding a system-level counterexample, then this is passed on to engineers who then cross-check whether this issue can indeed occur. Otherwise, we go back to the previous step and try to find more unit-level counterexamples. (Our technical report [12] describes the working of the falsifier (Breach [2])). Briefly, we formulate an STL property that states that the value of the input variables of the software component $\boldsymbol{x}$ always remain at least an $\epsilon > 0$ distance away from the counterexample assignment $\widehat{\boldsymbol{x}}$, as shown below:

$$\varphi(\boldsymbol{x}) = \Box \left(\sqrt{\sum_{i=1}^{n} (\boldsymbol{x}_i(t) - \widehat{\boldsymbol{x}}_i)^2} \geq \epsilon \right) \tag{2}$$

Algorithm 1 AF target decision

```
1: if $60.0 \leq throttleAngle \leq 62.0$ and $-2.0 \leq waterTemp \leq 2.0$
2: and (($airFlow$[g/s]$< 0.0$) or ($10.0 \leq airFlow \leq 11.0$)) then
3:     $airFuelRatioTarget \leftarrow 12.5$            ▷ injected fault
4: else
5:     $airFuelRatioTarget \leftarrow 14.7$            ▷ original code
6: end if
```

We refer to this property as Property Eq. 2. Breach uses numerical optimization to search for a counterexample. If it finds one, this is a system-level counterexample showing how $\widehat{\boldsymbol{x}}$ can be extended to the entire closed-loop system.

III. Case Study 1: Abstract Fuel Control System

In this section, we present an evaluation of our methodology on an Abstract Fuel Control System (AFC) model [5]. This model was proposed by Toyota researchers [5] as a synthetic challenge problem representative of some of the key verification challenges faced (see [12] for details).
Description. This fuel control model is a subsystem of gasoline engine and implemented in Simulink [10]. The purpose of this model is to control the engine air-fuel ratio so as to meet emissions targets — an important control functionality in a gasoline engine. The model contains the air-fuel controller and a mean value model of the engine dynamics, such as the throttle and intake manifold air dynamics. Inputs of this system model are $throttleAngle$, $engineSpeed$ and $waterTemp$. Outputs are $airFuelRatio$, $airFuelRatioTarget$ and $controllerMode$.
Injected fault: We revised this model to inject a rare case malfunction into one of the software components. This software component has 3 inputs: $airFlow$, $throttleAngle$ and $waterTemp$ and one output:$airFuelRatioTarget$. The injected malfunction sets $airFuelRatioTarget$ to 12.5 under a rare condition (see Alg. 1). The post-condition for this model is that $airFuelRatioTarget \geq 13.0$.

The injected fault is highly representative of true issues that come up during production. This model is a functional approximation of a more complex A/F reference decision unit that would include a latent malfunction. It is desirable to find such issues early in the development cycle. However, such a rare case malfunction is difficult to find using random testing or simulation-based methods in general.
Experimental Results: The input variables $\boldsymbol{x}$ are $airFlow$, $throttleAngle$, and $waterTemp$. The result of range mining provides $airFlow$[g/s]$= [2.6, 34.0]$, $throttleAngle$[deg]$= [0.0, 90.0]$ and $waterTemp$[°C]$= [-30.0, 100.0]$. Note that $airFlow$ is the only intermediate variable whose range was unknown before range mining. We applied SLDV with and

without the mined input ranges as a pre-condition. In both cases, SLDV finds counterexamples that violate the post-condition described above, but they are different. We show the counterexamples in Table I. We indicate the pre-condition ranges in the absence of range mining as no-condition, indicating that there is no constraint on the value of that input variable. The counterexample obtained without range mining turns out to not be feasible due to the negative value for $airFlow$ which is not possible when accounting for the physical plant dynamics.

In our methodology, we take the counterexample obtained with range mining, namely, $\widehat{\boldsymbol{x}} = [10.0, 60.0, -2.0]$ and run Breach to falsify Property Eq. 2. Breach finds a system-level counterexample that violates the post-condition. (Our technical report [12] shows this system-level counterexample.)

