Proceedings of the 15th Conference on

Formal Methods in Computer-Aided Design (FMCAD 2015)

Austin, Texas, USA, September 27 - 30, 2015

Edited by Roope Kaivola and Thomas Wahl

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 15th Conference on

Formal Methods in Computer-Aided Design

FMCAD 2015

September 27 - 30, 2015

Austin, Texas, USA

Edited by Roope Kaivola and Thomas Wahl

ISBN: 978-0-9835678-5-1

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, in the broadest sense of the word. FMCAD covers the spectrum of formal aspects of specification, verification, synthesis, and testing, and is intended as a leading forum for researchers and practitioners in academia and industry alike. The fifteenth meeting in the series was held in Austin/Texas, USA, September 27–30, 2015.

FMCAD 2015 featured a rich program. In addition to regular paper presentations, the conference offered a tutorial day (joint with the SAT conference and the DIFTS workshop) on September 27. The tutorial topics included, in the order of appearance in the program,

- "Proving Hybrid Systems", by Andre Platzer (Carnegie Mellon University)
- "Formal Verification of Arithmetic Datapaths using Algebraic Geometry and Symbolic Computation", by Priyank Kalla (University of Utah)
- "Reactive Synthesis", by Roderick Bloem (Graz University of Technology)
- "Abductive Inference and Its Application in Program Analysis, Verification, and Synthesis", by Işil Dillig (University of Texas at Austin)

FMCAD 2015 further featured three keynote talks, again in order of appearance:

- "Democratization of Formal Verification with Collective Intelligence", by Ziyad Hanna (Cadence Design Systems)
- "Detecting Hardware Trojans: A Tale of Two Techniques", by Sharad Malik (Princeton University)
- "The Genesis and Development of Model Checking: Fact vs. Fiction", by Allen Emerson (University of Texas at Austin, *2007 ACM Turing Award winner*)

FMCAD also offered the third (and so far annual) edition of the Student Forum (organized by Georg Weissenbacher [Vienna University of Technology]; the Forum is described in more detail later in these proceedings); an industrial panel discussion on "Formal Verification in the Industry – a 2020 Vision", with participation of Per Bjesse (Synopsys), David Hardin (Rockwell Collins), Roope Kaivola (Intel; organizer), Naren Narasimhan (Calypto), Vigyan Singhal (OSKI), and Daryl Stewart (ARM); and finally a report on the 2015 Hardware Model Checking competition, organized and presented by Armin Biere (Johannes Kepler University Linz/Austria).

Those still not exhausted by this program could take advantage of several co-located events, namely the conferences MEMOCODE (Sept. 21-23) and SAT (Sept. 24-27), and the workshops DIFTS (Sept. 26-27) and ACL2 (Oct. 01-02).

FMCAD 2015 received about 75 abstracts at paper registration time, which materialized into 53 full submissions. Each full submission was reviewed by at least four program committee members. For a few submissions we solicited extra reviews, from within or outside the program committee. After the initial reviews were available, authors had the opportunity to respond to them in writing ("rebuttal"), to point out misunderstandings or errors. This opportunity was taken by almost all authors. After a thorough round of discussions among the program committee members, 21 submissions were selected for presentation at the conference. The themes ranged from traditional software verification topics such as Invariant Generation, over the development and use of SAT and SMT technology, to concurrency and protocol verification, and to circuit analysis and synthesis. Each paper was presented in a 30min conference talk.

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 (spare) time, dedication to the purpose of FMCAD, their willingness to help the authors improve their manuscripts, and their help with the selection of the various Best Paper Awards. Our sincere gratitude further goes out to the Publication Chair Ruzica Piskac (Yale University; in charge of these proceedings), and the Tutorial Chairs Malay Ganai (Atrenta) and Chao Wang (Virginia Tech). Georg Weissenbacher (Vienna University of Technology, Austria) 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 Shilpi Goel (University of Texas at Austin), formally the Local Arrangements Chair & Webmaster, a title that does not do justice to the immense range of tasks that Shilpi took care of, from managing the FMCAD website, to the organization of the Welcome Reception and the Banquet, and everything in between. Shilpi was assisted by Lindy Aleshire, an Administrative Associate at the University of Texas (UT) at Austin, in setting up the logistics within the UT venue. We are greatly indebted to Shilpi and Lindy for their help. 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 Jason Baumgartner (IBM).

We would like to express our gratitude to our industrial sponsors ARM Ltd., Cadence Design Systems, Inc., Centaur Technology, Inc., IBM Research, Intel Corporation, Mentor Graphics, Microsoft Research, and Real Intent, Inc. 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 2015 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 2015 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.

Finally, 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 2015, for playing their part in making FMCAD a continued success story.

Roope Kaivola and Thomas Wahl
FMCAD 2015 Program Chairs
Austin/TX, September 2015

Organization Committee

Program Co-Chairs

Roope Kaivola	Intel
Thomas Wahl	Northeastern University

Local Arrangement Chair & Webmaster

Shilpi Goel	The University of Texas at Austin

Publication Chair

Ruzica Piskac	Yale University

Tutorial Chairs

Malay Ganai	Atrenta
Chao Wang	Virginia Tech

Student Forum Chair

Georg Weissenbacher	Vienna University 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

Program Committee

Additional Reviewers

Table of Contents

Proving Hybrid Systems

André Platzer
Carnegie Mellon University

Abstract of Tutorial Talk

Cyber-physical systems (CPS) combine cyber aspects such as communication and computer control with physical aspects such as movement in space, which arise frequently in many safety-critical application domains, including aviation, automotive, railway, and robotics. But how can we ensure that these systems are guaranteed to meet their design goals, e.g., that an aircraft will not crash into another one?

This tutorial focuses on the most elementary CPS model: hybrid systems, which are dynamical systems with interacting discrete transitions and continuous evolutions along differential equations. It describes a compositional programming language for hybrid systems and shows how to specify and verify correctness properties of hybrid systems in differential dynamic logic. Extensions of this logic that support CPS models with more general dynamics will be surveyed briefly.

In addition to providing a strong theoretical foundation for CPS, differential dynamic logics have also been instrumental in verifying many applications, including the Airborne Collision Avoidance System ACAS X, the European Train Control System ETCS, several automotive systems, mobile robot navigation with the dynamic window algorithm, and a surgical robotic system for skull-base surgery. The approach is implemented in the theorem prover KeYmaera X.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Formal Verification of Arithmetic Datapaths using Algebraic Geometry and Symbolic Computation

Priyank Kalla
University of Utah

ABSTRACT OF TUTORIAL TALK

Algebraic geometry is the study of the geometry of solutions to a system of multivariate polynomial equations. Modern algebraic geometry does not explicitly solve the system of equations to enumerate the solutions, but rather reasons about the presence, absence, dimensions or intersection properties of the solution-sets, etc. Abstract and computational algebra is often used for this purpose – particularly the theory and technology of Gröbner bases, which provides a very powerful set of tools to solve many polynomial decision problems. In this talk, I will present a tutorial on how some of these techniques from algebraic geometry and commutative algebra can be used for formal verification of RTL datapaths and arithmetic circuits.

Datapath designs implement arithmetic computations over finite word-length operands, say, over k-bit vectors. These circuits implement functions that are mappings over k-dimensional Boolean spaces $f : \mathbb{B}^k \rightarrow \mathbb{B}^k$. Such functions can also be construed as mappings over: i) finite integer rings of the type $\mathbb{Z}_{2^k} \equiv \mathbb{Z} \pmod{2^k}$, i.e. as functions $f : \mathbb{Z}_{2^k} \rightarrow \mathbb{Z}_{2^k}$; or ii) as functions over the Galois field of 2^k elements, i.e. $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$. The designs can then be modeled as a system of polynomial functions over $\mathbb{Z}_{2^k}$ or $\mathbb{F}_{2^k}$, and Gröbner basis techniques can be applied for verification by reasoning about the solutions (functions) of the polynomial systems (circuits). Given the arithmetic nature of the designs, such an approach provides a natural *word-level abstraction* which can enable efficient verification.

While Gröbner basis techniques are very powerful, the computation suffers from high complexity. Therefore, the main focus of the tutorial will be on how to overcome this complexity. I will describe:

- How to formulate various verification problems using ideal membership, Nullstellensatz, elimination theory and Gröbner bases;
- How to exploit the number-theoretic properties of finite rings and fields to simplify the problems;
- How to analyze the structure/topology of the given circuits to get more theoretical insights into the corresponding polynomial ideals, and use this information to improve the computation; and
- How to implement the aforementioned concepts using modern symbolic computation algorithms, *e.g. Faugére's* F_4-style reductions, for practical datapath verification.

Arithmetic datapaths are usually custom designed, and they often exhibit some structure or symmetry in the implementations. Gröbner bases can help identify this inherent symmetry. By exploiting this information, efficient symbolic computation algorithms can then be devised for scalable verification.

The verification context will be motivated by applications such as elliptic curve cryptography, error correcting circuits, polynomial signal processing, word-level RTL synthesis, etc. I will provide information on various resources – publications, design benchmarks and the verification tools developed by us – so that interested participants can explore this exciting area of work. I will conclude by describing important unsolved problems in this specific area, and the challenges that need to be overcome to fully exploit the potential of the theory and technology.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Reactive Synthesis

Roderick Bloem
Graz University of Technology

Abstract of Tutorial Talk

Synthesis is the question of how to construct a correct system from a specification. In recent years, synthesis has made major steps from a theoretists dream towards a practical design tool. While synthesis from a language like LTL has very high complexity, synthesis can be quite practical when we are willing to compromise on the specification formalism. Similarly, we can take a pragmatic approach to synthesize small distributed systems, a problem that is in general undecidable.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Abductive Inference and Its Applications in Program Analysis, Verification, and Synthesis

Işil Dillig
The University of Texas at Austin

ABSTRACT OF TUTORIAL TALK

Abductive inference is a form of backwards logical reasoning that infers likely hypotheses from a given conclusion. In other words, given an invalid implication of the form A => B, abduction asks the question "What formula C do we need to conjoin with the antecedent A so that (i) A & C => B is logically valid and (ii) C is consistent with A?" Abductive reasoning has found many applications in program verification and synthesis, particularly in modular program analysis, invariant generation, and automated inference of missing program expressions. This tutorial will give an overview of logical abduction and algorithms for performing abductive inference. We will also survey several use cases of abductive inference in the context of program analysis, verification, and synthesis.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Democratization of Formal Verification with Collective Intelligence

Ziyad Hanna
Cadence Design Systems

Abstract of Invited Talk

Formal verification has become an essential method for design, integration and verification of the emerging design IPs and complex systems-on-chips. Despite the great success in proliferating formal in the industry and making it a great companion to simulation and emulation methods, its full power has not been leveraged yet to address the larger spectrum of applications by non-formal experts. To handle this challenge a major effort is still needed to boost its scalability and usability to cope with the emerging complexity of design under verification. In this talk we discuss a new approach for boosting the productivity of formal verification users using an expert system, which is powered by collective intelligence technology, human-machine interface and self guiding and learning rules. Besides the emerging advancements in model checking and proof strategies, expert system helps to break the formal verification complexity and scalability barriers, and make it affordable for a larger set of users, for yet another major leap in the applications and productivity of formal.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Detecting Hardware Trojans: A Tale of Two Techniques

Sharad Malik
Princeton University

ABSTRACT OF INVITED TALK

Integrated Circuits (ICs) are designed and fabricated in a globalized multi-vendor environment making them vulnerable to malicious design changes and the insertion of hardware Trojans/malware. In this talk I will cover two distinct techniques to address the problem of detecting hardware Trojans. The first uses SAT and BDD-based functional analysis to reverse engineer ICs. The goal here is to derive the higher- level function of IC through algorithmic analysis of its netlist to help expose the Trojan logic. The second uses statistical analysis of chip simulation data in a clustering algorithm to isolate the Trojan logic. I will discuss these techniques, their practical application on benchmark circuits and their complementary strengths.

This is joint work with Burcin Cakir and Pramod Subramanyan.

The Genesis and Development of Model Checking: Fact vs. Fiction

Allen Emerson
The University of Texas at Austin

Abstract of Invited Talk

Clarke and Emerson are noted for the invention and development of model checking [1]. In this presentation, I will recall the roles of the principals. I was responsible for the initial conception of model checking, as a spinoff of my work on program synthesis. This entailed the checking of candidate models of a temporal specification algorithmically, and explains the coining of the term model checking. Model checking plainly was a form of verification circumventing proofs. As a part of some consulting work, Ed Clarke had been consulting on verification of protocols using temporal specifications and deductive proofs. As I discussed model checking with Ed, he envisioned that it could be a really practical verification method, applicable to protocols and more. Later, Ed commented that neither one of us could have achieved model checking without the other.

There are alternative perspectives on this history. In broad brush, these agree, but they vary in detail. I will elaborate with additional crucial facts, and examine their impact on various apparent discrepancies. I would like to suggest the relevance of Pnueli's Principle here: The junior researcher lives in the problem, while the senior researcher lives in a constant stream of interrupts. A Corollary is: The junior researcher will singlemindedly acquire a detailed memory concerning the problem and its solution; The overburdened, timesharing senior researcher will lack sufficient focused attention, blocks of time, and depth of understanding to have good recall.

References

[1] Edmund M. Clarke and E. Allen Emerson: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. Logics of Programs, Workshop, Yorktown Heights, New York, May 1981.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

The FMCAD 2015 Graduate Student Forum

Georg Weissenbacher
TU Wien, Austria

***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 2015, the event took place in Austin, Texas, as integral part of the FMCAD conference. Sixteen students were invited to give a short talk and present a poster illustrating their work. The presentations covered a broad range of topics in the field of verification, such as automated reasoning, model checking of hardware, software, as well as hybrid systems, verification of concurrent programs, and checking of security properties.**

Since 2013, the FMCAD conference features a Student Forum, providing a platform for graduate students at any career stage to introduce their research to the wider Formal Methods community. The FMCAD 2015 Graduate Student Forum follows the tradition of its predecessors, which took place in Lausanne, Switzerland in 2014 [1] and in Portland, Oregon, USA in 2013 [2].

Graduate students were invited to submit short reports describing their ongoing research in the scope of the FMCAD conference. We received 21 submissions, covering novel technical contributions and outlining future research planned by the authors. The presentations covered a broad range of topics in the field of verification, such as automated reasoning, model checking of hardware, software, as well as hybrid systems, verification of concurrent programs, and checking of security properties. While all contributions were of high quality, the limitations of the venue and conference schedule forced us to reject some of the submissions. Based on the reviews provided by members of the organizing committee as well as a number of external reviewers, 16 submissions were finally 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:

- Konstantinos Athanasiou: *A Constraint-Based Approach to Multi-Threaded Program Location Reachability*
- Peter Backeman and Aleksandar Zeljic: *Approximations for Deciding Quantified Floating-Point Constraints*
- Cuong Chau: *ACL2(r) Formalization of Fourier Series' Properties*
- Shaobo He: *Towards Automated Differential Program Verification For Approximate Computing*
- Egor Karpenkov and David Monniaux: *Program Analysis with Local Policy Iteration*
- Guy Katz and David Harel: *Concurrency Idioms and their Effect on Program Analysis*
- Shou-Pon Lin and Nicholas Maxemchuk: *Probabilistic Model Checking of Systems with a Large State Space: A Stratified Approach*
- Rajdeep Mukherjee: *How Efficient are Software Verifiers for Hardwares?*
- Luan Nguyen and Taylor T Johnson: *Towards Bounded Model Checking for Timed and Hybrid Automata with a Quantified Encoding*
- Daniel Poetzl: *Efficient Checking of Thread Refinement*
- Moritz Sinn: *Bound Analysis of Heap-Manipulating Programs*
- Pramod Subramanyan: *Specification and Scalable Verification of Security Properties in Contemporary SoCs*
- Jiaqi Tan, Rajeev Gandhi and Priya Narasimhan: *White-box Software Isolation with Fully Automated Black-box Proofs*
- Danilo Vendraminetto: *Exploiting Craig Interpolants in Unbounded Model Checking of Hardware Designs*
- Vadim Zaliva and Franz Franchetti: *Formal Verification of HCOL Rewriting*
- Lu Zhang: *Classifying Race Conditions in Web Applications*

The 2015 student forum is also the first 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]

The Student Forum would not have been possible without the excellent contributions of the student authors. The generous support of the National Science Foundation and FMCAD's sponsors enabled us to subsidize the travel cost of the participating students. The help and advice of Thomas Wahl and Ruzica Piskac, who organized the first two student forums, as well as Roope Kaivola and Shilpi Goel, who took care of all local arrangements, was crucial to the success of the event. We are also indebted to Ezio Bartocci and Vijay D'Silva for their help with the reviews.

References

[1] R. Piskac, "The FMCAD 2014 graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2014, p. 13.
[2] 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/FMCAD15/

Verification of Cache Coherence Protocols wrt. Trace Filters

Parosh Aziz Abdulla*, Mohamed Faouzi Atig*, Zeinab Ganjei†, Ahmed Rezine† and Yunyun Zhu*
**Department of Information Technology*
Uppsala University
Uppsala, Sweden
†*Deptartment of Computer and Information Science*
Linköping University
Linköping, Sweden

Abstract—**We address the problem of parameterized verification of cache coherence protocols for hardware accelerated transactional memories. In this setting, transactional memories leverage on the versioning capabilities of the underlying cache coherence protocol. The length of the transactions, their number, and the number of manipulated variables (i.e., cache lines) are parameters of the verification problem. Caches in such systems are finite-state automata communicating via broadcasts and shared variables. We augment our system with filters that restrict the set of possible executable traces according to existing conflict resolution policies. We show that the verification of coherence for parameterized cache protocols with filters can be reduced to systems with only a finite number of cache lines. For verification, we show how to account for the effect of the adopted filters in a symbolic backward reachability algorithm based on the framework of constrained monotonic abstraction. We have implemented our method and used it to verify transactional memory coherence protocols with respect to different conflict resolution policies.**

1. Introduction

The behavior of many types of systems can be described using one or more *parameters* such as the number of processes, or the number of variables that may be used in a given run of the system. Parameterized systems are ubiquitous and serve as natural models of mutual exclusion algorithms, bus protocols, distributed algorithms, telecommunication protocols, and cache coherence protocols. The goal of *parameterized verification* is to prove (or refute) the correctness of the system for all values of the parameters. For instance, in a cache coherence protocol, copies of a variable may exists in an arbitrary number of caches. It is then relevant to verify exclusive ownership of the cache line regardless of the number of caches in a particular session of the protocol. The state space of a such a system is infinite since we are dealing with an unbounded number of instances, namely one for each size.

Several techniques for the verification of parameterized systems have been developed during the last two decades [1], [2], [3], [4], [5]. One approach, related to this paper, is *monotonic abstraction* [6]. It defines an abstraction that allows to apply the framework of well quasi-ordered systems (wqo for short) [7] and based on backward reachability analysis in order to perform parameterized verification. Monotonic abstraction has been successfully applied to several non-trivial examples of mutual exclusion, leader election, and cache coherence protocols.

This paper addresses parameterized verification of transactional memory cache coherence protocols. Such protocols are not expected to guarantee coherence under arbitrary sequences of transitions. However, coherence should be guaranteed for all sequences that respect the transactional memory. Transactional memories usually make use of conflict tables in order to track read/write and write/write conflicts at a cache line granularity. Detected conflicts can be resolved according to different policies. For instance, in an eager policy, the conflict is resolved by aborting a transaction as soon as the conflict is detected. In a lazy policy, the resolution can wait until the commit before deciding on which transaction to abort.

Since the numbers of transactions, caches and cache lines are arbitrary, we need to consider systems that are parameterized in multiple dimensions. Furthermore, conflict policies can in general not be definable by finite-state automata since they quantify over the sets of threads and variables both of which are unbounded. Hence, parameterized verification of such systems is beyond the applicability of existing techniques. In this work, we present for the first time a method for automatic verification of cache coherence in the presence of transactional memories. We capture the conflict resolution mechanism, one for each policy, using so called *filters*, each of which is a set of forbidden "patterns". All traces of the system that do not match the patterns are allowed to occur. For instance, an eager conflict resolution will forbid traces where two different transactions continue running although a write/write conflict has been detected. Given a filter, we check reachability for the cache coherence protocol under the constraints imposed by the filter. For this we proceed in two steps. First, we give a small model theorem establishing that if coherence is violated then it is also violated using only a fixed small number of cache lines. Then we perform backward reachability analysis by

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

modifying classical monotonic abstraction by accounting for information from the filters in order to exclude traces that are eliminated by the conflict resolution mechanism. We show that this is possible for the class of filters we use, and establish termination of the analysis.

We have implemented our approach and managed to show, for arbitrarily many caches on which arbitrary transactions are repeatedly run, that transactional memories such as FlexTM and DynTM with their proper cache coherence protocol extensions cannot violate coherence.

Related Work. To the best of our knowledge, this is the first work that considers parameterized verification of cache protocols in the presence of conflict policies.

Regular model checking [8], [9] performs parameterized verification by encoding the set of configurations using finite-state automata. The method has been augmented with techniques such as widening [10], [11], abstraction [12], and acceleration [13].

There are numerous techniques less general than regular model checking, but that are lighter and more dedicated to the problem of parameterized verification. The idea of *counter abstraction* is to keep track of the number of processes which satisfy a certain property [14], [15], [16], [17]. In general, counter abstraction is designed for systems with unstructured or clique architectures, but may be used for systems with other topologies too [18].

Several works reduce parameterized verification to the verification of finite-state models. Among these, the *invisible invariants* method [19], [20] and the work of [21] exploit cut-off properties to check invariants for mutual exclusion protocols.

Monotonic abstraction [6], [22], [23] combines regular model checking with abstraction in order to produce systems that have monotonic behaviors wrt. a wqo on the state-space.

Methods relying on dynamic detection of cutoff conditions are described in [1] and [24].

2. Motivating example

We use a hardware accelerated transactional memory in order to describe the different steps we use to establish coherence in the presence of execution filters.

An example of a hardware accelerated transactional memories. FlexTM [25] is a hardware accelerated transactional memory that orchestrates the execution of concurrent transactions by only allowing a subset of the possible traces (this subset includes the strictly serializable ones [26]). A finite but arbitrary number of caches participate in such executions. At most one transaction is run on each cache. Transactions can access arbitrarily many cache lines. The lines that do not fit in the caches are handled in software, and hence do not affect the cache coherence. We assume, to simplify the presentation, that the caches are large enough to hold all lines accessed by the transactions. Each transaction consists in some arbitrary sequence of read and write instructions on an arbitrary number of cache lines[1]. At any moment, transactions are either pending, committed or aborted. FlexTM tracks all transactions and might decide to abort a transaction based on some conflict resolution policy (e.g., lazy or eager). A transaction can therefore be aborted at any time, in which case a new arbitrary transaction might be started.

Like other hardware accelerated transactional memories [25], [27], FlexTM builds on the inherent versioning capabilities of the underlying cache coherence protocol. In the case of FlexTM, the MESI [28] protocol is extended. Schematically, FlexTM makes use of an extension of the MESI cache protocol, called TMESI [25] in order to maintain tentative versions of the accessed cache lines. In this protocol, a cache line can be in one of the states in $\{\mathtt{I}, \mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}, \mathtt{TMI}\}$. The four first states are the usual MESI states: Invalid, Shared, Exclusive and Modified. The last two ones are FlexTM additions. `TMI` and `TI` respectively correspond to a tentative written copy or to a read copy that is threatened by a tentative write by another transaction.

Table 1 depicts a run of two transactions reading and writing to cache lines l and l'. In the first transition (t_1), a read instruction from an invalid cache line (state `I`) results in an exclusive state `E`. We will say that the cache line "takes" the transition and changes its state from `I` to `E`. This transition is enabled if the state of the same line in all other caches is `I`. This appears in the transition because we forbid all other states using $f[\mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}, \mathtt{TMI}]$.

The second transition (t_2) is also a read. Here, a cache line that takes the transition moves from the invalid state to the shared one. This transition requires that at least another cache has the same line at a shared, exclusive, modified or threatened state (hence the $r[\mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}]$). In addition, the transition is not enabled if another cache associates the same line to `TMI` (hence the forbid $f[\mathtt{TMI}]$). If enabled, the transition t_2 performs a broadcast where it moves all exclusive or modified states to shared (hence the $b[(\mathtt{E}, \mathtt{S})(\mathtt{M}, \mathtt{S})]$). Except for the line firing the transition, a broadcast keeps all non mentioned lines unmodified. For instance, in this transition, the states of all `I` lines remain unchanged.

Transitions t_1, t_2, t_3, t_4 are said to be *horizontal transitions* because they focus on a particular cache line in all caches. More concretely, a cache line that "takes" such a transition changes state if there is at least another cache where the same line is at a state specified by the $r[\ ]$ part and if none of the other caches associates the same line to one of the states mentioned in $f[\ ]$. In this case, the line that takes the transition changes its state and moves the state of the same line in all other caches as described in the broadcast part $b[\ ]$.

Some transitions are said to be *vertical transitions* when they focus on all the lines of the same cache (as opposed to the same line in all caches). In FlexTM, commit and aborts correspond to vertical transitions. When a transaction is aborted (t_5), all lines in the cache running the transaction

1. Of course, transactions read and write variables, but as far as the cache protocol is concerned, these are tracked at a cache line granularity.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

$$t_1 = \left(\mathtt{I} \xrightarrow[read]{r[\ ]\ f[\mathtt{S,E,M,TI,TMI}]\ b[\ ]} \mathtt{E} \right)$$

$$t_2 = \left(\mathtt{I} \xrightarrow[read]{r[\mathtt{S,E,M,TI}]\ f[\mathtt{TMI}]\ b[(\mathtt{E,S})(\mathtt{M,S})]} \mathtt{S} \right)$$

$$t_3 = \left(\mathtt{I} \xrightarrow[write]{r[\ ]\ f[\ ]\ b[(\mathtt{S,I})(\mathtt{E,I})(\mathtt{M,I})]} \mathtt{TMI} \right)$$

$$t_4 = \left(\mathtt{I} \xrightarrow[read]{r[\mathtt{TMI}]\ f[\ ]\ b[\ ]} \mathtt{TI} \right)$$

$$t_5 = \left(\bullet \xrightarrow[abort]{b[(\mathtt{TMI,I})(\mathtt{TI,I})]} \bullet \right)$$

$$t_6 = \left(\bullet \xrightarrow[commit]{b[(\mathtt{TMI,M})(\mathtt{TI,I})]} \bullet \right)$$

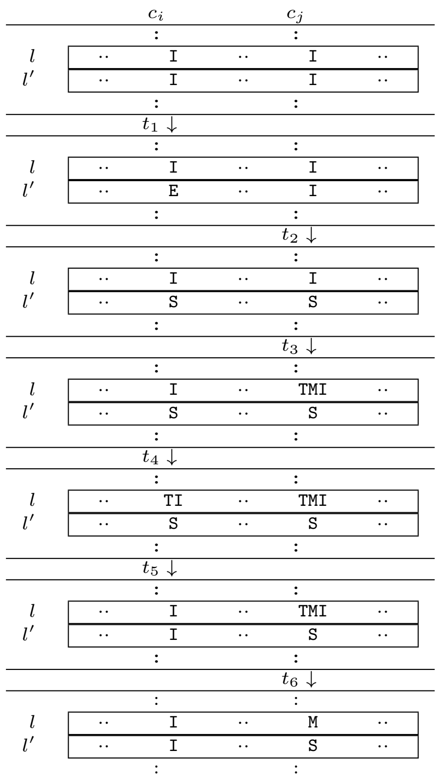


TABLE 1: A possible FlexTM [25] run is depicted to the right. At least two transactions are running on the caches c_i and c_j. In this execution, the c_i transaction tm_i reads line l', the c_j transaction tm_j reads line l' and writes line l, then tm_i reads l and is aborted by FlexTM before tm_j commits. This results in the TMESI transitions $t_1, \ldots t_6$ listed to the left.

that is to be aborted are invalidated. In a commit transition (t_6) all TMI lines are changed to M, and all TI lines are invalidated.

Coherence for transactional memory cache protocols. It turns out that cache coherence is violated if no restrictions are imposed on the sequences of horizontal (i.e., read and write) and vertical (i.e., abort and commit) transitions. For instance, assume that two transactions start running on caches c_1 and c_2 from a cache configuration where the line l is mapped to I in both caches (written $(\mathtt{I}, \mathtt{I})$). The sequence $(write, l, c_1)(write, l, c_2)(commit, c_1)(commit, c_2)$ where both transactions write the same l line and commit would result in executing transitions t_3, t_3, t_6 and t_6 by, respectively, caches c_1, c_2, c_1 and c_2. This sequence translates, for the l cache line, into the following states:

$$(\mathtt{I}, \mathtt{I}) \xrightarrow{write,l,c_1} (\mathtt{TMI}, \mathtt{I}) \xrightarrow{write,l,c_2} (\mathtt{TMI}, \mathtt{TMI})$$

$$(\mathtt{TMI}, \mathtt{TMI}) \xrightarrow{commit,c_1} (\mathtt{M}, \mathtt{TMI}) \xrightarrow{commit,c_2} (\mathtt{M}, \mathtt{M})$$

Coherence is violated in the last cache configuration. This is because the same cache line is mapped to the modified state M in two different caches. Intuitively, such configurations are bad because it is not clear which version to use if a transaction was to read a value as two possibly different versions coexist.

As it happens, FlexTM forbids such bad traces, based on some conflict resolution policy, by aborting transactions if certain conflicts arise.

In this work, we aim to show coherence in the presence of conflict resolution policies. Observe that the numbers of transactions, caches and cache lines are arbitrary. In other words, we are tackling coherence in the presence of conflict resolution policies for systems that are parameterized in the number of transactions, caches and cache lines.

Capturing transactional memory policies. FlexTM makes use of conflict tables in order to track read write and write write conflicts at a cache line granularity. Detected conflicts can be resolved according to different policies. For instance, in an eager policy, the conflict is resolved by aborting a transaction as soon as the conflict is detected. In a lazy policy, the resolution can wait until the commit before deciding on which transaction to abort.

We are interested in cache coherence in this work. We capture the conflict resolution mechanism using what we call *filters*. These consist in simple "patterns" that are going to be forbidden by the conflict resolution mechanisms. All traces that do not match the patterns are allowed. There will be simple patterns for each conflict resolution policy. Soundness requires that the patterns we use do not eliminate traces allowed by FlexTM. For instance, an eager conflict resolution will forbid traces where two different transactions continue running although a write write conflict has been detected.

Given such filters, we check reachability on the product of the cache coherence protocol and the filter and establish coherence for arbitrary transactions running on arbitrarily many caches and involving arbitrarily many cache lines.

3. Preliminaries

Let $\mathbb{N}$ denote the set of natural numbers. Given two natural numbers $i, j \in \mathbb{N}$, we use $[i, j]$ to denote the set $\{k \in \mathbb{N} \mid i \leq k \leq j\}$. For sets A and B, we use $f : A \mapsto B$ to denote that f is a function that maps any element from A to an element of B. Let $[A \mapsto B]$ denote the set of all functions from A to B. For $a \in A$ and $b \in B$, we use $f[a \leftarrow b]$ to denote the function f' where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$. For a set $A' \subseteq A$, we use $f(A')$ to denote the set $\{f(a) \mid a \in A'\}$.

For a set Σ, we use Σ^* to denote the set of finite words over Σ. We use ϵ to denote the empty word. For a word $w \in \Sigma^*$, we use $|w|$ to denote its length (observe that $|\epsilon| = 0$). For $1 \leq i \leq |w|$, we use $w[i]$ to denote the letter at position i in w.

Let Θ be a subset of Σ. Given two words w and w', we define $w \sqsubseteq_\Theta w'$ to denote that there is a function $h : [1, |w|] \mapsto [1, |w'|]$ such that: (1) for every $i, j \in [1, |w|]$ such that $i < j$, $h(i) < h(j)$, (2) for every $i \in [1, |w|]$, $w'[h(i)] = w[i]$, and (3) $\{i \mid w'[i] \in \Theta\} \subseteq h([1, |w|])$.

4. Parameterized Cache Protocols with Filters

In this section, we introduce a formal model for parameterized cache protocols with filters, and define their coverability problem.

4.1. Parameterized Cache Protocols

A parameterized cache protocol consists of an arbitrary (but finite) number of caches. Each cache is a finite-state system manipulating an arbitrary (but finite) set of cache lines. Each cache can perform two kinds of operations: (1) *vertical* actions that only affect the states of the lines of one single cache, and (2) *horizontal* actions that affect the states of the same line but for different caches.

Formally, a parameterized cache protocol $\mathcal{P}$ is a tuple $(Q, A, \Delta, q_{\texttt{init}})$ where Q is a finite set of states, A is a finite set of actions partitioned into two sets: the set of *vertical* actions $A_{\texttt{ver}}$ and the set of *horizontal* actions $A_{\texttt{hor}}$, $q_{\texttt{init}} \in Q$ is the initial state, and Δ is a finite set of transitions. A transition can be of one of the following two forms: (1) $q \xrightarrow[a_{\texttt{hor}}]{r[Q_1]\ f[Q_2]\ b[(q_1,q'_1),\ldots,(q_m,q'_m)]} q'$ or (2) $\bullet \xrightarrow[a_{\texttt{ver}}]{b[(q_1,q'_1),\ldots,(q_m,q'_m)]} \bullet$ where: (i) q, q' in Q are cache line states, (ii) $a_{\texttt{ver}}$ is a vertical action in $A_{\texttt{ver}}$ and $a_{\texttt{hor}}$ is a horizontal action in $A_{\texttt{hor}}$, (iii) $Q_1 \subseteq Q$ is the set of *existentially required* states, (iv) $Q_2 \subseteq Q$ is the set of *universally forbidden* states, and (v) the sequence of pairs $(q_1, q'_1), \ldots, (q_m, q'_m) \in Q \times Q$, such that $q_i \neq q_j$ for all $i \neq j$, corresponds to a broadcast.

Let C be a finite set of caches and L be a finite set of cache lines. We write c to mean a cache in C and l to mean a cache line in L. A configuration ν over (C, L) is a mapping $\nu : C \mapsto [L \mapsto Q]$. We write $\nu_{(C,L)}$ to make the sets of caches and lines explicit. We use $\texttt{cachesOf}(\nu_{(C,L)})$ and $\texttt{linesOf}(\nu_{(C,L)})$ to respectively mean C and L. Let $\nu^{init}_{(C,L)}$ denote the configuration that associates $q_{\texttt{init}}$ to all cache lines, i.e., $\nu^{\texttt{init}}_{(C,L)}(c)(l) = q_{\texttt{init}}$ for all $c \in C$ and $l \in L$.

Let $A^{\texttt{ext}}_{\texttt{ver}} = (A_{\texttt{ver}} \times C)$ and $A^{\texttt{ext}}_{\texttt{hor}} = (A_{\texttt{hor}} \times C \times L)$ respectively be the sets of extended vertical and horizontal actions over (C, L). Let $A^{\texttt{ext}} = A^{\texttt{ext}}_{\texttt{ver}} \cup A^{\texttt{ext}}_{\texttt{hor}}$ be the set of extended actions. Given an extended action $\mathbf{a}$ of the form (a, c, l) or (a, c), we let $\texttt{cacheOf}(\mathbf{a})$ mean the associated cache c.

Let ν and ν' be two configurations over (C, L). Let $\mathbf{a} \in A^{\texttt{ext}}$ be an extended action. We use $\nu \xrightarrow{\mathbf{a}}_{(C,L)} \nu'$ to denote that one of the following cases holds:

Case 1: $\mathbf{a} = (a, c)$ for some vertical action $a \in A_{\texttt{ver}}$ and cache $c \in C$, and there is a transition $t \in \Delta$ of the form $\bullet \xrightarrow[a_{\texttt{ver}}]{b[(q_1,q'_1),\ldots,(q_m,q'_m)]} \bullet$ such that the following conditions are satisfied:

- For every cache line $l \in L$ such that $\nu(c)(l) = q_i$ for some $i \in [1, m]$, we have $\nu'(c)(l) = q'_i$. This corresponds to a transition resulting from a vertical action that changes the state of each cache line at q_i to q'_i.
- For every cache line $l \in L$ such that $\nu(c)(l) \notin \{q_i | i \in [1, m]\}$, we have $\nu'(c)(l) = \nu(c)(l)$, i.e., all the remaining cache lines keep their states.
- For every cache $c' \in C$ such that $c' \neq c$, we have $\nu'(c') = \nu(c')$, i.e., states of lines belonging to other caches remain unchanged.

Case 2: $\mathbf{a} = (a, c, l)$ for some $a \in A_{\texttt{hor}}$, $c \in C$ and $l \in L$, and there are a transition $t \in \Delta$ of the form $q \xrightarrow[a]{r[Q_1]\ f[Q_2]\ b[(q_1,q'_1),\ldots,(q_m,q'_m)]} q'$, and a cache $c' \in C$, with $c \neq c'$, such that the following conditions are satisfied:

- $\nu(c)(l) = q$ and $\nu'(c) = \nu(c)[l \leftarrow q']$. The state of line l of the cache c changes from q to q'.
- $\nu(c')(l) \in Q_1$. This condition corresponds to the existential requirement. It states that the line l of at least another cache c' belongs to Q_1.
- $\nu(c'')(l) \notin Q_2$ for all $c'' \in C \backslash \{c, c'\}$. This condition corresponds to the universal requirement. It states that none of the lines l belonging to any cache other than c and c' is in Q_2.
- Any cache $c'' \in C \setminus \{c\}$ such that $\nu(c'')(l) = q_i$ for some $i \in [1, m]$, will change the state of l according to $\nu'(c'') = \nu(c'')[l \leftarrow q'_i]$. This corresponds to a horizontal broadcast where the state of the line l in any other cache is changed from q_i to q'_i.

- All other lines remain unchanged. In other words, for all caches $c'' \in C \setminus \{c\}$ with $\nu(c'')(l) \notin \{q_i | i \in [1,m]\}$ we have $\nu'(c'') = \nu(c'')$.

A trace $\sigma \in (A^{\mathtt{ext}})^*$ over (C,L) is a sequence of *extended* actions. We use $\nu \xrightarrow{\sigma}_{(C,L)} \nu'$ to denote that one of the following two cases hold: (1) $\sigma = \epsilon$ and $\nu = \nu'$, or (2) there is a sequence of configurations $\nu_0, \ldots, \nu_n$ over (C,L) such that $\nu_0 = \nu$, $\nu_n = \nu'$, and for every $i \in [0, n-1]$, we have $\nu_i \xrightarrow{\mathbf{a}_i}_{(C,L)} \nu_{i+1}$ with $\sigma = \mathbf{a}_0\mathbf{a}_1 \cdots \mathbf{a}_{n-1}$. In this case, we say that the configuration ν' is reachable from ν. Finally we say that the configuration ν' is reachable if it is reachable from $\nu^{\mathtt{init}}_{(C,L)}$.

4.2. Filter Model

Let C be a finite set of caches and L be a finite set of cache lines. A pattern π over (C,L) is a finite sequence in $(A^{\mathtt{ext}})^*$ of extended actions. We define a filter over (C,L) to be a finite set of *forbidden* patterns over (C,L).

Let C' be a set of caches and L' be a set of cache lines. Let σ be a trace over (C',L'). Let us assume that $\pi = \mathbf{a}_1\mathbf{a}_2 \cdots \mathbf{a}_n$ and $\sigma = \mathbf{b}_1\mathbf{b}_2 \cdots \mathbf{b}_m$. We say that the pattern π appears in σ (denoted by $\sigma \models \pi$) if and only if there are injective functions $\phi : C \mapsto C'$, $\psi : L \mapsto L'$ and $h : [1,n] \mapsto [1,m]$ such that:

- For every $i,j \in [1,n]$ such that $i < j$, $h(i) < h(j)$.
- For every $i \in [1,n]$, we have $\sigma[h(i)] = (a_i, \phi(c_i), \psi(l_i))$ if $\pi[i]$ is of the form (a_i, c_i, l_i) and $\sigma[h(i)] = (a_i, \phi(c_i))$ if $\pi[i]$ is of the form (a_i, c_i).
- For every $i \in [1,n]$ such that $\mathbf{a}_i$ is of the form (a_i, c_i, l_i) and there is an index j such that $i < j$ and $\mathtt{cacheOf}(\mathbf{a}_j) = c_i$, we have $\mathbf{b}_k \notin (A_{\mathtt{ver}} \times \phi(c_i))$ for all $h(i) < k < h(j')$ with j' is the minimal index such that $i < j'$ and $\mathtt{cacheOf}(\mathbf{a}_{j'}) = c_i$.

A filter F over (C,L) is a finite set of *forbidden* patterns over (C,L). We say that a trace σ over (C',L') is *valid* with respect to a filter F if and only if $\sigma \not\models \pi$ for all $\pi \in F$.

4.3. Coverability Problem

Let ν and ν' be two configurations respectively over (C,L) and (C',L'). Let $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ be two injective functions. We use $\nu \preceq_{(\phi,\psi)} \nu'$ to denote that for every cache $c \in C$ and every line $l \in L$, we have $\nu'(\phi(c))(\psi(l)) = \nu(c)(l)$. We use $\nu \preceq \nu'$ to denote that there are two injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ such that $\nu \preceq_{(\phi,\psi)} \nu'$. Intuitively, this means that ν (modulo renaming of the caches and lines) is the restriction of ν' to the subsets of caches $\phi(C) \subseteq C'$ and lines $\psi(L) \subseteq L'$.

Let $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$ be a parameterized cache coherence protocol and F be a filter over a set of caches C and a set of lines L. The coverability problem for $\mathcal{P}$ with respect to the filter F and a configuration ν over (C,L), consists in checking whether there is a configuration ν' over (C',L'), with $\nu \preceq \nu'$, such that $\nu^{\mathtt{init}}_{(C',L')} \xrightarrow{\sigma}_{(C',L')} \nu'$ for some trace σ over (C',L') with $\sigma \not\models \pi$ for any $\pi \in F$.

5. Small Model Theorem

In this section, we show that it is possible to restrict the analysis of the coverability problem for parameterized cache protocols to the subclass where only finite number of variables are used. Let $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$ be a parameterized cache protocol. We will first introduce some notations.

Notations. Let C and C' be two sets of caches and L and L' be two sets of cache lines. Given two injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$, we use $\sigma[\phi,\psi]$ to denote the trace σ' over (C',L') such that $|\sigma'| = |\sigma|$ and for every $i \in [1, |\sigma'|]$, $\sigma'[i] = (a, \phi(c), \psi(l))$ if $\sigma[i] = (a,c,l)$, and $\sigma'[i] = (a, \phi(c))$ if $\sigma[i] = (a,c)$. Given a trace τ over (C',L') We use $\tau[\phi^-,\psi^-]$ to denote the set of traces τ' over (C,L) such that $\tau'[\phi,\psi] \sqsubseteq_{(A_{\mathtt{ver}} \times \phi(C))} \tau$. Intuitively, τ' corresponds to some trace obtained from τ by only deleting some horizontal actions and renaming caches and lines.

In the following, we will establish two closure properties of the considered cache protocols.

Closure property of the cache protocol. Our first property concerns the parameterized cache protocol. Intuitively, we show that if a configuration ν' is reachable and ν' is larger than a configuration ν (w.r.t. the ordering $\preceq$) then ν is also reachable.

Lemma 1. *Let C and C' be two sets of caches such that $|C| = |C'|$. Let L and L' be two sets of cache lines such that $|L| \leq |L'|$. Let ν be a configuration over (C,L) and ν' be a configuration over (C',L'). If $\nu^{\mathtt{init}}_{(C',L')} \xrightarrow{\sigma'}_{(C',L')} \nu'$ for some trace σ' over (C',L') and $\nu \preceq_{(\phi,\psi)} \nu'$ for some injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$, then $\nu^{\mathtt{init}}_{(C,L)} \xrightarrow{\sigma}_{(C,L)} \nu$ with $\sigma \in \sigma'[\phi^-,\psi^-]$.*

Closure property of the filter. Our second property concerns the filter. We show that if a trace σ' is valid wrt. a filter then any trace which is an *extended-vertical-actions-preserving-subword* (modulo renaming of the caches) of σ' is also valid wrt. the filter.

Lemma 2. *Let C and C' be two sets of caches such that $|C| \leq |C'|$. Let L and L' be two sets of cache lines such that $|L| \leq |L'|$. Let $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ be two injective functions. Let σ' be a valid trace over (C',L') with respect to a given filter F. Then every trace $\sigma \in \sigma'[\phi^-,\psi^-]$ is valid with respect to the filter F.*

Bounding the number of cache lines. We are now ready to state our main theorem which is a consequence of Lemma 1 and Lemma 2. Intuitively, we will show that checking the coverability problem for parameterized cache protocols can be restricted to instances where the number of cache lines is bounded.

Theorem 3. *Let F be a filter over a set of caches C and a set of cache lines L. Let ν^{bad} be a configuration over (C,L). Let C' be a set of caches and L' be a set of cache lines. If $\nu^{\mathtt{init}}_{(C',L')} \xrightarrow{\sigma'}_{(C',L')} \nu'$ for some valid trace σ' with respect to F and $\nu^{\mathsf{bad}} \preceq \nu'$, then there is a configuration ν over*

(C', L) such that $\nu^{\mathsf{bad}} \preceq \nu$ and $\nu^{\mathtt{init}}_{(C',L)} \xrightarrow{\sigma}_{(C',L)} \nu$ for some valid trace σ w.r.t. F.

As an immediate consequence of Theorem 3, we can restrict the coverability problem for parameterized cache protocols where the set of cache lines is restricted to L. More formally, we define the *restricted* coverability problem as follows: The *restricted* coverability problem for $\mathcal{P}$ wrt. a filter F and a configuration ν^{bad} over a set of caches C and a set of cache lines L, consists in checking whether there is a configuration ν over (C', L), such that: (1) $\nu^{\mathsf{bad}} \preceq_{(\phi,\psi)} \nu$ for some injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L$ such that $\psi(l) = l$ for all $l \in L$, and (2) $\nu^{\mathtt{init}}_{(C',L)} \xrightarrow{\sigma}_{(C',L)} \nu$ for some trace σ over (C', L) with $\sigma \not\models \pi$ for any $\pi \in F$. As a corollary of Theorem 3, we have:

Corollary 4. *Let F be a filter over a set of caches C and a set of cache lines L. Let ν^{bad} be a configuration over (C, L). Then, the coverability problem for $\mathcal{P}$ wrt. F and ν^{bad} can be reduced to the* restricted *coverability problem for $\mathcal{P}$ wrt. F and ν^{bad}.*

As a consequence of Corollary 4, we will use from now on the term coverability problem to mean its restricted form.

6. Checking Trace Sensitive Coverability

Assume a cache protocol $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$, a set of forbidden patterns F and a configuration ν^{bad} capturing some violation of cache coherence. Section 5 ensures that it is enough to check for the existence or absence of F-valid traces that cover ν^{bad} (i.e. violate coherence) on systems with the same number of cache lines as the number of lines in $\mathtt{linesOf}(\nu^{\mathsf{bad}})$. Observe that the length of the transactions and the number of caches (i.e. of concurrent transactions) is still arbitrary.

In fact, state reachability for any given two counters Minsky machine can be encoded using a parameterized cache protocol with a single cache line. The idea is to capture the value of each counter using the number of caches having their line at some cache state. Tests for zero are captured with the forbidding part of horizontal transitions. Coverability is therefore undecidable even in the case of a single cache line per cache. We use over-approximated systems where the analysis is exact and terminates and we refine the approximation in case of false positives.

The tail recursive procedure checkCov is used to check coherence. It takes three arguments. A cache protocol $\mathcal{P}$, a filter F, a configuration ν^{bad} and a preorder $\trianglelefteq$ on pairs of configurations and traces. All manipulated configurations have $L = \mathtt{linesOf}(\nu^{\mathsf{bad}})$ lines. The procedure is invoked with $checkCov(P, F, \nu^{\mathsf{bad}}, \trianglelefteq_0)$ where $(\nu, \sigma) \trianglelefteq_0 (\nu', \sigma')$ iff there are renamings $\phi : \mathtt{cachesOf}(\nu) \mapsto \mathtt{cachesOf}(\nu')$ and $\psi : L \mapsto L$ such that $\nu \preceq_{(\phi,\psi)} \nu'$ and $\mathtt{truncate}_F(\sigma[\phi, \psi]) \sqsubseteq_{(A_{\mathsf{ver}} \times \phi(\mathtt{cachesOf}(\nu)))} \mathtt{truncate}_F(\sigma')$ (see 3, 4.3). The result of $\mathtt{truncate}_F(\sigma)$ is defined to be the longest prefix of σ that does not contain more vertical instructions than the number of vertical instructions appearing in any of the patterns in F. Observe that such a prefix can be arbitrarily long. The idea is that the traces will be checked against the filter incrementally while being constructed, so we only need to check the "freshest" part of it. Intuitively, $(\nu, \sigma) \trianglelefteq_0 (\nu', \sigma')$ holds if, up to eliminating some caches, ν and ν' coincide and the $\mathtt{truncate}_F(\sigma)$ sequence can be obtained from the $\mathtt{truncate}_F(\sigma')$ sequence by deleting the same caches and some horizontal (but not vertical) instructions. The idea is that vertical instructions are not deleted from larger traces because this would not preserve F-validity. However, considering whole traces without applying $\mathtt{truncate}_F(\sigma)$ would result in a non wqo $\trianglelefteq_0$ for which there is no guarantee of termination even without refinement [7].

Lemma 5. *The preorder $\trianglelefteq_0$ is a wqo on* $\{(\nu_{(C,L)}, \mathtt{truncate}_F(\sigma)) | \sigma \in ((A_{\mathsf{ver}} \times C) \cup (A_{\mathsf{hor}} \times C \times L))^*\}$.

Procedure checkCov checks whether an F-valid trace σ can cover ν^{bad}. The procedure tracks pairs of the form (ν, σ), where ν is a configuration and σ is a trace. Intuitively, such a pair denotes all pairs (ν', σ') that are larger wrt. the current ordering, i.e. an upward closed set wrt. the current ordering $\trianglelefteq$. The procedure is a classical working list algorithm that maintains two sets of pairs, namely the working set W of pairs that have not been treated yet, and the visited set V of pairs that have been treated. The union of the two sets is minimal in the sense that one cannot find a pair of $\trianglelefteq$-related pairs. Given a pair in W (i.e., that has not been treated yet), the procedure computes the predecessor image wrt. each action that would not violate, given the trace in the pair, the filter F. For this reason, the trace σ that lead from ν^{bad} to the current configuration ν is maintained in each pair. Notice that the same configuration ν can participate in two $\trianglelefteq$-unrelated pairs (ν, σ) and (ν, σ'). The procedure makes use of the following operations:

```
   Input: A protocol $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$, a filter $F$, a bad configuration
          $\nu^{\mathsf{bad}}$ and a wqo $\trianglelefteq$ on pairs of configurations and $(A^{\mathsf{ext}})^*$
   Output: uncoverable or an $F$-valid trace covering $\nu^{\mathsf{bad}}$
 1 W, V := $\{(\nu^{\mathsf{bad}}, \epsilon)\}, \{\}$;
 2 while W is not empty do
 3     remove a $(\nu, \sigma)$ from W and add it to V;
 4     if $(\nu = \nu^{\mathtt{init}}_{\mathtt{cachesOf}(\nu),\mathtt{linesOf}(\nu)})$ then
 5         if $\sigma$ is possible in $\mathcal{P}$ then return $\sigma$;
 6         else return $checkCov(\mathcal{P}, F, \nu^{\mathsf{bad}}, \mathtt{strengthen}(\trianglelefteq, \sigma))$;
 7     foreach $c \in \mathtt{cachesOf}(\nu) \cup \mathtt{newCache}(\mathtt{cachesOf}(\nu))$ do
 8         $\Sigma := \{\}$;
 9         foreach $a \in A_{hor}$ and $l \in \mathtt{linesOf}(\nu^{\mathsf{bad}})$ do add $(a, c, l)$ to $\Sigma$;
10         foreach $a \in A_{ver}$ do add $(a, c)$ to $\Sigma$ ;
11         foreach $\mathbf{a} \in \Sigma$ do
12             $\sigma' := \mathbf{a}\sigma$;
13             if $\sigma' \models \pi$ for some $\pi \in F$ then continue;
14             $\Gamma := \mathtt{minOf}_{\trianglelefteq}(\mathtt{preOf}(\mathbf{a}, \mathtt{upOf}_{\trianglelefteq}(\nu)))$;
15             foreach $\nu' \in \Gamma$ do
16                 if $(\nu'', \sigma'') \not\trianglelefteq (\nu', \sigma')$ for each $(\nu'', \sigma'') \in$ W $\cup$ V then
17                     remove from W $\cup$ V each $(\nu'', \sigma'')$ s.t. $(\nu', \sigma') \trianglelefteq (\nu'', \sigma'')$;
18                     add $(\nu', \sigma')$ to W;
19 return uncoverable
```
Procedure checkCov($\mathcal{P}, F, \nu^{\mathsf{bad}}, \trianglelefteq$)

1) at line 6, $\mathtt{strengthen}(\trianglelefteq, \sigma)$ is invoked in case the obtained trace σ is a false positive due to the application of the upward closure. It returns

a stronger ordering $\unlhd'$. The new ordering can be chosen to be a wqo in case $\unlhd$ is a wqo [29], [30].
2) at line 7, $\texttt{newCache}(C)$ returns a singleton c that is not in the set C of caches (i.e., $c \notin C$).
3) at line 14, $\texttt{upOf}_{\unlhd}(\nu)$ is the upward closure of ν wrt. the current ordering $\unlhd$.
4) at line 14, $\texttt{preOf}(\mathbf{a}, \Gamma)$ returns a representation of the (possibly infinite) set of configurations that can reach the upward set of configurations Γ in one step with the action $\mathbf{a}$.
5) at line 14, $\texttt{minOf}_{\unlhd}(\Gamma)$ returns a finite set of configurations that are pairwise $\unlhd$ unrelated and such that each element in Γ is larger than some of them.

Lemma 6. *The operations 1-5 are effectively computable.*

Assume each $\unlhd$ is a wqo and the operations are as stated above. Termination of each non recursive call to checkCov is obtained using a wqo argument. Intuitively each call to checkCov terminates and results in `uncoverable`, an F-valid trace, or in another call to checkCov with a stronger ordering. Indeed, an infinite execution that involves only a finite number of recursive calls would mean that there is a call where `W` never gets empty. This means that we keep on finding new pairs that cannot be eliminated by the elements in $\texttt{W} \cup \texttt{V}$ in lines 15-18. This infinite sequence of new elements contradicts that $\unlhd$ is a wqo.

Lemma 7. *Each infinite execution of checkCov contains an infinite number of recursive calls where each call is made with a preorder that is stronger than the orderings of the previous calls.*

Restriction to pairs corresponding to F-valid executions is obtained because lines 12-13, together with the fact that $\unlhd$ is stronger than $\unlhd_0$, ensure we discard actions and pairs that violate the filter. Soundness is guaranteed by the fact that line 14 computes an over-approximation of the predecessor configurations, that we consider all actions and that we eliminate pairs only if they denote less configurations and stronger traces. Returned traces are valid by construction.

Theorem 8. *If checkCov returns* `uncoverable`*, then none of the F-valid executions from $\nu^{\texttt{init}}$ cover $\nu^{\texttt{bad}}$. If it returns an F-valid trace σ, then $\nu^{\texttt{bad}}$ is coverable using σ.*

7. Experimental Results

We have implemented our techniques from Section 6 as an extension of the tool ZAAMA [30]. ZAAMA implements constrained monotonic abstraction [29]. The tool can address the parameterized verification problem for cache coherence protocols (without any restriction on the input sequence of traces). The input of our prototype includes the description of the parameterized cache protocol, the set bad configurations and the filter.

We have applied our prototype to a number of different cache coherence protocols and filters. In fact, we have considered two cache protocols: The TMESI protocol [25] and the UTCP protocol [27]. Both of them are adaptations of the well-known MESI protocol [28] to the case of transactional memories. TMESI is used in the hardware accelerated transactional memory FlexTM [25], while UTCP in the hybrid transactional memory DynTM [27].

These hardware accelerated transactional memories come with conflict resolution policies describing the set of forbidden traces. We model these policies using our filter models. FlexTM admits two conflict resolution policies which are the lazy and eager policies. In the lazy policy, the resolution can wait until the commit before deciding on which transaction to abort. While in the eager policy the conflict is resolved by aborting a transaction as soon as the conflict is detected. Therefore, FlexTM can be run with different modes. On the other hand, DynTM allows the eager and lazy modes to execute simultaneously. Furthermore, we have also defined a new filter for the lazy execution mode of FlexTM which allows the transactions whose read instructions precede all the conflicting writing instructions to survive when a conflicting transaction commits. For instance, the transaction running on c_1 would survive in $(read, l, c_1)(write, l, c_2)(commit, c_2)(commit, c_1)$. This behavior does not cause incoherent states and still satisfies the strict serializability definition [31]. We have also considered the filter allowing only strict serializable traces [26], [31].

The results of our analyses can be seen in Table 2. Our results show that TMESI (resp. UTCP) cannot violate coherence when run together with its proper filters, namely *lazy* FlexTM or *eager* FlexTM (resp. *eager* & *lazy* DynTM). To the best of our knowledge, this is the first time that coherence of such hardware accelerated transactional memories is proven automatically. Our results show that coherence is still preserved when TMESI is run together with the *new lazy* filter in spite of the fact that it allows for more traces than the ones allowed by the *lazy* FlexTM. Finally, our results show that both TMESI and UTCP become incoherent when considering only strict serializable traces.

All experiments were performed on a 2.9 Ghz Intel Core i7 with 8GB of RAM.

8. Conclusion

In this paper, we have addressed for the first time the parameterized verification of cache coherence protocols in the presence of transactional memories. We have first proposed a formal model for this class of systems in order to capture behaviours of parameterized cache coherence protocols as restricted by filters to capture transactional memories conflict resolution policies. Our first contribution was a small model theorem allowing us to restrict the analysis of such systems to only a fixed number of cache lines. Our second contribution was an non-trivial extension of the classical framework of monotonic abstraction in order to exclude the traces that are not allowed by our filter. Finally, we have implemented a prototype that is able to successfully establish or refute coherence for several challenging examples.

Cache protocol (filter)	#rules	# bad states	Reachable (Y/N)	Execution time
TMESI (eager FlexTM)	92	36	N	48.7s
TMESI (lazy FlexTM)	48	34	N	12.7s
UTCP (eager & lazy DynTM)	128	137	N	236.8s
UTCP (serial. filter)	70	47	Y, bad state (M, M)	117.3s
TMESI (new lazy filter)	47	34	N	13.5s
TMESI (serial. filter)	42	38	Y, bad state (M, M)	35.8s

TABLE 2: Experimental Results. The columns "#rules" and "# bad" states give the number of rules and the number of bad states used to model the cache coherence protocols, respectively. A "N" in the column "Reachable (Y/N)" means that the parameterized cache protocol with filter is coherent. A "Y" in the column "Reachable (Y/N)" means that the parameterized cache protocol with filter is not coherent and we provide the first reachable bad state. Finally, the column "Execution time" gives the running time in seconds.

A direction for future work is to address the problem of automatically synthesizing filters in order to ensure the coherence of a given cache protocol.

References

[1] A. Kaiser, D. Kroening, and T. Wahl, "Dynamic cutoff detection in parameterized concurrent programs," in *CAV'10*, ser. LNCS, vol. 6174. Springer, 2010, pp. 645–659.

[2] P. Liu and T. Wahl, "Infinite-state backward exploration of boolean broadcast programs," in *FMCAD*, 2014.

[3] E. M. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *VMCAI'06*, ser. LNCS, vol. 3855. Springer, 2006, pp. 126–141.

[4] D. Sethi, M. Talupur, and S. Malik, "Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom," in *ATVA*, ser. LNCS, vol. 8837, 2014.

[5] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *CHARME*, ser. Lecture Notes in Computer Science, vol. 2144, 2001.

[6] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine, "Regular model checking without transducers (on efficient verification of parameterized systems)," in *TACAS'07*, ser. LNCS, vol. 4424. Springer, 2007, pp. 721–736.

[7] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS'96*, 1996, pp. 313–321.

[8] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, "Symbolic model checking with rich assertional languages," *Theor. Comput. Sci.*, vol. 256, no. 1-2, pp. 93–112, 2001.

[9] D. Dams, Y. Lakhnech, and M. Steffen, "Iterating transducers," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001.

[10] B. Boigelot, A. Legay, and P. Wolper, "Iterating transducers in the large," in *CAV'03*, ser. LNCS, vol. 2725. Springer, 2003, pp. 223–235.

[11] T. Touili, "Regular Model Checking using Widening Techniques," *ENTCS*, vol. 50, no. 4, 2001, proc. of VEPAS'01.

[12] A. Bouajjani, P. Habermehl, and T. Vojnar, "Abstract regular model checking," in *CAV'04*, ser. LNCS, vol. 3114. Springer, 2004, pp. 372–386.

[13] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine, "Simulation-based iteration of tree transducers," in *TACAS'05*, ser. Lecture Notes in Computer Science, vol. 3440. Springer, 2005, pp. 30–44.

[14] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, pp. 675–735, 1992.

[15] G. Delzanno, "Automatic verification of cache coherence protocols," in *CAV'00*, ser. LNCS, Emerson and Sistla, Eds., vol. 1855. Springer, 2000, pp. 53–68.

[16] ——, "Verification of consistency protocols via infinite-state symbolic model checking," in *FORTE'00*, ser. IFIP Conference Proceedings, vol. 183. Kluwer, 2000, pp. 171–186.

[17] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0,1,infinity)-counter abstraction," in *CAV'02*, ser. LNCS, vol. 2404. Springer, 2002.

[18] P. Ganty and A. Rezine, "Ordered counter-abstraction," in *Language and Automata Theory and Applications*, ser. LNCS. Springer International Publishing, 2014, vol. 8370, pp. 396–408.

[19] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 221–234.

[20] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS'01*, ser. LNCS, vol. 2031. Springer, 2001, pp. 82–97.

[21] K. S. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *VMCAI'07*, ser. LNCS, vol. 4349. Springer, 2007, pp. 299–313.

[22] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine, "Handling parameterized systems with non-atomic global conditions," in *VMCAI'08*, ser. LNCS, vol. 4905. Springer, 2008, pp. 22–36.

[23] N. Yonesaki and T. Katayama, "Functional specification of synchronized processes based on modal logic," in *IEEE 6th International Conference on Software Engineering*, 1982, pp. 208–217.

[24] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few (parameterized verification through view abstraction)," in *VMCAI*, ser. LNCS, vol. 7737, 2013, pp. 476–495.

[25] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *ISCA'08*. IEEE Computer Society, 2008, pp. 139–150.

[26] P. A. Abdulla, S. Dwarkadas, A. Rezine, A. Shriraman, and Y. Zhu, "Verifying safety and liveness for the flextm hybrid transactional memory," in *DATE 13*. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 785–790.

[27] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *MICRO 10*. IEEE Computer Society, 2010, pp. 27–38.

[28] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA 84*. ACM, 1984, pp. 348–354.

[29] P. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine, "Constrained monotonic abstraction: A cegar for parameterized verification," in *CONCUR 2010*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6269, pp. 86–101.

[30] Z. Ganjei, A. Rezine, P. Eles, and Z. Peng, "Abstracting and counting synchronizing processes," in *VMCAI 05*, ser. LNCS, vol. 8931. Springer, 2015, pp. 227–244.

[31] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

Compositional Reasoning Gotchas in Practice

Chirag Agarwal*, Paul Hylander†, Yogesh Mahajan‡, Jonathan Michelson‡ and Vigyan Singhal§
*Oski Technology, Gurgaon, India
†Ikanos Communications, Fremont, California, USA
‡NVIDIA, Santa Clara, California, USA
§Oski Technology, Mountain View, California, USA

***Abstract*—Model checking has become a formal sign-off requirement in the verification plans of many hardware designs. For design sizes encountered in practice, compositional assume-guarantee reasoning is often necessary to achieve satisfactory results. However, many pitfalls exist that can create unsound or unexpected results for users of commercial model checking tools. Users need to watch out for circularity in properties, for dead-ends getting trimmed by tools, as well as understand the differences in proof composition for liveness and safety properties. We present many real design examples to illustrate these points, as well as describe our experiences with compositional reasoning in practice.**

I. Introduction

Compositional proofs, while highly desirable, are sometimes tricky to apply correctly in practice. Compositional reasoning is probably the most widely used form of assume-guarantee reasoning. Assume-guarantee reasoning does not necessarily cut out the guaranteeing logic, e.g. when establishing inductive invariants or helper lemmas that simplify the proofs for some properties. Compositional reasoning focuses on cutting out the guaranteeing logic when assuming the property which has been guaranteed, thereby reducing complexity by analyzing smaller chunks of logic. Cutting out logic is often implemented by running model checkers on sub-modules or by black boxing some sub-modules.

Compositional reasoning is naturally suited to hardware designs, which are parallel compositions of thousands or millions of processes. Adoption of compositional reasoning is also driven by the existence of interfaces, which are natural boundaries for contracts between the designers. The contracts may or may not always be explicit, but there's a good chance that a well-designed hardware interface obeys some relatively simple contracts, as would be consistent with following good design principles like encapsulation. The designed interfaces and protocols are intended to guarantee high-level properties that are targets of verification for us. Using compositional reasoning and relying on interfaces, we can verify higher-level system properties, such as absence of system-level deadlocks.

Given the size of industrial designs, the system-level verification problem is impractical to solve in a formal verification setup using the entire chip system as the design-under-test (DUT). Rather, one aims to find related proof obligations on the logic of smaller sub-units (typically coded by one RTL designer), which when taken together are sufficient to imply that the original system properties hold. This is the most practical way that model checkers are able to verify system properties to the level of confidence desired. As a side benefit, using assume-guarantee for properties on interfaces between neighboring units, both of which have model checking setups, can yield insight about design invariants and ultimately help formalize the inter-designer contracts that may not have been explicitly or precisely specified earlier.

However, as we discuss in this paper, it is important to figure out the process to get compositional reasoning right. There are plenty of pitfalls in this activity, and we would like to publicize some of the gotchas.

Our discussion in this paper is based only on our practical experience in a setting that is limited to applying commercial model checking tools on synthesizable RTL designs while verifying properties written in the popular SystemVerilog Assertion (SVA) language [1].

In Section II, we describe some previous work on compositional reasoning, and some challenges in using that in our setting. In Section III, we describe how we apply compositional reasoning. We need to watch out for false positives when tools trim dead-ends silently (Section IV). In Section V, we caution about changes in the SVA liveness syntax and go on to detail an example of a missed deadlock bug when compositional reasoning is applied for liveness properties without appropriate care. We conclude in Section VI by pointing out some steps end-users can take to avoid such pitfalls.

II. Prior Results and Discussion

When applying compositional assume-guarantee reasoning, it is important to be able to tell what may be safely inferred about properties at the system level based on the results seen at the unit level. To begin with, one may ask if it is sound to conclude that some high-level property holds at the system level if all unit level obligations are proven. This is a reasonable question since one can encounter circularity in the assume-guarantee argument, and circularity in the argument could lead to unsoundness. For example, given two neighboring modules A and B, property P_A might be verified on A assuming P_B holds, and then P_B might be verified on B assuming P_A holds, which is a form of circular argument. To help address the issue of potential unsoundness in the compositional reasoning arguments, there are many theoretical results which characterize sound applications of compositional reasoning. The work by McMillan [2] is widely known. A good overview of compositional reasoning with a focus on completeness and soundness is presented by Namjoshi and

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

$$\begin{tikzcd}
p_2 \arrow[r] \arrow[dr] & p_4 \arrow[r] & p_3 \\
& p_1 \arrow[ur] &
\end{tikzcd}$$

$\mathcal{O}$

$$(p_2 \wedge p_4) \rhd (p_2 \Rightarrow p_1)$$

$$(p_1 \wedge p_3) \rhd p_2$$

$$\texttt{true} \rhd ((p_1 \wedge p_4) \Rightarrow p_3)$$

$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4) \rhd (p_2 \Rightarrow p_4)$$

Fig. 1. Example illustrating some concepts involved in applying McMillan's compositional rule for 4 properties p_1, p_2, p_3, p_4. The graph $\mathcal{O}$ is used to order the properties. On the right are shown four target proof obligations that are consistent with applying this rule.

Trefler [3]. An alternative scheme for compositional reasoning is presented in follow up work by Amla *et al.* [4].

A. Terminology and Notation

We use SVA to write properties. SVA extends Linear Temporal Logic (LTL) [5] with operations which increase expressiveness[1] and succinctness. An assertion written as **assert P** indicates that property **P** is expected to be `true` starting anywhere along the trace. If property **P** corresponds to temporal logic formula p, **assert P** corresponds to the temporal logic formula $G(p)$. Similarly, an assumption written as **assume P** requires that $G(p)$ does not fail.

When discussing the soundness results, it is useful to know the distinction between *safety* and *liveness* properties [6]. A safety property is one whose failures can be witnessed by finite traces. Once a safety property is witnessed to fail, no further extension of the trace can make the property hold. A liveness property is one whose failures cannot be witnessed by finite traces. For a liveness property, every finite trace can be extended such that the property does not fail, i.e. every failure trace is of infinite length. Every property can be rewritten as a conjunction of a safety property and a liveness property. In this paper, we will assume that every property is written as either a safety or liveness property.

The suggestive notation $q \rhd p$ (read as "q constrains p") is used below for consecution claims. $q \rhd p$ means that it is never the case that p is `false` in cycle n and q is `true` in all prior cycles. It is equivalent to $\neg(qU\neg p)$.

B. McMillan's circular compositional reasoning rule

McMillan's compositional reasoning result provides a sufficient condition for concluding that $G(\bigwedge p_i)$ holds for the design given that some local proof obligations are met. Each proof obligation takes the form of a consecution claim, i.e. something is claimed about cycle n of a trace if a related claim applies during cycles $0 \ldots (n-1)$ of the trace. We will consider only the case with a finite number of properties p_i.

A key part of McMillan's result involves partially ordering the properties, or equivalently, viewing the properties as nodes of some acyclic directed graph $\mathcal{O}$.

The main result is that one can conclude that $\mathcal{S} \models G(\bigwedge p_i)$ if we establish for all i that $\mathcal{A}_i \models \{\Delta_i \rhd ((\bigwedge_{p \in T_i} p) \Rightarrow p_i)\}$, where

[1]Unlike LTL, SVA has the expressive power of ω-regular languages.

i. $\mathcal{S}$ is the original design with its environment constraints
ii. $\mathcal{A}_i$ is a valid abstraction of $\mathcal{S}$ (typically obtained by retaining the guaranteeing logic for p_i and cutting out unrelated logic, e.g. via black-boxing some modules)
iii. $T_i \subset \{p_1, p_2, \ldots, p_k\}$ such that $p_j \in T_i$ implies that there is a (nontrivial) path from p_j to p_i in the acyclic graph $\mathcal{O}$
iv. Δ_i is $(\bigwedge_{p \in D_i} p)$ with $D_i \subseteq \{p_1, p_2, \ldots, p_k\}$

(An illustrative example showing an application of the rule is in Fig. 1 for the case of 4 properties.)

In this result, the key part is the definition of T_i using the ordering implied by $\mathcal{O}$. This allows one to construct an argument that $\bigwedge p_i$ is not `false` at cycle n assuming $\bigwedge \Delta_i$ is not `false` at all prior cycles. This then allows one to conclude that Δ_i is not `false` at cycle n and enables the inductive step for the next cycle.

The result applies irrespective of whether the p_i's are safety or liveness properties. Since liveness properties do not fail on finite traces, and all failures of safety properties are witnessed by finite traces, the inductive argument may be somewhat surprising! The potential for confusion arises because saying "p_i is not `false` at cycle n" is not the same as saying "p_i does not fail at cycle n", especially when p_i is not a combinational property. Each property p_i is itself a path formula which can be evaluated as `true` or `false` for the path beginning in cycle n. A (safety) property p_i fails at cycle n if cycles m through n of the trace form a bad prefix for p_i for some $m \leq n$.

For the case where all the p_i are safety properties, the induction argument *can* in fact be of the form that $\bigwedge p_i$ does not fail at cycle n assuming $\bigwedge \Delta_i$ does not fail at any prior cycle. Then Δ_i does not fail at cycle n and enables the next inductive step. In this case, the induction is over the length of finite traces and we seek the shortest trace which witnesses a failure of the safety properties involved.

C. Challenges applying McMillan's result

Implementing McMillan's compositional assume-guarantee result as given above is not simple in practice:

i. For a large number of properties, providing and maintaining a valid and effective ordering of the properties can get unwieldy and burdensome for the end user.
ii. Writing the compositional proof obligations using SVA can get complicated. For example, writing $\neg(qU\neg p)$ in SVA naively as "**assert property** `(not (q s_until (not p)));`" actually expresses $G\neg(qU\neg p)$ which evaluates the path formula $\neg(qU\neg p)$ at each cycle of the trace. We wish to evaluate $\neg(qU\neg p)$ only at the first cycle.

In addition to the above issues which are specific to applying McMillan's result, there is also the broader question of how a particular compositional reasoning result applies when restricted to finite traces. This is important because we frequently reason about finite traces in practice. Which semantics for finite traces are compatible with the result of a given compositional reasoning theorem? (For example, the safety-only variant of the argument above which uses the notion of "p_i does not fail in cycle k" has a dependency on the semantics

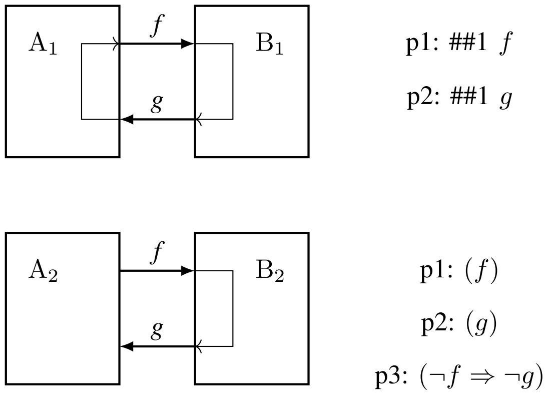


Fig. 2. Zero-delay loops

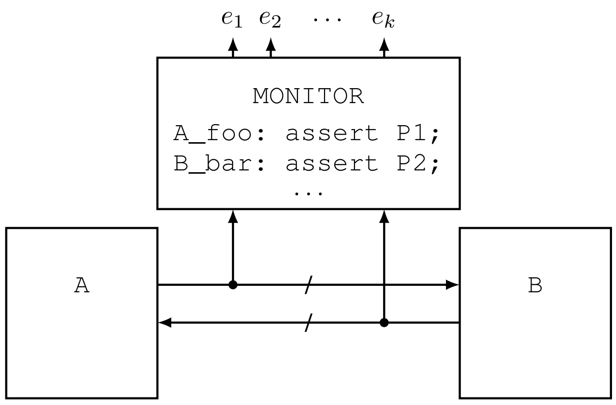


Fig. 3. Structure used for compositional reasoning about safety properties

used for finite traces. The user needs to confirm that the semantics SVA uses for finite words do not break the inductive argument. Further, certain related tool settings/behaviors like trimming dead-ends can break the inductive argument and lead to unsound results when reasoning about finite traces.)

D. Another Compositional Reasoning Approach

Since McMillan's rule can be challenging to apply, a simpler compositional reasoning rule as described below is often used.

Model checking setups are created to check properties on abstractions $\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_m$. Each property p_i is associated with the abstraction it is intended to be checked on. Let M_k be the set of all properties intended to be proven on $\mathcal{A}_k$.

In order to claim that $\mathcal{S} \models G(\bigwedge p_i)$, establish for all k that $\mathcal{A}_k \models G(\bigwedge_{p \notin M_k} p) \Rightarrow G(\bigwedge_{p \in M_k} p)$, where

i. $\mathcal{S}$ is the original design with its environment constraints
ii. $\mathcal{A}_k$ is a valid abstraction of $\mathcal{S}$ containing the guaranteeing logic for all $p \in M_k$

Unlike in McMillan's rule, there is no longer any requirement to explicitly order the properties. Further, SVA can express the proof obligations very simply. In the setup for $\mathcal{A}_i$, use **assume P** for all $p \notin M_i$ and **assert P** for all $p \in M_i$.

Unfortunately, this argument is not sound in general, as summarized in detail in [4]. It is unsound for liveness properties. It can be used only with safety properties, but that too has a few corner cases where it can yield unsound results.

The tradeoff is that the user now needs to be aware of the assumptions under which this compositional reasoning claim is valid and to have a way of knowing when these extra assumptions are getting violated, in addition to the problem of not accidentally using tool settings that interfere with the induction argument.

Since the compositional reasoning argument for safety properties involves an inductive argument on the length of the shortest trace which violates one of the safety properties, the potential for unsoundness lies in either the base case or the inductive step being spuriously claimed to hold.

One way the inductive step can spuriously be claimed to hold occurs when there are zero-delay loops in the logic involved. Fig. 2 pictorially depicts two ways a zero-delay loop can occur in the logic.

In the first case shown in the upper half of Fig. 2, there is a combinational loop going through modules A_1 and B_1, and we are skipping the check in the cycle after reset. All tools complain about such combinational loops, and so we need not worry about this pitfall.

In the second case shown in the lower half of Fig. 2, property *p1* will get proven on A_2 after assuming *p2* and *p3*, irrespective of the driving logic of f (assuming the base case holds at reset), and properties *p2* and *p3* will get proven on B_2 assuming *p1*. This situation is due to a zero-delay loop in the logic involving f getting reflected as g by B_2 combined with the structure of the properties. In this case, unfortunately, tools do not complain about the combinational zero-cycle dependency, and the user needs to be careful that the properties do not cause such a loop.

Further examples illustrating unsoundness are presented in Section IV (safety case) and Section V-B (liveness case). Section III briefly describes the methodology used in these unsoundness examples.

III. Our Methodology

The structure we use for compositional assume-guarantee reasoning is illustrated in Fig. 3 for interconnected modules A and B. It uses an SVA bind to hook a monitor module to the interface between the modules. The model checking setup for A black-boxes B and vice versa. Properties are written in the monitor module using a naming convention where the name of the property indicates which module is expected to contain the guaranteeing logic for the property. When running the setup with A as the DUT, the properties in the monitor with A expected to be guaranteeing logic are used as SVA asserts, while the properties expected to be guaranteed by B are used as SVA assumes. For example, the property named A_foo is used as an assertion for A and an assumption for B. If the properties asserted on A and B do not fail in their respective setups, it is claimed (possibly unsoundly) that the properties hold for the composed system up to the minimum bounded proof depth achieved. This structure implements the approach described in Section II-D. It is unsound to use this structure for composing liveness properties (refer to Section V-B).

Note that for reset, we (and tools) assume that the hardware logic resets to a state consistent with 3-valued simulation. Our SVA assumptions are disabled during the reset analysis phase.

IV. Deadends and Impact of Trimming Deadends

Writing constraints[2] is often a challenging and time-consuming endeavor. Over time, users of formal verification tools have developed their own guidelines to converge on a "good" set of constraints. Some different guidelines are:

- Begin with the environment as under-constrained as possible, so you avoid missing bugs by accidentally over-constraining the setup.
- Begin with an over-constrained setup, to minimize false failures and avoid inefficient use of designer debug time.
- Avoid writing constraints on the outputs or on internal signals of the DUT.
- If possible, write constraints as implications with the constrained input appearing in the consequent of the implications.
- Avoid dead-ends.

A. Intentional dead-ends

Dead-ends are artifacts of using constraints. In any RTL design, without any constraints, any sequence of input values is permitted (and the design will produce some output, no matter what the input sequence is). In the presence of constraints, however, we can have permissible finite sequences of input values that cannot be extended any further if every choice of the next input value violates at least one constraint. Dead-ends are recursively defined as states from which either no transition is possible (0-cycle dead-ends), or the only transitions are to states that are dead-ends (multi-cycle dead-ends).

The SystemVerilog standard [7] does not mandate what the tools should do with dead-ends, and unfortunately, different commercial tools treat dead-ends differently – some tools never trim dead-ends (i.e. a finite trace is considered a legal counter-example to a safety property even if it ends in a dead-end) unless explicitly instructed by the user, whereas other tools trim dead-ends, and sometimes they trim 0-cycle and 1-cycle dead-ends, but not other dead-ends!

The above-mentioned guideline of writing constraints carefully to avoid dead-ends, is religious, and different opinions exist. Sometimes the process of avoiding dead-ends can make the code less intuitive or less readable. Consider a design with one input *a*, and where *a* should be constrained such that it is never asserted in three consecutive cycles, nor deasserted in three consecutive cycles. One popular method of implementing constraint models is through state machines. The state machine in Fig. 4 can be used to implement this constraint via the following code:

[2]In the following sections, we will use the word *constraints* to refer to assumptions, because the use of *constraints* as well as *under-constraints* and *over-constraints* is more popular in commercial tools.

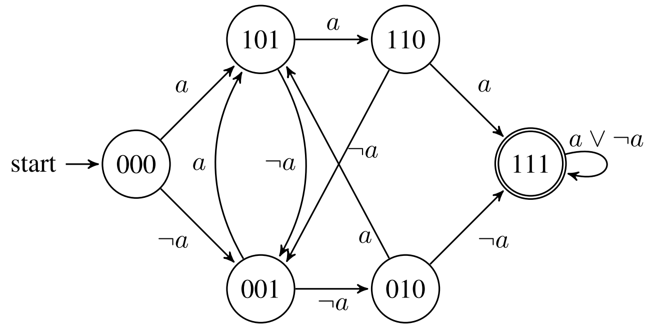


Fig. 4. Constraint implemented using a state machine

```
no_000_111_c: assume property(
  @(posedge clk) disable iff(!rstn)
     sm != 3'b111
);
```

The state 111, the error state, is a dead-end. Using the code above, the user intends for the tool to remove any input sequences that will cause the state machine to go to this error state. Consider a 5-cycle long input sequence $0{\cdot}1{\cdot}0{\cdot}0{\cdot}0$, which violates the desired constraint that *a* should not be deasserted for 3 consecutive cycles, causing the state machine to arrive in state 111 in the 6-th cycle. If some design assertion fails on the 5-th cycle on this input sequence, the questions is whether that failure is a legal counter-example or not. Likely, the author of this constraint code considers that sequence illegal, and would not be happy with the false failure. We think that might be the reason why some tools trim dead-ends, perhaps in response to such seemingly reasonable expectations. In fact, using the same state machine, coding constraints to avoid any dead-ends in the first place would require code like the following:

```
no_000_c: assume property(
  @(posedge clk) disable iff(!rstn)
     (sm == 3'b010) |-> a
);
no_111_c: assume property(
  @(posedge clk) disable iff(!rstn)
     (sm == 3'b110) |-> (!a)
);
```

This is more verbose than the previous code, and one can imagine it being even more so, if there were more arcs to the error state.

B. Compositional reasoning with dead-ends

Even though it might seem reasonable for tools to trim dead-ends, we show a serious problem that can happen during compositional reasoning due to this.

Consider modules A and B shown in Fig 5. B receives packets sent by A, buffers them in a FIFO structure which is 8 entries deep, and then sends the packets downstream. A is responsible for making sure that it does not send more packets than B can hold, while B is responsible for notifying A when packets are being drained from the FIFO. The interface signals depicted in the figure show a simplified view of the flow control used in the actual design – A sets signal *push*

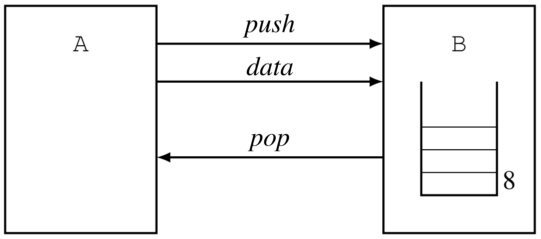


Fig. 5. A sends packets to B. B buffers the packets in a FIFO of size 8.

when valid data is sent and B uses signal *pop* to indicate when an entry has been dequeued from the FIFO.

The checks that the FIFO does not overflow or underflow are implemented using a counter *ctr* in the interface monitor as follows:

```
logic [3:0] ctr;
always @(posedge clk or negedge rstn)
begin
  if (!rstn)
    ctr <= 0;
  else
    ctr <= ctr + push - pop;
end

A_no_overflow: assert property(
@(posedge clk) disable iff(!rstn)
  ctr + push <= 8
);

B_no_underflow: assert property(
@(posedge clk) disable iff(!rstn)
  pop <= ctr
);
```

Since *ctr* is 4-bit wide, an underflow makes it wrap around to 15. Suppose the design has a bug where B sends *pop* when *ctr* is 0, but this bug can only happen when *push* is 0. When this failure of *B_no_underflow* happens, *ctr* will have the value 15 in the next cycle and that will cause *A_no_overflow* to fail. Since *A_no_overflow* is used as an assumption when checking *B_no_underflow* on B, the failure of *B_no_underflow* is witnessed only by 0-cycle dead-end traces.

If the user enables the tool setting to hide failure traces that cannot be extended by at least one more cycle, or if the tool trims dead-ends by default, then the tool will trim all failures of *B_no_underflow* and report it as proven.

We recommend implementing constraints so dead-ends are avoided altogether, even if that makes the code more verbose, as in Section IV-A. For the current example, one way to avoid the dead-ends is to rewrite the *A_no_underflow* property as:

```
A_no_overflow: assert property(
@(posedge clk) disable iff(!rstn)
  (ctr == 8) |-> !push
);
```

The above approach to removing unintentional dead-ends is reactive. We would instead like the tools to support a check that the formal testbench permits no dead-ends, and if the check fails, to produce a finite witness ending in a dead-end.

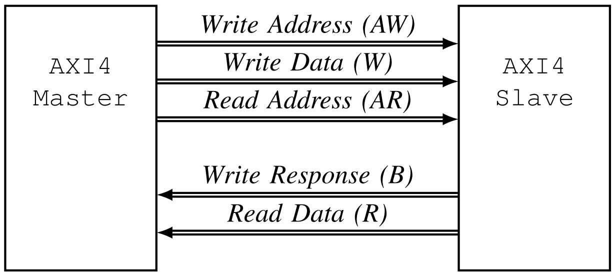


Fig. 6. ARM AMBA AXI4 interface

V. Compositional Proofs of Forward Progress

We describe a system we have seen where a deadlock may be missed in the process of compositional proofs, if the properties are expressed as liveness properties.