IV. Case Study 2: Production Powertrain System

Our second case study is a production powertrain system which is one of the production models under development. It is based on SMiL [11] which is an in-house SILS (Simulation-in-the-Loop-Simulation) environment developed at Toyota [12].

Description: This power train model comprises engine and transmission sub-systems as well as the entire controller code in C. The model has 5 external inputs: $pedalAngle$, $brakeAngle$, $shift$, $waterTemp$ and $airTemp$. $shift$ position is always fixed as "D" (drive) in this evaluation.

Injected Fault: Motivated by an actual issue that occurred during development, dealing with a malfunction triggered under a very specific combination of conditions in the C code, we injected a fault into this model. Alg. 2 shows the injected fault in code that decides a control target in a closed loop and has 8 input variables:

- $waterTemp$[°C]: Temperature of engine coolant.
- $atmosphericPressure$[hPa]: Atmospheric pressure.
- $gear$: Current gear position in transmission.
- $gearHoldFlag$: Status of lock-up.
- $idlFlag$: Status of engine idling.
- $catalystTempHIGHflag$: Turned ON when catalyst temperature becomes high.
- $fuelCutFlag$: Status of fuel cut, triggered when negative torque is required e.g. braking.
- $engRpm$[rpm]: Rotational speed of engine.

The post-condition of this code is $target < 150.0$. We now discuss how we attempt to find a system-level counterexample that violates this post-condition.

Experimental Results: We applied the methodology of Sec. II to this case study. Once again, Breach was used for the range mining and falsification steps, while, in this case, CBMC [9] was used as the software verification tool. As a reference, we also applied software model checking without range mining. Table II shows the counterexamples obtained at the unit level with and without range mining.

The counterexample obtained without range mining is not a true system-level counterexample, because, e.g., it assigns $atmosphericPressure$ to be greater than 2.0 hPa.

TABLE I
Counterexamples from SLDV with and without range mining. "ce" indicates the counterexample values.

input variable	with mining		no mining	
	range	ce	range	ce
$airFlow$[g/s]	[2.6, 34.0]	10.0	no-condition	-0.5
$throttleAngle$[deg]	[0.0, 90.0]	60.0	no-condition	60.0
$waterTemp$[°C]	[-30.0, 100.0]	-2.0	no-condition	-2.0

Algorithm 2 Injected issue on power train model

```
1: if $waterTemp > WARMINGUP$
2: and $atmosphericPressure > THRESHOLD$
3: and $((4th \leq Gear \leq 6th)$ or $(gearHoldFlag = OFF))$
4: and $idlFlag = OFF$ and $fuelCutFlag = OFF$
5: and $catalystTempHIGHflag = ON$ then
6:     if $2600.0 \leq engRpm \leq 2610.0$
7: and $89.0 \leq waterTemp \leq 91.0$ then
8:         $target \leftarrow 150.0$                ▷ injected fault
9:     else
10:        $target \leftarrow orignalTarget$        ▷ original code
11:    end if
12: end if
```

However, when combined with range mining using Breach, our methodology can be used to lift CBMC's counterexample to the system level. For this, we once again use Breach's falsification feature to find a violation of Property Eq. 2 where $\widehat{\boldsymbol{x}} = [90.0, 1.0, 6, 0, 0, 1, 2605.0]$.

The system-level counterexample is visualized in Fig. 2. The triangles show that the post-condition is violated at around 21.0 sec. This violation occurs through a sequence of events involving both continuous signals in the physical plant and changes in discrete variables. We trace the sequence of events backwards (see Fig. 2). For the post-condition to be violated, $engRpm$ must reach 2605.0 and $catalystTempHIGHflag$ must be set to True. The latter condition occurs when high temperature of exhaust gas are present, which occurs in turn when a high value of $pedalAngle$ and heavy engine load ($engRpm$) are kept on for a certain amount of time. Further, to reach $engRpm$[rpm] = 2605.0, the system must start from low rotation such as idle mode and start mode. In addition, the $gear$ must change in a specified pattern based on the current $gear$, $engRpm$ and the vehicle speed. Finding such a complex sequence of events involving physical plant signals and software variables requires an approach such as ours that analyzes the closed-loop system.

To summarize, our methodology combines the unit-level exhaustiveness of software model checking with the system-level scalability of simulation-driven requirement mining and falsification. The system-level counterexamples obtained greatly enhance the productivity with which issues arising the development process can be debugged and fixed. In our experience, this approach significantly eliminates the manual effort in finding good preconditions (20% of total person hours of an engineer trained in formal methods) and validating a counterexample (50% of total person hours).

V. Discussion

We conducted another set of experiments to check whether using simulation-driven falsification directly to violate the unit

TABLE II
Counterexamples from CBMC with and without range mining. "ce" indicates the counterexample values.

input variable	with mining		no mining
	range	ce	ce
$waterTemp$[°C]	[-30.0, 100.0]	90.0	89.4
$atmosphericPressure$[hPa]	[0.0, 1.0]	1.0	3.5
$gear$	[0,6]	6	5
$gearHoldFlag$	0	0	0
$idlFlag$	[0,1]	0	0
$catalystTempHIGHflag$	[0,1]	1	1
$fuelCutFlag$	[0,1]	0	0
$engRpm$[rpm]	[0.0, 5310.9]	2605.0	2600.0

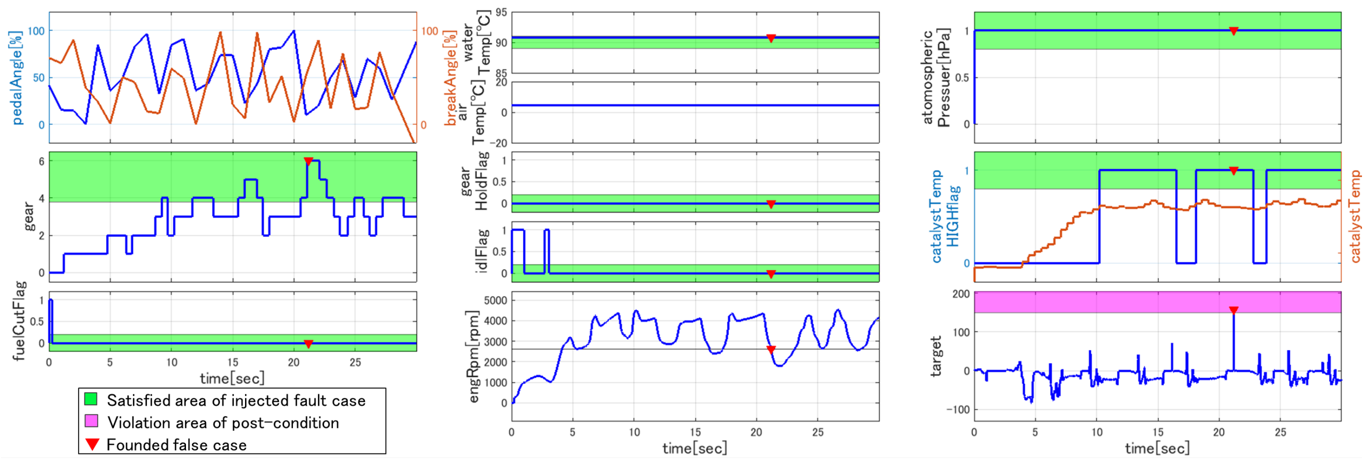


Fig. 2. System-level counterexample ("false case") on production powertrain system model (larger version in our technical report [12].)