A. Liveness syntax and semantics in SystemVerilog 2009

Before going on to describe the missed deadlock, we want to alert the reader that the SystemVerilog semantics adopted in 2009 (IEEE Standard 1800-2009 [7]) significantly change the meaning of property expressions like the one below:

```
a |-> ##[0:$] b
```

Before 2009, this syntax was used to express the property that if *a* is true, eventually *b* must be true [1]. However, with the change in the standard, the property expression written above is equivalent to `true`, even if *a* and *b* are primary inputs with no constraints on them! This was a surprise to the authors, and given that tools do not typically warn the users that such property expressions are very likely not doing what the user intended, we consider this very dangerous.

In contrast, the tools do error out on a property expression like the one below which is deemed illegal:

```
a |-> eventually b
```

The safer way to express liveness in SystemVerilog 2009 is using syntax like the following:

```
a |-> s_eventually b
```

B. AXI4 deadlock missed while composing liveness properties

This design[3], a simplified version of a real-system design, is based on the popular ARM AMBA AXI4 on-chip interface standard [8]. The AXI4 standard connects a master to a slave via five asynchronous channels (Fig. 6): Write Address (AW), Write Data (W), Read Address (AR), Write Response (B) and Read Data (R). The standard assumes a synchronous clock *ACLK*.

The Write Address and Write Data transactions are responded by a single-beat Write Response (B) that indicates whether the write succeeded without errors, or not. Two important signals in this B channel are *BVALID* and *BREADY*. *BVALID* indicates that the slave is sending the write response on this cycle, and *BVALID* stays asserted until the master

[3]The RTL of our simplified design is available at http://www.oskitechnology.com/wp-content/uploads/2015/09/fmcad15.tar.gz

acknowledges the response with *BREADY*. Similarly, the Read Data (R) channel is used to send the read response, which includes the read data. Besides the read data itself, three important signals in the R channel are *RVALID*, *RLAST* and *RREADY*. *RVALID* is asserted on each cycle the slave is sending the read data, and since the read could be a burst read, *RLAST* is used to indicate whether the current beat of data is the last beat. Like with the B channel, the ready signal *RREADY* is used by the master to indicate to the slave that it has accepted the current beat – until that happens, the slave is required to hold it values of *RVALID*, *RLAST* and read data.

To describe the deadlock situation, it is convenient to assume that each read is either 1 or 2 beats, no longer. We will only need to look at the B and R channels. Two AXI4 properties will participate in this deadlock, one master property (*M*) and one slave property (*S*):

1) master property *M*: once the master receives *BVALID*, eventually it must assert *BREADY*
2) slave property *S*: once the slave sends *RVALID* with *RLAST* deasserted (i.e. not for the last beat) and it is accepted by the master which asserts *RREADY*, eventually the slave must send *RVALID* with *RLAST* asserted

These two properties can be implemented in SystemVerilog 2009 as the following liveness properties:

```
property master_liveness_bready;
  @(posedge aclk) disable iff (!aresetn)
    (bvalid && !bready) |->
      s_eventually bready;
endproperty

property slave_liveness_rlast;
  @(posedge aclk) disable iff (!aresetn)
    (rvalid && !rlast && rready) |->
      s_eventually (rvalid && rlast);
endproperty
```

The AXI4 protocol allows the B and R channels to be completely decoupled from each other. However, to optimize resources or area, some master or slave may choose to share a FIFO for both the B and R responses. We will call such a device a serializing device.

The deadlock happened because a serializing master was connected to a serializing agent. This deadlock scenario is depicted in Fig. 7. In the simplified version of the deadlock, the serializing master has a 2-deep FIFO that stores the B and R responses. For the R responses, the master needs to receive the entire read response (whether it is 1 beat or 2 beats), before it dequeues the read response from the queue. For design-specific reasons, the master processes the requests in order – so if a B response arrives later than a previous R response, the prior R response must be processed before the B response is accepted by asserting *BREADY*. Similarly, the serializing slave also has a 2-deep FIFO that stores the B and R responses that are queued up to be sent to the master. For the deadlock to happen, we have three transactions: a Read Response R1 composed of two beats R1.F and R1.L, and two Write Responses B1 and B2. The slave decides to send R1.F, followed by B1, followed

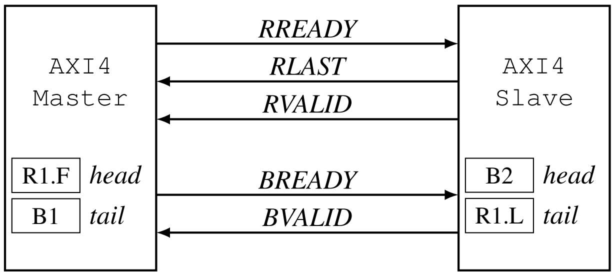


Fig. 7. AXI4 system deadlock

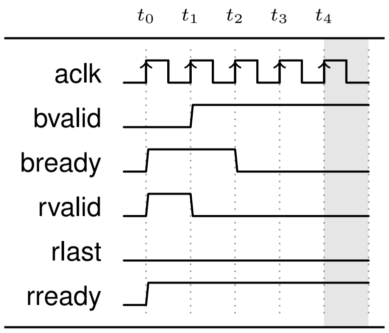


Fig. 8. Deadlock failure in the composed design (last 1 cycle loops forever)

by B2, followed by R1.L. This results in a deadlock shown in Fig. 7: the master does not assert *BREADY* for B2, because it is waiting for R1.L. The slave will not send R1.L, until B2 is dequeued first, causing the deadlock.

In fact, if the model checking setup has the DUT as the entire system containing both the master and the slave RTL, each of the two liveness properties fails, with infinite-length counter-examples showing the deadlock (Fig. 8 shows one of these two failures). The reader may observe that the deadlock can be avoided by forcing either the master or the slave to be non-serializing, and in that sense, it is arguable if the deadlock is due to a bug in the master or in the slave!

However, the situation becomes interesting when we use compositional reasoning. In the real system, the master and slave modules were large enough (and designed by different RTL designers) that it was important to verify the modules separately. We wanted to prove the property *M* on the master, and the property *S* on the slave. Since each module is serializing, and depends on fairness constraints from the other, it seems natural to want to prove *M* on the master while assuming *S*; and conversely, to prove *S* on the slave while assuming *M*. The methodology described in Section III was used to carry out this compositional argument. An expert reader may realize at this point that each of these liveness checks may now actually pass. In fact, that is exactly what happens! A naive user might then incorrectly conclude that *M* and *S* are true on the composed system and miss the deadlock bug.

When using the methodology of Section III, the proof decomposition attempted is to prove $G(M) \Rightarrow G(S)$ on the slave and $G(S) \Rightarrow G(M)$ on the master. Suppose it happens that whenever S is `false` for the slave, property M must also necessarily be `false` somewhere further along the same trace. Then if we assume $G(M)$ when proving $G(S)$, we will no longer see any failures for $G(S)$.

Suppose we had instead tried to prove $G(M \Rightarrow S)$ on the slave and $G(S \Rightarrow M)$ on the master. This proof decomposition would be equivalent to checking $\texttt{true} \rhd (M \Rightarrow S)$ on the slave and $\texttt{true} \rhd (S \Rightarrow M)$ on the master. For this decomposition, we cannot use McMillan's rule to infer that $G(M \wedge S)$ holds for the composed design, because the implied ordering graph would be cyclic $$\begin{tikzcd} M \arrow[r, bend left] & S \arrow[l, bend left] \end{tikzcd}$$

(Note that $\texttt{true} \rhd p$ is equivalent to $\neg(\texttt{true} U \neg p)$ which can be rewritten as $\neg(F \neg p)$ and then as Gp.)

C. Using liveness properties safely with McMillan's rule

To confirm that we cannot miss the deadlock when applying McMillan's circular compositional rule with the liveness properties M and S, we ran the four checks listed below:

i. on the master, check $(M \wedge S) \rhd S$
ii. on the master, check $(M \wedge S) \rhd (M \Rightarrow S)$
iii. on the slave, check $(M \wedge S) \rhd S$
iv. on the slave, check $(M \wedge S) \rhd (M \Rightarrow S)$

All the above checks fail, implying that all attempts to apply McMillan's circular compositional rule to infer $G(M \wedge S)$ will involve a failing check. Hence, the user will not be led to the incorrect conclusion that there is no deadlock.

For reference, the SVA implementation is shown below:

```
wire mlhs = (bvalid && (!bready));
wire mrhs = bready;
wire slhs = (rvalid && (!rlast) && rready);
wire srhs = (rvalid && rlast);
reg aresetn_d;
always @(posedge aclk) begin
  aresetn_d <= aresetn;
end

property m;
  (mlhs |=> s_eventually mrhs);
endproperty

property s;
  (slhs |=> s_eventually srhs);
endproperty

property T(e);
@(posedge aclk) disable iff (!aresetn)
  e;
endproperty

property F(p);
  !aresetn_d |-> (p);
endproperty

property K(l,r);
  not ((l) s_until (not (r)));
endproperty

chk0_s: // most constrained check
  assert property (
    T(F(K(m and s,s)))
);
```

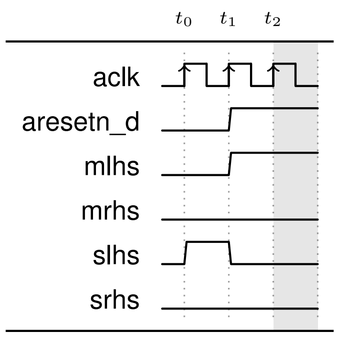


Fig. 9. Failure for chk0_s on the slave (last 1 cycle loops forever). The same trace also shows chk0_m_s failing.

```
chk0_m_s: // most constrained check
  assert property (
    T(F(K(m and s,m implies s)))
);
```

The failure trace for *chk0_s* on the slave is shown in Fig. 9 and shows the property *s* is `false` in the first cycle. Since property *m* is `true` in the first cycle, the same trace is also a failure trace for *chk0_m_s* on the slave.

For sake of completeness, it may be noted that the only checks that do not fail (and get proven) are the following:

i. on the master, check $(M \wedge S) \rhd M$
ii. on the master, check $(M \wedge S) \rhd (S \Rightarrow M)$
iii. on the master, check $S \rhd M$
iv. on the master, check $S \rhd (S \Rightarrow M)$

D. Using safety properties to express forward progress

Next, we explore the use of safety properties for compositional reasoning about forward progress properties. Users are often divided about their preference for using liveness versus safety properties to prove forward progress or absence of deadlocks [9]. Liveness is usually more elegant, although it may require some iterations to identify appropriate fairness constraints. Safety properties can be used by picking a design-specific constant to require the consequent to be satisfied within this constant number of cycles, but it may need some iterations to figure out the value of the constant.

For the example as described previously, the master and slave properties can be written as the following safety variants:

```
property master_safety_bready;
  @(posedge aclk) disable iff (!aresetn)
    (bvalid && (!bready)) |->
      ##[1:`B_TIMEOUT] bready;
endproperty

property slave_safety_rlast;
  @(posedge aclk) disable iff (!aresetn)
    (rvalid && (!rlast) && rready) |->
      ##[1:`R_TIMEOUT] (rvalid && rlast);
endproperty
```

Of course, the two defined constants $B\_TIMEOUT$ and $R\_TIMEOUT$ must be selected to be large enough that

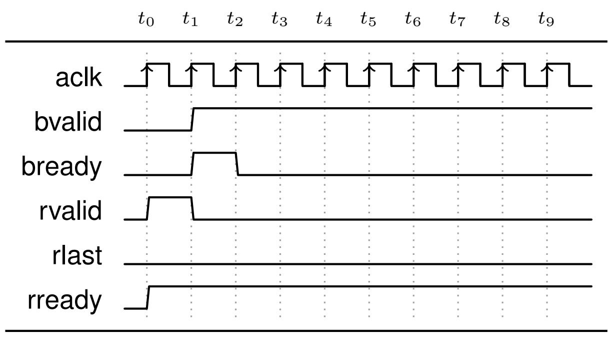


Fig. 10. Slave failure (*B_TIMEOUT=8*; *R_TIMEOUT=8*)

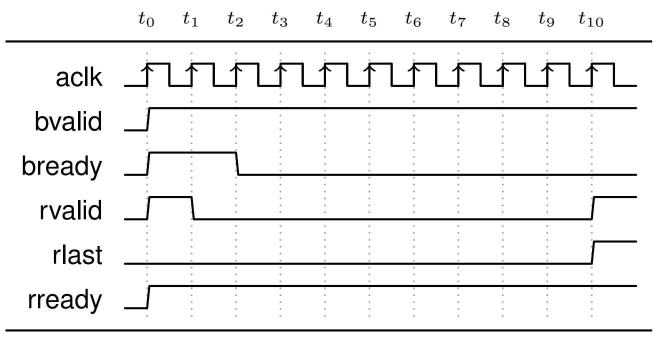


Fig. 11. Master failure (*B_TIMEOUT=8*; *R_TIMEOUT=12*)

desirable design behaviors are not flagged as errors. When these properties are checked on a system composed of the master and slave, a value of 8 for each of the two constants shows the deadlock scenario. The counter-example trace is very similar to that in Fig. 8, except that instead of the lasso at the end, the trace is stretched to about 8 extra cycles.

Further, as discussed in Section V-B, for this to work in practice on real-sized designs, we need to prove these two properties separately on the master and slave RTL modules. Choosing constant values of $B\_TIMEOUT = 8$ and $R\_TIMEOUT = 8$, if we assume *slave_safety_rlast*, the master property *master_safety_bready* passes on the master RTL module. However, doing the reverse, while assuming $master\_safety\_bready$, the slave property *slave_safety_rlast* fails with the waveform in Fig. 10.

The root cause of the failure appears to be that the master might be responsible for this since it is not accepting the second *BVALID* in the trace with a corresponding *BREADY*. So, the user is tempted to re-run by increasing *R_TIMEOUT* relative to *B_TIMEOUT*. Choosing constant values of $B\_TIMEOUT = 8$ and $R\_TIMEOUT = 12$, indeed, the slave property *slave_safety_rlast* passes while assuming *master_safety_bready*. However, now the master property *master_safety_bready* fails on the master RTL with the waveform in Fig. 11.

In fact, one can try all possible values of $B\_TIMEOUT$ and $R\_TIMEOUT$, and observe that no matter which values are chosen, either the master RTL or the slave RTL or both show a failure. Table I shows the results for

TABLE I
ASSUME-GUARANTEE RESULTS WITH SAFETY PROPERTIES

B_TIMEOUT	R_TIMEOUT	master result	slave result
8	≤ 9	Pass	Fail
8	10 or 11	Fail	Fail
8	≥ 12	Fail	Pass

$B\_TIMEOUT = 8$, and similar results are seen for other values of $B\_TIMEOUT$. This is a good result, because unlike the liveness situation (Section V-B), a naive user does not have to take on the burden of avoiding the circularity pitfall.

VI. Conclusion

While doing compositional reasoning, users need to be careful about avoiding circularity. They need to be careful that the properties combined with the hardware design do not create zero-delay loops. If tools trim dead-ends, compositional proofs may not work, unless constraints are written in a specific coding style. Liveness properties with compositional reasoning are dangerous unless users have taken care to order properties, or otherwise use McMillan's method accurately.

We hope the examples and our experiences are useful for other users practicing formal verification on hardware designs.

Acknowledgment

The authors would like to thank Prashant Aggarwal and Arun Khurana for experimenting and sharing their results on some design examples we discussed in this paper, and Jin Zhang for her review on an earlier version of the paper. We also thank the reviewers for their useful comments.

References

[1] F. Haque, J. Michelson, and K. Khan, "The art of verification with SystemVerilog assertions," *Verification Central*, 2006.

[2] K. L. McMillan, "Circular compositional reasoning about liveness," in *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME 99), volume 1703 of Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 342–345.

[3] K. S. Namjoshi and R. J. Trefler, "On the completeness of compositional reasoning," in *Computer Aided Verification*. Springer, 2000, pp. 139–153.

[4] N. Amla, E. A. Emerson, K. Namjoshi, and R. Trefler, "Abstract patterns for compositional reasoning," in *In Concurrency Theory (CONCUR), LNCS 2761*. SpringerVerlag, 2003, pp. 423–448.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[6] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed computing*, vol. 2, no. 3, pp. 117–126, 1987.

[7] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. 1–1285, Dec 2009.

[8] ARM Ltd., "AMBA AXI and ACE protocol specification, issue D," 2011.

[9] B. Krishna, J. Michelson, V. Singhal, and A. Jain, "Liveness vs safety–a practical viewpoint," in *Hardware and Software: Verification and Testing*. Springer, 2012, pp. 80–94.

Universal Boolean Functional Vectors

Jesse Bingham
Intel Corporation
Hillsboro, Oregon, U.S.A.
Email: jesse.d.bingham@intel.com

Abstract*—In the simplest setting, one represents a boolean function using expressions over variables, where each variable corresponds to a function input. So-called *parametric representations*, used to represent a function in some restricted subspace of its domain, break this correspondence by allowing inputs to be associated with *functions*. This can lead to more succinct representations, for example when using binary decision diagrams (BDDs). Here we introduce *Universal Boolean Functional Vectors* (UBFVs), which also break the correspondence, but done so such that all input vectors are accounted for. Intelligent choice of a UBFV can have a dramatic impact on BDD size; for instance we show how the hidden weighted bit function can be efficiently represented using UBFVs, whereas without UBFVs BDDs are known to be exponential for any variable order. We show several industrial examples where the UBFV approach has a huge impact on proof performance, the "killer app" being floating point addition, wherein the wide case-split used in the state-of-the-art approach is entirely done away with, resulting in 70-fold reduction in proof runtime. We give other theoretical and experimental results, and also provide two approaches to verifying the crucial "universality" aspect of a proposed UBFV. Finally, we suggest several interesting avenues of future research stemming from this program.

I. Introduction

Binary Decision Diagram (BDD) techniques for formal verification (FV) have fallen out of fashion as a research topic in the past decade. Nevertheless, the data structure is still widely used in industry to solve real-world hardware verification problems, for example at companies such as Intel [23], IBM [24], [29], and Centaur [28]. Furthermore, contemporary commercial FV tools also include BDDs in their spectrum of technologies. Perhaps one of the most successful domains for these techniques is FV of arithmetic data-path hardware designs. The efficacy of BDDs stems from the fact that they constitute a canonical representation of boolean functions, and for many functions of practical importance, the BDDs are of tractable size. It is well-known that the variable order can drastically influence the size of a BDD. Unfortunately, there are many functions, both artificial and practical, that have been shown to have exponentially large BDDs regardless of the ordering. To combat these roadblocks, techniques such as *case-splitting* and proof *decomposition* have been explored.

In this paper, we propose an orthogonal approach to avoiding blow up. Typically, one constructs a BDD for a function f in a setting wherein BDD variables and the inputs of f are in one-to-one correspondence. This correspondence can be broken when using parametric substitutions [3] to perform case-splitting; there, (not necessarily variable) functions are associated with f's inputs, with the goal of restricting the space in which f is represented. Our approach, called *Universal Boolean Functional Vectors* (UBFV) is similar to parametric substitutions in that *functions* are associated with f's inputs, however, unlike parametric substitutions, a UBFV representation of f does not restrict the space of representation (ergo, "universal"). In other words, all assignments to f's inputs are implicitly represented in the UBFV representation.

To illustrate this concept, consider the function $f(v_1, v_2, v_3) = v_1 \vee v_2\overline{v_3}$. Suppose we perform the substitution $(v_1, v_2, v_3) \mapsto (a \vee b, d, \overline{b}\overline{c})$, where a, b, c, and d are fresh variables. Applying this substitution to f yields a new function $f' = a \vee b \vee dc$. Even though f' bears no syntactic resemblance to f, the former completely characterizes the latter in the sense that one can evaluate f for any input using only f' and the substitution. The complete characterization is possible only because $(a \vee b, d, \overline{b}\overline{c})$ is *universal* in the sense that all 2^3 of the possible boolean assignments to (v_1, v_2, v_3) can be realized via assignments to (a, b, c, d); such a substitution is what we call a UBFV.

But what advantage can be achieved by performing these UBFV substitutions? We show that there exists functions, both theoretical examples and those arising in practical FV, with BDD representations being exponentially more compact when one selects an appropriate UBFV substitution. For the practical FV problems, this approach has a profound impact on proof runtime requirements. This is because where the current solution to avoid the exponential blow up involves performing many case-splits, by employing UBFVs we can reduce the number of cases drastically, often eliminating the need for case-splitting altogether. We also give a practical example where using UBFVs removes the need for proof decomposition – this can reduce proof *development* effort.

Our contributions are as follows. We lay down the groundwork for the theory of universal boolean functional vectors. and prove a key result (Theorem 2) showing that those functions that have small *partitioned BDDs* [22] also have UBFV representations with small BDDs. As a corollary, we prove that the hidden weighted bit function [5] has a cubic BDD for an appropriate UBFV. Deciding if a given substitution constitutes a UBFV is NP-hard. However we provide a BDD-based algorithm that is sufficient for the UBFV for the hidden weighted bit function and for our industrial examples, and also a user-assisted approach with low complexity. Finally, we report very encouraging performance speed-ups for real industrial proofs, namely floating point (FP) addition (FADD) and FP fused-multiply add (FMA) instructions.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

II. Related Work

The idea of using parametric representations of boolean functions (what we term BFVs) goes back to the early 1970s [6], [9]. Such a representation, generated by the *generalized co-factor* operation (GCF) (a.k.a. *constrain*) was introduced in the setting of hardware model checking by Coudert and Madre [11]. Later, Jones *et al.* [3], [18] proposed a specialized variation of GCF called *param*, for use in symbolic simulation, e.g. STE [26]. Similar to us, Jain and Gopalakrishnan [17] employ problem-specific recipes (rather than use a generic algorithm) to create parametric representations for hardware verification. However, a common theme in these works is the restriction of the space of a boolean function to simplify its representation (often BDDs); we believe ours is the first published approach that strives to simplify the representation *without* a space restriction.[1]

The industrial example where our approach is extremely effective is the verification of FADD. Published FADD proofs [10], [3], and those solving the related problem of FMA verification [16], [21], [27] require wide case-splitting, which we altogether eliminate (or at least drastically reduce, in the case of FMA).

A good introduction to the use of BDDs in hardware FV is the paper by Hu [14].

III. Mathematical Foundations

A. Boolean Functions

Let $\mathbb{B}$ be the boolean constants $\{0,1\}$ and let V be a finite set of *boolean variables*. A *V-assignment* (or simply *assignment* if V is understood) is a function $\alpha : V \to \mathbb{B}$. A *(boolean) function (over V)* is a function f taking V-assignments to $\mathbb{B}$, i.e. $f : (V \to \mathbb{B}) \to \mathbb{B}$. An assignment α is said to *satisfy* f if $f(\alpha) = 1$; f is said to be *satisfiable* (resp. *tautological*) if f is satisfied by some (resp. by all) assignments. We denote a tautological function as $\mathbf{1}$ and an unsatisfiable function by $\mathbf{0}$. We will employ overbar for boolean negation, juxtaposition or $\wedge$ for conjunction, $\vee$ for disjunction, and $\leftrightarrow$ for boolean equality. As is well known, any function over V can be represented as a formula using these operators and V. We will often leave α implicit, for instance $xy \vee z$ represents the function f over $\{x, y, z\}$ such that $f(\alpha) = \alpha(x)\alpha(y) \vee \alpha(z)$. Also, if $V \subseteq V'$, and f is defined to be a function over V, we can freely employ f as a function over V' in the obvious way.Finally, we say the function f over V is a *variable* if there exists $v \in V$ such that $f(\alpha) = \alpha(v)$ for all assignments α.

Much of what follows involves placing a total order $\preceq$ on sets of variables; if the set of variables is subscripted with integers, we say the *natural order* is the ordering that simply applies $\leq$ to the subscripts.

[1]Anecdotes within the walls of Intel recall similar *ad-hoc* trickery done in the past [15], though the idea was not explored thoroughly as we do in this paper, nor was the application to FADD/FMA known.

B. (Universal) Boolean Functional Vectors

A common operation on boolean functions is that of *substitution*, in which functions are substituted for the variables of another function. In this paper, the objects that describe substitutions are called *boolean functional vectors* (BFV) [13]. Given (not necessarily disjoint) sets of variables V and V', a BFV over (V, V') is a function $\psi : V \to ((V' \to \mathbb{B}) \to \mathbb{B})$ that takes the variables V to boolean functions over the other set of variables V'. We transpose the arguments of ψ to obtain the function $\psi^* : (V' \to \mathbb{B}) \to (V \to \mathbb{B})$ defined by $\psi^*(\alpha')(v) = \psi(v)(\alpha')$. Now applying a BFV ψ over (V, V') as a substitution against any function f over V is achieved via the composition

$$f \circ \psi^* : (V' \to \mathbb{B}) \to \mathbb{B}$$

Aside from being employable as a substitution for functions over V, a BFV ψ is also characterized by a particular V-function c. We say that c is the *characteristic* of the BFV ψ if for all V-assignments α we have that α satisfies c if and only if there exists a V'-assignment α' such that $\alpha = \psi^*(\alpha')$. Said another way, as we range across all possible V'-assignments α', $\psi^*(\alpha')$ ranges across exactly the set of V-assignments that satisfy c. When necessary to distinguish between the two variable sets V and V', we will respectively refer to them as the *primary* and the *secondary* variables.

Now we may define the central concept of this paper: a *universal BFV* (UBFV) is simply a BFV that has as the characteristic the tautological function $\mathbf{1}$. Equivalently, ψ is universal iff ψ^* is surjective. The key observation about UBFVs is that when one is used as a substitution against a function f, the result incurs no loss of information regarding f. This is formalized by the following lemma.

Lemma 1. *Let ψ be a UBFV over (V, V') and let f be a function over V. Then for any $\alpha : V \to \mathbb{B}$, there exists $\alpha' : V' \to \mathbb{B}$ such that $\alpha = \psi^*(\alpha')$ and thus $(f \circ \psi^*)(\alpha') = f(\alpha)$*
.

Proof. Follows from the fact that ψ is a UBFV; note however that α' is not necessarily unique. □

Lemma 1 says that $f \circ \psi^*$ is a legitimate representation of f; given only $f \circ \psi^*$ and ψ, we can evaluate f for any V-assignment. Since BFV substitution commutes with any boolean connective, we can apply boolean connectives in the "domain" of a UBFV ψ, for instance for two functions f_1 and f_2 and a boolean connective $\odot$, we have

$$(f_1 \circ \psi^*) \odot (f_2 \circ \psi^*) \;=\; (f_1 \odot f_2) \circ \psi^*$$

It follows from Lemma 1, importantly, that we can check for function equality in this domain as well — f_1 and f_2 are identical functions iff $f_1 \circ \psi^*$ and $f_2 \circ \psi^*$ are also identical.

C. Binary Decision Diagrams

A *branching program* (BP) [4] is an acyclic, labelled, directed graph with labelling function $\ell : N \to (V \cup \{1, 0\})$, where N is a set of nodes and V is a set of boolean variables.

A BP has exactly one source node, called the *root*. The labelling function is such that $\ell(\sigma) \in \{1, 0\}$ if and only if σ is a sink; the sinks are called *terminal nodes*. Each non-terminal node σ has exactly two direct successors, called $lo(\sigma)$ and $hi(\sigma)$. Given a V-assignment α and non-terminal node σ, the *active* child of σ is $lo(\sigma)$ if $\alpha(\ell(\sigma)) = 0$ and $hi(\sigma)$ otherwise. A BP represents the boolean function f over V defined so that $f(\alpha)$ is the label of the terminal node found by starting at the root, and following the path of active children according to α. The number of non-terminal nodes in a BP is called its *size*.

We now introduce two sub-classes of BPs that involve placing a total order $\preceq$ on V. A *$\preceq$-ordered binary decision diagram* ($\preceq$-OBDD, or OBDD if $\preceq$ is understood) is a BP such that for all non-terminal nodes σ and σ' where σ' is a successor of σ, we have that $\ell(\sigma) \neq \ell(\sigma')$ and $\ell(\sigma) \preceq \ell(\sigma')$. In other words, all paths from the root to a sink respect $\preceq$. The *sub-OBDD* rooted at a node σ is the OBDD formed by deleting all nodes other than σ and its descendants. We say two BPs are *isomorphic* if they are isomorphic in the traditional graph theoretic sense, and the isomorphism preserves ℓ, lo, and hi. A *reduced OBDD*, which we simply call a *BDD*, is an OBDD such that no two sub-OBDDs are isomorphic, and no node σ has $lo(\sigma) = hi(\sigma)$.

Theorem 1 (Bryant [7]). *For any function f and $\preceq$, there exists a $\preceq$-BDD that represents f and it is unique up to isomorphism.*

Thanks to Theorem 1, we can refer to a $\preceq$-BDD that represents f as *the $\preceq$-BDD for f*; we denote the size of this BDD by $\mathrm{Sz}(f, \preceq)$. For many functions that arise in practice, the BDD provides a compact representation. Also it is well-known that the choice of $\preceq$ can often make the difference between having an exponentially- or compactly-sized BDD. Furthermore, there are some practical functions that have been proven to have exponentially sized BDDs for *any* choice of $\preceq$. The middle bit of the output of an integer multiplier is a standard such example [8], as too is the so-called *hidden weighted bit function* [5], which we define in Sect. IV-C.

It is well-known that BDDs are the most compact OBDDs:

Lemma 2. *For any function f and $\preceq$, $\mathrm{Sz}(f, \preceq)$ is not greater than the size of any $\preceq$-OBDD for f.*

D. Generalized Cofactor

Given a total order $\preceq$ over a set of indexed variables $\{x_1, \dots, x_n\}$, the *permutation induced by* $\preceq$ is the permutation π on $\{1, \dots, n\}$ defined by $\pi(i) = j$ iff x_j is the ith element in the order $\preceq$, thus $x_{\pi(1)} \preceq \cdots \preceq x_{\pi(n)}$. Given functions f and h over $\{x_1, \dots, x_n\}$, and a variable order $\preceq$, the *generalized cofactor* [11] of f and h is the function defined by $\mathit{GCF}(f, h, \preceq)(\alpha_1) = f(\alpha_2)$, where α_2 is the unique assignment such that $h(\alpha_2) = 1$, and the following distance d between α_1 and α_2 is minimized. Here π is the permutation induced by $\preceq$, and $\oplus$ is exclusive-OR.

$$d(\alpha_1, \alpha_2) = \sum_{i=1}^{n} 2^{n-i} (\alpha_1(x_{\pi(i)}) \oplus \alpha_2(x_{\pi(i)})) \qquad (1)$$

The intuition behind (1) is that differences between α_1 and α_2 are weighted greater for variables that are smaller in $\preceq$. Although d depends on π and thus $\preceq$, we leave this implicit.

There are many papers that employ the generalized cofactor operation, but we could not find an explicit statement of the following lemma (that we use):

Lemma 3. $\mathrm{Sz}(\mathit{GCF}(f, h, \preceq), \preceq) \ \leq \ \mathrm{Sz}(f, \preceq)\mathrm{Sz}(h, \preceq)$

Proof. By inspection of the pseudo-code for $gcf(f, h)$ of Franco and Weaver [12], we see that a new node is generated at most once per recursive call to $gcf(f', h')$. A recursive call is done at most once on each pair of nodes (f', h') where f' is a node of f and h' is a node of h; the result follows. □

As noted by Jones [18], the *Param* operation [3] can be synthesized using *GCF*. We will be effectively using *Param* on several occasions, but will express it using *GCF* and hence not mention *Param* explicitly.

IV. Partitioned BDD and UBFV Size Complexity

A partitioned BDD for a function f is a set of functions that each characterize f in a subspace of assignments, and each subspace can use a different variable order. This freedom can result in smaller BDDs than a "monolithic" BDD representation. In this section we show that if f has a compact representation as a partitioned BDD, then there exists a UBFV ψ for f such that $\psi(v)$ has a compact BDD for each v, and so too does $f \circ \psi^*$. Hence, we needn't exploit disparate variable orderings as afforded by partitioned BDDs; UBFVs allow for compact representations using a single order. Consequently, whereas techniques for using partitioned BDDs in proofs effectively involve doing the proof once for each partition, we need only run the proof once using an UBFV representation.

As just stated, our result is almost obvious: one could create a per-partition copy of each primary variable, and construct the UBFV representation using an order that preserves each partition's ordering on its copy. This would totally eliminate the possibility of inter-partition sharing of sub-BDDs, and provide no interesting advantage over doing per-partition proofs. On the contrary, we use the same set of variables for each partition in the UBFV representation, which allows us to prove our result while only introducing a logarithmic increase in the number of secondary variables. Though this enables our per-partition sub-BDDs to share BDD nodes, our upper-bound result assumes no sharing. However, we show in Sect. V empirically and by example that sharing can have a profound effect; this is also evident in our experimental results of Sect. VII.

A. Partitioned BDDs

We now formalize partitioned BDDs, more or less following Narayan *et al.* [22], with one material difference.[2]

[2]The difference being that [22] requires $f_j = w_j f$ while we use the weaker condition $w_j f_j = w_j f$. We effectively give f_i the freedom to behave arbitrarily in the "don't care" space $\overline{w_i}$. This allows Theorem 2 to potentially yield tighter bounds than if we used the definitions from [22] verbatim.

A *partitioned BDD* for V-function f is a set of triples $\{(w_j, f_j, \preceq_j) : 1 \le j \le k\}$ where

1) each w_j and f_j are boolean functions over V such that $w_j f_j = w_j f$ and $w_j \neq \mathbf{0}$
2) each $\preceq_j$ is a V-order
3) $w_1 \vee \cdots \vee w_k = \mathbf{1}$

Typically, partitioned BDDs are of interest when f does not have a sufficiently small BDD representation, but each w_j and f_j do have compact $\preceq_j$-BDDs.

B. UBFV Construction

Suppose we have a partitioned BDD for f as in Sect. IV-A, over the variables $V = \{v_1, \dots, v_n\}$. Our construction of a corresponding UBFV representation involves secondary variables $C = \{c_{\log(k)}, \dots, c_0\}$ that, when assigned to, uniquely select a partition. Two key insights come into play:

1) We employ secondary variables $Y = \{y_1, \dots, y_n\}$ such that, once we have selected partition j, the variable y_i represents the ith element of the V-order $\preceq_j$. In other words, the variable v_q represented by y_i differs according to the selected partition. This trickery allows us to use the *same* order on Y across all partitions and preserve the BDD sizes in the partitioned BDD.
2) Our UBFV is constructed so that once we select partition j, the V-assignment that is induced satisfies w_j. To achieve this we utilize the generalized co-factor operation, explained in Sect. III-D.

Formalizing this, we define the *repartitioned BFV* γ. Let $c[j]$ denote the condition that $c_{\log(k)} \dots c_0$, as a binary number, is equal to j. Let π_j be the permutation induced by $\preceq_j$, and in a slight abuse, for any V-function h we let $\pi_j(h)$ be the Y-function formed by substituting $y_{\pi_j(i)}$ for v_i, $1 \le i \le n$, in h. Letting $\preceq$ be the natural order over Y, γ is the BFV over $(V, C \cup Y)$ defined by

$$\gamma(v_i) \;=\; \bigvee_{j=1}^{k} c[j]\mathit{GCF}(\pi_j(v_i), \pi_j(w_j), \preceq) \tag{2}$$

The above expression integrates both of our key insights. The use of π_j to turn V-functions v_i and w_j into Y-functions is the manifestation of the first insight, while the use of *GCF* is that of the second. Lemma 4 below, which is proven in the appendix of the web version [1] asserts that the repartitioned BFV γ is in fact universal, and also gives an expression for $f \circ \gamma^*$.

Lemma 4. *γ is a* universal *BFV, and furthermore*

$$f \circ \gamma^* \;=\; \bigvee_{j=1}^{k} c[j]\mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq) \tag{3}$$

Lemma 4 is instrumental in proving our upper bound result Theorem 2 below. The BDD for $f \circ \gamma^*$ described in the proof is shown in Figure 1.

Theorem 2. *Suppose f is a function over V with a partitioned BDD $\{(w_1, f_1, \preceq_1), \dots, (w_k, f_k, \preceq_k)\}$. Then there exists a UBFV ψ for f with secondary variables V' and total order $\preceq$ on V' such that*

(a) For all $v \in V$, $\mathrm{Sz}(\psi(v), \preceq) = \mathcal{O}\left(\sum_{j=1}^{k} \mathrm{Sz}(w_j, \preceq_j)\right)$

(b) $\mathrm{Sz}(f \circ \psi^*, \preceq) = \mathcal{O}\left(\sum_{j=1}^{k} \mathrm{Sz}(w_j, \preceq_j)\mathrm{Sz}(f_j, \preceq_j)\right)$

(c) $|V'| = \lceil \log_2 k \rceil + |V|$

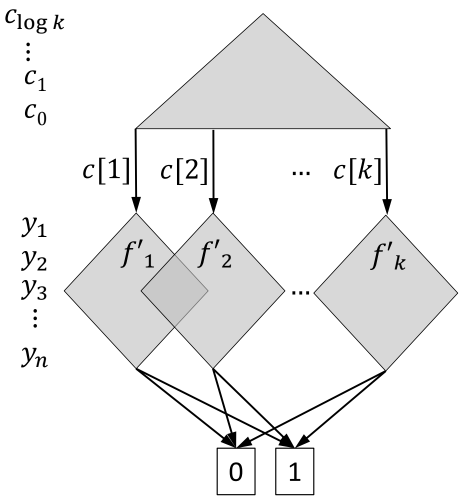


Fig. 1. The OBDD for $f \circ \gamma^*$ described in the proof of Theorem 2. The variable order is shown on the left; the triangular part at the top of the OBDD is the incomplete OBDD that determines which $c[j]$ holds. Each f'_j is the BDD for $\mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq)$, which is the GCF of f_j with respect to w_j, but with variables renamed according to π_j. Thus each f'_j has the exact same structure as the $\preceq_j$-BDD of $\mathit{GCF}(f_j, w_j, \preceq_j)$. The overlap between f'_1 and f'_2 emphasizes that sub-BDD sharing is possible between all the f'_j's.

Proof. (Sketch) We choose the repartitioned BFV γ to serve as the witness. Thanks to Lemma 4, we have that γ is universal; Also condition (c) holds of γ by definition. To prove conditions (a) and (b), let us employ the notion of an *incomplete OBDD* as an OBDD wherein some sink nodes are not terminal nodes, but rather unlabelled *placeholders*; by appropriately plugging in other OBDDs at placeholders, an incomplete OBDD can be turned into an OBDD.

For each v_i, an $\preceq$-OBDD for $\gamma(v_i)$ (2) can be constructed as follows. Start with an incomplete OBDD over variables C with k placeholder nodes, such that the path to the jth placeholder is active when $c[j]$ holds of an assignment. Note that this incomplete OBDD has size $\mathcal{O}(k)$. For each $1 \le j \le k$, at the jth placeholder we insert the $\preceq$-BDD of $\mathit{GCF}(\pi_j(v_i), \pi_j(w_j), \preceq)$. From Lemma 3 and the fact that the size of a variable function is 1, the size of this BDD is bounded by $\mathrm{Sz}(\pi_j(w_j), \preceq) = \mathrm{Sz}(w_j, \preceq_j)$; condition (a) follows.

We build an $\preceq$-OBDD for $f \circ \gamma^*$ by starting with the same incomplete OBDD as above. At the jth placeholder, we insert the $\preceq$-BDD of $\mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq)$. Again appealing to Lemma 3 we find that size of this OBDD to be bounded by

$$\mathrm{Sz}(\pi_j(f_j), \preceq)\mathrm{Sz}(\pi_j(w_j), \preceq) = \mathrm{Sz}(f_j, \preceq_j)\mathrm{Sz}(w_j, \preceq_j)$$

□

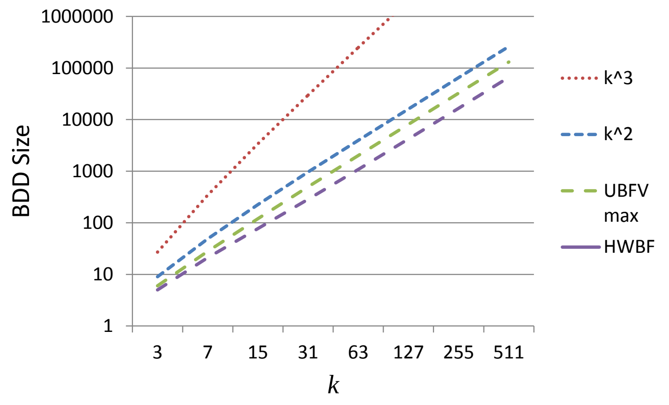


Fig. 2. Computed BDD sizes for the hidden weighted bit function using UBFVs (log-log scale). Data points are for $k = 2^e - 1$ with $2 \leq e \leq 9$. "UBFV max" is the largest BDD in the UBFV ψ, while "HWBF" is the size of $HWB_k \circ \psi^*$. The curves k^3 and k^2 are plotted for comparison, from which it seems evident that the BDDs grow only quadratically, an improvement over our proven cubic bounds.

C. Application to Hidden Weighted Bit Function

As a corollary to Theorem 2, we can prove the existence of a good UBFV representation for the hidden weighted bit function. For any $k \geq 1$, HWB_k is the function over the k variables $\{x_1, \ldots, x_k\}$ defined by $HWB_k = x_w$, where $w = \textsc{Weight}(\{x_1, \ldots, x_k\})$ and Weight counts the number of variables assigned to 1 in its argument. For the case that $w = 0$, we set $HWB_k = 0$.

Corollary 1. *For $k \geq 1$ there exists a UBFV ψ for HWB_k and secondary variable ordering $\preceq$ such that for each $1 \leq i \leq k$, $\textsc{Sz}(\psi(x_i), \preceq) = \mathcal{O}(k^3)$, and $\textsc{Sz}(HWB_k \circ \psi^*, \preceq) = \mathcal{O}(k^3)$.*

Proof. Let $\{(w_0, f_0, \preceq), \ldots, (w_k, f_k, \preceq)\}$ be the partitioned BDD for HWB_k such that

- $w_j = 1$ iff $j = \textsc{Weight}(x_k, \ldots, x_1)$
- $f_0 = \mathbf{0}$ and $f_j = x_j$ for $1 \leq j \leq k$
- $\preceq$ is any ordering.

Since each w_j is a totally symmetric function, $\textsc{Sz}(w_j, \preceq) = \mathcal{O}(k^2)$ [7], and clearly $\textsc{Sz}(f_j, \preceq) = \mathcal{O}(1)$. The result then follows by Theorem 2. $\square$

V. Discussion

In practice, due to the reduction and sharing inherent in BDDs, the bounds afforded by Theorem 2 can be quite loose. For example, empirically we observe quadratically sized UBFV representations for HWB_k, as plotted in Figure V. Here we give a couple other arm-chair constructions that illustrate the versatility afforded by UBFVs.