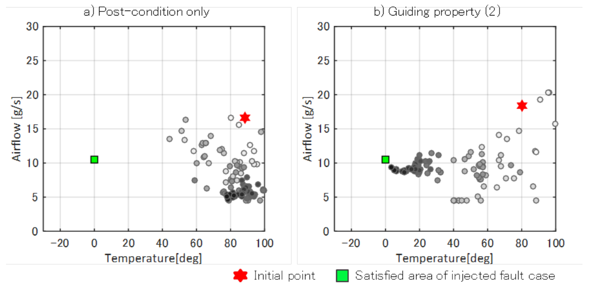


Fig. 3. Comparison between direct falsification of post-condition (left plot) and falsification guided using a counterexample and mined pre-conditions (right) for the AFC model. Each circle is an input found during falsification.

level post-condition, without the use of model checking, can be as effective as using software model checking first and then simulation to falsify Property Eq. 2. Specifically, we re-ran just the falsification step for 100 different trials on the AFC model with different initial input values, varying the property between one that tries to directly violate the post-condition and Property Eq. 2. Note that simulation-based falsification can be sensitive to the choice of initial input values, since it performs numerical optimization from this initial valuation.

We found that the combined approach could find the fault (and a system-level counterexample) 59.0% of the time, while a pure simulation-based approach could only find the fault 17.0% of the time. Further, as seen in Fig. 3, we see the visualization of one pair of trials for the different properties. The red star denotes the initial input valuation and the green box indicates the unit level counterexample to be hit. We can see that if we directly try to violate the post-condition, the optimizer gets stuck in a local minimum in the parameter space away from the fault region; whereas Property Eq. 2 is effective at guiding the search towards the unit level counterexample.

We also compared these options on the production power train model, and found that, on average, using Property Eq. 2 can find the injected malfunction faster than just using post-condition. The data is presented in our technical report [12].

In conclusion, in this paper we have shown that a combination of simulation-driven requirement mining, software model checking, and simulation-based falsification can be significantly more effective than using just software model checking or just simulation-based verification.

Going forward, we plan to expand the adoption of this methodology and also consider more complex requirements to be mined at the interface between the software components and the physical plant.

Acknowledgments

We thank the anonymous referees for their comments. We are grateful to Jyotirmoy Deshmukh, Xiaoqing Jin, James Kapinski, Hisahiro Ito, Arthur Wu and Ken Butts from Toyota Motor Engineering & Manufacturing North America, Inc. (TEMA) for their insightful comments and suggestions. We thank James Kapinski for providing the Abstract Fuel Control Model. We acknowledge the support on CBMC from Daniel Kroening, Martin Brain and Peter Schrammel. The UC Berkeley authors were supported in part by Toyota through the CHESS center.

References

[1] Edmund M. Clarke, Orna Grumberg, and Doron Peled. Model Checking. MIT Press, 2000.
[2] Donzé, A. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. CAV 2010: 167-170.
[3] ANNPUREDDY, Yashwanth, et al. S-taliro: A tool for temporal logic falsification for hybrid systems. Springer Berlin Heidelberg, 2011.
[4] Jin, X., Donzé, A., Deshmukh, J. V., & Seshia, S. A. Mining requirements from closed-loop control models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2015, 34.11: 1704-1717.
[5] JIN, Xiaoqing, et al. Powertrain control verification benchmark. In: Proceedings of the 17th international conference on Hybrid systems: computation and control. ACM, 2014. p. 253-262.
[6] Tomoya Yamaguchi, et al. A model checking application to software development of automobile control systems. Embedded System Symposium 2012, 2012. p. 188-196. (Japanese)
[7] Maler, Oded; Nickovic, Dejan; Pnueli, Amir. Checking temporal properties of discrete, timed and continuous behaviors. In: Pillars of computer science. Springer Berlin Heidelberg, 2008. p. 475-505.
[8] http://www.mathworks.com/products/sldesignverifier
[9] Kroening, Daniel and Tautschnig, Michael. CBMC bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2014. p. 389-391.
[10] http://www.mathworks.com/products/simulink
[11] Fukuoka Koji, et al. Development of CRAMAS-VF. In: Fujitsu Ten technical report, 2014, 31.1: 15-20.
[12] Tomoya Yamaguchi, Tomoyuki Kaga, Alexandre Donzé and Sanjit A. Seshia. Combining Requirement Mining, Software Model Checking, and Simulation-Based Verification for Industrial Automotive Systems. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-124, June 30, 2016