A. Inverting an Adder

In this section we give an example of a UBFV that does not resemble the repartitioned UBFV of Sect. IV-B and demonstrates how counter-intuitive some UBFV representations can be. Consider an unsigned modulo-2^n adder; in our formalism, this is a list of n functions $\textsc{Add} = \textsc{Add}_{n-1}, \ldots, \textsc{Add}_0$ over $X \cup Y$, where $X = \{x_{n-1}, \ldots, x_0\}$ and $Y = \{y_{n-1}, \ldots, y_0\}$. An assignment α encodes a binary number on X and Y, when we evaluate the functions of Add at α we obtain a binary encoding of $(X + Y) \mod 2^n$. It is well-known that by using a variable ordering that interleaves X and Y, the BDDs of Add are of linear size. However, one can formulate a UBFV representation of Add with *constant* size simply by exploiting the fact that modulo addition is invertible.

Letting $Z = \{z_{n-1}, \ldots, z_0\}$ be fresh variables, we create a BFV ψ over $(X \cup Y, Z \cup Y)$ such that $\psi(y_i) = y_i$ for each $y_i \in Y$, and $\psi(x_i)$ is the function representing the ith bit of the binary number $Z - Y \mod 2^n$. Universality of ψ follows from the fact that given any naturals $x, y < 2^n$, there exists a natural $z < 2^n$ such that $x = z - y \mod 2^n$. More interestingly, consider $\textsc{Add} \circ \psi^*$ (by which we mean ψ applied to each element of Add piecemeal). Now, slightly abusing notation, $\textsc{Add} = X + Y \mod 2^n$, hence

$$\begin{aligned} \textsc{Add} \circ \psi^* &= (X \circ \psi^*) + (Y \circ \psi^*) \mod 2^n \\ &= (Z - Y + Y) \mod 2^n \\ &= Z \end{aligned}$$

Hence the ith bit of $\textsc{Add} \circ \psi^*$ is simply the variable z_i.

This is rather surprising; we claim that a non-trivial arithmetic function is represented simply by a vector of unique variables. But we must keep in mind that $\textsc{Add} \circ \psi^*$ only represents Add when we know what ψ is. And in a sense, we have simply transposed complexity from the outputs of Add to (half of) its inputs.[3]

B. Disguising a Function as a Variable

Sect. V-A showed how in a particular case we can construct a UBFV that reduces a list of functions to a list of unique variables. In general this is not always possible, but when we consider the output of a single function, we have the following result.

Theorem 3. *Let f be a function that is not identically $\mathbf{1}$ or $\mathbf{0}$. Then there exists a UBFV ψ for f such that $f \circ \psi^*$ is a variable.*

Proof. (Sketch) Let $V = \{v_1, \ldots, v_n\}$ be the variables of f, let $\preceq$ be the natural V-order, and let y be a fresh variable. For all $1 \leq i \leq n$ we define

$$\psi(v_i) = (y \wedge GCF(v_i, f, \preceq)) \vee \left(\overline{y} \wedge GCF(v_i, \overline{f}, \preceq)\right)$$

It can be seen that ψ is a UBFV, and that $f \circ \psi^* = y$; the condition that f is not $\mathbf{1}$ or $\mathbf{0}$ is necessary since the generalized co-factor cannot be taken of $\mathbf{0}$. $\square$

Of course Theorem 3 does not directly save us any BDD complexity since the largest BDD in ψ is the same size as the BDD for f; the UBFV ψ simply moves the complexity of the "output" to the "inputs". However, if f is actually an intermediate function involved in symbolically simulating a

[3]The result is not quite so elegant if we use a non-modulo adder, i.e. one with an $(n+1)$th bit of output. In this case we can construct a similar UBFV such that the UBFV representation of bit i of the output will again be z_i, except when $i = n$, in which case it is a nontrivial function.

specification and/or implementation, and having a nontrivial BDD on f leads to downstream blow-up, it an UBFV along the lines of Theorem 3 could be a remedy.

VI. Checking BFV Universality

So far we have ignored a crucial question: given a BFV ψ, how does one verify that it is a *universal* BFV? This is an important problem to solve; we propose to allow users to concoct intricate, problem-specific UBFVs, and soundness of any proof involving UBFVs hinges on them being universal, hence we require a way to certify BFV universality. This issue is not merely theoretical — when experimenting with UBFVs for the case studies of this paper, on more than one occasion the author ended up with a BFV that subtly failed to be universal. In this section we give several solutions to this problem, varying in automation.

A. Algorithmic Approach

Let us denote the elements of V and V' respectively by $\{v_1, \ldots, v_n\}$ and $\{u_1, \ldots, u_m\}$, and for this section we will consider V and V' to be disjoint. The V-function that is the characteristic of ψ can be expressed as:

$$\exists u_1, \ldots, u_m . \bigwedge_{i=1}^{n} (v_i \leftrightarrow \psi(v_i)) \tag{4}$$

This is simply a logical expression of the set represented by a BFV from Goel and Bryant's paper [13]. It follows that ψ is a UBFV iff (4) is the function $\mathbf{1}$. This can be checked using BDD techniques, although we have found that often computing the BDD for the n-way conjunction or the subsequent existential quantification caused BDD blow-up.

B. Semi-automatic Inverse Approach

A non-automatic, but computationally less demanding approach to checking universality can use BDDs or SAT-solving as a propositional engine. The user provides a proof of universality; the engine then checks the proof via computations that are much simpler than a direct computation of (4). The proof takes the form of an inverse $\psi^{-1} : V' \to ((V \to \mathbb{B}) \to \mathbb{B})$ of ψ. We say "an inverse" since ψ^{-1} is usually not unique. The idea is that for each $v' \in V'$ and V-assignment α, the V'-assignment α' defined by $\alpha'(v') = \psi^{-1}(v')(\alpha)$ is such that $\alpha = \psi^*(\alpha')$. In this way, ψ^{-1} maps α to a "witness" α' such that $\alpha = \psi^*(\alpha')$, the existence of such a witness for each α is tantamount to universality of ψ.

Expounding on this, we employ ψ^{-1} to simplify (4) by using it to pick values for the existentially quantified variables of V', which allows us to strip off the quantifier:

$$\bigwedge_{i=1}^{n} \left(v_i \leftrightarrow (\psi(v_i) \circ \psi^{-1*}) \right) \tag{5}$$

Since we aspire to prove that (5) is tautological, which is the case iff each conjunct is tautological, this reduces to checking tautology of each $v_i \leftrightarrow \psi(v_i) \circ \psi^{-1*}$ individually.

Recollecting the example in the introduction of this paper, suppose we have $V = \{v_1, v_2, v_3\}$, $V' = \{a, b, c, d\}$, and $\psi = \{v_1 \mapsto a \vee b, v_2 \mapsto d, v_3 \mapsto \overline{b}\overline{c}\}$. Then a suitable inverse is $\psi^{-1} = \{a \mapsto v_1, b \mapsto 0, c \mapsto \overline{v_3}, d \mapsto v_2\}$; the reader can check that (5) holds.

instruction	case-splits		runtime		memory	
	classic	UBFV	classic	UBFV	classic	UBFV
SP FADD	113	1	22.1	0.2	2.5	2.4
DP FADD	231	1	51.5	0.5	2.6	6.7
SP FMA	173	5	40.1	2.6	12.8	11.7
DP FMA	2374	28	497.6	47.4	10.3	37.5
PCMPISTRI	82	1	0.7	1.3	2.0	17.1
SP FDIV Pre	335	16	17.2	0.9	4.1	4.1

TABLE I
Results comparing our UBFV approach to the classical approach. Times are in hours and memory usage is the maximum number of GB used by any case-split.

VII. Case Studies

In this section we report on application of the UBFV approach to the problem of verifying hardware implementations of several instructions in recent CPU designs done at Intel. All examples use symbolic simulation with BDDs, specifically using the rSTE symbolic simulator [23] and the underlying forte [25] tool. We contrast our results against the "classic" proofs, which use the same tool suite and were done by Intel FV experts other than the author.[4] The results are summarized in Table VII; note that all results involving a case-split include the time taken to prove that the cases are exhaustive. Note that the UBFVs for these case studies were developed manually prior to the theory of Sect. IV, and thus do not explicitly use the repartitioned BFV construction of that section.

A. Floating Point Addition

Floating Point Addition (FADD) is a family of instructions that perform addition of FP numbers. BDDs for the outputs of *integer* addition are linear in the bit-width of the operands, using the variable order that simply interleaves the operand variables. For FADD, however, the BDDs are exponential. This is because the input mantissas must be aligned, according to the exponent difference *expdiff*, prior to performing the (integral) addition [10], [3], which means there is no good ordering that covers all values of *expdiff* at once. The current state of the art thus involves case-splitting based on *expdiff*. For double precision (DP), there are roughly 2^{12} possible *expdiff*'s, however it is common practice to reduce this significantly by bucketing near-by and extremal *expdiff*'s into the same case. For example, the classic DP proof ran 4 consecutive *expdiff*'s per case, yielding 231 cases total.

We constructed a UBFV for FADD where the mantissa of the second operand m is effectively symbolically shifted by the exponent difference, except in the opposite direction as the FADD alignment. This UBFV pre-shifting has the result (in both the hardware and the specification code) that after

[4]It should be noted that the classic proofs might not be as optimized with regard to case-splitting as possible — as the work was done under project schedule constraints, when the verification engineer achieves a reasonable proof configuration, he or she moves on to other work. Nevertheless, they provide a meaningful benchmark against which to compare our approach.

alignment, the i bit of m collapses to a relatively simple BDD involving just a variable m_i and variables from the exponents. We can think of m_i as representing the bit of m that is weighted the same as the ith bit of the first operand, *after alignment*. Thus, when the symbolic addition is performed and we use an interleaving variable order, the exponential blow-up one faces in the classic proof (without case-splitting) is avoided. Table VII shows the incredible impact this has; for double precision, the 231-way case-split that takes 38 hours is reduced to a single case that takes but half an hour.

B. Fused Multiply-Add

Fused Multiply-Add (FMA) is a three operand FP instruction that computes $x+yz$ in one fell-swoop, incurring only a single rounding error. FVing an FMA design with symbolic simulation requires both the decomposition involving in verifying a multiplier [20], and the wide case-split for FADD discussed above. Our FMA proofs generally follow the decomposition described by Slobodová [27], the final stage of which uses a cut-point at the product p of the mantissas of y and z, and proves that p is added to x and rounded correctly, given the exponents and signs of x, y, and z. This looks more or less like an FADD, with the exception that p is roughly twice as wide as the input mantissa width, which increases BDD complexity and doubles the number of (non-extremal) $\mathit{expdiff}$'s. Thus the case-splitting is more extensive than for FADD.

An aspect of our FMA hardware that proved to be a challenge is the support for *denormal* FP inputs. As a result, the product p can too be denormal in the sense that its leading one can be in any position. We have yet to pin down why this constituted a challenge for our UBVF approach, especially since our FADD examples also supported denormals but they weren't problematic. As a result we needed to employ some case-splitting on top of the UBVF; 5-way for SP and 28-way for DP. This case-splitting involved the conditions $\mathit{expdiff} < -1$, $\mathit{expdiff} \in \{-1, 0\}$, $0 < \mathit{expdiff}$, as well as additional splitting based on the position of the leading one when p is denormal. Nevertheless, we find our results extremely encouraging — even with dozens of machines on which to concurrently run these cases, the classic DP FMA proof still took several days to run, and would often fail due to a machine going down or infrastructure issues. Reducing to under 2 days of compute time is a huge win; the 28 cases with 12 concurrent worker threads only took 6.5 hours of real time. Also, further splitting could reduce the memory footprint significantly; although the maximum memory was 37.5 GB, the average across all 28 cases was only 17.3 GB[5]

Apart from the using the $\mathit{expdiff}$ pre-shifting discussed in Sect. VII-A, our FMA study employed a second trick afforded by the UBFV framework. The mantissa product p, from the specification's point of view, is a (binary encoding of) a single positive integer. However, this number is never calculated explicitly in most hardware designs; but rather appears as a *sum/carry pair* of values. A verification engineer constructs a mapping that appropriately sums these two vectors; the result of which serves as the p input to the specification. Using a trick very similar to that of Sect. V-A, our UBFV was set up so after summing, p collapses to more or less a vector of unique variables, hence simplifying the downstream computations in the specification and the design as well.

C. SSE4 String Instruction

Our third case study is an example string processing instruction from Intel's SSE4 instruction set [2] called PCMPISTRI. Logically[6], this instruction takes two arrays $s1$ and $s2$, each having 8 entries, and each entry being a 16-bit word, and returns $\mathit{ind} \in \{0, \ldots, 8\}$. Let $\mathit{len1}$ (resp. $\mathit{len2}$) be the smallest $i < 8$ such that the $s1[i] = 0$ (resp. $s2[i] = 0$), or 8 if no such i exists. Then the returned value ind is the smallest such that $\mathit{ind} < \mathit{len1}$ and $s1[\mathit{ind}] = s2[j]$ for some $0 \leq j < \mathit{len2}$, or $\mathit{ind} = 8$ if no such ind exists.

The classic proof we looked at involves a decomposition point; there is an internal 8-bit vector $\mathit{IntRes1}$ that is calculated such that $\mathit{IntRes1}[i] = 1$ iff $s1[i] = s2[j]$ for some $0 \leq j < \mathit{len2}$ and $i \leq \mathit{len1}$. Once $\mathit{IntRes1}$ is computed, the return value ind is simply the index of the lowest set bit of $\mathit{IntRes1}$, or $\mathit{ind} = 8$ if it is all 0s. The first stage of the decomposition uses an 81-way case-split, according to the possible values of $(\mathit{len1}, \mathit{len2})$. Referring to Table VII, we note that although the classic proof was a bit faster and used significantly less memory, we still see this result as a "win" for the UBFV approach; we have eliminated the need to decompose the proof, which introduces lots of human effort into the proof (decomposing the specification, mapping to the cut-point in the RTL, etc). Finally, to show that the UBFV win was not simply an artifact of it using more memory, we ran the class proof for the case $(\mathit{len1}, \mathit{len2}) = (8, 8)$ but *without* decomposition – the memory footprint grew to 100 GB after 20 hours of runtime and we killed the process.

D. Floating Point Division

FP Division (FDIV) proofs are highly decomposed [19]. One part of this decomposition involves proving that a pre-processing step (Pre), prior to the main iterative algorithm, is correct. The Pre proof originally required holding 4 bits of the mantissa to constants, yielding 16 cases. In a subsequent chip, support for denormal inputs was added. To perform the analogous case-split for denormals, a second level of case-splitting based on the position of the leading one in the denormal mantissa was needed, resulting in roughly $22{\times}16$ additional cases. We employed a pre-shifted UBFV that allowed us to handle both normal and denormal inputs using only the original 16 cases. This pre-shifting was based on a vector of secondary variables that point to the leading-one position.

We originally deemed UBFV to be overkill for this problem since one can play pre-shifting trickery when crafting the

[5]For the classic FMA approach, we were unable to compile definitive data on maximum memory usage; the given numbers are the memory footprint of the case-split that involved the most BDD nodes, which isn't necessarily the one that used the most memory. Hence they are lower bounds.

[6]Actually what we describe here is how the instruction behaves when the immediate bits are set appropriately.

case-split. However, doing this lead to non-trivial BDDs on the *non-constant* mantissa bits, which incurred BDD blow up. Using our UBFV, these BDDs involve only a single mantissa variable, which makes the BDDs behave very similarly to the purely normal cases. Finally we mention that the same UBFV approach also had a dramatic impact on double precision FDIV Pre, however we were not able to compile data for Table VII in time for this submission.

VIII. Conclusions and Future Work

We have proposed Universal Functional Boolean Vectors as a means of alleviating BDD complexity and demonstrated a profound impact of this approach on difficult hardware data-path FV problems. Though concocting a good UBFV requires human insight, like a BDD variable order, the recipe is often *specification* specific (rather than *implementation* specific) — thus the effort is amortized over many generations of hardware designs. Here we now ponder directions for future work.

One of our case studies showed that it is possible to use UBFVs to make it unnecessary to decompose a proof, for a rather esoteric string processing instruction. But what about classically complex arithmetic functions, for instance multiplication? We conjecture that there does not exist a polynomial-sized UBFV that would elicit a polynomial-sized representation for the list of functions that define the output of integer multiplication, but can this be proven?

The focus here has been on BDD-representations. The other propositional reasoning workhorse is the SAT solver – can UBFV representations be employed to alleviate time complexity in this domain? Are there algorithms and heuristics for generating good UBFVs? Finally, are their applications in other domains, for instance fix-point BDD model checking?

Acknowledgement

Thanks to Sava Krstić and the anonymous reviewers for their suggestions, and to John Franco and Sean Weaver for answering some of our questions.

References

[1] Web version of this paper. http://www.cs.ubc.ca/~jbingham/fmcad2015.pdf.

[2] *Intel SSE4 Programming Reference*. Intel Corporation, 2007.

[3] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Design Automation Conference (DAC 1999)*, July 1999.

[4] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

[5] B. Bollig, M. Lobbing, M. Sauerhoff, and I. Wegener. On the complexity of the hidden weighted bit function for various BDD models. *Theoretical Informatics and Applications*, 33:33–103, 1998.

[6] F. M. Brown. Reduced solutions of boolean equations. *IEEE Trans. Comput.*, 19(10):976–981, Oct. 1970.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, 1986.

[8] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[9] E. Cerny and M. A. Marin. A computer algorithm for the synthesis of memoryless logic circuits. *Computers, IEEE Transactions on*, C-23(5):455–465, May 1974.

[10] Y.-A. Chen and R. Bryant. Verification of floating-point adders. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 488–499. 1998.

[11] O. Coudert and J. Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129, Nov 1990.

[12] J. Franco and S. Weaver. *Handbook of Combinatorial Optimization*, chapter Algorithms for the Satisfiability Problem. Springer, New York, 2013.

[13] A. Goel and R. E. Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, 2003.

[14] A. J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682, 1997.

[15] S. Huddleston, R. Kaivola, and C. Seger. personal communication, 2015.

[16] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. Automatic formal verification of fused-multiply-add fpus. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1298–1303, 2005.

[17] P. Jain and G. Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 13(8):1005–1015, 2006.

[18] R. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Springer US, 2002.

[19] R. Kaivola and K. R. Kohatsu. Proof engineering in the large: formal verification of pentium?4 floating-point divider. *Software Tools for Technology Transfer*, 4(3):323–334, 2003.

[20] R. Kaivola and N. Narasimhan. Formal verification of the pentium 4 floating-point multiplier. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 20–. IEEE Computer Society, 2002.

[21] V. 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*, pages 149–156, Oct 2012.

[22] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs-a compact, canonical and efficiently manipulable representation for boolean functions. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 547–554, Nov 1996.

[23] J. O'Leary, R. Kaivola, and T. Melham. Relational STE and theorem proving for formal verification of industrial circuit designs. In B. Jobstmann and S. Ray, editors, *Formal Methods in Computer-Aided Design (FMCAD 2013)*, pages 97–104. IEEE, Oct. 2013.

[24] V. Paruthi, C. Jacobi, and K. Weber. Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting. In *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, pages 114–128, 2005.

[25] C. J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(9):1381–1405, 2006.

[26] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[27] A. Slobodová. Challenges for formal verification in industrial setting. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2006.

[28] A. Slobodová, J. Davis, S. Swords, and W. A. Hunt. A flexible formal verification framework for industrial scale validation. In S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors, *MEMOCODE*, pages 89–97. IEEE, 2011.

[29] J. Xu, M. Williams, H. Mony, and J. Baumgartner. Enhanced reachability analysis via automated dynamic netlist-based hint generation. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 157–164, Oct 2012.

Compositional Safety Verification with Max-SMT

Marc Brockschmidt [*], Daniel Larraz [†], Albert Oliveras [†], Enric Rodríguez-Carbonell [†] and Albert Rubio [†]
[*]Microsoft Research, Cambridge
[†]Universitat Politècnica de Catalunya

***Abstract*—We present an automated compositional program verification technique for safety properties based on *conditional inductive invariants*. For a given program part (e.g., a single loop) and a postcondition φ, we show how to, using a Max-SMT solver, an inductive invariant together with a precondition can be synthesized so that the precondition ensures the validity of the invariant and that the invariant implies φ. From this, we build a bottom-up program verification framework that propagates preconditions of small program parts as postconditions for preceding program parts. The method recovers from failures to prove the validity of a precondition, using the obtained intermediate results to restrict the search space for further proof attempts.**

As only small program parts need to be handled at a time, our method is scalable and distributable. The derived conditions can be viewed as implicit contracts between different parts of the program, and thus enable an incremental program analysis.

I. Introduction

To have impact on everyday software development, a verification engine needs to be able to process the millions of lines of code often encountered in mature software projects. At the same time, the analysis should be repeated every time developers commit a change, and should report feedback in the course of minutes, so that fixes can be applied promptly. Consequently, a central theme in recent research on automated program verification has been *scalability*. As a natural solution to this problem, *compositional* program analyses [1]–[3] have been proposed. They analyze program parts (semi-)independently and then combine the results to obtain a whole-program proof.

For this, a compositional analysis has to predict likely intermediate assertions that allow us to break whole-program reasoning into many instances of local reasoning. This strategy simplifies the individual reasoning steps and allows distributing the analysis [4]. The disadvantage of compositional analyses has traditionally been their precision: local analyses must blindly choose the intermediate assertions. While in some domains (*e.g.* heap) some heuristics have been found [2], effective strategies for guessing and/or refining useful intermediate assertions or summaries in arithmetic domains remains an open problem.

In this paper we introduce a new method for predicting and refining intermediate arithmetic assertions for compositional reasoning about sequential programs. A key component in our approach is Max-SMT solving. Max-SMT solvers can deal with hard and soft constraints, where hard constraints are mandatory, and soft constraints are those that we would like to hold, but are not required to. Hard constraints express what is needed for the soundness of our analysis, while soft ones favor the solutions that are more useful for our technique. More precisely, we use Max-SMT to iteratively infer *conditional inductive invariants*[1], which prove the validity of a property, given that a precondition holds. Hence, if the precondition holds, the program is proved safe. Otherwise, thanks to a novel program transformation technique we call *narrowing*[2], we exploit the failing conditional invariants to focus on what is missing in the safety proof of the program. Then new conditional invariants are sought, and the process is repeated until the safety proof is finally completed. Based on this, we introduce a new bottom-up program analysis procedure that infers conditional invariants in a goal-directed manner, starting from a property that we wish to prove for the program. Our approach makes distributing analysis tasks as simple as in other bottom-up analyses, but also enjoys the precision of CEGAR-based provers.

II. Illustration of the Method

In this section, we illustrate the core concepts of our approach by using some small examples. We will give the formal definition of the used methods in Sect. IV.

We handle programs by considering one strongly connected component (SCC) $\mathcal{C}$ of the control-flow graph at a time, together with the sequential parts of the program leading to $\mathcal{C}$, either from initial states or other SCCs.

Instead of program invariants, for each SCC we synthesize *conditional inductive invariants*. These are inductive properties such that they may not always hold whenever the SCC is reached, but once they hold, then they are always satisfied.

a) Conditional Inductive Invariants: As an example, consider the program snippet in Fig. 1, where we do not assume any knowledge about the rest of the program. To prove the assertion, we need an inductive property $\mathcal{Q}$ for the loop such that $\mathcal{Q}$ together with the negation of the loop condition $\mathsf{i} > 0$ implies the assertion. Using our constraint-solving based method CondSafe (cf. Sect. IV-A), we find $\mathcal{Q}_1 = \mathsf{x} + 5 \cdot \mathsf{i} \geq 0$. The property $\mathcal{Q}_1$ can be seen as a *precondition* at the loop entry for the validity of the assertion.

```
while i > 0 do
  x := x + 5;
  i := i − 1;
done
assert(x ≥ 0);
```

Fig. 1.

[1]This concept was previously introduced with the name "quasi-invariant" in [5] in the different context of proving program non-termination.

[2]This narrowing is inspired by the narrowing in term rewrite systems, and is unrelated to the notion with the same name used in abstract interpretation.

This work has been supported by Spanish MINECO under the grant TIN2013-45732-C4-3-P (project DAMAS) and the FPI grant BES-2011-044621 (Larraz).

b) Combining Conditional Inductive Invariants: Once we have found a conditional inductive invariant for an SCC, we use the generated preconditions as postconditions for its preceding SCCs in the program.

```
while $j > 0$ do
  $j := j - 1$;
  $i := i + 1$;
done
```

Fig. 2.

As an example, assume that the loop from Fig. 1 is directly preceded by the loop in Fig. 2. We now use the precondition $\mathcal{Q}_1$ we obtained earlier as input to our conditional invariant synthesis method, similarly to the assertion in Fig. 1. Thus, we now look for an inductive property $\mathcal{Q}_2$ that, together with $\neg(j > 0)$, implies $\mathcal{Q}_1$. In this case we obtain the conditional invariant $\mathcal{Q}_2 = j \geq 0 \wedge x + 5 \cdot (i + j) \geq 0$ for the loop. As with $\mathcal{Q}_1$, now we can see $\mathcal{Q}_2$ as a precondition at the loop entry, and propagate $\mathcal{Q}_2$ up to the preceding SCCs in the program.

c) Recovering from Failures: When we cannot prove that a precondition always holds, we try to recover and find an alternative precondition. In this process, we make use of the results obtained so far, and *narrow* the program using our intermediate results. As an example, consider the loop in Fig. 3.

```
while $unknown()$ do
  assert($x \neq y$);
  $x := x + 1$;
  $y := y + 1$;
done
```

Fig. 3.

We again apply our method CondSafe to find a conditional invariant for this loop which, together with the loop condition, implies the assertion in the loop body. As it can only synthesize conjunctions of linear inequalities, it produces the conditional invariant $\mathcal{Q}_3 = x > y$ for the loop. However, assume that the precondition $\mathcal{Q}_3$ could not be proven to always hold in the context of our example. In that case, we use the obtained information to narrow the program and look for another precondition.

```
if $\neg(x > y)$ then
  while $\neg(x > y)$ do
    assert($x \neq y$);
    $x := x + 1$;
    $y := y + 1$;
  done
fi
```

Fig. 4.

Intuitively, our program narrowing reflects that states represented by the conditional invariant found earlier are already proven to be safe. Hence, we only need to consider states for which the negation of the conditional invariant holds, i.e., we can add its negation as an assumption to the program. In our example, this yields the modified version of Fig. 3 displayed in Fig. 4. Another call to CondSafe then yields the conditional invariant $\mathcal{Q}'_3 = x < y$ for the loop. This means that we can ensure the validity of the assertion if before the conditional statement we satisfy that $\neg(x > y) \Rightarrow x < y$, or equivalently, $x \neq y$. In general, this narrowing allows us to find (some) disjunctive invariants.

III. Preliminaries

1) SAT, Max-SAT, and Max-SMT: Let $\mathcal{P}$ be a fixed set of *propositional variables*. For $p \in \mathcal{P}$, p and $\neg p$ are *literals*. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A (CNF) *propositional formula* is a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$. The problem of *propositional satisfiability* (*SAT*) is to determine whether a propositional formula is *satisfiable*. An extension of SAT is *satisfiability modulo theories (SMT)* [6], where satisfiability of a formula with literals from a given background theory is checked. We will use the theory of *quantifier-free integer (non-)linear arithmetic*, where literals are inequalities of linear (resp. polynomial) arithmetic expressions.

Another extension of SAT is *Max-SAT* [6], which generalizes SAT to finding an assignment such that the number of satisfied clauses in a given formula F is maximized. Finally, *Max-SMT* combines Max-SAT and SMT. A *(weighted partial) Max-SMT* problem is a formula of the form $H_1 \wedge \ldots \wedge H_n \wedge [S_1, \omega_1] \wedge \ldots \wedge [S_m, \omega_m]$, where the hard clauses H_i and the soft clauses S_j (with weight ω_j) are disjunctions of literals over a background theory, and the aim is to find a model of the hard clauses that maximizes the sum of the weights of the satisfied soft clauses.

2) Programs and States: We make heavy use of the program structure and hence represent programs as graphs. For this, we fix a set of (integer) program *variables* $\mathcal{V} = \{v_1, \ldots, v_n\}$ and denote by $\mathcal{F}(\mathcal{V})$ the formulas consisting of conjunctions of linear inequalities over the variables $\mathcal{V}$. Let $\mathcal{L}$ be the set of program *locations*, which contains a *canonical start location* ℓ_0. Program *transitions* $\mathcal{T}$ are tuples (ℓ, τ, ℓ'), where ℓ and $\ell' \in \mathcal{L}$ represent the pre- and post-location respectively, and $\tau \in \mathcal{F}(\mathcal{V} \cup \mathcal{V}')$ describes its transition relation. Here $\mathcal{V}' = \{v'_1, \ldots, v'_n\}$ are the *post-variables*, i.e., the values of the variables after the transition.[3] A transition is *initial* if its source location is ℓ_0. A *program* is a set of transitions. We view a program $\mathcal{P} = (\mathcal{L}, \mathcal{T})$ as a directed graph (the *control-flow graph*, CFG), in which edges are the transitions $\mathcal{T}$ and nodes are the locations $\mathcal{L}$.[4]

A *state* $s = (\ell, \boldsymbol{v})$ consists of a location $\ell \in \mathcal{L}$ and a *valuation* $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. A state $(\ell, \boldsymbol{v})$ is *initial* if $\ell = \ell_0$. We denote an *evaluation step* with transition $t = (\ell, \tau, \ell')$ by $(\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v'})$, where the valuations $\boldsymbol{v}$, $\boldsymbol{v'}$ satisfy the formula τ of t. We use $\to_\mathcal{P}$ if we do not care about the executed transition, and $\to^*_\mathcal{P}$ to denote the transitive-reflexive closure of $\to_\mathcal{P}$. We say that a state s is *reachable* if there exists an initial state s_0 such that $s_0 \to^*_\mathcal{P} s$.

3) Safety and Invariants: An *assertion* (t, φ) is a pair of a transition $t \in \mathcal{T}$ and a formula $\varphi \in \mathcal{F}(\mathcal{V})$. A program $\mathcal{P}$ is *safe* for the assertion (t, φ) if for every evaluation $(\ell_0, \boldsymbol{v}_0) \to^*_\mathcal{P} \circ \to_t (\ell, \boldsymbol{v})$, we have that $\boldsymbol{v} \models \varphi$ holds.[5] Note that proving that a formula φ always holds at a location ℓ can be handled in this setting by adding an extra location ℓ^* and an extra transition $t^* = (\ell, \mathrm{true}, \ell^*)$ and checking safety for (t^*, φ).

We call a map $\mathcal{I} : \mathcal{L} \to \mathcal{F}(\mathcal{V})$ a *program invariant* (or often just *invariant*) if for all reachable states $(\ell, \boldsymbol{v})$, we have $\boldsymbol{v} \models \mathcal{I}(\ell)$ holds. An important class of program invariants are inductive invariants. An invariant $\mathcal{I}$ is *inductive* if the following conditions hold:

Initiation:		$\top$	$\models \mathcal{I}(\ell_0)$
Consecution:	For $(\ell, \tau, \ell') \in \mathcal{P}$:	$\mathcal{I}(\ell) \wedge \tau$	$\models \mathcal{I}(\ell')'$

[3]For $\varphi \in \mathcal{F}(\mathcal{V})$, $\varphi' \in \mathcal{F}(\mathcal{V}')$ is the version of φ using primed variables.

[4]Since we label transitions only with conjunctions of linear inequalities, disjunctive conditions are represented using several transitions with the same pre- and post-location. Thus, $\mathcal{P}$ is actually a multigraph.

[5]Here, $\to^*_\mathcal{P} \circ \to_t$ denotes arbitrary program evaluations that end with an evaluation step using t.

4) Constraint Solving for Verification: Inductive invariants can be generated using a *constraint-based* approach [7], [8]. The idea is to consider templates for candidate invariant properties, such as (conjunctions of) linear inequalities. These templates contain both the program variables $\mathcal{V}$ as well as template variables $\mathcal{V}_T$, whose values have to be determined to ensure the required properties. To this end, the conditions on inductive invariants are expressed by means of *constraints* of the form $\exists\, \mathcal{V}_T\, .\forall\, \mathcal{V}\, \ldots$.. Any solution to these constraints then yields an invariant. In the case of linear arithmetic, Farkas' Lemma [9] is often used to handle the quantifier alternation in the generated constraints. Intuitively, it allows one to transform $\exists\forall$ problems encountered in invariant synthesis into $\exists$ problems. In the general case, an SMT problem over non-linear arithmetic is obtained, for which effective SMT solvers exist [10], [11]. By assigning weights to the different conditions, invariant generation can be cast as an optimization problem in the Max-SMT framework [5], [12].

IV. Proving Safety

Most automated techniques for proving program safety iteratively construct *inductive program invariants* as over-approximations of the reachable state space. Starting from the known set of initial states, a process to discover more reachable states and refine the approximation is iterated, until it finally reaches a fixed point (i.e., the invariant is inductive) and is strong enough to imply program safety. However, this requires taking the whole program into account, which is sometimes infeasible or undesirable in practice.

In contrast to this, our method starts with the known unsafe states, and iteratively constructs an under-approximation of the set of safe states, with the goal of showing that all initial states are contained in that set. For this, we introduce the notion of *conditional safety*. Intuitively, when proving that a program is $(\tilde{t}, \tilde{\varphi})$-*conditionally safe for the assertion* (t, φ) we consider evaluations starting after a $\rightarrow_{\tilde{t}} (\tilde{\ell}, \tilde{\boldsymbol{v}})$ step, where $\tilde{\boldsymbol{v}}$ satisfies $\tilde{\varphi}$, instead of evaluations starting at an initial state. In particular, a program that is $(t_0, \top)$-conditionally safe for (t, φ) for all initial transitions t_0 is (unconditionally) safe for (t, φ).

Definition 1 (Conditional safety)**.** *Let $\mathcal{P}$ be a program, $t, \tilde{t}$ transitions and $\varphi, \tilde{\varphi} \in \mathcal{F}(\mathcal{V})$. The program $\mathcal{P}$ is $(\tilde{t}, \tilde{\varphi})$-*conditionally safe *for the assertion (t, φ) if for any evaluation that contains $\rightarrow_{\tilde{t}} (\tilde{\ell}, \tilde{\boldsymbol{v}}) \rightarrow^*_{\mathcal{P}} (\bar{\ell}, \bar{\boldsymbol{v}}) \rightarrow_t (\ell, \boldsymbol{v})$, we have $\tilde{\boldsymbol{v}} \models \tilde{\varphi}$ implies that $\boldsymbol{v} \models \varphi$. In that case we say that the assertion $(\tilde{t}, \tilde{\varphi})$ is a* precondition *for the* postcondition *(t, φ).*

Conditional safety is "transitive" in the sense that if a set of transitions $\mathcal{E} = \{\widetilde{t_1}, \ldots, \widetilde{t_m}\}$ dominates t,[6] and for all $i = 1, \ldots, m$ we have $\mathcal{P}$ is $(\widetilde{t_i}, \widetilde{\varphi}_i)$-conditionally safe for (t, φ) and $\mathcal{P}$ is safe for $(\widetilde{t_i}, \widetilde{\varphi}_i)$, then $\mathcal{P}$ is also safe for (t, φ). In what follows we exploit this observation to prove program safety by means of conditional safety.

A *program component* $\mathcal{C}$ of a program $\mathcal{P}$ is an SCC of the control-flow graph, and its *entry transitions* (or *entries*) $\mathcal{E}_{\mathcal{C}}$ are those transitions $t = (\ell, \tau, \ell')$ such that $t \notin \mathcal{C}$ but ℓ' appears in $\mathcal{C}$. By considering each component as a single node, we can obtain from $\mathcal{P}$ a DAG of SCCs, whose edges are the entry transitions. Our technique analyzes components independently, and communicates the results of these analyses to other components along entry transitions.

[6]We say a set of transitions $\mathcal{E}$ dominates transition t if every path in the CFG from ℓ_0 that contains t must also contain some $\tilde{t} \in \mathcal{E}$.

```
...
while i > 0 do
  x := x + 5;
  i := i − 1;
done
assert(x ≥ 50)
```

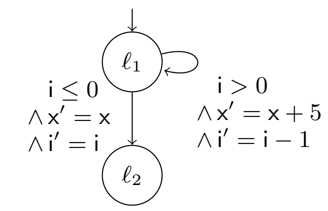

ℓ_1
$\mathsf{i} \le 0$
$\wedge\, \mathsf{x}' = \mathsf{x}$
$\wedge\, \mathsf{i}' = \mathsf{i}$
$\mathsf{i} > 0$
$\wedge\, \mathsf{x}' = \mathsf{x} + 5$
$\wedge\, \mathsf{i}' = \mathsf{i} - 1$
ℓ_2

Fig. 5. Source code of program snippet and its CFG.

Given a component $\mathcal{C}$ and an assertion (t, φ) such that $t \notin \mathcal{C}$ but the source node of t appears in $\mathcal{C}$, we call t an *exit transition* of $\mathcal{C}$. For such exit transitions, we compute a *sufficient* condition $\psi_{\tilde{t}}$ for each entry transition $\tilde{t} \in \mathcal{E}_{\mathcal{C}}$ such that $\mathcal{C} \cup \{t\}$ is $(\tilde{t}, \psi_{\tilde{t}})$-conditionally safe for (t, φ). Then we continue reasoning backwards following the DAG and try to prove that $\mathcal{P}$ is safe for each $(\tilde{t}, \psi_{\tilde{t}})$. If we succeed, following the argument above we will have proved $\mathcal{P}$ safe for (t, φ).

In the following, we first discuss how to prove conditional safety of single program components in Sect. IV-A, and then present the algorithm that combines these local analyses to construct a global safety proof in Sect. IV-B.

A. Synthesizing Local Conditions

Here we restrict ourselves to a program component $\mathcal{C}$ and its entry transitions $\mathcal{E}_{\mathcal{C}}$, and assume we are given an assertion $(t_{\text{exit}}, \varphi)$, where $t_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \tau_{\text{exit}}, \ell_{\text{exit}})$ is an exit transition of $\mathcal{C}$ (i.e., $t_{\text{exit}} \notin \mathcal{C}$ and $\tilde{\ell}_{\text{exit}}$ appears in $\mathcal{C}$). We show how a precondition (t, ψ) for $(t_{\text{exit}}, \varphi)$ can be obtained for each $t \in \mathcal{E}_{\mathcal{C}}$. Here we only consider the case of φ being a single clause (i.e., a disjunction of literals); if φ is in CNF, each conjunct is handled separately. The preconditions on the entry transitions will be determined by a *conditional inductive invariant*, which like a standard invariant is inductive, but not necessarily initiated in all program runs. Indeed, this initiation condition is what we will extract as precondition and propagate backwards to preceding program components in the DAG.

Definition 2 (Conditional Inductive Invariant)**.** *We say a map $\mathcal{Q} : \mathcal{L} \rightarrow \mathcal{F}(\mathcal{V})$ is a* conditional (inductive) invariant *for a program (component) $\mathcal{P}$ if for all $(\ell, \boldsymbol{v}) \rightarrow_{\mathcal{P}} (\ell', \boldsymbol{v}')$, we have $\boldsymbol{v} \models \mathcal{Q}(\ell)$ implies $\boldsymbol{v}' \models \mathcal{Q}(\ell')$.*

Conditional invariants are convenient tools to express conditions for safety proving, allowing reasoning in the style of "if the condition for $\mathcal{Q}$ holds, then the assertion (t, φ) holds".

Example 1. *Consider the program snippet in Fig. 5. A conditional inductive invariant supporting safety of this program part is $\mathcal{Q}_5(\ell_1) \equiv \mathsf{x} + 5 \cdot \mathsf{i} \ge 50$, $\mathcal{Q}_5(\ell_2) \equiv \mathsf{x} \ge 50$. In fact, any conditional invariant $\mathcal{Q}_m(\ell_1) \equiv \mathsf{x} + m \cdot \mathsf{i} \ge 50$ with $0 \le m \le 5$*

would be a conditional inductive invariant that, together with the negation of the loop condition $\mathsf{i} \leq 0$*, implies* $\mathsf{x} \geq 50$.

We use a Max-SMT-based constraint-solving approach to generate conditional inductive invariants. Unlike in [5], to use information about the initialization of variables before a program component, we take into account the entry transitions $\mathcal{E}_\mathcal{C}$. The precondition for each entry transition is the conditional invariant that has been synthesized at its target location.

To find conditional invariants, we construct a constraint system. For each location ℓ in $\mathcal{C}$ we create a template $I_{\ell,k}(\mathcal{V}) \equiv \wedge_{1 \leq j \leq k} I_{\ell,j,k}(\mathcal{V})$ which is a conjunction of k linear inequations[7] of the form $I_{\ell,j,k}(\mathcal{V}) \equiv i_{\ell,j} + \sum_{v \in \mathcal{V}} i_{\ell,j,v} \cdot v \leq 0$, where the $i_{\ell,j}$, $i_{\ell,j,v}$ are fresh variables from the set of template variables $\mathcal{V}_T$. We then transform the conditions for a conditional invariant proving safety for the assertion $(t_{\text{exit}}, \varphi)$ to the constraints in Fig. 6. Here, e.g., $I'_{\ell',k}$ refers to the variant of $I_{\ell',k}$ using primed versions of the program variables $\mathcal{V}$, but unprimed template variables $\mathcal{V}_T$.

In the overall constraint system, we mark the **Consecution** and **Safety** constraints as hard requirements. Thus, any solution to these constraints is a conditional *inductive* invariant *implying our assertion*. However, as we mark the **Initiation** constraints as soft, the found conditional invariants may depend on preconditions not implied by the direct context of the considered component. On the other hand, the Max-SMT solver prefers solutions that require fewer preconditions. Overall, we create the following Max-SMT formula

$$\mathbb{F}_k \overset{def}{=} \bigwedge_{t \in \mathcal{C}} \mathbb{C}_{t,k} \wedge \bigwedge_{t \in \mathcal{E}_\mathcal{C}, 1 \leq j \leq k} (\mathbb{I}_{t,j,k} \vee \neg p_{\mathbb{I}_{t,j,k}}) \wedge \mathbb{S}_k \wedge \bigwedge_{t \in \mathcal{E}_\mathcal{C}, 1 \leq j \leq k} [p_{\mathbb{I}_{t,j,k}}, \omega_\mathbb{I}],$$

where the $p_{\mathbb{I}_{t,j,k}}$ are propositional variables which are true if the **Initiation** condition $\mathbb{I}_{t,j,k}$ is satisfied, and $\omega_\mathbb{I}$ is the corresponding weight. [8] We use $\mathbb{F}_k$ in our procedure CondSafe in Algo. 1.

Algorithm 1 Proc. CondSafe computing conditional invariant

Input: component $\mathcal{C}$, entry transitions $\mathcal{E}_\mathcal{C}$, assertion $(t_{\text{exit}}, \varphi)$ s.t. t_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause

Output: None | $\mathcal{Q}$, where $\mathcal{Q}$ maps locations in $\mathcal{C}$ to conjunctions of inequations

```
1: $k \leftarrow 1$
2: repeat
3:     construct formula $\mathbb{F}_k$ from $\mathcal{C}$, $\mathcal{E}_\mathcal{C}$ and $(t_{\text{exit}}, \varphi)$
4:     $\sigma \leftarrow$ Max-SMT-solver($\mathbb{F}_k$)
5:     if $\sigma$ is a model then
6:         $\mathcal{Q} \leftarrow \{\ell \mapsto \sigma(I_{\ell,k}) \mid \ell$ in $\mathcal{C}\}$ return $\mathcal{Q}$
7:     $k \leftarrow k + 1$
8: until $k >$ MAX_CONJUNCTS  return None
```

In CondSafe, we iteratively try "larger" templates of more conjuncts of linear inequations until we either give up (in our implementation, MAX_CONJUNCTS is 3) or finally find a conditional invariant. Note, however, that here we are only trying to prove safety for *one* clause at a time, which reduces the number of required conjuncts as compared to dealing with a whole CNF in a single step. If the Max-SMT solver is able to find a model for $\mathbb{F}_k$, then we instantiate our invariant templates $I_{\ell,k}$ with the values found for the template variables in the model σ, obtaining a conditional invariant $\mathcal{Q}$. When we obtain a result, for every entry transition $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$ the conditional invariant $\mathcal{Q}(\ell')$ is a precondition that implies safety for the assertion $(t_{\text{exit}}, \varphi)$. The following theorem states the correctness of this procedure.

Theorem 1. *Let* $\mathcal{C}$ *be a component,* $\mathcal{E}_\mathcal{C}$ *its entry transitions, and* $(t_{\text{exit}}, \varphi)$ *an assertion with* t_{exit} *an exit transition of* $\mathcal{C}$ *and* φ *a clause. If the procedure call* CondSafe$(\mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi))$ *returns* $\mathcal{Q} \neq$ None*, then* $\mathcal{Q}$ *is a conditional inductive invariant for* $\mathcal{C}$ *and* $\mathcal{P}$ *is* $(t, \mathcal{Q}(\ell'))$*-conditionally safe for* $(t_{\text{exit}}, \varphi)$ *for all* $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$*.*

Proof. All proofs can be found in our technical report [13]. □

B. Propagating Local Conditions

In this section, we explain how to use the local procedure CondSafe to prove safety of a full program. To this end we now consider the full DAG of program components. As outlined above, the idea is to start from the assertion provided by the user, call the procedure CondSafe to obtain preconditions for the entry transitions of the corresponding component, and then use these preconditions as assertions for preceding components, continuing recursively. If eventually for each initial transition the transition relation implies the corresponding preconditions, then safety has been proven. If we fail to prove safety for certain assertions, we backtrack, trying further possible preconditions and conditional invariants.

The key to the precision of our approach is our treatment of failed proof attempts. When the procedure CondSafe finds a conditional invariant $\mathcal{Q}$ for $\mathcal{C}$, but proving $(t, \mathcal{Q}(\ell'))$ as a postcondition of the preceding component fails for some $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$, we use $\mathcal{Q}$ to *narrow* our program representation and filter out evaluations that are already known to be safe.

As outlined above, in our proof process we treat each clause of the conjunction $\mathcal{Q}(\ell')$ separately, and pass each one as its own assertion to preceding program components, allowing for a fine-grained *program-narrowing* technique. By construction of $\mathcal{Q}$, evaluations that satisfy all literals of $\mathcal{Q}(\ell')$ after executing $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$ are safe. Thus, among the evaluations that use t, we only need to consider those where at least one literal in $\mathcal{Q}(\ell')$ does not hold. Hence, we *narrow* each entry transition by conjoining it with the negation of the conjunction of all literals for which we could not prove safety (see line 13 in Algo. 2). Note that if there is more than one literal in this conjunction, then the negation is a disjunction, which in our program model implies splitting transitions.

We can narrow program components similarly. For a transition $t = (\ell, \tau, \ell') \in \mathcal{C}$, we know that if either $\mathcal{Q}(\ell)$ or $\mathcal{Q}(\ell')'$ holds in an evaluation passing through t, the program

[7]In our overall algorithm, k is initially 1 and increased in case of failures.

[8]Farkas' Lemma is applied *locally* to the subformulas $\mathbb{C}_{t,k}$, $\mathbb{I}_{t,j,k}$ and $\mathbb{S}_k$, and weights are added on the resulting constraints over the template variables.

$$
\begin{array}{llrcrl}
\textbf{Initiation:} & \text{For } t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}, 1 \leq j \leq k\text{:} & \mathbb{I}_{t,j,k} & \overset{def}{=} & \tau & \Rightarrow I'_{\ell',j,k} \\
\textbf{Consecution:} & \text{For } t = (\ell, \tau, \ell') \in \mathcal{C}\text{:} & \mathbb{C}_{t,k} & \overset{def}{=} & I_{\ell,k} \wedge \tau & \Rightarrow I'_{\ell',k} \\
\textbf{Safety:} & \text{For } t_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \tau_{\text{exit}}, \ell_{\text{exit}})\text{:} & \mathbb{S}_k & \overset{def}{=} & I_{\tilde{\ell}_{\text{exit}},k} \wedge \tau_{\text{exit}} & \Rightarrow \varphi'
\end{array}
$$

Fig. 6. Constraints used in $\mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_{\mathcal{C}}, (t_{\text{exit}}, \varphi))$

Algorithm 2 Procedure CheckSafe for proving a program safe for an assertion

```
Input: Program $\mathcal{P}$, a component $\mathcal{C}$, entries $\mathcal{E}_\mathcal{C}$, assertion
    $(t_{\text{exit}}, \varphi)$ s.t. $t_{\text{exit}}$ is an exit transition of $\mathcal{C}$ and $\varphi$ a clause
Output: Safe | Maybe
 1: let $(\ell_{\text{exit}}, \tau_{\text{exit}}, \ell'_{\text{exit}}) = t_{\text{exit}}$
 2: if $(\tau_{\text{exit}} \Rightarrow \varphi')$ then return Safe
 3: else if $\ell_{\text{exit}} = \ell_0$ then return Maybe
 4: $\mathcal{Q} \leftarrow \mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi))$
 5: if $\mathcal{Q} = $ None then return Maybe
 6: for all $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$, $L \in \mathcal{Q}(\ell')$ do
 7:     $\tilde{\mathcal{C}} \leftarrow \mathsf{component}(\ell, \mathcal{P})$
 8:     $\mathcal{E}_{\tilde{\mathcal{C}}} \leftarrow \mathsf{entries}(\tilde{\mathcal{C}}, \mathcal{P})$
 9:     $\mathsf{res}[t, L] \leftarrow \mathsf{CheckSafe}(\mathcal{P}, \tilde{\mathcal{C}}, \mathcal{E}_{\tilde{\mathcal{C}}}, (t, L))$
10: if $\forall t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}, L \in \mathcal{Q}(\ell')$ . $\mathsf{res}[t, L] = $ Safe then
11: return Safe
12: else
13:     $\hat{\mathcal{E}}_\mathcal{C} \leftarrow \{(\ell, \tau \wedge \neg(\bigwedge_{\substack{L \in \mathcal{Q}(\ell') \\ \mathsf{res}[t,L] = \mathsf{Maybe}}} L'), \ell') \mid t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}\}$
14:     $\hat{\mathcal{C}} \leftarrow \{(\ell, \tau \wedge \neg \mathcal{Q}(\ell')' \wedge \neg \mathcal{Q}(\ell), \ell') \mid (\ell, \tau, \ell') \in \mathcal{C}\}$
15: return $\mathsf{CheckSafe}(\mathcal{P}, \hat{\mathcal{C}}, \hat{\mathcal{E}}_\mathcal{C}, (t_{\text{exit}}, \varphi))$
```

is safe. Thus, we narrow the program by replacing τ by $\tau \wedge \neg\, \mathcal{Q}(\ell) \wedge \neg\, \mathcal{Q}(\ell')'$ (see line 14 in Algo. 2).

This narrowing allows us to generate disjunctive conditional invariants, where each result of CondSafe is one disjunct. Note that not *all* disjunctive invariants can be discovered like this, as each intermediate result needs to be inductive using the disjuncts found so far. However, this is the pattern observed in *phase-change* algorithms [14].

Our overall safety proving procedure CheckSafe is shown in Algo. 2. The helper procedures component and entries are used to find the program component for a given location and the entry transitions for a component. The result of CheckSafe is either Maybe when the proof failed, or Safe if it succeeded. In the latter case, we have managed to create a chain of conditional invariants that imply that $(t_{\text{exit}}, \varphi)$ always holds.

Theorem 2. *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component and $\mathcal{E}_\mathcal{C}$ its entries. Given an assertion $(t_{\text{exit}}, \varphi)$ such that t_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause, if* $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi)) = $ *Safe, then $\mathcal{P}$ is safe for* $(t_{\text{exit}}, \varphi)$.

Example 2. *We demonstrate* CheckSafe *on the program displayed on Fig. 7, called $\mathcal{P}$ in the following, which is an extended version of the example from Fig. 3.*

We want to prove the assertion $(t_5, \mathsf{x} \neq \mathsf{y})$. Hence we make a first call $\mathsf{CheckSafe}(\mathcal{P}, \{t_4\}, \{t_3\}, (t_5, \mathsf{x} {\neq} \mathsf{y}))$*: the non-trivial SCC containing ℓ_2 is $\{t_4\}$ and its entry transitions are $\{t_3\}$. Hence, we call* $\mathsf{CondSafe}(\{t_4\}, \{t_3\}, (t_5, \mathsf{x} \neq \mathsf{y}))$ *and the resulting conditional invariant for ℓ_2 is either $\mathsf{x} < \mathsf{y}$ or $\mathsf{y} < \mathsf{x}$. Let us assume it is $\mathsf{y} < \mathsf{x}$. In the next step, we propagate this to the predecessor SCC $\{t_2\}$, and call* $\mathsf{CheckSafe}(\mathcal{P}, \{t_2\}, \{t_1\}, (t_3, \mathsf{y} < \mathsf{x}))$.

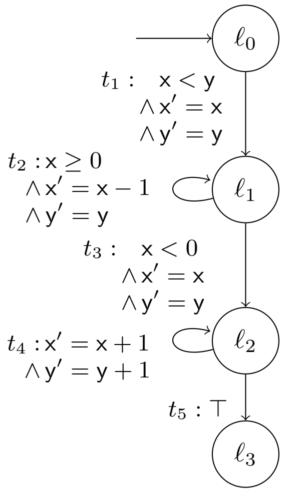
In turn, this leads to calling $\mathsf{CondSafe}(\{t_2\}, \{t_1\}, (t_3, \mathsf{y} < \mathsf{x}))$ *to our synthesis subprocedure. No conditional invariant supporting this assertion can be found, and hence None is returned by* CondSafe*, and consequently Maybe is returned by* CheckSafe*. Hence, we return to the original SCC $\{t_4\}$ and its entry $\{t_3\}$, and then by narrowing we obtain two new transitions:*

$$t'_4 = (\ell_2, \mathsf{x}' = \mathsf{x} + 1 \wedge \mathsf{y}' = \mathsf{y} + 1 \wedge \neg(\mathsf{y} < \mathsf{x}), \ell_2),$$
$$t'_3 = (\ell_1, \mathsf{x} < 0 \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y} \wedge \neg(\mathsf{y} < \mathsf{x}), \ell_2).$$

Using these, we call $\mathsf{CheckSafe}(\mathcal{P}, \{t'_4\}, \{t'_3\}, (t_5, \mathsf{x} {\neq} \mathsf{y}))$*. The next call to* CondSafe *then yields the conditional invariant $\mathsf{x} < \mathsf{y}$ at ℓ_2, which is in turn propagated backwards with the call* $\mathsf{CheckSafe}(\mathcal{P}, \{t_2\}, \{t_1\}, (t'_3, \mathsf{x} < \mathsf{y}))$*. This then yields a conditional invariant $\mathsf{x} < \mathsf{y}$ at ℓ_1, which is finally propagated back in the call* $\mathsf{CheckSafe}(\mathcal{P}, \{\}, \{\}, (t_1, \mathsf{x} < \mathsf{y}))$*, which directly returns Safe.*

C. Improving Performance

The basic method CheckSafe can be extended in several ways to improve performance. We now present a number of techniques that are useful to reduce the runtime of the algorithm and distribute the required work. Note that none of these techniques influences the precision of the overall framework.



ℓ_0
$t_1 : \mathsf{x} < \mathsf{y} \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}$
$t_2 : \mathsf{x} \geq 0 \wedge \mathsf{x}' = \mathsf{x} - 1 \wedge \mathsf{y}' = \mathsf{y}$
ℓ_1
$t_3 : \mathsf{x} < 0 \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}$
$t_4 : \mathsf{x}' = \mathsf{x} + 1 \wedge \mathsf{y}' = \mathsf{y} + 1$
ℓ_2
$t_5 : \top$
ℓ_3

Fig. 7.

a) Using conditional invariants to disable transitions: When proving an assertion, it is often necessary to find invariants that show the unfeasibility of some transition, which allows disabling it. In our framework, the required invariants can be conditional as well. Therefore, CheckSafe must be called recursively to prove that the conditional invariant is indeed invariant. In our implementation, we generate constraints such that every solution provides conditional invariants either implying the postcondition or disabling some transition. By imposing different weights, we make the Max-SMT solver prefer solutions that imply the postcondition.

b) Handling unsuccessful proof attempts: One important aspect is that the presented algorithm does not *learn* facts about the reachable state space, and so duplicates work when assertions appear several times. To alleviate this for *unsuccessful* recursive invocations of CheckSafe, we introduce a simple memoization technique to avoid repeating such calls. So when $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (t, \varphi)) = \mathsf{Maybe}$, we store this result, and use it for all later calls of $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (t, \varphi))$. This strategy is valid as the return value Maybe indicates that our method cannot prove the assertion (t, φ) at all, meaning that later proof attempts will fail as well. In our implementation, this memoization of unsuccessful attempts is local to the initial call to CheckSafe. The rationale is that, when proving unrelated properties, it is likely that few calls are shared and that the book-keeping does not pay off.

c) Handling successful proof attempts: When a recursive call yields a *successful* result, we can *strengthen* the program with the proven invariant. Remember that $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (t, \varphi)) = \mathsf{Safe}$ means that whenever the transition t is used in *any* evaluation, φ holds in the succeeding state. Thus, we can add this knowledge explicitly and change the transition in the original program. In practice, this strengthening is applied only if the first call to CheckSafe was successful, i.e, no narrowing was applied. The reason is that, if the transition relation of t was obtained through repeated narrowing, in general one needs to split transitions, and it is not correct to just add φ' to t. Namely, assume that $t_o = (\ell, \tau_o, \ell')$ is the original (unnarrowed) version of a transition $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$. As t is an entry transition of $\mathcal{C}$, we have $\tau = \tau_o \wedge \neg\psi'_1 \wedge \ldots \wedge \neg\psi'_m$ by construction, where ψ_i is the additional constraint we added in the i-th narrowing of component entries. Thus what we proved is that $\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi'$ always holds after using transition t_o. So we should replace t_o in the program with a transition labeled with $\tau_o \wedge (\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi')$. As we cannot handle disjunctions natively, this implies replacing t_o by $m + 1$ new transitions.

Note that this program modification approach, unlike memoization, makes the gained information available to the Max-SMT solver when searching for a conditional invariant. A similar strategy can be used to strengthen the transitions in the considered component $\mathcal{C}$.

d) Parallelizing & distributing the analysis: Our analysis can easily be parallelized. We have implemented this at two stages. First, at the level of the procedure CondSafe, we try at the same time different numbers of template conjuncts (lines 3-6 in Algo. 1), which requires calling several instances of the solver simultaneously. Secondly, at a higher level, the recursive calls of CheckSafe (line 9 in Algo. 2) are parallelized. Note that, since narrowing and the "learning" optimizations described above are considered only locally, they can be handled as asynchronous updates to the program kept in each worker, and do not require synchronization operations. Hence, distributing the analysis onto several worker processes, in the style of Bolt [4], would be possible as well.

Other directions for parallelization, which have not been implemented yet, are to return different conditional invariants in parallel when the Max-SMT problem in procedure CondSafe has several solutions. Moreover, based on experimental observations that successful safety proofs have a short successful path in the tree of proof attempts, we are also interested in exploring a look-ahead strategy: after calling CondSafe in CheckSafe, we could make recursive calls of CheckSafe on some processes while others are already applying narrowing.

e) Iterative proving: Finally, one could store the conditional invariants generated during a successful proof, which are hence invariants, so that they can be re-used in later runs. E.g., if a single component is modified, one can reprocess it and compute a new precondition that ensures its postcondition. If this precondition is implied by the previously computed invariant, the program is safe and nothing else needs to be done. Otherwise, one can proceed with the preceding components, and produce respective new preconditions in a recursive way. Only when proving safety with the previously computed invariants in this way fails, the whole program needs to be reprocessed again. This technique has not been implemented yet, as our prototype is still in a preliminary state.

V. Related Work

Safety proving is an active area of research. In the recent past, techniques based on variations of counterexample guided abstraction refinement have dominated [15]–[24]. These methods prove safety by repeatedly unfolding the program relation using a symbolic representation of program states, starting in the initial states. This process generates an over-approximation of the set of reachable states, where the coarseness of the approximation is a consequence of the used symbolic representation. Whenever a state in the over-approximation violates the safety condition, either a true counterexample was found and is reported, or the approximation is refined (using techniques such as predicate abstraction [25] or Craig interpolation [26]). When further unwinding does not change the symbolic representation, all reachable states have been found and the procedure terminates. This can be understood as a "top-down" ("forward") approach (starting from the initial states), whereas our method is "bottom-up" ("backwards"), i.e. starting from the assertions.

Techniques based on Abstract Interpretation [27] have had substantial success in the industrial setting. There, an abstract interpreter is instantiated by an abstract domain whose elements are used to over-approximate sets of program states. The interpreter then evaluates the program on the chosen abstract domain, discovering reachable states. A widening operator, combining two given over-approximations to a more general one representing both, is employed to guarantee termination of the analysis when handling loops.

"Bottom-up" safety proving with preconditions found by abduction has been investigated in [28]. This work is closest to ours in its overall approach, but uses fundamentally different techniques to find preconditions. Instead of applying Max-SMT, the approach uses an abduction engine based on maximal universal subsets and quantifier elimination in Presburger arithmetic. Moreover, it does not have an equivalent to our narrowing to exploit failed proof attempts. In a similar vein, [29] uses straight-line weakest precondition computation and backwards-reasoning to infer loop invariants supporting validity of an assertion. To enforce a generalization towards inductive invariants, a heuristic syntax-based method is used.

Automatically constructing program proofs from independently obtained subproofs has been an active area of research in the recent past. Splitting proofs along syntactic boundaries (e.g., handling procedures separately) has been explored in [1], [2], [4], [30]. For each such unit, a summary of its behavior is computed, i.e., an expression that connects certain (classes of) inputs to outputs. Depending on the employed analyzers, these summaries encode under- and over-approximations of reachable states [1] or changes to the heap using separation logic's frame rule [2]. Finally, [4] discusses how such compositional analyses can leverage cloud computing environments to parallelize and scale up program proofs.

VI. Implementation and Evaluation

We have implemented the algorithms from Sect. IV-A and Sect. IV-B in our early prototype VeryMax, using the Max-SMT solver for non-linear arithmetic [31] in the Barcelogic [32] system. We evaluated a sequential (VeryMax-Seq) and a parallel (VeryMax-Par) variant on two benchmark sets.

The first set (which we will call HOLA-BENCHS) are the 46 programs from the evaluation of safety provers in [28] (which were collected from a variety of sources, among others, [14], [33]–[43], the NECLA Static Analysis Benchmarks, etc.). The programs are relatively small (they have between 17 and 71 lines of code, and between 1 and 4 nested or consecutive loops), but expose a number of "hard" problems for analyzers. All of them are safe.

On this first benchmark set we compare with three systems. The first two were leading tools in the Software Verification Competition 2015 [44]: CPAchecker[9] [45], which was the overall winner and in particular won the gold medal in the *"Control Flow and Integer Variables"* category, and SeaHorn [46], which got the silver medal, and also won the *"Simple"* category. We also compare with HOLA [28], an abduction-based backwards reasoning tool. Unfortunately, we were not able to obtain an executable for HOLA. For this reason we have taken the experimental data for this tool directly from [28], where it is reported that the experiments were performed on an Intel i5 2.6 GHz CPU with 8 Gb of memory. For the sake of a fair comparison, we have run the other tools on a 4-core machine with the same specification, using the same timeout of 200 seconds. Tab. I summarizes the results, reporting the number of successful proofs, failed proofs, and timeouts (TO), together with the respective total runtimes. Both versions of VeryMax are competitive, and our parallel version was two times faster than our sequential one on four cores. As a reference, on these examples VeryMax-Seq needed 2.8 overall calls (recursive or after narrowings) on average, with a maximum of 16. The number of narrowings was approximately 1, with a maximum of 13. Our memoization technique making use of already failed proof attempts was employed in about one third of the cases.

[9]We ran CPAchecker with two different configurations, predicateAnalysis and sv-comp15.

In our second benchmark set (which we will refer to as NR-BENCHS) we have used integer abstractions of 217 numerical algorithms from [47]. For each procedure and for each array access in it, we have created two safety problems with one assertion each, expressing that the index is within bounds. In some few cases the soundness of array accesses in the original program depends on properties of floating-point variables, which are abstracted away. So in the corresponding abstraction some assertions may not hold. Altogether, the resulting benchmark suite consists of 6452 problems, of up to 284 lines of C code. Due to the size of this set, and to give more room to exploit parallelism (both tools with which we compare on these benchmarks, CPAchecker and SeaHorn, make use of several cores), we performed the experiments with a more powerful machine, namely, an 8-core Intel i7 3.4 GHz CPU with 16 GB of memory. The time limit is 300 seconds.

The results can be seen in Tab. II. On these instances, VeryMax is able to prove more assertions than any of the other tools, while being about as fast as SeaHorn, and significantly faster than CPAchecker. Note that many examples are solved very quickly in the sequential solver already, and thus do not profit from our parallelization. VeryMax is at an early stage of development, and is hence not yet fully tuned. For example, a number of program slicing techniques have not been implemented yet, which would be very useful for handling larger programs. Thus, we expect that further development will improve the tool performance significantly. The benchmarks and our tool can be found at http://www.cs.upc.edu/~albert/VeryMax.html.

VII. Conclusion

We have presented a novel approach to compositional safety verification. Our main contribution is a proof framework that refines intermediate results produced by a Max-SMT-based precondition synthesis procedure. In contrast to most earlier work, we proceed *bottom-up* to compute summaries of code that are guaranteed to be relevant for the proof.

We plan to further extend VeryMax to cover more program features and include standard optimizations (e.g., slicing and constraint propagation with simple abstract domains). It currently handles procedure calls by inlining, and does not support recursive functions yet. However, they can be handled by introducing templates for function pre/postconditions.

In the future, we are interested in experimenting with alternative precondition synthesis methods (e.g., abduction-based ones). We also want to combine our method with a Max-SMT-based termination proving method [12] and extend it to *existential* properties such as reachability and non-termination [5]. We expect to combine all of these techniques in an *alternating* procedure [1] that tries to prove properties at

Tool	Safe	Σ s	Fail	Σ s	TO	Total s
CPAchecker sv-comp15	33	2424.41	3	61.28	10	4489.73
CPAchecker predicateAnalysis	25	503.05	11	19.72	10	2271.12
SeaHorn	32	7.95	13	3.477	1	211.56
HOLA	43	23.53	0	0	3	623.53
VeryMax-Seq	44	293.14	2	50.69	0	343.83
VeryMax-Par	45	138.40	1	12.81	0	151.21

TABLE I
EXPERIMENTAL RESULTS ON HOLA-BENCHS BENCHMARK SET.

Tool	Safe	Σ s	Unsafe	Σ s	Fail	Σ s	TO	Total s
CPAchecker sv-comp15	5570	614803.98	251	6188.30	326	28749.78	305	735336.82
CPAchecker predicateAnalysis	5928	23417.15	170	495.13	234	9105.69	120	64652.29
SeaHorn	6077	4276.21	233	135.25	80	529.09	62	24167.11
VeryMax-Seq	6105	5940.88	0	0	326	26739.30	21	38980.80
VeryMax-Par	6106	4789.73	0	0	346	18878.42	0	23668.15

TABLE II
EXPERIMENTAL RESULTS ON NR-BENCHS BENCHMARK SET.

the same time as their duals, and which uses partial proofs to narrow the state space that remains to be considered. Eventually, these methods could be combined to verify arbitrary temporal properties. In another direction, we want to consider more expressive theories to model program features such as arrays or the heap.

REFERENCES

[1] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *POPL*, 2009.

[2] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *JACM*, vol. 58, no. 6, 2011.

[3] B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv, "Synthesis of circular compositional program proofs via abduction," in *TACAS*, 2013.

[4] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," in *PLDI*, 2012.

[5] D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving non-termination using Max-SMT," in *CAV*, 2014.

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

[7] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear Invariant Generation Using Non-linear Constraint Solving," in *CAV*, 2003.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma, "Linear ranking with reachability," in *CAV*, 2005.

[9] A. Schrijver, *Theory of Linear and Integer Programming*. Wiley, 1998.

[10] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT Modulo Linear Arithmetic for Solving Polynomial Constraints," *JAR*, vol. 48, no. 1, 2012.

[11] D. Jovanovic and L. M. de Moura, "Solving non-linear arithmetic," in *IJCAR*, 2012.

[12] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving termination of imperative programs using Max-SMT," in *FMCAD*, 2013.

[13] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Compositional Safety Verification with Max-SMT," 2015, http://arxiv.org/abs/1507.03851.

[14] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *CAV*, 2011.

[15] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *CAV*, 2001.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *SPIN*, 2003.

[17] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, 2005.

[18] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.

[19] A. Podelski and A. Rybalchenko, "ARMC: The logical choice for software model checking with abstraction refinement," in *PADL*, 2007.

[20] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011.

[21] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.

[22] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Whale: an interpolation-based algorithm for inter-procedural verification," in *VMCAI*, 2012.

[23] A. Cimatti and A. Griggio, "Software model checking via IC3," in *CAV*, 2012.

[24] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2014.

[25] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *POPL*, 2002.

[26] K. L. McMillan, "Craig interpolation and reachability analysis," in *SAS*, 2003.

[27] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977.

[28] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, 2013.

[29] C. S. Pasareanu and W. Visser, "Verification of Java programs using symbolic execution and invariant generation," in *SPIN*, 2004.

[30] G. Yorsh, E. Yahav, and S. Chandra, "Generating precise and concise procedure summaries," in *POPL*, 2008.

[31] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Minimal-model-guided approaches to solving polynomial constraints and extensions," in *SAT*, 2014.

[32] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The barcelogic SMT solver," in *CAV*, 2008.

[33] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *CAV*, 2009.

[34] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *PLDI*, 2008.

[35] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Path invariants," in *PLDI*, 2007.

[36] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Automatically refining abstract interpretations," in *TACAS*, 2008.

[37] A. R. Bradley and Z. Manna, "Property-directed incremental invariant generation," *Formal Asp. Comput.*, vol. 20, no. 4-5, 2008.

[38] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, 2006.

[39] R. Jhala and K. L. McMillan, "A practical and complete approach to predicate refinement," in *TACAS*, 2006.

[40] R. Sharma, A. V. Nori, and A. Aiken, "Interpolants as classifiers," in *CAV*, 2012.

[41] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "SYNERGY: a new algorithm for property checking," in *FSE*, 2006.

[42] B. S. Gulavani and S. K. Rajamani, "Counterexample driven refinement for abstract interpretation," in *TACAS*, 2006.

[43] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *PLDI*, 2012.

[44] D. Beyer, "Software verification and verifiable witnesses - (report on SV-COMP 2015)," in *TACAS*, 2015.

[45] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *CAV*, 2011.

[46] T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli, "The SeaHorn verification framework," in *CAV*, 2015.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C++ (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 2002.

Formal Verification of Automatic Circuit Transformations for Fault-Tolerance

Dmitry Burlyaev
Univ. Grenoble Alpes; INRIA
dmitry.burlyaev@inria.fr

Pascal Fradet
INRIA; Univ. Grenoble Alpes
pascal.fradet@inria.fr

***Abstract*—We present a language-based approach to certify fault-tolerance techniques for digital circuits. Circuits are expressed in a gate-level Hardware Description Language (HDL), fault-tolerance techniques are described as automatic circuit transformations in that language, and fault-models are specified as particular semantics of the HDL. These elements are formalized in the Coq proof assistant and the properties, ensuring that for all circuits their transformed version masks all faults of the considered fault-model, can be expressed and proved. In this article, we consider Single-Event Transients (SETs) and fault-models of the form “at most 1 SET within k clock cycles”. The primary motivation of this work was to certify the Double-Time Redundant Transformation (DTR), a new technique proposed recently [1]. The DTR transformation combines double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers. That intricacy requested a formal proof to make sure that no single-point of failure existed. The correctness of DTR as well as two other transformations for fault-tolerance (TMR & TTR) have been proved in Coq.**

I. Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors has become a design characteristic as important as performance and power consumption [2]. The increased risk of soft errors results from the continuous shrinking of transistor size that makes components more sensitive to radiation [3].

The most widely-used methods to make circuits fault-tolerant rely on hardware redundancy. Triple-Modular Redundancy (TMR) [4] remains the most popular technique along with Finite State Machine (FSM) encoding (one hot, hamming, *etc.*). Some more complex ones are based on time-redundancy (re-execution) [1], [5], [6]. All these techniques can be realized through automatic circuit transformations and some of them are already supported by CAD tools. Since fault-tolerance is typically used in critical domains (aerospace, nuclear power, etc.), the correctness of such transformations is essential. If there is little doubt about the correctness of simple transformations such as TMR, this is not the case for more intricate ones.

The overall correctness of an automatic circuit transformation for fault tolerance consists not only in its functional correctness when no soft errors occur but also in its proper behavior under error occurrences. Widely-used post-synthesis verification tools (*e.g.,* model checking) are simply inappropriate to prove that a transformation ensures some property for all possible circuits; only proof-based approaches are suitable.

We propose an approach using the Coq proof assistant [7] to formally verify the functional and fault-tolerance properties of circuit transformations. We define the syntax and semantics of a simple gate-level functional HDL to describe circuits. Fault models, that specify the kind and occurrences of faults to be masked, are formalized in the language semantics. In this paper, we focus on SETs and fault-models of the form “at most 1 SET every k cycles”. Fault-tolerance transformations are defined as recursive functions on the syntax of the language. Proofs rely mainly on relating the execution of the source circuit without faults to the execution of the transformed circuit *w.r.t.* the considered fault-model. They make use of several techniques (case analysis, induction on the type or the structure of circuits, co-induction on input streams).

While our approach is general, it has been originally developed to prove DTR, an involved transformation combining double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers [1]. If manual checks were quite useful to develop that transformation, they were error-prone and not convincing enough. This transformation served as an advanced case study. The correctness of DTR as well as two other transformations (among which TMR) have all been proved in Coq.

Section II introduces the syntax and semantics of our gate-level HDL. In section III, we present the specification of fault-models in the language formal semantics. Section IV explains the proof methodology adopted to show the correctness of circuit transformations. It is illustrated by examples taken from the simplest transformation: TMR. Section V introduces the DTR circuit transformation [1] and sketches the associated proofs. Section VI presents related work, summarizes our contributions and suggests a few extensions.

Throughout this article, we use standard mathematical and semantic notations. The corresponding Coq specifications and proofs are available online [8].

II. Circuit Description Language

We describe circuits at the gate level using a purely functional language inspired from Sheeran’s combinator-based languages such as μFP [9] or Ruby [10]. We equip our language with dependent types which, along with the language syntax, ensure that circuits are well-formed by construction (gates correctly plugged, no dangling wires, no combinational loops, . . .).

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Contrary to μFP or Ruby, our primary goal is not to make the description of circuits easy but to keep the language as simple and minimal as possible to facilitate formal proofs. Our language contains only 3 logical gates, 5 plugs and 3 combining forms. It is best seen as a low-level core language used as the object code of a synthesis tool. We denote it as LDDL (Low-level Dependent Description Language).

A. *Syntax of* LDDL

A *bus* of signals is described by the following type

$$B := \omega \mid (B_1 * B_2)$$

A bus is either a single wire (ω) or a pair of buses. In other terms, buses are defined as nested pairs. The constructors of LDDL annotated with their types are gathered in Fig. 1. A circuit takes as parameters its input and output types and is either a logic gate, a plug, or composition of circuits.

Gates

$$\textsc{not} \;:\; \mathsf{Gate}\ \omega\ \omega \qquad \textsc{and}, \textsc{or} \;:\; \mathsf{Gate}\ (\omega * \omega)\ \omega$$

Plugs

$$\begin{array}{lcl}
\textsc{id} & : & \forall \alpha, \mathsf{Plug}\ \alpha\ \alpha \\
\textsc{fork} & : & \forall \alpha, \mathsf{Plug}\ \alpha\ (\alpha * \alpha) \\
\textsc{swap} & : & \forall \alpha\ \beta, \mathsf{Plug}\ (\alpha * \beta)\ (\beta * \alpha) \\
\textsc{lsh} & : & \forall \alpha\ \beta\ \gamma, \mathsf{Plug}\ ((\alpha * \beta) * \gamma)\ (\alpha * (\beta * \gamma)) \\
\textsc{rsh} & : & \forall \alpha\ \beta\ \gamma, \mathsf{Plug}\ (\alpha * (\beta * \gamma))\ ((\alpha * \beta) * \gamma)
\end{array}$$

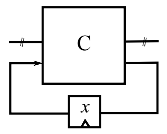
Circuits

$$\begin{array}{rlcl}
C ::= & \mathit{Gates} & & \\
\mid & \mathit{Plugs} & & \\
\mid & C_1 \mathbin{-\!\circ\!-} C_2 & : & \forall \alpha\ \beta\ \gamma, \mathsf{Circ}\ \alpha\ \beta \to \mathsf{Circ}\ \beta\ \gamma \\
 & & & \quad \to \mathsf{Circ}\ \alpha\ \gamma \\
\mid & [\![C_1, C_2]\!] & : & \forall \alpha\ \beta\ \gamma\ \delta, \mathsf{Circ}\ \alpha\ \gamma \to \mathsf{Circ}\ \beta\ \delta \\
 & & & \quad \to \mathsf{Circ}\ (\alpha * \beta)\ (\gamma * \delta) \\
\mid & \boxed{x}\!\!-\!C & : & \forall \alpha\ \beta, \mathsf{bool} \to \mathsf{Circ}\ (\alpha * \omega)\ (\beta * \omega) \\
 & & & \quad \to \mathsf{Circ}\ \alpha\ \beta
\end{array}$$

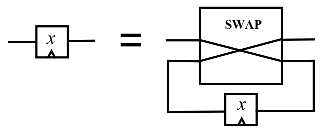
Fig. 1: LDDL Syntax

The sets of logical gates and plugs are minimal but expressive enough to specify any combinational circuit. The type of AND and OR, $\mathsf{Gate}\ (\omega{*}\omega)\ \omega$, indicates that they are gates taking a bus made of two wires and returning one wire. Likewise, NOT has type $\mathsf{Gate}\ \omega\ \omega$. Plugs, used to express (re)wiring, are polymorphic functions that duplicate or reorder buses: ID leaves its input bus unchanged, FORK duplicates its input bus, SWAP inverts the order of its two input buses, LSH and RSH reorder their three input buses.

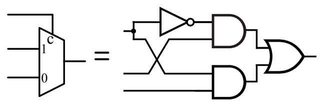
Circuits are either a gate, a plug, a sequential composition ($. \mathbin{-\!\circ\!-} .$), a parallel composition ($[\![.,.]\!]$), or a composition with a cell (flip-flop) within a feedback loop ($\boxed{\ .\ }\!\!-\!.$). The typing of the sequential operator ensures that the output bus of the first circuit has the same type as the input bus of the second one. The typing of the parallel operator expresses the fact that the inputs (resp. outputs) of the resulting circuit is made of the inputs (res. outputs) of the two sub-circuits. The last operator (related to the μ operator of μFP) is the only way to introduce feedback loops in the circuit. $\boxed{x}\!\!-\!C$ is better seen graphically as the circuit



The circuit C can have any input/output bus but it also takes and returns a wire connected to a memory cell set to the Boolean value x (tt or ff). The main advantage of that operator is to ensure that any loop contains a cell. It prevents combinational loops by construction. Of course, it does not force all cells to be within loops. A simple cell without feedback is expressed as $\boxed{x}\!\!-\!\textsc{swap}$:



To illustrate the language, a multiplexer



can be expressed in LDDL as the expression

$$[\![\textsc{fork}, \textsc{id}]\!] \mathbin{-\!\circ\!-} \textsc{lsh} \mathbin{-\!\circ\!-} [\![\textsc{not}, \textsc{rsh} \mathbin{-\!\circ\!-} \textsc{swap}]\!] \mathbin{-\!\circ\!-} \textsc{rsh}$$
$$\mathbin{-\!\circ\!-} [\![\textsc{and}, \textsc{and}]\!] \mathbin{-\!\circ\!-} \textsc{or}$$

As common with low-level or assembly-like languages, LDDL is quite verbose. Recall that it is not meant to be used directly. It is best seen as a back-end language produced by synthesis tools. On the other hand, it is simple and expressive; its dependent types make inputs and outputs of each sub-circuit explicit and ensure that all circuits are well-formed.

B. *Semantics of* LDDL

From now on, to alleviate notations, we leave typing constraints implicit. All input and output types of circuits and corresponding buses always match.

The semantics of gates and plugs are given by functions denoted by $[\![.]\!]$. For instance, the semantics of ID, is the identity function ($[\![\textsc{id}]\!]x = x$) or the semantics of FORK is the function duplicating its bus argument ($[\![\textsc{fork}]\!]x = (x, x)$).

Taking into account errors (in particular, SETs) makes the semantics non deterministic. When a glitch produced by an SET reaches a flip-flop, it may be latched non-deterministically as tt or ff. Therefore, the standard semantics of circuits is not described as functions but as predicates. The second issue is the representation of a circuit state (*i.e.,* the current values of its cells). A solution could be to equip the semantics with an environment ($\mathit{cell} \to \mathsf{bool}$). We choose here to use the circuit itself to represent its state which is made explicit by the $\boxed{x}\!\!-\!C$ constructs.

The semantics of circuits is described by the inductive predicate $\mathsf{step} : \mathsf{Circ}\ \alpha\ \beta \to \alpha \to \beta \to \mathsf{Circ}\ \alpha\ \beta$. The expression step $C\ a\ b\ C'$ can be read as "after one clock cycle, the circuit C applied to the inputs a may produce the outputs b and the new circuit (state) C' ". The rules are gathered in Fig. 2.

$$\text{Gates \& Plugs}\ \frac{[\![G]\!]a = b}{\mathsf{step}\ G\ a\ b\ G}$$

$$\text{Seq}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \qquad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{step}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\text{Par}\ \frac{\mathsf{step}\ C_1\ a\ c\ C_1' \qquad \mathsf{step}\ C_2\ b\ d\ C_2'}{\mathsf{step}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]}$$

$$\text{Loop}\ \frac{\mathsf{step}\ C\ (a, \mathsf{b2s}\ x)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{step}\ \boxed{x}\!-\!C\ a\ b\ \boxed{y}\!-\!C'}$$

Fig. 2: LDDL semantics for a clock cycle

Gates (or plugs) are stateless: they are always returned unchanged by step. The rules for sequential and parallel compositions are standard. The rule for $\boxed{x}\!-\!C$ makes use of the b2s function which converts the Boolean value of a cell into a signal and of the s2b predicate which relates a signal to a Boolean. The outputs and the new state (circuit) depend on the reduction of C applied to the inputs a and the signal corresponding to x. Non-determinism may come from the predicate s2b which relates a glitch to both tt and ff.

The complete semantics is given by a co-inductive predicate eval : Circ $\alpha\ \beta \to$ Stream $\alpha \to$ Stream β which describes the circuit behavior for any infinite stream of inputs.

$$\text{Eval}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{eval}\ C'\ is\ os}{\mathsf{eval}\ C\ (i : is)\ (o : os)}$$

If C applied to the inputs i returns after a clock cycle the outputs o and the circuit C' and if C' applied to the infinite stream of inputs is returns the output stream os then the evaluation of C with the input stream $(i : is)$ returns the stream $(o : os)$.

The variable-less nature of LDDL spares the semantics to deal with bindings and environments. It avoids many administrative matters (reads, updates, well-formedness of environments) and facilitates formalization and proofs.

III. Specification of Fault Models

There are two main types of soft errors caused by particle strikes: Single-Event Upsets (SEUs) (*i.e.,* bit-flips in flip-flops) and SETs (*i.e.,* glitches propagating in the combinational circuit). An SEU can be modeled by changing the value of an arbitrary memory cell between two clock cycles. In this article, we focus on SETs and fault-models allowing at most 1 SET within k clock cycles, written $SET(1,k)$. An SET in a combinational circuit can lead to the non-deterministic corruption of any memory cell connected (by a purely combinational path) to the place where the SET occurred. Since an SET may potentially lead to several bit-flips, the $SET(1,k)$ model subsumes $SEU(1,k)$. In order to model SETs, glitches and their propagation must be represented in the semantics. We use signals that can take 3 values: 0, 1, or a glitch written $\lightning$. We often abuse the notation and denote a wire by its signal value.

Glitches propagate through plugs and gates (*e.g.,* AND$(1,\lightning) = \lightning$) but can be also logically masked (*e.g.,* OR$(1,\lightning) = 1$ or AND$(0,\lightning) = 0$). If a corrupted signal is not masked, it is latched as tt or ff (both (s2b $\lightning$ tt) and (s2b $\lightning$ ff) hold).

The semantics of circuits for a cycle with an SET is represented as the inductive predicate stepg $C\ a\ b\ C'$ that can be read as "after one cycle with an SET occurrence, the circuit C applied to the inputs a may produce the outputs b and the new circuit/state C'". The main rules for stepg are gathered in Fig. 3.

$$\text{Gates}\ \frac{}{\mathsf{stepg}\ G\ a\ \lightning\ G}$$

$$\text{SeqL}\ \frac{\mathsf{stepg}\ C_1\ a\ b\ C_1' \qquad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\text{SeqR}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \qquad \mathsf{stepg}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\dots$$

$$\text{LoopC}\ \frac{\mathsf{stepg}\ C\ (a, b2s\ x)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\!-\!C\ a\ b\ \boxed{y}\!-\!C'}$$

$$\text{LoopM}\ \frac{\mathsf{step}\ C\ (a, \lightning)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\!-\!C\ a\ b\ \boxed{y}\!-\!C'}$$

Fig. 3: LDDL semantics with SET (main rules)

The rule (Gates) asserts that stepg introduces a glitch after a logical gate. The two rules for sequential composition represents two mutually exclusive cases where the SET occurs in left sub-circuit (SegL) or in the right one (SegR). The rule for the parallel operator is similar. The rule (LoopC) represents the case where an SET occurs inside C. The rule (LoopM) represents the case where an SET occurs at the output of the memory cell x which is taken as an input by C. To summarize, stepg introduces non-deterministically a single glitch after a cell or a logical gate. Hence, if a circuit has n gates and m cells, it specifies $n+m$ possible executions.

The fault-model $SET(1,k)$ is expressed by the predicate setk_eval : Nat $\to$ Circ $\alpha\ \beta \to$ Stream $\alpha \to$ Stream β:

$$\text{SetN}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (n-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ n\ C\ (i : is)\ (o : os)}$$

$$\text{SetG}\ \frac{\mathsf{stepg}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (k-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ 0\ C\ (i : is)\ (o : os)}$$

The first argument of setk_eval plays the role of a clock counter. A glitch can be introduced (by stepg) only if the counter is 0 (SetG). When a glitch is introduced, the counter is reset to enforce at least $k-1$ normal execution steps (SetN).

IV. Overview of Correctness Proofs

Describing the proofs in details is out of scope of this paper (and would be tiresome). Instead, we outline the common proof structure of the transformations we have studied. We illustrate the main steps using examples taken from the correctness proof of the simplest one: TMR.

Transformation

Each fault-tolerance technique is specified by a program transformation on the syntax of LDDL. They are all defined by induction of the syntax and replacement of each memory cell by a memory block (a small circuit). The TMR transformation takes a circuit of type Circ $\alpha\ \beta$ and returns a circuit of type Circ $((\alpha*\alpha)*\alpha)\ ((\beta*\beta)*\beta)$. Inputs/outputs are triplicated to play the role of the inputs/outputs of each copy.

$$\begin{array}{lcl}\textsc{tmr}(X) & = & [\![[\![X,X]\!],X]\!] \quad \textit{with } X \textit{ a gate/plug}\\ \textsc{tmr}(C_1 \multimap C_2) & = & \textsc{tmr}(C_1) \multimap \textsc{tmr}(C_2)\\ \textsc{tmr}([\![C_1,C_2]\!]) & = & \textsc{s}_1 \multimap [\![\textsc{tmr}(C_1),\textsc{tmr}(C_2)]\!] \multimap \textsc{s}_2\\ \textsc{tmr}(\boxed{x}\!-\!C) & = & \boxed{x}\!-\!\boxed{x}\!-\!\boxed{x}\!-\!(\mathsf{vot} \multimap \textsc{tmr}(C) \multimap \textsc{s}_3)\end{array}$$

where $\textsc{s}_1$, $\textsc{s}_2$, $\textsc{s}_3$ are reshuffling plugs (*e.g.,* $\textsc{s}_1$ has type Plug $(((\alpha*\beta)*(\alpha*\beta))*(\alpha*\beta))\ (((\alpha*\alpha)*\alpha)*((\beta*\beta)*\beta))$ and reshuffles the input bus accordingly). Each cell is replaced by three cells followed by a triplicated voter (vot) made of a majority voter for each copy.

Relations between source and transformed circuits

The correctness property relates the execution of the source circuit without fault to the execution of the transformed circuit under a fault-model. Most of the lemmas also relate the states and executions of the source and transformed circuits. These relations are expressed as inductive predicates.

For TMR, a key property is that an SET can corrupt only a single redundant copy and that such corruption stays confined in that copy. To express corruption, we use a predicate relating source and transformed programs expressed on the syntax of LDDL. The corruption of the first copy of a transformed circuit C^T *w.r.t.* to its source circuit C is expressed by the predicate $\overset{c}{\sim}_1$. The main rule is

$$\text{CLoop}\ \frac{C\ \overset{c}{\sim}_1\ C^T}{(\boxed{x}\!-\!C)\ \overset{c}{\sim}_1\ (\boxed{z}\!-\!\boxed{x}\!-\!\boxed{x}\!-\!(\mathsf{vot} \multimap C^T \multimap \textsc{s}_3))}$$

which states that if C is in relation with C^T and the second and third memory cells of the transformed circuit are the same as the cell of the source circuit, then $\boxed{x}\!-\!C$ and its transformed version are in relation. The other rules just check recursively this source/transformed circuit relationship. For instance, the rule for the parallel construct is

$$\text{CPar}\ \frac{C_1\ \overset{c}{\sim}_1\ C_1^T \qquad C_2\ \overset{c}{\sim}_1\ C_2^T}{([\![C_1,C_2]\!])\ \overset{c}{\sim}_1\ (\textsc{s}_1 \multimap [\![C_1^T,C_2^T]\!] \multimap \textsc{s}_2)}$$

The same relations exist for other options of redundant copy corruption ($\overset{c}{\sim}_2$ and $\overset{c}{\sim}_3$) and for each possible corruption of the triplicated bus ($\overset{b}{\sim}_1$, $\overset{b}{\sim}_2$, $\overset{b}{\sim}_3$). In the following, we write $\overset{c}{\sim}$ for the relation $\overset{c}{\sim}_1 \vee \overset{c}{\sim}_2 \vee \overset{c}{\sim}_3$.

Key properties and proofs

Properties and their associated proofs can be classified as:

- properties "for all circuits" relating their source and transformed versions for a one cycle reduction. They are usually proved by a simple structural induction on the structure of LDDL expressions;
- similar properties but for known sub-circuits introduced by the transformations (*e.g.,* voters). They are proved by examining all possible cases of corruption or SET occurrences.
- properties about the complete (infinite) execution of source and transformed circuits. They are proved by co-induction on the stream of inputs.

The main lemmas state how the transformed circuit evolves when it is in a correct state and one SET occurs (stepg), or when it is in a corrupted state and it executes normally (by step). For TMR we have for instance:

$$\begin{array}{ll} & \mathsf{step}\ C_1\ a\ b\ C_2\ \wedge\ \mathsf{stepg}\ \textsc{tmr}(C_1)\ (a,a,a)\ b3\ C_2^T\\ \Rightarrow & C_2\ \overset{c}{\sim}\ C_2^T\end{array}$$

It can be read as: if C_1 reduces by step in C_2, and its transformed version $\textsc{tmr}(C_1)$ reduces by stepg in a circuit C_2^T, then C_2^T is the transformed version of C_2 with at most one corrupted redundant copy ($C_2\ \overset{c}{\sim}\ C_2^T$). In other terms, a glitch can corrupt only one of copies of the TMR circuit.

The following lemma

$$\begin{array}{ll} & C_1\ \overset{c}{\sim}\ C_1^T\ \wedge\ \mathsf{step}\ C_1\ a\ b\ C_2\\ \Rightarrow & \mathsf{step}\ C_1^T\ (a,a,a)\ (b,b,b)\ \textsc{tmr}(C_2)\end{array}$$

ensures that a corrupted transformed circuit comes back to a valid state after one normal reduction step.

The main correctness theorems state that for related inputs the normal execution of the source circuit and the execution (under the considered fault-model) of the transformed circuit give related outputs. A complete execution is modeled using infinite streams of inputs/outputs and the proof should proceed by co-induction.

The correctness of the TMR transformation is expressed as

$$\begin{array}{ll} & \mathsf{eval}\ C\ i\ o\ \wedge\ \mathsf{setk\_eval}\ 2\ \textsc{tmr}(C)\ n\ (\mathsf{tripl}\ i)\ o3\\ \Rightarrow & o\ \overset{s}{\sim}\ o3\end{array}$$

TMR masks all faults of the fault-model $SET(1,2)$, so it tolerates an SET every other cycle. The stream of primary inputs for the transformed circuit is the input stream i where each element (bus) is triplicated (tripl i). The stream of primary outputs of the transformed circuit ($o3$: Stream $((\beta*\beta)*\beta)$) is a triplicated version of the output stream (o : Stream β) with at most one corrupted element in each triplet ($\overset{s}{\sim}$ relation). Indeed, the fault-model allows an SET to occur after the final voters. These SETs cannot be corrected internally but, since the outputs are triplicated, masking is still possible by voting in the surrounding circuit.

Practical issues

Taylor-made tactics had to be written for LDDL syntax and semantics. They helped to shorten and to automatize parts of the proofs.

All the transformations use known sub-circuits (*e.g.,* voters) and many basic properties must be shown on them. Such properties are often of the form

$$\text{Pstepg}\ \frac{P\ a \qquad \mathsf{stepg}\ C\ a\ b\ C'}{Q(a,b,C')}$$

with P and Q representing pre- and post-conditions, respectively. These properties on stepg entail to consider all possible SET occurrences. For TMR, which introduces triplicated voters, this can be done using standard proofs. The transformation DTR introduces much bigger sub-circuits, which would lead to very large proofs since dozens of different cases of SET need to be considered. Fortunately, Coq permits proofs by reflection which, in some cases, permits to replace manual proofs by automatic computations. We use largely this feature for known circuits. It amounts to

- define fstepg a functional version of stepg, which, for a given circuit and particular input, computes the set of the possible outputs and circuits in relation by stepg;
- prove that if $(b, C') \in$ (fstepg $C\ a$) then stepg $C\ a\ b\ C'$;
- define (or generate) equivalent functional (Boolean) versions P_b and Q_b of the predicates P and Q.

Then, a proof by reflection of the property (Pstepg) proceeds by generating all possible inputs, then it filters them by P_b, executes fstepg on all elements of that set and, finally, checks that Q_b returns true on all results. In this way, reflection automatizes the exploration of all fault occurrences and most of the proof boils down to computations.

V. Correctness of Double-Time Redundancy

The initial motivation of this work was to certify DTR, an involved circuit transformation that we recently proposed [1]. Hereafter, we outline DTR and the main parts of its proof.

A. DTR Transformation Overview

The main assets of DTR are its much lower hardware overhead than TMR and its ability to mask SETs using double-time redundancy instead of a triple overhead (in time or in space). DTR uses double-time redundancy to detect errors and a micro-checkpointing and a rollback mechanisms to re-execute the faulty cycle for recovery. Since, according to the fault-model, no error can occur immediately after the last error, time-redundancy can be switched-off during the recovery phase to "accelerate" the circuit twice. Along with input and output buffers to record inputs and to produce delayed outputs, it makes errors and recovery absolutely transparent to the surrounding circuit. Error detection followed by the recovery to a correct state may take up to 9 clock cycles; therefore DTR masks errors from the $SET(1, 10)$ fault-model.

The DTR transformation consists of four parts (see Fig. 4):

1) substitution of each original memory cell with a *memory block* and threading of control wires within the circuit;

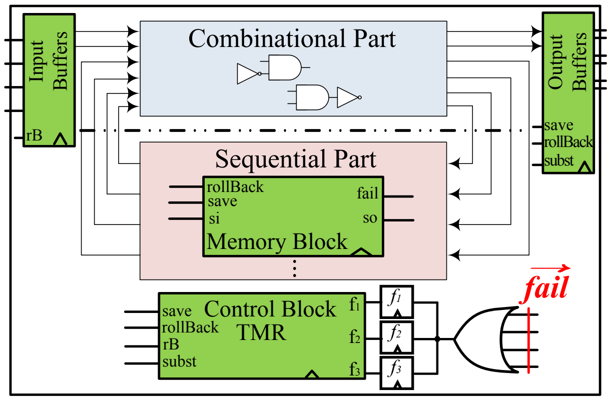


Fig. 4: Transformed circuit for DTR.

2) addition of a *control block*;
3) addition of *input buffers* to all circuit primary inputs;
4) addition of *output buffers* to all circuit primary outputs.

Further, the input stream should be upsampled twice in order to introduce enough redundancy in the transformed circuit to detect errors caused by SETs.

1) Memory blocks: Each original memory cell is substituted with a memory block (see Fig. 5) that stores the results of signal propagation through the combinatorial circuit along with recovery bits (or checkpoint bits). It consists of:

- two cells d and d' (the data bits) to save redundant information for comparison (with EQ) to detect errors; since the input stream is upsampled twice, d and d' normally contain the same value each every other cycle;
- two cells r and r' (the recovery bits) with enable-input to keep the value of the input during four clock cycles. If an error is detected in any memory block, a synchronous rollback occurs in all blocks. It retrieves correct values from r' cells and the circuit recovers from the erroneous state using a third recomputation.

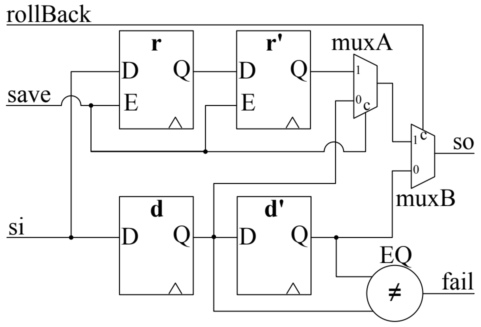


Fig. 5: DTR memory block.

2) Control block: When a memory block detects an error, the *fail* signal, going to the control block, is raised and latched in three error signaling cells (f_i in Fig. 4). Then, the control block emits a series of control signals to memory blocks (*e.g., save* and *rollBack*) to schedule rollback and recovery. Its functionality can be described as the FSM of Fig. 6. The states

0 and 1 compose the $normal$ mode which raises alternatively the $save$ signal used as an enable signal to organize a 4-cycle delay in the r-r' memory block cells. When an error is detected (*i.e.,* $f_i = 1$), the FSM enters the recovery mode for 4 cycles (states $2, 3, 4, 5$) and raises appropriate signals.

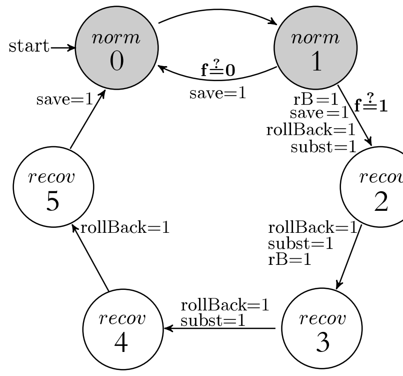


Fig. 6: FSM of the DTR control block: "$\overset{?}{=}$" denotes a guard, "$=$" an assignment and, by default, signals are set to 0.

During the recovery process, the control block switches-off double time redundancy speeding-up the circuit which, in a few cycles, catches up the state it should have had if no error had occurred. According to the fault-model, no error may occur immediately after the last error which allows us to perform such "acceleration". The control block itself is a small circuit protected against SETs using TMR.

3) Input/Output Buffers: To prevent disrupted input/output behavior during recovery and to guarantee transparency for the surrounding circuit, additional input and output buffers are necessary. They are inserted at each primary input and output of the original circuit. Input buffers keep the two last inputs which are only used for re-computation during recovery. Output buffers delay outputs (and introduce a two-cycles latency) in order to emit the previously recorded correct outputs during the recovery process. They are designed to be also fault-tolerant to any SET occurring inside or even at their outputs. To achieve this property, the primary outputs are triplicated in space. Buffers are controlled by the rB, $rollBack$, $subst$, $save$ signals. We refer the reader to [1] for a detailed presentation of their internal structure and behavior.

We illustrate the basic functionalities of DTR using a simple scenario where the upsampled stream $a\ a\ b\ b\ c\ c\ d\ d\ e\ e\ f\ f\ \ldots$ is sent to the circuit but a SET in the combinational part corrupts the first occurrence of b (written $\tilde{b}$). We use quadruplets like $(c, s, [d, d', r, r'], s)$ to denote the cycle number (c), the state of the control block (s) at the beginning of the cycle (see Fig. 6) and the state of the memory blocks (the values to be used as output are in bold). We start when the error is about to be detected: the four first values $a, a, \tilde{b}, b$ have been stored in memory blocks. The execution proceeds as follows:

$$\begin{array}{lll}(1,0,[b,\tilde{\mathbf{b}},b,a]); & (2,1,[\tilde{c},b,b,\mathbf{a}]); & (3,2,[\mathbf{b},\tilde{c},b,b]);\\ (4,3,[\mathbf{c},b,b,b]); & (5,4,[\mathbf{d},c,b,b]); & (6,5,[e,\mathbf{d},b,b]);\\ (7,0,[e,\mathbf{e},e,b]); & (8,1,[f,\mathbf{e},e,b]); & (9,0,[f,\mathbf{f},f,e])\end{array}$$

In cycle 1, the error is detected and the f_i cells (see Fig. 4) are set to 1. The value $\tilde{c}$ is read; it is potentially corrupted since it may be the result of the propagation of $\tilde{b}$ (which is corrupted) through the combinational circuit. In cycle 2, the control block goes in the recovery mode and performs the rollback (transition $1 \mapsto 2$, Fig. 6). It emits control signals that ensure that the values stored in the recovery bits r' (as well as those stored the input buffers) are used instead of the usual inputs for the third recomputation. The circuit is now in a speedup mode and double time redundancy is suspended. The cycles 2 to 4 are computed using the values a, b, c. The control signals entail that the read value, which is stored in d, is used as output in the next cycle. Then, double redundancy resumes and the recovery line r-r' retakes correct values in the next cycles. At the 9^{th} cycle, the state is exactly the state that would have been reached at the same 9^{th} cycle without error. During the recovery, input buffers also re-inject previous values synchronously with the memory blocks, and output buffers produce delayed correct outputs (see [1] for the complete description).

B. Formal Specification and Proofs

While TMR is a well-established transformation and its properties are doubtless, DTR is a novel and much more complex technique. Our goal was to ensure that no single point of failure existed: in particular, any SET in memory blocks, combinational logic, input or output buffers, control block, and control wires should be masked. The number of possible error scenarios is very large (about 10 cases each for memory block and output buffers). Moreover, the normal execution mode has a two-cycle period which doubles the number of corruption cases. Full confidence in DTR correctness for all possible circuits and errors can only be achieved with a formal proof-based approach.

Transformation

The core DTR transformation is defined very much like TMR as presented in Section IV. It takes an original circuit of type $\mathsf{Circ}\ \alpha\ \beta$ and substitutes each memory cell with a memory block returning a circuit of type $\mathsf{Circ}\ (\alpha * ((\omega * \omega) * \omega))\ (\beta * ((\omega * \omega) * \omega))$. The three wires $((\omega * \omega) * \omega)$ correspond to the control signals $((save * rollBack) * fail)$ that propagate through all memory blocks. Input (resp. output) buffers are plugged to each primary input (resp. output) by recursion on the input type α (resp. output type β). Plugging input/output buffers and the control block to the transformed circuit returns a circuit of type $\mathsf{Circ}\ \alpha\ ((\beta * \beta) * \beta)$. The triplicated output interface of type $((\beta * \beta) * \beta)$ represents the triplicated original output bus.

In the following, we write $\textsc{mb}(d, d', r, r', C)$ to denote a memory block with values d, d', r, r' (see Fig. 5) plugged to a circuit C.

Relations between source and transformed circuits

Most of the inductive predicates relating states and executions of the source and transformed circuits have several versions depending on the state of the control block $(0, 1, \ldots)$.

For instance, the predicate $\mathsf{dtr0}$ expresses the relation between a transformed circuit and its source version(s) when the control block is in state 0. The state of a memory block is of the form $[y, y, y, x]$ where the values x and y are the two values taken successively by the corresponding cells of the source version. Therefore, the state of the transformed circuit is in relation with two successive source circuits. $\mathsf{dtr0}$ is defined inductively in a similar way as $\overset{c}{\sim}$ predicates in Sec. IV. The main rule relates the memory block to the states of the two source circuits:

$$\frac{\mathsf{dtr0}\ C_0\ C_1\ C^T}{\mathsf{dtr0}\ (\boxed{x}\!-\!C_0)\ (\boxed{y}\!-\!C_1)\ \textsc{mb}(y, y, y, x, C^T)}$$

The memory block should be of the form ($d = d' = r = y$; $r' = x$) where x and y are values of the corresponding cells of the circuits $\boxed{x}-C_0$ and $\boxed{y}-C_1$, respectively. Those two circuits represent two successive states of the source circuit.

The corresponding predicate when the control block is in state 1 relates a transformed circuit to three successive source circuits. Indeed, in that state, the memory block is of the form $[z, y, y, x]$ where x, y and z are three successive values taken by the source circuit.

Several versions of these predicates are needed to represent the corruption cases. For instance, the predicate $\mathsf{dtr1d}$ expresses the relation between a transformed circuit whose d cells are potentially corrupted and its source version when the control block is in state 1. The main rule is:

$$\frac{\mathsf{dtr1d}\ C_0\ C_1\ C_2\ C^T}{\mathsf{dtr1d}\ (\boxed{x}\!-\!C_0)\ (\boxed{y}\!-\!C_1)\ (\boxed{z}\!-\!C_2)\ \textsc{mb}(w, y, y, x, C^T)}$$

that is, r', (resp. d' and r) should hold the same values are the first (resp. second) source circuit; d has no constraint (*i.e.,* can be corrupted). Other predicates are also needed to relate the source and transformed versions when the control block is in the recovery mode.

Key lemmas

Using the aforementioned predicates, we can define lemmas that show how the transformed circuit evolves with and without SETs. First, it can be shown that, initially, the transformed circuit is in relation with the source circuit *i.e.,*

$$\mathsf{dtr0}\ C\ C\ \textsc{dtr}(C)$$

Then, all cases of state evolution are covered. For instance, the following property for a reduction with no SET

$$\begin{array}{ll} \mathsf{dtr0}\ C_0\ C_1\ C^T & \Rightarrow\ \mathsf{step}\ C_1\ a\ b\ C_2 \\ & \Rightarrow\ \mathsf{step}\ C^T\ \{a, \{0, 0, 0\}\}\ b'\ C'^T \\ & \Rightarrow\ b' = \{b, \{0, 0, 0\}\} \\ & \quad \wedge\ \mathsf{dtr1}\ C_0\ C_1\ C_2\ C'^T \end{array}$$

states that, if the original circuit evolves from C_1 to C_2 with input a, then the corresponding transformed circuit C^T with input a and signals $\mathit{save} = 0$, $\mathit{rollBack} = 0$, and $\mathit{fail} = 0$ returns the same output b and the same global signals. Further, if C^T is related to (C_0, C_1) with $\mathsf{dtr0}$, the returning state C'^T is related to (C_0, C_1, C_2) with $\mathsf{dtr1}$.

Similarly, if the $\mathit{rollBack}$ signal is corrupted (has a glitch), then memory blocks may output an incorrect value (wrongly selected by muxB, Fig.5). Since this value goes through the combinational circuit and may be fetched by memory blocks, their d values may be corrupted. This is formalized as

$$\begin{array}{ll} \mathsf{dtr0}\ C_0\ C_1\ C^T & \Rightarrow \mathsf{step}\ C_1\ a\ b\ C_2 \\ & \Rightarrow \mathsf{step}\ C^T\ \{a, \{0, \lightning, 0\}\}\ b'\ C'^T \\ & \Rightarrow \exists x, b' = \{x, \{0, \lightning, 0\}\} \\ & \quad \wedge\ \mathsf{dtr1d}\ C_0\ C_1\ C_2\ C'^T \end{array}$$

These properties are shown by simple structural induction. Similar properties on input and output buffers are proved using reflection. The proofs for the collection of input and output buffers plugged to the primary input and output buses are proved by induction of the input and output types. Proofs for the triplicated control block make a critical use of the main properties proved for the TMR transformation.

The property corresponding to a reduction by stepg of the whole transformed circuit proceeds by inspection of all cases of fault occurrences and application of the properties mentioned above. It can be shown that in all cases the transformed circuit returns to a correct state after less than 10 reduction steps after an SET.

Main theorem

The main correctness theorem is expressed as

$$\begin{array}{ll} & \mathsf{step}\ C_0\ a\ b\ C_1 \\ \wedge & \mathsf{step}\ \textsc{dtr}(C_0)\ a\ b_1\ C^T\ \wedge\ \mathsf{step}\ C^T\ a\ b_2\ C_1^T \\ \wedge & \mathsf{eval}\ C_1\ i\ o\ \wedge\ \mathsf{setk\_eval}\ 10\ C_1^T\ n\ (\mathsf{upsampl}\ i)\ oo \\ \Rightarrow & \mathsf{outDTR}\ (b, o)\ oo \end{array}$$

It assumes that no error occurs during the first two cycles (second line of the theorem). This is due to the arbitrary initialization of memory cells (buffers, memory blocks) performed by the transformation. Since the recovery bits are not properly set, a rollback and the following recovery would be incorrect. The stream of primary inputs of the transformed circuit ($\mathsf{upsampl}\ i$) is the input stream i where each bit is repeated twice. The fault-model $SET(1, 10)$ is expressed by the predicate ($\mathsf{setk\_eval}\ 10$) that may use stepg at most once every 10 cycles (and uses step otherwise). The predicate outDTR relates the output stream (of type $\mathsf{Stream}\ \beta$) produced by the source circuit to the output stream (oo of type $\mathsf{Stream}\ (\beta * (\beta * \beta))$) of the transformed circuit. The two first values of the transformed stream are not meaningful since output buffers introduce a latency of two cycles. The predicate outDTR states that if the first stream has value a at position i, then the second stream will have a triplet with at least two a's at position $2 * i + 1$. We can guarantee the correctness of only two values because we allow an SET to occur even at the primary outputs.

VI. Conclusions

Many efforts have been devoted to the formal functional verification of circuits [11]. It is usually performed for specific circuits using model-checking or SAT solving techniques.

However, such an approach is inappropriate to prove the correctness of a synthesis or transformation tool for all possible circuits; theorem proving must be used instead.

Still, proof-assistants have been mostly used for functional circuit verification. Let us cite, among many others, the application of ACL2 to prove the out-of-order microprocessor architecture FM9801 [12], HOL for the Uinta pipelined microprocessor [13], and Coq for an ATM Switch Fabric [14]. The language proposed by Braibant [15] is close to our LDDL language and has been used to prove the correctness of parametric combinational circuits (*e.g.,* n bits adders).

Proof-assistants have also been used to certify tools used in circuit synthesis. An old survey of formal circuit synthesis is given in [16]. More recently, S. Ray et al. proved circuit transformations used in high-level synthesis with ACL2 [17]. Braibant and Chlipala certified in Coq a compiler from a simplified version of BlueSpec to RTL [18].

To the best of our knowledge, our work is the first to certify automatic circuit transformations for fault-tolerance. Contrary to most of the works which specify circuits within the logic of the prover, we use a gate-level HDL. This approach permits to reason on circuits (gates and wires) and to model SETs as glitches occurring at specific places. Automatic fault-tolerance techniques are easily specified by program transformations on the syntax of LDDL. Furthermore, its variable-less nature allowed a simple semantics (without environments) that facilitated formalization and proofs.

Our approach is general and applicable to many fault-tolerant transformations. We used it to prove the correctness of TMR and DTR but also of Triple-Time Redundant Transformation (TTR), a simpler and more straightforward time redundancy technique where each computation cycle is triplicated and followed by votings. However, our initial motivation was the proof of DTR whose correctness was far from obvious. While we relied on many manual checks to design the transformation, only Coq allowed us to get complete assurance. The formalization of DTR did not reveal real errors but a few imprecisions. For instance, we stated in [1] that the control block was protected using TMR without making it clear how it was connected to the rest of the circuit. We had to introduce three cells to record the value of the *fail* signal and to slightly change the definition of its internal FSM.

The approach makes an essential use of two features of Coq: dependent types and reflection. Dependent types provided an elegant solution to ensure that all circuits were well-formed. Such types are often presented as tricky to use but, in our case, that complexity remained confined to the writing of libraries for the equality and decomposition of buses and circuits. Reflection was very useful to prove properties of known subcircuits; it would had been much harder without it.

The size of specifications and proofs for the common part (LDDL syntax and semantics, libraries) is 5000 lines of Coq (excluding comments and blank lines), 700 for TMR, 3500 for TTR and 7000 for DTR. Checking all the proofs takes around 45 min on an average laptop. The overall effort for the complete development is hard to estimate. Completing the proof of DTR alone took roughly 5 man-months. The Coq files for these proofs are available online [8].

We believe that additional user-defined tactics could make the proofs of LDDL transformations much smaller and automatic. Indeed, the key parts are to define the predicates relating the source and transformed circuits and to state the lemmas. The proofs themselves are, for the most part, straightforward inductions. The proposed framework could also be used to prove other fault-tolerance mechanisms (*e.g.,* the transformations for adaptive fault-tolerance we present in [19]) or well-known techniques used in circuit synthesis (*e.g.,* FSM-encoding). More generally, proof-assistants are now sufficiently mature to consider the formal certification of the whole circuit synthesis chain including optimizations.

Acknowledgments

Thanks are due to Frédéric Besson and Damien Pous for answering several questions about Coq. We got inspiration from James R. Wilcox's blog to make our proofs axiom free.

References

[1] D. Burlyaev, P. Fradet, and A. Girault, "Automatic time-redundancy transformation for fault-tolerant circuits," in *FPGA*, 2015, pp. 218–227.

[2] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer*, vol. 38, no. 2, pp. 43–52, Feb. 2005.

[3] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks*, 2002, pp. 389–398.

[4] J. von Neumann, "Probabilistic logic and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[5] C. Chan, D. Schwartz-Narbonne, D. Sethi, and S. Malik, "Specification and synthesis of hardware checkpointing and rollback mechanisms," in *Design Automation Conference*, 2012, pp. 1222–1228.

[6] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *FPGA*, 2007, pp. 188–196.

[7] Coq development team. The coq proof assistant, software and documentation available at http://coq.inria.fr/, 1989-2014.

[8] "Coq proofs of circuit transformations for fault-tolerance," available at https://team.inria.fr/spades/fthwproofs/, 2014-2015.

[9] M. Sheeran, "muFP, A language for VLSI design," in *LISP and Functional Programming*, 1984, pp. 104–112.

[10] G. Jones and M. Sheeran, "Designing arithmetic circuits by refinement in Ruby," *Sci. Comput. Program.*, vol. 22, no. 1-2, pp. 107–135, 1994.

[11] A. Gupta, "Formal hardware verification methods: A survey," *Form. Methods in System Design*, vol. 1, no. 2-3, pp. 151–238, Oct. 1992.

[12] J. Sawada and W. A. Hunt Jr., "Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability," *Formal Methods in System Design*, pp. 187–222, 2002.

[13] P. J. Windley and M. L. Coe, "A correctness model for pipelined multiprocessors," in *Theor. Provers in Circuit Design*, 1994, pp. 33–51.

[14] S. Coupet-Grimal and L. Jakubiec, "Certifying circuits in type theory," *Formal Asp. Comput.*, vol. 16, no. 4, pp. 352–373, 2004.

[15] T. Braibant, "Coquet: A coq library for verifying hardware," in *Proc. of Certified Programs and Proofs - CPP*, 2011, pp. 330–345.

[16] R. Kumar, C. Blumenrhr, D. Eisenbiegler, and D. Schmid, "Formal synthesis in circuit design. A classification and survey," in *FMCAD*, 1996, pp. 294–309.

[17] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang, "Formal verification for high-assurance behavioral synthesis," in *Int. Symposium on Automated Technology for Verification and Analysis*, 2009, pp. 337–351.

[18] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*, 2013, vol. 8044, pp. 213–228.

[19] D. Burlyaev, P. Fradet, and A. Girault, "Time-redundancy transformations for adaptive fault-tolerant circuits," in *NASA/ESA Conf. in Adaptive Hardware and Systems, AHS*, june 2015.

An SMT-based Approach to Fair Termination Analysis

Javier Esparza
Institut für Informatik
Technische Universität München
Garching bei München, Germany
Email: esparza@in.tum.de

Philipp J. Meyer
Institut für Informatik
Technische Universität München
Garching bei München, Germany
Email: meyerphi@in.tum.de

***Abstract*—Algorithms for the coverability problem have been successfully applied to safety checking for concurrent programs. In a former paper (An SMT-based Approach to Coverability Analysis, CAV14) we have revisited a constraint approach to coverability based on classical Petri net analysis techniques and implemented it on top of state-of-the-art SMT solvers. In this paper we extend the approach to fair termination; many other liveness properties can be reduced to fair termination using the automata-theoretic approach to verification. We use T-invariants to identify potential infinite computations of the system, and design a novel technique to discard false positives, that is, potential computations that are not actually executable. We validate our technique on a large number of case studies.**

I. Introduction

In recent years, verification problems for concurrent shared-memory or asynchronous message-passing software have been attacked by means of Petri net techniques. In particular, it has been shown that safety properties or fair termination can be solved by constructing and analyzing the coverability graph of a Petri net, or some related object [1]–[5]. This renewed interest on the coverability problem has led to numerous algorithmic advances for the construction of coverability graphs [4], [6]–[9].

Despite this success, the coverability problem remains computationally expensive [10], since it involves exhaustive state-space exploration. This motivates the study of cheaper incomplete procedures: algorithms much faster than the construction of the coverability graph, which may prove the property true, but also answer "don't know". In a recent paper, the authors, together with other colleagues, have revisited and further developed tests based on the marking equation and traps, two classical Petri net analysis techniques [11]. These techniques allow one to efficiently compute program invariants expressed as constraints of linear arithmetic [12]–[14]. If the states violating the property also correspond to those satisfying a linear constraint, unsatisfiability of the complete constraint system proves the property true. In the test suite analyzed in [11], 83% of the positive problem instances (that is, the instances for which the property holds) could be proved in this way. Moreover, due to advances in SMT-solving, the constraint systems could be solved at a fraction of the cost of state-exploration techniques. So the technique makes sense as a preprocessing that allows to prove many easy cases at low cost; if the technique fails, then we can always resort to complete state-space exploration methods.

In this paper we extend the approach to liveness properties. As in [11], which revisited and expanded previous work, we revisit an idea initially presented in [15], based on the use of transition invariants. Since liveness is typically harder than safety, and the constraint technology of 1997 was very primitive compared to state-of-the-art SMT-solvers, the work of [15] only explored a rather straightforward test, and only considered one case study. In this paper we improve the test of [15], design different implementations, compare their performance, and validate them on numerous case studies coming from different areas: distributed algorithms, workflow processes, Erlang programs, and asynchronous programs.

We conclude this introduction with a brief outline of our technique. Given an infinite execution σ of a Petri net model, let $\mathit{inf}(\sigma)$ be the set of transitions that occur infinitely often in σ. We consider liveness properties such that whether σ satisfies the property or not depends only on $\mathit{inf}(\sigma)$. (This is not an important restriction because, by taking the product of the Petri net model with a suitable Büchi automaton, every LTL property can be reduced to a property of this kind.) We say that a set T of transitions is *feasible* if $T = \mathit{inf}(\sigma)$ for some σ. We use T-invariants (more precisely, T-surinvariants) to extract Boolean constraints that must be satisfied by every feasible set of transitions. However, these constraints are typically quite weak, and have spurious solutions, that is, they are satisfied by unfeasible sets of transitions. So we design a refinement loop that, given a solution, tries to construct an additional constraint that excludes it. If the refinement procedure terminates, then the model satisfies the property.

The paper is structured as follows. Section II contains basic definitions. Section III introduces the main technique. In Section IV and V, we describe two methods to refine the main technique. Section VI contains the experimental evaluation. Finally, Section VII presents conclusions.

II. Preliminaries

A *net* is a triple (P, T, F), where P is a set of *places*, T is a (disjoint) set of *transitions*, and $F : (P \times T) \cup (T \times P) \to \{0, 1\}$ is the *flow function*. For $x \in P \cup T$, the *pre-set* is ${}^\bullet x = \{y \in P \cup T \mid F(y, x) = 1\}$ and the *post-set* is $x^\bullet = \{y \in P \cup T \mid$

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

```
     procedure PROCESS 1                 procedure PROCESS 2
     begin                               begin
       $bit_1$ := $false$                  $bit_2$ := $false$
       while $true$ do                     while $true$ do
$p_1$:    $bit_1$ := $true$        $q_1$:     $bit_2$ := $true$
$p_2$:    while $bit_2$ do skip od $q_2$:     if $bit_1$ then
$p_3$:    (* critical section *)   $q_3$:       $bit_2$ := $false$
          $bit_1$ := $false$       $q_4$:       while $bit_1$ do skip od
       od                                       goto $q_1$
     end                                      fi
                                   $q_5$:     (* critical section *)
                                              $bit_2$ := $false$
                                           od
                                         end
```

Fig. 1. Lamport's 1-bit algorithm for mutual exclusion [16].

$F(x, y) = 1\}$. We extend the pre- and post-set to a subset of $P \cup T$ as the union of the pre- and post-sets of its elements. A *subnet* of a Petri net (P, T, F) is a triple (P', T', F') such that $P' \subseteq P$, $T' \subseteq T$, and $F' : (P' \times T') \cup (T' \times P') \to \{0, 1\}$ with $F'(x, y) = F(x, y)$. Since F' is completely determined by F,P', and T', we often speak of the subnet (P', T').

A *marking* of a net (P, T, F) is a function $m : P \to \mathbb{N}$. Assuming an enumeration $p_1, \ldots, p_n$ of P, we often identify m and the vector $(m(p_1), \ldots, m(p_n))$. For a subset $P' \subseteq P$ of places, we write $m(P') = \sum_{p \in P'} m(p)$. A *Petri net* is a tuple $N = (P, T, F, m_0)$, where (P, T, F) is a net and m_0 is a marking called the *initial marking*. Petri nets are represented graphically as follows: places and transitions are represented as circles and boxes, respectively. For $x, y \in P \cup T$, there is an arc leading from x to y iff $F(x, y) = 1$. The initial marking is represented by putting $m_0(p)$ black tokens in each place p.

A transition $t \in T$ is *enabled at* m iff $m(p) \geq 1$ for every $p \in {}^\bullet t$. A transition t enabled at m may *fire*, yielding a new marking m' (denoted $m \xrightarrow{t} m'$), where $m'(p) = m(p) + F(t, p) - F(p, t)$.

A sequence of transitions, $\sigma = t_1 t_2 \ldots t_r$ is an *occurrence sequence* of N iff there exist markings $m_1, \ldots, m_r$ such that $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \ldots \xrightarrow{t_r} m_r$. The marking m_r is said to be *reachable* from m_0 by the occurrence of σ (denoted $m_0 \xrightarrow{\sigma} m_r$).

An infinite sequence of transitions, $\sigma = t_1 t_2 \ldots$ is an *infinite occurrence sequence* of N iff every finite prefix of σ is an occurrence sequence of N (denoted $m_0 \xrightarrow{\sigma}$). The set $inf(\sigma)$ contains the transitions occurring infinitely often in σ.

A. Liveness properties

We consider a restricted notion of liveness property. Section II-C briefly sketches how to handle general LTL properties.

A *liveness property* φ of a net $N = (P, T, F, m_0)$ is a Boolean constraint over the free variables T. The property φ holds for an infinite occurrence sequence σ (denoted $\sigma \models \varphi$) iff $I_\sigma \models \varphi$, where $I_\sigma(t) = 1$ if $t \in inf(\sigma)$ else 0. A Petri net N satisfies a property φ (denoted $N \models \varphi$) iff $\sigma \models \varphi$ for every infinite occurrence sequence $m_0 \xrightarrow{\sigma}$. Note that a liveness property is always satisfied if the Petri net has no infinite occurrence sequences. Therefore the property $\varphi = false$ is equivalent to termination of the Petri net. Fair termination properties can be expressed by means of more complex formulas φ.

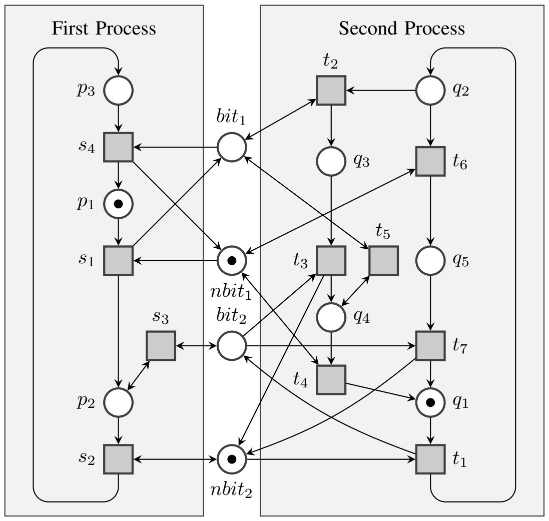


Fig. 2. Petri net for Lamport's 1-bit algorithm.

B. Two examples

As a first example, consider Lamport's 1-bit algorithm for mutual exclusion [16], shown in Fig. 1. Fig. 2 shows a Petri net model for the code. The two grey blocks model the control flow of the two processes. For instance, the token in place p_1 models the current position of process 1 at program location p_1. The four places in the middle of the diagram model the current values of the variables. For instance, a token in place $nbit_1$ indicates that the variable *bit*$_1$ is currently set to *false*.

The main liveness property for the processes states that, assuming a fair scheduler that allows both processes to execute actions infinitely often, each process enters the critical section infinitely often. For the first process, this corresponds to the property that every infinite occurrence sequence in which at least one of $s_1, \ldots, s_4$ and one of $t_1, \ldots, t_7$ occur infinitely often, contains infinitely many occurrences of s_2. As a Boolean formula, we get

$$\left(\bigvee_{i=1}^{4} s_i \right) \wedge \left(\bigvee_{j=1}^{7} t_j \right) \Rightarrow s_2$$

For the second process we obtain a similar property.

As a second example, consider the fairly terminating asynchronous program [17] given in Fig. 3. Here, the **post** command is a non-blocking operation for launching a process in parallel. Initially, the process INIT is executed, which sets x to *true* and launches H. Process H launches new instances of H and G until G sets x to *false*. Assuming a fair scheduler, i.e., one that will execute each process eventually, the program

```
    procedure H          procedure G              procedure INIT
    begin                begin                    begin
h:    if $x$ then        g:  $x$ := $false$           $x$ := $true$
        post H           end                        post H
        post G                                    end
      fi
    end
```

Fig. 3. Asynchronous program [17].

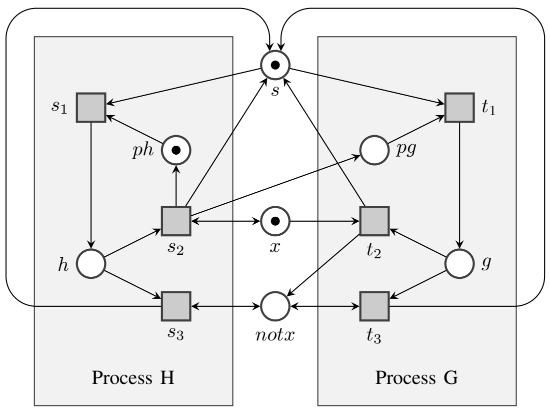


Fig. 4. Petri net for the asynchronous program.

should terminate. This fair termination is the liveness property we want to prove.

Transforming the program into a Petri net gives us the net in Fig. 4. The place s models the scheduler, ph and pg are pending instances of H and G, respectively, and h and g are program locations. The transitions t_1 and s_1 dispatch the processes, while the other transitions exit the processes depending on the value of x. Note that the net is unbounded, as repeatedly firing $s_1 s_2$ puts arbitrarily many tokens in pg.

If the scheduler is fair and continues dispatching instances of H and G infinitely often, the program should terminate, giving us the liveness property $s_1 \wedge t_1 \implies \mathit{false}$, equivalent to $\neg(s_1 \wedge t_1)$.

C. LTL properties

To check general LTL properties we can use the automata-theoretic approach. Given a property φ, we construct the product of the Petri net model of the system and a Büchi automaton for $\neg\varphi$. The product yields a new Petri net with a set of accepting places. The initial net violates the property iff the product net has an infinite sequence σ such that $\mathit{inf}(\sigma)$ contains at least one of the input transitions of the accepting places. A detailed construction can be found in [15].

III. T-SURINVARIANTS

We present a procedure, called LIVENESS, which checks a sufficient condition for a given Petri net to satisfy a liveness property. The condition is unsatisfiability of an appropriate linear arithmetic formula.

Definition 1 (Incidence matrix)**.** The *incidence matrix* C of a Petri net N is a $|P| \times |T|$ matrix given by

$$C(p,t) = F(t,p) - F(p,t)$$

Definition 2 (T-surinvariant)**.** A vector $X : T \to \mathbb{Z}$ is a *T-surinvariant* of a Petri net N iff $C \cdot X \geq 0$. If moreover $C \cdot X = 0$, then X is a *T-invariant*.

A T-surinvariant X is *semi-positive* iff $X \geq 0$ and $X \neq 0$. The *support* of a T-surinvariant X is given by $\|X\| = \{t \in T \mid X(t) > 0\}$.

Loosely speaking, X is a surinvariant if for every place p and for every occurrence sequence $m \xrightarrow{\sigma} m'$, if σ fires each transition t exactly $X(t)$ times, then $m(p) \leq m'(p)$, that is, the number of tokens in p can only increase. The following theorem, where we identify X with the multiset of transitions containing each $t \in T$ exactly $X(t)$ times, shows that the T-surinvariants of a Petri net provide information about its infinite runs.

Theorem 1. *[13], [14] Let σ be an infinite sequence of transitions and N a Petri net. If σ is an infinite occurrence sequence of N, then there is a semi-positive T-surinvariant X satisfying $\|X\| = \mathit{inf}(\sigma)$.*

Proof. Let σ' be a suffix of σ containing only transitions of $\mathit{inf}(\sigma)$, and let $\sigma' = \sigma'_1\sigma'_2\sigma'_3\ldots$ such that each σ'_i contains every transition of $\mathit{inf}(\sigma)$ at least once. Since σ is an infinite occurrence sequence of N, there exist markings $m_1, m_2, m_3, \ldots$ such that $m_1 \xrightarrow{\sigma'_1} m_2 \xrightarrow{\sigma'_2} m_3 \xrightarrow{\sigma'_3} \ldots$. By Dickson's lemma, there exist indices $i < j$ such that $m_i \leq m_j$. Let X be the Parikh vector of $\sigma'_i \ldots \sigma'_{j-1}$, i.e., the vector assigning to each transition its number of occurrences in the sequence. By the definition of the firing rule and the incidence matrix C, for every place p we have $m_j(p) - m_i(p) = \sum_{t \in T} C(p,t)X(t)$ or, in matrix form, $m_j - m_i = C \cdot X$. Since $m_j \geq m_i$, we have $m_j - m_i \geq 0$, and so X is a semi-positive T-surinvariant. Since $\sigma'_i \ldots \sigma'_{j-1}$ contains all transitions of $\mathit{inf}(\sigma)$, we have $\|X\| = \mathit{inf}(\sigma)$. □

However, a T-surinvariant does not guarantee the existence of a corresponding occurrence sequence. Consider the net in Fig. 4. The multiset $X = \{s_1, s_2, t_1, t_3\}$ is a semi-positive T-invariant, but, as we will see later, no infinite occurrence sequence σ satisfies $\mathit{inf}(\sigma) = \{s_1, s_2, t_1, t_3\}$. We say that a T-surinvariant X is *realizable* if there is an infinite occurrence sequence σ with $\|X\| = \mathit{inf}(\sigma)$.

For a T-surinvariant X and a liveness property φ, we denote by $\varphi(X)$ the constraint of linear arithmetic obtained by substituting $X(t) > 0$ for every occurrence of t in φ. So, for instance, if $\varphi = t_1 \vee t_2$, then $\varphi(X) = X(t_1) > 0 \vee X(t_2) > 0$. By Theorem 1, if there is an infinite sequence σ such that $\sigma \models \varphi$, then there is also a semi-positive T-surinvariant X such that $\varphi(X)$ holds. Taking the contrapositive, we have: if no semi-positive T-surinvariant X satisfies $\neg\varphi(X)$, then no sequence σ satisfies $\neg\varphi$, and so $N \models \varphi$. This directly leads to a semi-decision procedure for checking if a liveness property

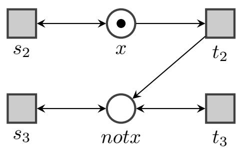


Fig. 5. Subnet of the net of Fig. 4.

φ is a property of a Petri net N: If the following constraints are unsatisfiable, then $N \models \varphi$.

$$\mathcal{C}(N,\varphi) :: \begin{cases} C \cdot X \geq 0 & \text{T-surinvariant condition} \\ X \geq 0 & \text{non-negativity condition} \\ X \neq 0 & \text{non-zero condition} \\ \neg\varphi(X) & \text{property condition} \end{cases} \qquad (1)$$

In practice, the procedure is very efficient, but often fails to prove the property. As an example, consider Lamport's algorithm. The negation of the fairness property for the first process yields

$$(s_1 \vee s_2 \vee s_3 \vee s_4) \wedge (t_1 \vee t_2 \vee t_3 \vee t_4 \vee t_5 \vee t_6 \vee t_7) \wedge \neg s_2$$

which corresponds to runs of the system where both processes are executed infinitely often, but where the first process never enters the critical section. However, $X = \{s_3, t_5\}$ is a solution to the constraints (1). The solution corresponds to the processes being stuck in the locations p_2 and q_4 while executing the **skip** commands. For this reason, in the next section we revisit an idea of [15] which leads to a more precise set of constraints.

IV. Refining T-surinvariants with P-components

The method Liveness can be strengthened by discarding T-surinvariants which are not realizable.

Consider again the net of Fig. 4. Recall that $X = \{s_1, s_2, t_1, t_3\}$ is a semi-positive T-invariant. We prove that it is not realizable. Consider the subnet $N' = (P', T')$, where $P' = \{x, notx\}$ and $T' = \{s_2, t_2, s_3, t_3\}$, shown in Fig. 5.

Inspection of the subnet shows that firing a transition does not change the total number of tokens in P'. For example, firing t_2 takes a token from x, but adds a token to $notx$. So this number is always equal to 1, and so it makes sense to speak of "the" token of N'. Assume now that X is realized by some infinite sequence σ, i.e., $\mathit{inf}(\sigma) = \|X\|$. Since both s_2 and t_3 occur infinitely often in σ, there are sequences $\sigma_1, \sigma_2, \sigma_3$ such that $\sigma = \sigma_1\, s_2\, \sigma_2\, t_3 \sigma_3$, and $\sigma_2 \in \|X\|^*$. After the occurrence of s_2 the token of N' is on x, and before the occurrence of t_3 it is on $notx$. But σ_2 cannot "move" the token from x to $notx$, as it does not contain any occurrence of t_2 (because $t_2 \notin \|X\|$). So we reach a contradiction, and σ does not exist.

In the rest of the section we show how to automatically search for proofs of non-realizability like this. We need the notion of a P-component of a net.

Definition 3 (P-component)**.** A *P-component* of a net $N = (P, T, F)$ is a subnet $N' = (P', T')$ such that $P' \neq \emptyset$ and $|t^\bullet \cap P'| = |{}^\bullet t \cap P'| = 1$ for all $t \in T'$ and $T' = P'^\bullet \cup {}^\bullet P'$ (where pre- and post-sets are taken with respect to N).

The subnet of Fig. 5 is a P-component. Note that the number of tokens in a P-component never changes, i.e., $m_0(P) = m(P)$ for all $m_0 \xrightarrow{\sigma} m$. Therefore, if initially a P-component only contains one token, then we know that the token will stay in the P-component.

Lemma 2. *Let X be a T-surinvariant of a Petri net N. If N has a P-component (P', T') such that $m_0(P') = 1$, and the subnet $(P', T' \cap \|X\|)$ is not strongly connected, then X is not realizable.*

Proof. (Sketch.) If $(P', T' \cap \|X\|)$ is not strongly connected, then by the definition of P-component there are two transitions $t_1, t_2 \in T' \cap \|X\|$ such that no path of $(P', T' \cap \|X\|)$ leads from t_1 to t_2. So the token of (P', T') cannot be transported from the output place of t_1 in $(P', T' \cap \|X\|)$ to the input place of t_2 in $(P', T' \cap \|X\|)$ by firing transitions of X only. Since every sequence realizing X must fire both t_1 and t_2 infinitely often, no such sequence exists. □

Lemma 2 provides a refinement condition. To find such a refinement, we encode the condition as a conjunction of linear arithmetic constraints. A pair (P', T') is the set of places and transitions of a P-component such that $m_0(P') = 1$ iff it satisfies these constraints:

$$\begin{array}{ll} \forall t \in T' : |t^\bullet \cap P'| = 1 & \quad P'^\bullet \cup {}^\bullet P' = T' \\ \forall t \in T' : |{}^\bullet t \cap P'| = 1 & \quad m_0(P') = 1 \end{array}$$

For the strong connectedness condition, we use that a graph (V, E) is not strongly connected iff there is a partition $V = V_1 \uplus V_2$ such that no edge $(v, v') \in E$ satisfies $v \in V_1, v' \in V_2$. In our case, V is the set $T' \cap \|X\|$, and E is the set of pairs (t_1, t_2) such that some place $p \in P'$ satisfies $(t_1, p), (p, t_2) \in F'$. So $(P', T' \cap \|X\|)$ is not strongly connected iff the following constraints are satisfiable:

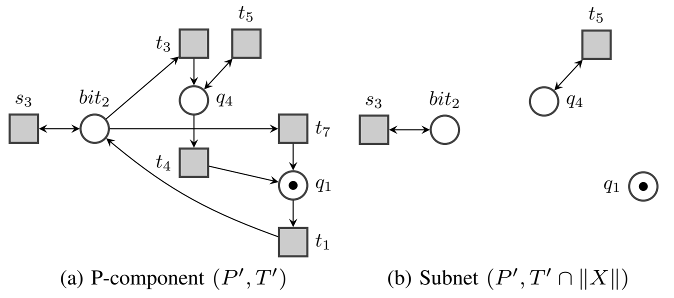
$$\begin{array}{rl} T' \cap \|X\| = T_1 \uplus T_2 & \quad T_1 \neq \emptyset \\ (T_1^\bullet \cap P')^\bullet \cap \|X\| \subseteq T_1 & \quad T_2 \neq \emptyset \end{array}$$

These constraints can be encoded by introducing an array of variables with range $\{0, 1\}$ for each set of places or transitions. For example, the constraint $\forall t \in T' : |t^\bullet \cap P'| = 1$ translates to the linear arithmetic constraint

$$\bigwedge_{t \in T} \left[T'(t) = 1 \implies \sum_{p \in t^\bullet} P'(p) = 1 \right]$$

where $(T'(t1), \ldots, T'(t_n))$ is the array of Boolean variables for the set T'.

If the constraints above are satisfiable for a given T-surinvariant X, then X is not realizable. We can exclude X (and any other T-surinvariant whose support has the same intersection with the P-component as $\|X\|$) by adding the constraint:



(a) P-component (P', T') (b) Subnet $(P', T' \cap \|X\|)$

Fig. 6. P-component and subnet of the Petri net for Lamport's algorithm.

$$\delta ::= \left[\bigvee_{t \in T_1} t\right] \wedge \left[\bigvee_{t \in T_2} t\right] \implies \bigvee_{t \in T' \setminus \|X\|} t \qquad (2)$$

to the set of constraints (1). We can iterate the process, until either the constraints are unsatisfiable, which means successfully proving the property, or no further P-components can be found to discard a T-surinvariant, which means failure.

For example, for Lamport's algorithm and the fairness property for the first process, the constraints (1) have the solution $X = \{s_3, t_5\}$. However, since $(P', T') = (\{bit_2, q_1, q_4\}, \{s_3, t_1, t_3, t_4, t_5, t_7\})$ is a P-component and $T_1 = \{s_3\}$, $T_2 = \{t_5\}$ satisfy the constraints above, we get that X is not realizable. The P-component (P', T') and the subnet $(P', T' \cap \|X\|)$ are shown in Fig. 6. We can immediately see that the token cannot be transported from the output place of s_3 to the input place of t_5.

We add the refinement constraint

$$s_3 \wedge t_5 \implies t_1 \vee t_3 \vee t_4 \vee t_7$$

to the set (1) and check again for satisfiability. The new set is still satisfiable with the solution $X = \{s_3, t_1, t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. In a second refinement step we find a P-component with $\{nbit_1, p_2, p_3\}$ as set of places, and add the refinement constraint

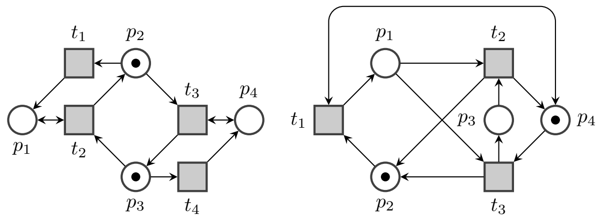
$$s_3 \wedge (t_4 \vee t_6) \implies s_1 \vee s_2 \vee s_4,$$

after which the constraints (1) are unsatisfiable, and we conclude that the fairness property for the first process holds.

For the second example (Fig. 3 and 4), we considered the fair termination property $\varphi = \neg(s_1 \wedge t_1)$. After adding $\neg\varphi = s_1 \wedge t_1$ to the constraints (1), we obtain a solution $X = \{s_1, s_2, t_1, t_3\}$. With the P-component $(P', T') = (\{x, notx\}, \{s_2, s_3, t_2, t_3\})$ and the partition $T_1 = \{s_2\}$ and $T_2 = \{t_3\}$, we can discard this T-invariant as unrealizable and obtain the refinement constraint

$$s_2 \wedge t_3 \implies s_3 \vee t_2,$$

after which the constraints (1) are unsatisfiable and we can prove fair termination.



(a) Net without P-components (b) Net without unmarked traps

Fig. 7. Terminating Petri nets for which refinement is insufficient.

V. Refining T-surinvariants with Traps

For some Petri nets, refinement with P-components is not sufficient for discarding unrealizable T-surinvariants. For example, we cannot prove the properties for the leader election algorithm by Dolev, Klawe and Rodeh [18] or the mutual exclusion algorithm by Szymanski [19]. These nets are too big to give as an example, but consider instead the net in Fig. 7a, which is similar to a subnet of the net for the leader election algorithm. The net has no infinite occurrence sequences and we would like to prove termination. The multiset $X = \{t_2, t_3\}$ is a T-surinvariant, but the net has no P-components, so we cannot refine the constraints. To solve this problem we develop a refinement technique based on traps.

Definition 4 (Trap)**.** A *trap* is a set of places $S \subseteq P$ such that $S^\bullet \subseteq {}^\bullet S$.

It follows immediately from the definition that marked traps stay marked: if a trap S is marked at some marking m, i.e. $m(S) > 0$, then it is also marked at all markings m' reachable from m, because every transition taking tokens from S also adds at least one token to S.

Given a T-surinvariant X, we consider the subnet $(P', T') = (\|X\|^\bullet, \|X\|)$. In the example of Fig. 7a, (P', T') is obtained by removing transitions t_1 and t_4, together with their input and output arcs. Assume $\|X\|$ is realized by an infinite occurrence sequence σ. Then there are sequences σ', σ'' such that $\sigma = \sigma'\sigma''$ and $\sigma'' \in \|X\|^\omega$. Since every place P' has an input transition in $\|X\|$, every place of P' gets marked during the execution of σ'', and therefore every trap of (P', T') becomes eventually marked. So we have the following lemma:

Lemma 3. *Let $N = (P, T, F, m_0)$ be a net and let $\|X\|$ be a realizable T-surinvariant. Then some marking m reachable from m_0 in N marks every trap of the subnet $(P', T') = (\|X\|^\bullet, \|X\|)$.*

By this lemma, if we show that no reachable marking marks every trap of (P', T'), then X is unrealizable. We use an iterative approach. Given a set of traps $\mathcal{Q}$, using the technique of [11] we can construct a set of constraints satisfied by every reachable marking that marks every trap of $\mathcal{Q}$.[1] If the constraints are satisfiable, then we extract from the solution a

[1]The constraints express that a solution m satisfies the marking equation and that $m(S) > 0$ for every trap $S \in \mathcal{Q}$.

marking m that marks all traps in $\mathcal{Q}$. Since m may not mark all traps, we search for a new trap $S \notin \mathcal{Q}$ not marked at m. If we find such S, we set $\mathcal{Q} = \mathcal{Q} \cup \{S\}$ and iterate, otherwise we give up. If the constraints are unsatisfiable, then no reachable marking marks all traps in $\mathcal{Q}$, which implies that X is not realizable. We can then add a new constraint excluding any solution with the same support as X. However, we can do better, and add a stronger constraint. Since we have shown that no infinite occurrence sequence σ can reach a marking that simultaneously marks all traps of $\mathcal{Q}$, we choose a constraint expressing that if $inf(\sigma)$ contains transitions marking all traps of (P', T'), then it must also contain at least one transition that empties a trap of (P', T'). (Of course, such a transition cannot belong to T', it must be a transition of $T \setminus T'$.)

$$\delta ::= \bigwedge_{S \in \mathcal{Q}} \left[\bigvee_{t \in {}^\bullet S} t \right] \implies \bigvee_{S \in \mathcal{Q}} \left[\bigvee_{t \in S^\bullet \setminus {}^\bullet S} t \right] \tag{3}$$

For example, for the Petri net in Fig. 7a, the method LIVENESS returns $X = \{t_2, t_3\}$ as a T-surinvariant. The corresponding subnet is $(P', T') = (\{p_1, p_2, p_3, p_4\}, \{t_2, t_3\})$. Initially, for $\mathcal{Q} = \emptyset$, we can take the initial marking m_0. In m_0, the trap $S_1 = \{p_1\}$ of (P', T') is unmarked. We search for a marking m satisfying the marking equation and $m(p_1) \geq 1$, and obtain as solution $m_1 = (1, 0, 1, 0)$. At this marking the trap $S_2 = \{p_4\}$ is unmarked. So we search for a marking m satisfying the marking equation, $m(p_1) \geq 1$ and $m(p_4) \geq 1$, and obtain as solution $m_2 = (1, 0, 0, 1)$. At this marking the trap $S_3 = \{p_2, p_3\}$ is unmarked. We search for a marking m satisfying the marking equation, $m(p_1) \geq 1$, $m(p_4) \geq 1$, and $m(p_2) + m(p_3) \geq 1$, and obtain that the constraints are unsatisfiable. So we generate the refinement constraint

$$(t_1 \vee t_2) \wedge (t_3 \vee t_4) \wedge (t_2 \vee t_3) \implies t_1 \vee t_4,$$

which excludes $\{t_2, t_3\}$. In fact, the additional constraint turns out to exclude not only $\{t_2, t_3\}$, but all T-surinvariants, which proves termination of the Petri net.

The refinement with traps is a generalization of the refinement with P-components. For a T-surinvariant X, assume there is refinement with a P-component (P', T') and a partition $T_1 \uplus T_2 = T' \cap \|X\|$. In the subnet $(\|X\|^\bullet, \|X\|)$, $S_1 = P' \cap T_1^\bullet$ and $S_2 = P' \cap T_2^\bullet$ are two different traps, and as the P-component has only one token, we can show with the marking equation that S_1 and S_2 cannot be marked at the same time.

Generally, refinement with P-components requires fewer calls to the SMT solver, and is therefore more efficient. In our experiments it is also sufficient for most cases, and if it fails, refinement with traps can be applied afterwards. So we always start with a refinement with P-components, and apply then a refinement with traps if necessary.

Even with both refinements, the method is still incomplete. Consider the Petri net in Fig. 7b, which appears in a Petri net model of the drinking philosopher's problem [20]. The net is terminating, but $X = \{t_1, t_1, t_2, t_3\}$ is a surinvariant (observe that X is a genuine multiset with two copies of t_1). The subnet corresponding to X is the complete net, and every trap is initially marked, so no refinement can be found.

If the property does not hold, our method fails and returns a surinvariant that cannot be excluded by our refinements. For example, for Lamport's algorithm, the fairness property for the second process is not satisfied, where the negation $\neg\varphi$ is:

$$(s_1 \vee s_2 \vee s_3 \vee s_4) \wedge (t_1 \vee t_2 \vee t_3 \vee t_4 \vee t_5 \vee t_6 \vee t_7) \wedge \neg t_6.$$

After two refinement steps, our method returns the T-surinvariant $X = \{s_1, s_2, s_3, s_4, t_1, t_2, t_3, t_4, t_5\}$, which satisfies $\neg\varphi$ and cannot be further refined. In this case we can use guided state-space exploration [21] to try to identify permutations of X that are actual repeatable occurrence sequences. In this case, $\sigma = s_1 t_1 s_3 t_2 t_3 s_2 t_5 s_4 t_4$ is indeed a matching occurrence sequence which can be repeated infinitely and violates the property.

VI. Experimental Evaluation

We extended our tool *Petrinizer* [11], implemented on top of the SMT solver Z3 [22], with the method LIVENESS. The method can be used without refinement, with only P-component or trap refinement, or with P-component refinement followed by trap refinement. In addition, the refinement structures can be minimized.

For our evaluation, we had three goals. First, we wanted to measure the success rates on a large number of case studies. The second goal was to investigate the usefulness and necessity of P-components, traps and minimization of them. As a third goal, we wanted to measure the performance of the method and compare it with the model checker SPIN[2] [23].

A. Benchmarks

For the evaluation, we used five different benchmark suites from various sources. The first two suites are workflow nets coming from business processes [24]. One is a collection of SAP reference models [25] and the other consists of IBM business process models [26]. We examined the nets for termination. In total, these suites contain 1976 models, out of which 1836 are terminating.

The third suite contains 50 examples that come from the analysis of Erlang programs [5], found on the website of the Soter tool[3]. Out of these, 33 are terminating.

For the fourth suite, we used classic asynchronous programs that can be scaled in the number of processes. These include a leader election algorithm [18], a snapshot algorithm [27] and three mutual exclusion algorithms [16], [19], [28]. Each of the 5 algorithms is scaled from $n = 2$ to 6 processes, resulting in 25 examples. For the former two distributed algorithms, the property is repeated liveness, i.e., infinitely often electing a leader or taking a snapshot infinitely often, while for the latter three mutual exclusion algorithms it is non-starvation for the first process. These properties all contain a fairness assumption for the scheduler, and they hold for all examples.

[2]http://spinroot.com/

[3]http://mjolnir.cs.ox.ac.uk/soter/

TABLE I
FAIRLY TERMINATING EXAMPLES WITH RATE OF SUCCESS BY DIFFERENT REFINEMENT METHODS.

Benchmark	No ref.	Ref. w/P-co.	Ref. w/traps	Terminating
SAP	1263	1263	1264	1264
IBM	571	571	572	572
Erlang	27	27	27	33
Asynchronous	0	14	20	25
Literature	0	3	5	5
Total	1861	1878	1888	1899

TABLE II
COMPARISON OF REFINEMENT WITH AND WITHOUT MINIMIZATION AND RUNTIME COMPARISON WITH SPIN. FOR AN EXECUTION, TO DENOTES EXCEEDING THE TIME LIMIT AND MO EXCEEDING THE MEMORY LIMIT.

		Refinement R_1			Ref. w/ min. R_2			SPIN
Benchmark	n	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{Q}\rvert$	T (s)	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{Q}\rvert$	T (s)	T (s)
Leader election by Dolev, Klawe and Rodeh [18]	2	0	4	2.53	0	4	2.30	0.69
	3	0	6	8.45	0	6	9.03	0.74
	4	0	8	35.5	0	8	38.4	15.7
	5	0	13	206	0	10	154	MO
	6	0	17	1104	0	12	728	MO
Snapshot algorithm by Bougé [27]	2	2	0	0.35	2	0	0.30	0.31
	3	3	0	0.50	3	0	0.81	0.72
	4	4	0	0.60	4	0	0.91	10.3
	5	5	0	0.73	5	0	1.41	218
	6	6	0	1.82	6	0	1.63	MO
Lamport's 1-bit algorithm for mutual exclusion [16]	2	2	0	0.50	3	0	0.43	0.69
	3	6	0	1.26	6	0	1.63	0.69
	4	12	0	2.83	13	0	5.50	0.92
	5	27	0	9.34	18	0	11.3	10.4
	6	26	0	13.4	23	0	20.6	MO
Peterson's mutual exclusion algorithm [28]	2	1	0	0.37	1	0	0.41	0.69
	3	13	0	6.57	7	0	8.55	0.71
	4	21	0	65.9	18	0	92.5	1.16
	5	285	0	2289	36	0	911	43.5
	6	-	-	TO	-	-	TO	MO
Szymanski's mutual exclusion algorithm [19]	2	21	6	10.9	26	6	17.6	0.70
	3							0.80
	4	Property cannot be proven with						5.83
	5	refinement for $n \geq 3$.						347
	6							MO

Finally, as the fifth suite, we collected 5 examples from the literature on termination and liveness analysis and modeled them as Petri nets. These are the programs from Fig. 2 in [29], Fig. 3 in [30], Fig. 1(b) in [17] and two variants of the Windows NT Bluetooth driver from [31]. These are all terminating programs.

The Petri nets for these benchmarks vary largely in size. The number of places ranges from 4 to 66950, with a mean of 116 and a median of 38. The number of transitions ranges from 3 to 213626, with a mean of 163 and a median of 30.

We try to prove the fairness property of the asynchronous programs and termination for the examples from the other benchmark suites. In total, we have 1899 examples where the property holds.

B. *Rate of success on terminating examples*

In Table I, the rate of success with different refinement methods is shown. Even without refinement, we can prove termination of all but 2 of the SAP and IBM examples, and of 27 of the 33 Erlang examples. However, without refinement we can prove none of the 30 examples from the other two suites. Refinement with P-components allows us to prove 14 of the asynchronous and 3 of the literature examples. Additional refinement with traps allows us to prove the 2 remaining SAP and IBM examples, 6 more asynchronous examples, and the remaining examples from the literature suite. In total, we can prove termination for 1888 of the 1899 terminating examples, and at least 80% of the terminating examples of each suite.

C. *Usefulness of refinement methods and minimization*

Table II presents results on the asynchronous benchmark suite and refinement with and without minimization. Minimization of the refinement components can result in better refinement constraints that exclude more T-surinvariants, at the price of a time overhead, since repeated calls to the SMT solver are needed until a minimal component is found. The default method, R_1, is refinement with P-components and traps without any minimization. Refinement method R_2 minimizes P-components (P', T') by $|P'|$ and traps S by $|S|$. Other criteria were also tested, but there was no optimal one working for all benchmarks. For each method, the number of P-components $|\mathcal{R}|$, number of trap refinements $|\mathcal{Q}|$ and total execution time in seconds for proving the property are given.

We observe cases where we need refinement only with P-components (Snapshot), only with traps (Leader election) or with both (Szymanski at $n = 2$). For Szymanski at $n \geq 3$ we cannot prove the property even with both refinement methods.

Minimization with method R_2 saves many refinement steps for Peterson and a few for Lamport and Leader election, while for Szymanski the number of steps increases. The time overhead when no steps are saved is not very large (up to $2\times$).

Our method produces a certificate for fair termination consisting of the P-components $\mathcal{R}$ and traps $\mathcal{Q}$. One can use independent methods to check that $\mathcal{R}$ and $\mathcal{Q}$ are indeed P-components and traps, and that the constraints (1) are unsatisfiable. The size of each P-component and trap is limited by the size of the net. The size of the whole certificate depends on the number of refinement steps, however it is usually much more compact than the whole state space.

D. *Performance*

All experiments were performed on the same machine, equipped with a Intel Core i7-4810MQ CPU at 2.8 GHz and 16 GB of memory, running Linux 3.18.6 in 64-bit mode. Execution time was limited to 2 hours and memory to 16 GB.

Table II shows the execution times of Petrinizer for the asynchronous benchmark suite and a comparison with SPIN. SPIN was used with a fairness strategy enforced and partial order reduction. Only for the snapshot algorithm, partial order reduction was turned off, as it is not supported together with fairness and the rendezvous operations used in the algorithm. For small examples, SPIN is usually faster. However, as n

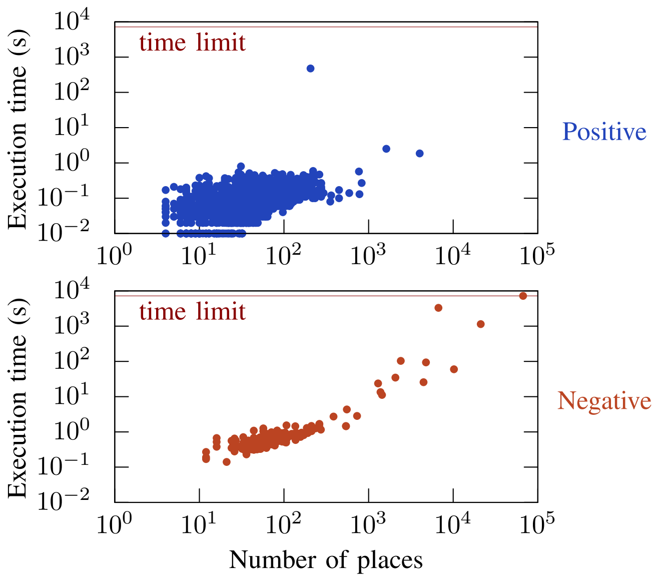


Fig. 8. Execution time in dependence on the number of places for the examples from the benchmark suites SAP, IBM, Erlang and Literature, depending on whether Petrinizer succeeds in proving termination.

grows to 5 or 6, SPIN quickly reaches the memory limit. Here, Petrinizer outperforms SPIN significantly on the examples Leader election, Snapshot and Lamport.

For the other four benchmark suites, Fig. 8 shows the performance of Petrinizer. For the positive examples (i.e., those where we can prove the property), we can prove all but one of the 1868 examples in under 3 seconds. The outlier is from the SAP suite, for which we need 320 refinement steps and 8 minutes. Even the largest positive example from the Erlang suite with 4014 places only needs 1.86 seconds. For the negative examples, Petrinizer performs worse, usually because it performs more refinement steps. However, it terminates in under 3 seconds for all nets with up to 1000 places. Only in one case we reach the time limit of 2 hours (our largest example with 66950 places).

We only need more than 3 refinement steps in one case (an outlier with 320 steps). The number of steps is not correlated to the net size.

VII. Conclusion

Transition invariants and P-components are classical analysis techniques for Petri nets. We have demonstrated that, combined with a state-of-the-art SMT solver, these techniques are very effective in proving fair termination for a large number of common benchmark examples. We have further developed a novel technique based on traps, which allows us to reach a high degree of completeness on these benchmarks. The constraint systems produced by our tool can be used as a certificate of fair termination.

Acknowledgments

We thank Filip Niksic, Corneliu Popeea, and Karsten Wolf for kindly providing examples for our experimental evaluation.

References

[1] A. Kaiser, D. Kroening, and T. Wahl, “Dynamic cutoff detection in parameterized concurrent programs,” in *CAV*, 2010, pp. 645–659.
[2] P. Ganty and R. Majumdar, “Algorithmic verification of asynchronous programs,” *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, p. 6, 2012.
[3] A. Bouajjani and M. Emmi, “Bounded phase analysis of message-passing programs,” in *TACAS*, 2012, pp. 451–465.
[4] A. Kaiser, D. Kroening, and T. Wahl, “Efficient coverability analysis by proof minimization,” in *CONCUR*, 2012, pp. 500–515.
[5] E. D’Osualdo, J. Kochems, and C.-H. L. Ong, “Automatic verification of Erlang-style concurrency,” in *SAS*, 2013, pp. 454–476.
[6] G. Geeraerts, J.-F. Raskin, and L. V. Begin, “Expand, enlarge and check: New algorithms for the coverability problem of WSTS,” *J. Comput. Syst. Sci.*, vol. 72, no. 1, pp. 180–203, 2006.
[7] P. Ganty, J.-F. Raskin, and L. Van Begin, “From many places to few: Automatic abstraction refinement for Petri nets,” *Fundam. Inform.*, vol. 88, no. 3, pp. 275–305, 2008.
[8] A. Valmari and H. Hansen, “Old and new algorithms for minimal coverability sets,” in *Petri Nets*, 2012, pp. 208–227.
[9] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac, “Incremental, inductive coverability,” in *CAV*, 2013, pp. 158–173.
[10] C. Rackoff, “The covering and boundedness problems for vector addition systems,” *Theor. Comput. Sci.*, vol. 6, pp. 223–231, 1978.
[11] J. Esparza, R. Ledesma-Garza, R. Majumdar, P. Meyer, and F. Niksic, “An SMT-based approach to coverability analysis,” in *CAV*, 2014, pp. 603–619.
[12] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
[13] J. Desel and J. Esparza, *Free Choice Petri Nets*. Cambridge University Press, 1995.
[14] W. Reisig, *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.
[15] J. Esparza and S. Melzer, “Model checking LTL using constraint programming,” in *ICATPN*, 1997, pp. 1–20.
[16] L. Lamport, “The mutual exclusion problem - part II: Statement and solutions,” *J. ACM*, vol. 33, no. 2, pp. 327–348, 1986.
[17] P. Ganty, R. Majumdar, and A. Rybalchenko, “Verifying liveness for asynchronous programs,” in *POPL*, 2009, pp. 102–113.
[18] D. Dolev, M. M. Klawe, and M. Rodeh, “An o(n log n) unidirectional distributed algorithm for extrema finding in a circle,” *J. Algorithms*, vol. 3, no. 3, pp. 245–260, 1982.
[19] B. K. Szymanski, “A simple solution to lamport’s concurrent programming problem with linear wait,” in *ICS*, 1988, pp. 621–626.
[20] K. M. Chandy and J. Misra, “The drinking philosopher’s problem,” *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, pp. 632–646, 1984.
[21] H. Wimmel and K. Wolf, “Applying CEGAR to the Petri net state equation,” *Logical Methods in Computer Science*, vol. 8, no. 3, 2012.
[22] L. M. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *TACAS*, 2008, pp. 337–340.
[23] G. J. Holzmann, “The model checker SPIN,” *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.
[24] W. M. P. van der Aalst, “Challenges in business process management: Verification of business processing using petri nets.” *Bulletin of the EATCS*, vol. 80, pp. 174–199, 2003.
[25] B. F. van Dongen, M. H. Jansen-Vullers, H. M. W. Verbeek, and W. M. P. van der Aalst, “Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants,” *Computers in Industry*, vol. 58, no. 6, pp. 578–601, 2007.
[26] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, “Instantaneous soundness checking of industrial business process models,” in *BPM*, 2009, pp. 278–293.
[27] L. Bougé, “Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP,” *Theor. Comput. Sci.*, vol. 49, pp. 145–169, 1987.
[28] G. L. Peterson, “Myths about the mutual exclusion problem,” *Inf. Process. Lett.*, vol. 12, no. 3, pp. 115–116, 1981.
[29] B. Cook, A. Podelski, and A. Rybalchenko, “Proving thread termination,” in *PLDI*, 2007, pp. 320–330.
[30] A. Podelski and A. Rybalchenko, “Transition invariants,” in *LICS*, 2004, pp. 32–41.
[31] S. Qadeer and D. Wu, “KISS: keep it simple and sequential,” in *PLDI*, 2004, pp. 14–24.

Compositional Recurrence Analysis

Azadeh Farzan
University of Toronto
azadeh@cs.toronto.edu

Zachary Kincaid
University of Toronto
zkincaid@cs.toronto.edu

Abstract—**This paper presents a new method for automatically generating numerical invariants for imperative programs. The procedure computes a transition formula which over-approximates the behaviour of a given input program. It is compositional in the sense that it operates by decomposing the program into parts, computing a transition formula for each part, and then composing them. Transition formulas for loops are computed by extracting recurrence relations from a transition formula for the loop body and then computing their closed forms. Experimentation demonstrates that this method is competitive with leading verification techniques based on abstraction refinement.**

I. Introduction

Compositional program analyses operate by decomposing a program into parts, computing an abstract meaning of each part, and then composing the meanings. Compositional analyses have a number of desirable properties, including scalability, parallelizability, and applicability to incomplete programs. However, compositionality comes at a price: since each program fragment is analyzed independently of its context, the analysis cannot benefit from contextual information. This paper presents a compositional analysis which, despite loss of contextual information, is capable of generating precise numerical invariants.

```
while (*) :
  x := x + 1
  y := y - 2
```

Compositional recurrence analysis, the technique proposed in this paper, aims to compute a *transition formula* that over-approximates the behaviour of a given program. As with any invariant generation technique, the crucial question is how to approximate the behaviour of loops. The basic idea can be illustrated with the example loop to the right, which loops for a non-deterministic number of iterations, adding 1 to x and subtracting 2 from y at each iteration. The body of this loop can be described by a system of recurrence equations:

$$\mathtt{x}^{(k)} = \mathtt{x}^{(k-1)} + 1$$
$$\mathtt{y}^{(k)} = \mathtt{y}^{(k-1)} - 2$$

(where $\mathtt{x}^{(k)}$ represents the value of x on the kth iteration of the loop). A transition formula for the loop can be computed by taking the closed form of this system. Using x and y to denote the values of those variables before executing the loop and x′ and y′ to their values after the loop, a transition formula that *precisely* describes this loop is:

$$\exists k \in \mathbb{N}. \mathtt{x}' = \mathtt{x} + k \land \mathtt{y}' = \mathtt{y} - 2k$$

The idea of using recurrences to abstract loops is classical, and there exist powerful techniques for solving very general classes of recurrence equations. However, two barriers stand in the way of applying recurrence analysis to real programs. First, loop bodies may have arbitrary control flow. Extracting recurrence equations from a straight-line loop body like the one above is straightforward, but what if the loop body contains branching, or nested loops, or even unstructured control flow? Second, the behaviour of a loop may not be describable as a system of recurrence equations (for example, consider a loop where x is non-deterministically incremented by either 1 or 2). How can recurrence analysis be used to over-approximate loops whose behaviour is not determined by system of recurrence equations?

Our approach exploits compositionality to overcome these barriers. The assumption of compositionality demands that a transition formula for a loop is computed from a transition formula for its body. This makes the control flow of the loop body irrelevant: whether it is a sequence of assignments or contains branching or nested loops – its transition formula is just a formula. While this circumvents the first barrier to practical recurrence analysis, it also presents a new challenge: how can we extract a system of recurrence equations from a formula representing a loop body? Our solution is to use a Satisfiablity Modulo Theories (SMT) solver to extract recurrence equations which are semantically implied by a loop body formula. In fact, our method goes beyond systems of recurrence equations over program variables: it extracts a system of recurrence *inequations* over *linear terms*. This allows compositional recurrence analysis to compute accurate over-approximations of loops which cannot be described as a system of recurrence equations, thereby overcoming the second barrier to practical recurrence analysis.

The rest of the paper is organized as follows. In the next section, we give a high-level overview of our algorithm. In Section III, we describe our strategy for over-approximating the behaviour of loop whose body is expressed as a linear arithmetic formula; Section IV describes a method for over-approximating a non-linear arithmetic formula by a linear arithmetic formula, so that our loop approximation procedure may be applied to any arithmetic formula. In Section V, we demonstrate experimentally that compositional recurrence analysis compares favourably with leading (non-compositional) verification techniques. Section VI compares with related work, and Section VII concludes.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

II. Overview

We will adopt a simple intra-procedural program model in which a program is represented by a control flow automaton (CFA) where edges are labeled by program statements. Figure 1(b) depicts such a CFA for a program that computes the quotient and remainder of division of a variable x by a variable y. Naturally, compositional recurrence analysis can be extended to a program model with procedures by using the analysis to compute procedure summaries [31], but for the sake of simplicity we will not discuss this extension formally.

Compositional recurrence analysis (CRA) is presented in the algebraic abstract interpretation framework described in [10]. In [10], a program analysis is defined by an *interpretation*, which consists of a *semantic algebra* and a *semantic function*. A semantic algebra consists of a *universe* which defines the space of possible program meanings, along with *sequencing* ($\odot$), *choice* ($\oplus$), and *iteration* ($\circledast$) operators, which define how to compose program meanings. A semantic function is a mapping from control flow edges to elements of the universe which defines the meaning of each control flow edge.

Path expression algorithms [7], [29], [32] form the algorithmic foundation of the algebraic framework. A *path expression* is a regular expression over an alphabet of control flow edges which recognizes the set of paths through a control flow automaton. Intuitively, path expression algorithms operate by computing a path expression to a control point of interest and then interpreting that path expression in the semantic algebra defining the analysis. The exponential blow-up of computing a regular expression from a control flow automaton is avoided by sharing sub-expressions and evaluating the path expression bottom-up. Path expression algorithms can share work over multiple queries to avoid duplicate work if there are multiple points of interest (e.g., if there is more than one assertion to be verified).

More concretely, suppose that we wish to prove that the assertion on the edge from v_8 to v^{exit} always succeeds using CRA. First, we compute a path expression for vertex v_8 (Figure 1(c)). This regular expression recognizes the set of paths from v^{entry} to v_8. Second, we evaluate this path expression in the semantic algebra of CRA by recursing on the regular expression, interpreting each edge using the semantic function and each regular expression operator by its counterpart in the semantic algebra that defines CRA. The result is a transition formula which approximates the executions which end at v_8. Third, we ask an SMT solver whether the transition formula implies that the assertion holds in the post-state. If the implication holds, then we may conclude that the assertion is safe. If not, then the verification is inconclusive: either the assertion fails in some execution, or the assertion is safe but the transition formula computed by CRA is not strong enough to prove it (the analysis cannot distinguish between these cases).

Keeping this framework in mind, we proceed to describe the interpretation which defines compositional recurrence analysis.

CRA Universe. The semantic universe of CRA (i.e., the space of program meanings) is the set of arithmetic *transition formulas*. Letting Var denote the set of program variables and Var' the set of "primed" copies of program variables, a transition formula is an arithmetic formula with free variables in $\mathsf{Var} \cup \mathsf{Var}'$.

Transition formulas may contain existential quantifiers and non-linear arithmetic. For readability, we will often simplify formulas by eliminating quantifiers in the remainder of the paper. However, CRA does not require quantifier elimination (and indeed, there is no quantifier elimination procedure for the class of formulas we consider, since we allow non-linear integer arithmetic).

CRA Semantic Function. The semantic function $[\![\cdot]\!]$ maps each edge of a control flow automaton to its interpretation as a transition formula. For example (again considering Figure 1), we have

$$[\![\langle v^{\mathsf{entry}}, v_1\rangle]\!] \triangleq \boxed{r' = x \land id(\{q,t,x,y\})}$$
$$[\![\langle v_2, v_3\rangle]\!] \triangleq \boxed{r \geq y \land id(\{q,r,t,x,y\})}$$

where for $X \subseteq \mathsf{Var}$, we define $id(X) \triangleq \boxed{\bigwedge_{x\in X} x' = x}$; we use id to factor out equalities from the formulas and make them more legible. Boxes around formulas have no meaning, and are used to make it easier to distinguish between equalities in formulas and the meta-language.

CRA Operators. The sequencing and choice operators of CRA are defined as follows:

$$\varphi \odot \psi \triangleq \boxed{\exists x''.\varphi[x''/x'] \land \psi[x''/x]} \qquad \text{Sequencing}$$
$$\varphi \oplus \psi \triangleq \boxed{\varphi \lor \psi} \qquad \text{Choice}$$

(where $\varphi[x''/x']$ denotes the formula obtained from φ by replacing each primed variable x' by its double-primed counterpart x'', and $\psi[x''/x]$ similarly replaces unprimed variables in ψ with double-primed variables).

The semantic function, sequencing, and choice operators are sufficient to analyze loop-free code. For example, CRA computes a transition formula for the body of the inner loop of Figure 1 as follows:

$$\begin{aligned}[\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle]\!] &= [\![\langle v_4, v_5\rangle]\!] \odot [\![\langle v_5, v_6\rangle]\!] \\ &\equiv \boxed{t \neq 0 \land r' = r - 1 \land id(\{q,t,x,y\})}\end{aligned}$$

$$\begin{aligned}[\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle \cdot \langle v_6, v_4\rangle]\!] &= [\![\langle v_4, v_5\rangle\cdot\langle v_5, v_6\rangle]\!] \odot [\![\langle v_6, v_4\rangle]\!] \\ &\equiv \boxed{t \neq 0 \land r' = r - 1 \land t' = t - 1 \land id(\{q,x,y\})}\end{aligned}$$

Having defined the semantic function, semantic universe, and sequencing and choice operators for CRA, it remains only to define its iteration operator. The formal definition of the iteration operator appears in the next section; in this section, we will illustrate the iteration operator on the running example.

Let $\varphi_{\mathsf{inner}} \triangleq [\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle \cdot \langle v_6, v_4\rangle]\!]$ be the formula above, which represents the body of the inner loop. The meaning of the inner loop is computed by applying the iteration operator to φ_{inner}.

Recurrence	Closed form
$r' = r - 1$	$r^{(k)} = r^{(0)} - k$
$t' = t - 1$	$t^{(k)} = t^{(0)} - k$

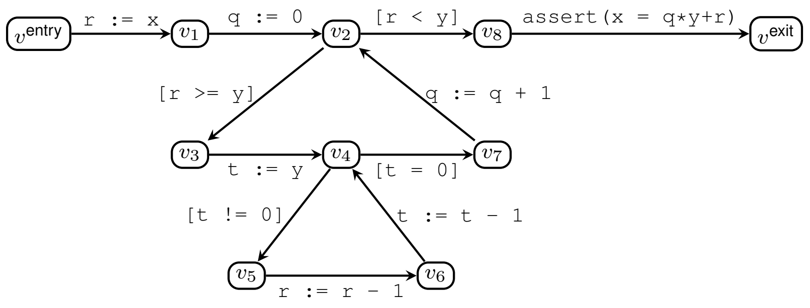
CRA's iteration operator begins by extracting the recurrence equations shown to the right (note

```
r := x  // remainder
q := 0  // quotient
while(r >= y):
   // subtract y from r
   t := y
   while(t != 0)
      r := r - 1
      t := t - 1

   q := q + 1

assert(x = q*y + r)
```

(a) Program text



(b) Control Flow Automaton for (a)

$$\langle v^{\text{entry}}, v_1\rangle\cdot\langle v_1,v_2\rangle\cdot\underbrace{\Big(\langle v_2,v_3\rangle\cdot\langle v_3,v_4\rangle\cdot\overbrace{\big(\langle v_4,v_5\rangle\cdot\langle v_5,v_6\rangle\cdot\langle v_6,v_4\rangle\big)^*}^{\text{Inner loop}}\cdot\langle v_4,v_7\rangle\cdot\langle v_7,v_2\rangle\Big)^*}_{\text{Outer loop}}\cdot\langle v_2,v_8\rangle$$

(c) Path expression to v_8

Fig. 1. An integer division program, computing a quotient and remainder. A statement $[\psi]$ denotes an *assumption* which blocks if ψ does not hold.

that this table omits "uninteresting" recurrences, such as $q' = q + 0$, which indicate that a variable does not change in a loop). It then computes closed forms for these recurrences, also shown to the right (where $x^{(k)}$ denotes the value that the variable x takes on the kth iteration of the loop). These closed forms are used to abstract the loop as follows:

$$\varphi^{\circledast}_{\text{inner}} = \boxed{\exists k. k \geq 0 \land r' = r - k \land t' = t - k \land id(\{q, x, y\})}$$
$$\equiv \boxed{r' = r + t' - t \land t' \leq t \land id(\{q, x, y\})}$$

The path expression algorithm uses the formula $\varphi^{\circledast}_{\text{inner}}$ for the inner loop to compute a transition formula representing the body of the outer loop as follows:

$$\varphi_{\text{outer}} = [\![\langle v_2, v_3\rangle]\!] \odot [\![\langle v_3, v_4\rangle]\!] \odot \varphi^{\circledast}_{\text{inner}} \odot [\![\langle v_4, v_7\rangle]\!] \odot [\![\langle v_7, v_2\rangle]\!]$$
$$\equiv \boxed{q' = q+1 \land r' = r+t'-y \land t' = 0 \land r \geq y \land id(\{x, y\})}$$

We then apply the iteration operator to φ_{outer} to compute a transition formula for the outer loop. The recurrences found for the outer loop and their closed forms are shown to the right (again, omitting "uninteresting" recurrences). Note that our algorithm extracts these recurrences from φ_{outer} using only semantic operations: the fact that φ_{outer} is an abstraction of a looping computation is completely transparent to the analysis. Using the closed forms of the recurrences to the right, we compute the following transition formula for the outer loop:

Recurrence	Closed form
$q' = q + 1$	$q^{(k)} = q^{(0)} + k$
$r' = r - y$	$r^{(k)} = r^{(0)} - y^{(0)}k$

$$\varphi^{\circledast}_{\text{outer}} = \boxed{\exists k. k \geq 0 \land q' = q + k \land r' = r - ky \land id(\{x, y\})}$$
$$\equiv \boxed{q' \geq q \land r' = r - (q' - q)y \land id(\{x, y\})}$$

Finally, we compute a transition formula that approximates all executions which end at v_8 as follows:

$$\varphi_P = [\![\langle v^{\text{entry}}, v_1\rangle \cdot \langle v_1, v_2\rangle]\!] \odot \varphi^{\circledast}_{\text{outer}} \odot [\![\langle v_2, v_8\rangle]\!]$$
$$\equiv \boxed{q' \geq 0 \land r' = x - q'y \land r \leq y \land id(\{x, y\})}$$

This formula is strong enough to imply that $x' = q'y' + r'$. Thus CRA verifies that the assertion holds at v_8.

III. CRA Iteration Operator

In this section, we describe the iteration operator of compositional recurrence analysis. Suppose that we have a formula φ_{body} which approximates the behaviour of the body of a loop. The goal of the iteration operator is to compute a formula $\varphi^{\circledast}_{\text{body}}$ which represents the effect of zero or more executions of the loop body. CRA's iteration operator works by extracting recurrence relations from the formula φ_{body} and then computing closed forms for these relations. We present the iteration operator in three stages, based on the types of recurrence relations being considered: *simple recurrence equations*, *stratified recurrence equations*, and *linear recurrence (in)equations*. Simple and stratified recurrences are classical types of recurrence equations. Linear recurrence (in)equations generalize classical recurrence equations by considering inequalities over linear terms (rather than equalities over variables).

In the remainder of this section, fix a formula φ_{body} representing the body of a loop. We assume that φ_{body} is expressed in linear (rational and integer) arithmetic (our strategy for linearizing non-linear arithmetic is described in Section IV). Additionally, we will assume that φ_{body} is satisfiable; if φ_{body} is unsatisfiable, then we may take $\varphi^{\circledast}_{\text{body}}$ to be $\boxed{\bigwedge_{x \in \text{Var}} x' = x}$, which represents zero iterations of the loop.

A. Simple recurrence equations

We start by defining simple recurrences and induction variables.

Definition 1. *A* simple recurrence *for a formula φ_{body} is an equation of the form $x' = x + c$ (for a constant c) such that $\varphi_{body} \models x' = x + c$. If $x' = x + c$ is a simple recurrence for φ_{body}, we say that x* satisfies *the recurrence $x' = x + c$, and if there is some c such that x satisfies the recurrence $x' = x + c$,* *we say that x is an* induction variable.

The set of all simple recurrences which are satisfied by a transition formula φ_{body} can be detected as follows. First, query an SMT solver for a model m of φ_{body}. Then, for each variable x, ask an SMT solver if φ_{body} implies the equation $x' = x + [\![x' - x]\!]^m$ (where $[\![x' - x]\!]^m$ denotes the interpretation of the term $x' - x$ in the model m). This implication holds iff x is an induction variable.

If x is an induction variable that satisfies the recurrence $x' = x + c$, then the closed form for x is $x^{(k)} = x^{(0)} + kc$ (writing $x^{(k)}$ for the value that x obtains on the kth iteration of the loop).

B. Stratified recurrences equations

Consider the loop shown to the right. One can see that x satisfies a simple recurrence equation $\mathrm{x}' = \mathrm{x}+1$, and that y satisfies a (non-simple) recurrence equation $\mathrm{y}' = \mathrm{y} + \mathrm{x} + 1$. A closed form for y's recurrence is $\mathrm{y}^{(k)} = \mathrm{y}^{(0)} + \sum_{i=0}^{k-1}(\mathrm{x}^{(i)}+1)$. Since x is an induction variable, we have a closed form for x ($\mathrm{x}^{(i)} = \mathrm{x}^{(0)} + i$), which we may use to simplify y's recurrence:

```
while(x ≤ 10):
  x := x + 1
  y := y + x
  z := 2 * x
```

$$\begin{aligned} \mathrm{y}^{(k)} &= \mathrm{y}^{(0)} + \sum_{i=0}^{k-1}(\mathrm{x}^{(0)} + i + 1) \\ &= \mathrm{y}^{(0)} + k\mathrm{x}^{(0)} + k + \sum_{i=0}^{k-1} i \\ &= \mathrm{y}^{(0)} + k\mathrm{x}^{(0)} + \frac{k(k+1)}{2}. \end{aligned}$$

Stratified recurrence equations generalize this idea: starting from simple recurrence equations, solve more and more complex recurrences using the closed forms for simpler ones. Stratified recurrence equations are formalized as follows:

Definition 2. *The stratified recurrence equations (and stratified induction variables) of a formula φ_{body} are defined recursively as:*

- *A simple recurrence equation which is satisfied by φ_{body} is a stratified recurrence equation of φ_{body} (and a simple induction variable is a stratified induction variable) at stratum 0.*
- *Let $\boldsymbol{y}$ denote a vector of the stratified induction variables of strata $\leq N$. A recurrence of the form $x' = x + \boldsymbol{cy} + d$ (where $\boldsymbol{c}$ is a rational vector and d is a rational) is a stratified recurrence at stratum $N+1$ (and if x satisfies such a recurrence, it is a stratified induction variable at stratum $N+1$).*

We now present a method of generating all stratified induction variables from loop body formula. In order to reduce the number of SMT queries made, our method begins by constructing an intermediate object from the loop body formula, from which recurrence equations may be easily extracted. This intermediate object is the affine hull $\mathit{aff}(\varphi_{\mathsf{body}})$ of φ_{body}. The *affine hull* $\mathit{aff}(\varphi_{\mathsf{body}})$ of a formula φ_{body} is the smallest affine set which contains φ_{body}, represented as (the set of solutions to) a system of equations $A\boldsymbol{x} = \boldsymbol{b}$,

Algorithm 1: Affine hull.

```
Input  : Satisfiable formula $\varphi_{\mathsf{body}}$
Output: Affine hull of $\varphi_{\mathsf{body}}$
$H \leftarrow \bot$; $\psi \leftarrow \varphi_{\mathsf{body}}$;
while there exists a model $m$ of $\psi$ do
    $H' \leftarrow \bigwedge\{x = [\![x]\!]^m : x \in \mathsf{Var} \cup \mathsf{Var}'\}$;
    $H \leftarrow H \sqcup^{=} H'$ ;        /*Affine equality join*/
    $\psi \leftarrow \psi \wedge \neg H$;
end
return $H$
```

where $\boldsymbol{x} = \begin{bmatrix} x_1, & \cdots, & x_n, & x_1', & \cdots, & x_n' \end{bmatrix}$ is a vector of all variables in $\mathsf{Var} \cup \mathsf{Var}'$. Logically, $\mathit{aff}(\varphi_{\mathsf{body}})$ is a system of equations such that (1) $\varphi_{\mathsf{body}} \models \mathit{aff}(\varphi_{\mathsf{body}})$, and (2) every affine equation over $\mathsf{Var} \cup \mathsf{Var}'$ which is implied by φ_{body} is also implied by $\mathit{aff}(\varphi_{\mathsf{body}})$. The affine hull of a formula may be computed using Algorithm 1 (a specialization of the algorithm $\widehat{\alpha}$ in [26] to the abstract domain of affine equalities). For example, one representation of the affine hull of the example loop given at the beginning of this section is:

$$\begin{bmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \\ \mathrm{x}' \\ \mathrm{y}' \\ \mathrm{z}' \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

Our strategy for generating stratified recurrence equations from $\mathit{aff}(\varphi_{\mathsf{body}})$ is based on the following lemma. Combined with property (2) of $\mathit{aff}(\varphi_{\mathsf{body}})$ above, this lemma implies that any affine equation implied by φ_{body} can be expressed as a linear combination of the equations in $\mathit{aff}(\varphi_{\mathsf{body}})$. Thus, after computing the affine hull of φ_{body}, determining whether a given variable satisfies a stratified recurrence is simply a matter of solving a system of linear equations (e.g., using Gaussian elimination).

Lemma 3 ([30], Corollary 3.1d)**.** *Let A be a matrix, $\boldsymbol{b}$ be a column vector, $\boldsymbol{c}$ be a row vector, and d be a constant. Assume that the system $A\boldsymbol{x} = b$ has a solution. Then $A\boldsymbol{x} = \boldsymbol{b}$ implies $\boldsymbol{cx} = d$ iff there is a row vector $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda}A = \boldsymbol{c}$ and $\boldsymbol{\lambda b} = d$.*

An algorithm for finding all stratified induction variables of φ_{body} is as follows. Let us write $\mathit{aff}(\varphi_{\mathsf{body}})$ as $A\boldsymbol{x} = \boldsymbol{b}$. The algorithm operates by induction on strata. In the base case, we compute all simple induction variables using the method of the previous section. For the induction step, we suppose that we have detected all induction variables of strata $< N$. Then for each variable x_i which is *not* an induction variable of stratum $< N$, we ask if there exists $\boldsymbol{\lambda}$, $\boldsymbol{c}$, and d such that:

- $\boldsymbol{\lambda}A = \boldsymbol{c}$ and $\boldsymbol{\lambda b} = d$ (i.e, $\boldsymbol{cx} = d$ is implied by $\mathit{aff}(\varphi_{\mathsf{body}})$ and thus by φ_{body}).
- $c_i = 1$ and $c_{i+n} = -1$ (the coefficients of x_i and x_i' are 1 and -1, respectively).
- For all j such that $j \neq i + n$ and $n \leq j \leq 2n$, we have $c_j = 0$ (except for x_i', all coefficients of primed variables are 0).
- For all $j \neq i$ such that x_j is *not* an induction variable of stratum $< N$, we have $c_j = 0$ (except for x_i and

induction variables of strata $< N$, all coefficients for unprimed variables are 0).

This system of linear equations has a solution if and only if x_i is an induction variable of stratum N. The algorithm terminates when it has computed a recurrence equation for every variable, or when it fails to detect any induction variables at some stratum.

Next, we give a procedure computing closed forms of stratified induction variables. Again, this procedure operates by induction on strata. For the base case, we compute closed forms for simple induction variables, as in the previous section. For the induction step, we make use of the induction hypothesis that *the closed form for a stratified induction variable of stratum N is of the form*

$$x^{(k)} = p_0(k) + p_1(k)y_1^{(0)} + \cdots + p_n(k)y_n^{(0)}$$

where each y_i is a stratified induction variable of stratum $< N$ and each $p_i(k) \in \mathbb{Q}[k]$ is a polynomial of one variable with rational coefficients.

Suppose that we have a recurrence equation at stratum N: $x' = x + c_1y_1 + \cdots + c_ny_n + b$ (all $y_1, ..., y_n$ are of strata $< N$). Then we may write

$$x^{(k)} = x^{(0)} + \sum_{i=0}^{k-1} \big(c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b\big).$$

By our induction hypothesis, each $y_j^{(i)}$ can be written as a linear term with coefficients from $\mathbb{Q}[k]$. It follows that there exists $p_0, ..., p_n \in \mathbb{Q}[k]$ so that

$$c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b = p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)}$$

Thus we have

$$\begin{aligned} x^{(k)} &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)} \\ &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + y_1^{(0)} \sum_{i=0}^{k-1} p_1(i) + \cdots + y_n^{(0)} \sum_{i=0}^{k-1} p_n(i) \end{aligned}$$

The closed form of a summation of a polynomial of degree m is a polynomial of degree $m+1$. Polynomial curve fitting is an elementary algorithm which can be used to compute the closed form for the summation: compute the first $m+1$ terms of the summation and then solve the corresponding linear system of equations for the coefficients of the polynomial.

C. Linear recurrence (in)equations

```
while(x ≥ 0 ∧ y ≥ 0):
  if(*): x := x - 1
  else: y := y - 1
```

Recurrence equations (such as the simple and stratified varieties) yield precise descriptions of the dynamics of *some* variables, but what about variables which do not satisfy *any* recurrence equation? For example, consider that neither x nor y satisfy a recurrence equation in the loop to the right. However, they *do* satisfy recurrence *inequations*: $\mathrm{x} - 1 \leq \mathrm{x}'$, $\mathrm{x}' \leq \mathrm{x}$, $\mathrm{y} - 1 \leq \mathrm{y}'$, and $\mathrm{y}' \leq \mathrm{y}$. These inequations can be closed to yield $\mathrm{x}^{(0)} - k \leq \mathrm{x}^{(k)}$ and $\mathrm{x}^{(k)} \leq \mathrm{x}^{(0)}$, $y^{(0)} - k \leq y^{(k)}$, and $y^{(k)} \leq y^{(0)}$. We will now describe a method for extracting and solving linear recurrence (in)equations, which allows CRA to compute accurate approximations for loops that cannot be completely described by a system of recurrence equations.

Definition 4. *A* linear recurrence (in)equation *of a formula φ is an (in)equation which is implied by φ and which is of the form*

$$\boldsymbol{c}\boldsymbol{x}' \bowtie \boldsymbol{c}\boldsymbol{x} + \boldsymbol{b}\boldsymbol{y} + d$$

where $\bowtie \in \{<, \leq, =\}$, $\boldsymbol{x}$ is any vector of variables, $\boldsymbol{y}$ is a vector of stratified induction variables in φ_{body}, $\mathbf{c}$, $\boldsymbol{b}$ are constant vectors, and d is a constant.

Linear recurrence (in)equations generalize recurrence equations in two ways: first, they allow for *inequalities* rather than equalities. Second, they allow recurrences for *linear terms*, rather than just variables. For example, the linear recurrence equation $(\mathrm{x}' + \mathrm{y}') = (\mathrm{x} + \mathrm{y}) + 1$ is satisfied by the body of the loop above, which can be closed to yield $(\mathrm{x}^{(k)} + \mathrm{y}^{(k)}) = (\mathrm{x}^{(0)} + \mathrm{y}^{(0)}) + k$.

We now describe a method for detecting and solving linear recurrence (in)equations. Let $\mathsf{Var}^\sharp$ denote the set of variables which are *not* stratified induction variables of φ_{body}. Introduce a set of *difference variables* δ_x, one for each variable x in $\mathsf{Var}^\sharp$ (stratified induction variables are precisely described by recurrence equations, so they need not be approximated). Construct the strongest formula $\delta(\varphi_{\mathsf{body}})$ which is implied by φ_{body} (conjoined with definitional equalities for each difference variable) and which the only free variables are the difference variables and the stratified induction variables of φ_{body} as:

$$\delta(\varphi_{\mathsf{body}}) \triangleq \exists \mathsf{Var}' \cup \mathsf{Var}^\sharp . \varphi_{\mathsf{body}} \wedge \bigwedge \{\delta_x = x' - x : x \in \mathsf{Var}^\sharp\}$$

Next, use Algorithm 2 to compute the convex hull of $\delta(\varphi_{\mathsf{body}})$. Geometrically, the convex hull *hull*(ψ) of a formula ψ is the smallest convex polyhedron which contains ψ. Logically, it is a set of (in)equations such that (1) every (in)equation in *hull*(ψ) is implied by ψ, and (2) any affine (in)equation (over the free variables of ψ) which is implied by ψ is also implied by *hull*(ψ). For example, *hull*$(\delta(\varphi_{\mathsf{body}}))$ for the loop above is:

$$0 \leq \delta_x \wedge \delta_x \leq 1 \wedge 0 \leq \delta_y \wedge \delta_y \leq 1 \wedge \delta_x + \delta_y = 1$$

Algorithm 2: Convex hull.

```
Input  : Formula of the form $\exists X.\psi$, where $\psi$ is satisfiable and
         quantifier-free
Output: Convex hull of $\exists X.\psi$
$P \leftarrow \bot$;
while there exists a model $m$ of $\psi$ do
    Let $Q$ be a cube of the DNF of $\psi$ s.t. $m \models Q$;
    $Q \leftarrow$ project$(Q, X)$;   /*Polyhedral projection*/
    $P \leftarrow P \sqcup Q$;          /*Polyhedral join*/
    $\psi \leftarrow \psi \wedge \neg P$;
end
return $P$
```

Note that the only variables which appear in the (in)equations in *hull*$(\delta(\varphi_{\mathsf{body}}))$ are (stratified) induction variables and difference variables. Thus, any (in)equation in *hull*$(\delta(\varphi_{\mathsf{body}}))$ may be written as $\boldsymbol{c}\boldsymbol{\delta} \bowtie \boldsymbol{b}\boldsymbol{y} + d$ (where $\boldsymbol{\delta}$ is the vector of difference variables, $\boldsymbol{y}$ is the vector of stratified induction variables, $\boldsymbol{c}$ and $\boldsymbol{b}$ are constant vectors, and d is a constant). Recalling the definition of the difference variables, such an inequation may be rewritten as $\boldsymbol{c}(\boldsymbol{x'} - \boldsymbol{x}) \bowtie \boldsymbol{by} + d$ and thus as $\boldsymbol{cx'} \bowtie \boldsymbol{cx} + \boldsymbol{by} + d$, which matches the definition of linear recurrence (in)equations given in Definition 4. The closed form of this recurrence inequation is

$$\boldsymbol{cx}^{(k)} \bowtie \boldsymbol{cx}^{(0)} + \sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$$

where the closed form of the summation $\sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$ is computed as in Section III-B.

D. Loop guards

Typically, there is crucial information about the execution of a loop that cannot be captured by recurrence relations. For example, consider the loop in Section III-B. Supposing that the loop executes n times, it must be the case that $\mathtt{x}^{(k)} \leq 10$ for each $k < n$. Further, consider that the variable $\mathtt{z}$ is a function of the simple induction variable $\mathtt{x}$, and so $\mathtt{z}^{(k)}$ can be described precisely in terms of the pre-state variables (even though it does not itself satisfy any recurrence):

$$\mathtt{z}^{(k)} = \begin{cases} \mathtt{z}^{(0)} & \text{if } k = 0 \\ 2(\mathtt{x}^{(0)} + k + 1) & \text{otherwise.} \end{cases}$$

The question is: how can this type of information be recovered from a loop body formula?

We define the *guard* of a transition formula φ_{body} as follows:

$$\mathit{guard}(\varphi_{\mathsf{body}}) \triangleq \boxed{(\exists \mathsf{Var}.\varphi_{\mathsf{body}}) \land (\exists \mathsf{Var}'.\varphi_{\mathsf{body}})}$$

If φ_{body} is a loop body formula, then $\mathit{guard}(\varphi_{\mathsf{body}})$ is a formula which over-approximates the effect of executing *at least one* execution of the loop. Intuitively, $(\exists \mathsf{Var}'.\varphi_{\mathsf{body}})$ is a precondition that must hold before every iteration of the loop and $(\exists \mathsf{Var}.\varphi_{\mathsf{body}})$ is a post-condition of the loop that must hold after each iteration.

Consider again the example loop in Section III-B. The loop body formula is as follows:

$$\varphi_{\mathsf{body}} = \boxed{\mathtt{x} \leq 10 \land \mathtt{x}' = \mathtt{x} + 1 \land \mathtt{y}' = \mathtt{y} + \mathtt{x}' \land \mathtt{z}' = 2\mathtt{x}'}$$

Following the definition of *guard*, we have:

$$\begin{aligned} \mathit{guard}(\varphi_{\mathsf{body}}) &\triangleq \boxed{(\exists \mathtt{x}, \mathtt{y}, \mathtt{z}.\varphi_{\mathsf{body}}) \land (\exists \mathtt{x'}, \mathtt{y'}, \mathtt{z'}.\varphi_{\mathsf{body}})} \\ &\equiv \boxed{(\mathtt{x'} \leq 11 \land \mathtt{z'} = 2\mathtt{x'}) \land (\mathtt{x} \leq 10)}\,. \end{aligned}$$

Thus, $\mathit{guard}(\varphi_{\mathsf{body}})$ recovers the desired information about $\mathtt{x}$ and $\mathtt{z}$.

Since loop body formulas may be large, in practice it may be advantageous to simplify the guard formula by eliminating the quantifiers (as we did above). A second option, which is more efficient but less precise, is to over-approximate quantifier elimination. Two possibilities are to use Algorithm 2 to compute the convex hull of $\mathit{guard}(\varphi_{\mathsf{body}})$, or to use optimization modulo theories [18] to compute intervals for each pre- and post-state variable in φ_{body}.

E. Bringing it all together

We close this section by describing how the pieces defined in this section fit into the iteration operator of compositional recurrence analysis. Let $\mathit{CR}(\varphi_{\mathsf{body}})$ denote the set of closed linear recurrence (in)equations (including simple and stratified recurrence equations) satisfied by φ_{body}. Each such (in)equation is of the form $\boldsymbol{cx}^{(k)} \bowtie t$, where the free variables of t are drawn from $\{x^{(0)} : x \in \mathsf{Var}\}$ and a distinguished variable $k \notin \mathsf{Var}$ indicating the loop iteration. Letting $t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}]$ denote the term t with every variable of the form $x^{(0)}$ is replaced by the corresponding variable x, we define $\varphi^{+}_{\mathsf{body}}$ to be the following formula:

$$\boxed{\exists k. k \geq 1 \land \bigwedge\{\boldsymbol{cx'} \bowtie t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}] : \boldsymbol{cx}^{(k)} \bowtie t \in \mathit{CR}(\varphi_{\mathsf{body}})\}}\,.$$

Finally, the iteration operator of CRA is defined as:

$$\varphi^{\circledast}_{\mathsf{body}} \triangleq \boxed{(\varphi^{+}_{\mathsf{body}} \land \mathit{guard}(\varphi_{\mathsf{body}})) \lor \textstyle\bigwedge_{x \in \mathsf{Var}} x' = x}\,.$$

IV. Linearization

The iteration operator presented in the previous section operates under the assumption that loop body formulas are expressed in linear arithmetic. However, a program may contain non-linear instructions, and even if it does not, CRA's iteration operator may introduce non-linearity (consider Example 1, where the transition formula for the outer loop $\varphi^{\circledast}_{\mathsf{outer}}$ contains the non-linear proposition $r' = x - q'y$). A solution to this problem is to *linearize* non-linear formulas before passing them to the iteration operator.

Linearization is an operation that, given an (arbitrary) arithmetic formula φ, computes a formula $\mathit{lin}(\varphi)$ which over-approximates φ (i.e., $\varphi \models \mathit{lin}(\varphi)$), but which is expressed in linear arithmetic. There is generally no best approximation of a non-linear formula as a linear formula, so our method is (necessarily) heuristic.

We explain our linearization algorithm informally using an example. Consider the following non-linear formula (where w, x, y, z are integers):

$$\psi \triangleq 1 \leq w = x < y < 5 \land wy \leq z \leq xy$$

Our algorithm begins by normalizing ψ, separating it into a linear part and a set of non-linear equations (introducing Skolem constants as necessary). For example, the result of normalizing ψ is:

$$\big(1 \leq w = x < y < 5 \land \gamma_0 \leq z \leq \gamma_1\big) \land \big(\gamma_0 = wy \land \gamma_1 = xy\big)$$

The left conjunct is a linear over-approximation of ψ, but it is very imprecise: semantically equal (but syntactically distinct) non-linear terms become semantically *un*equal in the over-approximation, and all information about the magnitude of non-linear terms is lost. To increase precision of this approximation, we use two strengthening steps.

1) Replace the non-linear operations with uninterpreted function symbols and compute the affine hull of the resulting formula to infer affine equalities between Skolem constants (representing non-linear terms). For our example ψ, this step discovers the equality $\gamma_0 = \gamma_1$.
2) Compute concrete and symbolic intervals for non-linear terms. Consider $\gamma_1 = xy$ from our example ψ. First

compute concrete ($x \in [1,3]$ and $y \in [2,4]$) and symbolic ($x \in [x,x]$ and $y \in [y,y]$) intervals for the operands x and y, using symbolic optimization [18] to compute the concrete intervals. Obtain a concrete interval for xy ($xy \in [2,12]$) by multiplying the concrete intervals of its operands. Obtain symbolic intervals for xy ($xy \in [y,3y]$ and $xy \in [2x,4x]$) by multiplying the concrete interval for x by the symbolic interval for y and vice-versa. As a result of interval computation, we discover: $2 \leq \gamma_1 \leq 12 \land y \leq \gamma_1 \leq 3y \land 2x \leq \gamma_1 \leq 4x$.

We take *lin*(ψ) to be the initial coarse linear approximation of ψ conjoined with the facts discovered by the two strengthening steps.

We expect linearization to have applications outside of the context in which we presented it, particularly in program analysis, where over-approximation can be tolerated but non-linear terms cannot. Finding improved linearization heuristics is an interesting direction of future work.

V. Experiments

In this section, we present an experimental evaluation of CRA. We aim to support our hypothesis that, despite the fact that CRA may not take advantage of contextual information, it is competitive with leading verification techniques based on abstraction refinement.

We implemented CRA in a tool that analyzes C code (using the CIL [24] front-end). We use Z3 [9] to resolve SMT queries that result from applying the iteration operator and checking assertion violations. Polyhedra operations are passed to the New Polka library implemented in Apron [4]. The quantifier elimination algorithm from [22] is used to compute loop guards. The tool and benchmarks are available at http://www.cs.toronto.edu/~zkincaid/cra.

We tested two different configurations of CRA: one which is fully compositional (CRA) and does not take advantage of contextual information, and one (CRA+Oct) which uses an intra-procedural octagon analysis [19] to gain *some* contextual information, but which is otherwise compositional. We compare CRA's performance against the state-of-the-art invariant generation and verification tools CPAChecker (overall winner of the 2015 Software Verification Competition) and SeaHorn (winner of the loops category among tools which are sound for verification).

To evaluate the precision of CRA we used it to verify the correctness of a suite of 119 small loop benchmarks of varying difficulty. Our benchmark suite was drawn from the *loops* category of the 2015 Software Verification Competition (SVComp-15), as well as a set of *non-linear* benchmarks (Non-linear), such as the one in Figure 1. The results for the 81 safe, integer-only benchmarks from these suites are shown in Table I. The suite also contains 38 *unsafe* benchmarks: CRA and CRA+Oct have no false negatives on these benchmarks; CPAChecker has 3 and SeaHorn has 2.

CRA can prove safety for 27 more programs than CPAChecker and 3 fewer than SeaHorn, thus demonstrating that CRA is capable of generating competitively precise invariants. This holds despite the fact that CRA is a compositional analysis which does not use contextual information or employ abstraction refinement. The performance of CRA+Oct (compared to CRA) indicates that CRA can be combined with other invariant generation techniques to increase precision.

Test Suite	#Tests	CRA+Oct	CRA	CPAChecker	SeaHorn
SVComp-15	74	65	60	37	65
Non-linear	7	6	5	1	3
Total	81	71 (88%)	65 (80%)	38 (47%)	68 (85%)
Running time across all test suites					
Mean		1.9s	1.8s	42.4s	37.7s
Median		0.6s	0.6s	1.6s	0.2s

TABLE I
Experimental results.

VI. Related work

In this section, we compare compositional recurrence analysis to a sampling of related work on recurrence analysis and compositional invariant generation.

Recurrence analysis. The idea of using closed forms of recurrence relations to approximate loops has appeared in a number of other papers. Generally speaking, CRA differs from previous work in two essential ways: first, CRA uses an SMT solver to extract *semantic* recurrences, rather than *syntactic* recurrences. Second, CRA goes beyond exact recurrences (equations over variables) to *approximate recurrences* (inequations over linear terms).

Ammarguellat and Harrison give a method for detecting induction variables which is compositional in the sense that it uses closed forms for inner loops in order to recognize nested recurrences [1]. Maps from variables to symbolic terms (effectively a symbolic constant propagation domain) are used as an abstract domain (in contrast to CRA's use of arbitrary arithmetic formulas). Rodríguez-Carbonell and Kapur [27] and Kovács [14] developed techniques for discovering invariant polynomial equations based on solving recurrence relations. The classes of simple and stratified recurrence equations are subsumed by the ones considered in [27], [14], but admit a simpler algorithm for computing closed forms. Kroening et al. [15] present a technique for computing *under*-approximations of loops which uses polynomial curve-fitting to directly compute closed forms for recurrences rather than extracting recurrences and then solving them in a separate step.

Ancourt et al. give a technique for computing recurrence *in*equations for **while** loops with affine bodies [2]. As with the method for computing linear recurrence inequations presented in Section III-C, their method is based on difference variables and polyhedral projection. CRA generalizes this work by (1) extending it to arbitrary control flow and non-linear arithmetic, (2) integrating recurrence inequations with stratified induction variables, thereby allowing enabling the computation of invariant polynomial inequations. Ancourt et al. discuss a method for computing invariant polynomial inequations which is based on higher-order differences rather than stratified recurrence inequations.

Acceleration. *Acceleration* is a technique closely related to recurrence analysis that was pioneered in infinite-state model checking [6], [11], [3], and which has recently found use in program analysis [12], [17], [13]. Given a set of reachable states and an affine transformation describing the body of a loop, acceleration computes an *exact* post-image which describes the set of reachable states after executing any number of iterations of the loop (although there is recent work on *abstract acceleration* that computes over-approximate post-images [12], [13]). In contrast, CRA is *approximate* rather than exact, and computes transition formulas rather than post-images. A result of these two features is that CRA can be applied to arbitrary loops, while acceleration is classically limited to simple loops where the body consists of a sequence of assignment statements.

Compositional program analysis. Compositional program analysis has a long history. Particular examples are interprocedural analyses based on summarization [31] and elimination-style dataflow analyses (a good overview of which can be found in [28]). The following surveys recent work on compositional analysis for numerical invariants.

Kroening et al. [16] and Biallas et al. [5] present compositional analysis techniques based on predicate abstraction. In addition to predicate abstraction, there are a few papers which use numerical abstract domains for compositional analysis. These include an algorithm for detecting affine equalities between program variables [23], an algorithm for detecting polynomial equalities between program variables [8], a disjunctive polyhedra analysis which uses widening to compute loop summaries [25], and a method for automatically synthesizing transfer functions for template abstract domains using quantifier elimination [21]. Our abstract domain is the set of arbitrary arithmetic formula, which is more expressive than these domains, but which (as usual) incurs a potential price in performance. It would be interesting to apply abstractions to our formulas to improve the performance of our analysis.

Linearization. The linearization algorithm in Section IV was inspired by Miné's procedure for approximating non-linear abstract transformers [20]. Miné's procedure abstracts non-linear terms by linear terms with interval coefficients using the abstract value in the pre-state to derive intervals for variables. Our algorithm abstracts non-linear terms by sets of symbolic and concrete intervals, and applies to the more general setting of approximating arbitrary formulas.

VII. Conclusion

This paper describes compositional recurrence analysis, a fully compositional algorithm for generating numerical invariants of imperative programs. There are two main points to take away. The first is that it is possible to design a fully compositional analysis (which makes no use of contextual information) that is competitive with state-of-the-art verification techniques based on abstraction refinement. Second, recurrence-based program analysis may be extended to programs with arbitrary control flow by exploiting compositionality and SMT solving.

References

[1] Z. Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI*, pages 283–295, 1990.

[2] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1):3–16, Oct. 2010.

[3] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA*, pages 474–488. 2005.

[4] J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[5] S. Biallas, J. Brauer, A. King, and S. Kowalewski. Loop leaping with closures. In *SAS*, pages 214–230, 2012.

[6] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67. 1994.

[7] K. Chatterjee, R. Ibsen-Jensen, A. Pavlogiannis, and P. Goyal. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, pages 97–109, 2015.

[8] M. A. Colón. Approximating the algebraic relational semantics of imperative programs. In *SAS*, pages 296–311. 2004.

[9] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[10] A. Farzan and Z. Kincaid. An algebraic framework for compositional program analysis. *CoRR*, abs/1310.3481, 2013.

[11] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST TCS*, pages 145–156, 2002.

[12] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS*, pages 144–160. 2006.

[13] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, pages 529–540, 2014.

[14] L. Kovács. Reasoning algebraically about P-solvable loops. In *TACAS*, pages 249–264. 2008.

[15] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, pages 381–396. 2013.

[16] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger. Loop summarization using abstract transformers. In *ATVA*, pages 111–125. 2008.

[17] J. Leroux and G. Sutre. Accelerated data-flow analysis. In *SAS*, pages 184–199, 2007.

[18] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *POPL*, pages 607–618, 2014.

[19] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[20] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.

[21] D. Monniaux. Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151, 2009.

[22] D. Monniaux. Quantifier elimination by lazy model enumeration. In *CAV*, pages 585–599, 2010.

[23] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[25] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2007.

[26] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[27] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*, pages 266–273, 2004.

[28] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, Sept. 1986.

[29] B. Scholz and J. Blieberger. A new elimination-based data flow analysis framework using annotated decomposition trees. In *CC*, pages 202–217, 2007.

[30] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[31] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[32] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.

Pushing to the Top

Alexander Ivrii
IBM Research
alexi@il.ibm.com

Arie Gurfinkel
Software Engineering Institute
http://arieg.bitbucket.org

***Abstract*—IC3 is undoubtedly one of the most successful and important recent techniques for unbounded model checking. Understanding and improving IC3 has been a subject of a lot of recent research. In this regard, the most fundamental questions are how to choose Counterexamples to Induction (CTIs) and how to generalize them into (blocking) lemmas. Answers to both questions influence performance of the algorithm by directly affecting the quality of the lemmas learned. In this paper, we present a new IC3-based algorithm, called QUIP[1], that is designed to more aggressively propagate (or push) learned lemmas to obtain a safe inductive invariant faster. QUIP modifies the recursive blocking procedure of IC3 to prioritize pushing already discovered lemmas over learning of new ones. However, a naive implementation of this strategy floods the algorithm with too many useless lemmas. In QUIP, we solve this by extending IC3 with may-proof-obligations (corresponding to the negations of learned lemmas), and by using an under-approximation of reachable states (i.e., states that witness why a may-proof-obligation is satisfiable) to prune non-inductive lemmas. We have implemented QUIP on top of an industrial-strength implementation of IC3. The experimental evaluation on HWMCC benchmarks shows that the QUIP is a significant improvement (at least 2x in runtime and more properties solved) over IC3. Furthermore, the new reasoning capabilities of QUIP naturally lead to additional optimizations and new techniques that can lead to further improvements in the future.**

I. Introduction

IC3 [1] (also known as PDR [2]) is one of the most powerful algorithms for unbounded model checking of hardware. It is highly customizable [3], [4], and was successfully extended to more general domains [5]–[7].

In a nutshell, IC3 aims at constructing an inductive invariant proving the property. IC3 works by iteratively detecting states that lead to a property violation (in IC3-literature these states are also identified with counterexamples-to-induction and are called CTIs) and by learning lemmas that demonstrate why these CTIs cannot be reached from the initial states within a bounded number of steps. In this way, IC3 incrementally refines over-approximations F_k of states that are reachable in up to k steps, and terminates when one of the sets F_k represents a safe inductive invariant, or when a counterexample is found. The general scope of this paper is to further improve on the invariant generation capabilities of IC3. In what follows, we first analyze and discuss some of the choices made by IC3, and then present our approach.

One of the most important decisions made by IC3 pertains to the process of generalization of new lemmas at the time when they are discovered. Ideally, given a CTI, we would like to generate the strongest possible lemma that excludes this CTI and holds on all reachable states. However, obviously the set of all reachable states is not available. IC3 solves this problem by attempting to find the strongest lemma φ that is relatively inductive with respect to the appropriate over-approximation F_k. However, as F_k is neither an over-approximation nor an under-approximation of the set of all reachable states, φ can be either too strong or too weak. Being too strong means that φ excludes some of the reachable states and hence has no chance to be in the final inductive invariant, while being too weak means that φ prunes less unreachable states which degrades convergence. Another deficiency of IC3 is that once a lemma is added, it remains in the system, and there is no mechanism to detect and prune non-inductive lemmas, which translates to the wasted effort spent to propagate them.

An important optimization that already exists in IC3 consists of blocking the same CTI at many different levels. In our experience, IC3 often discovers many different lemmas to block the same CTI. On the one hand, different lemmas are in general of different quality and so having a variety of lemmas to choose from is beneficial. On the other hand, keeping several lemmas for the same CTI leads to a wasted effort of storing and pushing multiple lemmas when one would be enough. IC3 partially addresses this concern by pushing each lemma as far as possible when it is created (which implicitly blocks the corresponding CTIs at higher levels); however, it often happens that a lemma φ cannot be pushed forward because the appropriate over-approximation F_k is not strong enough. An alternative solution is to derive additional supporting lemmas that enable pushing φ forward, thus prioritizing the usage of a lemma already in the system, at the expense of finding additional lemmas required to support it. We believe that the new strategy is superior, as it should lead to an inductive invariant faster. Unfortunately, a naïve implementation forces the algorithm to start discovering new lemmas to support lemmas already in the system, and then new lemmas to support these supporting lemmas, and so on – flooding the algorithm with a huge number lemmas. To some extent the problem again boils down to lack of control on the usefulness of lemmas in the system, and the need to detect and prune the less useful ones.

In this paper, we present an improvement to the core of the IC3 algorithm. Motivated by the considerations above, we present an algorithm, called Quip, that combines the following innovations:

1) In Quip, we periodically detect the maximal inductive subset of all lemmas discovered so far. These lemmas are stored separately (in F_∞ in the terminology of PDR) and

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002441.

[1]QUIP is an acronym for “a QUest for an Inductive Proof”.

represent *good* lemmas – lemmas that should always remain in the system.

2) In Quip, we turn existing lemmas into additional proof obligations (and prioritize considering these proof obligations over regular proof obligations). Given $\varphi \in F_k \setminus F_{k+1}$, we add $\neg\varphi$ at level $k+1$ as a *may-proof-obligation*. In this way, we either succeed to push φ further (if $\neg\varphi$ is blocked), or find a witness trace that explains why φ cannot be pushed. Since $\neg\varphi$ does not necessarily represent a CTI, the witness trace does not necessarily lead to a property violation; however, it produces a concrete forward reachable state that is excluded by φ and hence which explains why φ is not inductive. In particular, φ is a *bad* lemma – lemma that has no chance to be in the inductive invariant.

3) In Quip we dynamically discover reachable states. These reachable states are used in several ways. First, each time that a new reachable state is discovered, it is used to mark as bad all lemmas in the system that exclude this state. Second, reachable states are used to automatically invalidate other may-proof-obligations or to discover a real counterexample. Finally, they are used to effectively enlarge the set of initial states and take the enlarged initial states into account when generalizing lemmas in the future.

Note that the ideas above are highly interdependent: without considering may-proof-obligations there is no way to produce interesting reachable states, while without considering reachable states there is no way to prune lemmas in the system. We also claim that Quip partially addresses the problems described in the beginning. By prioritizing may-proof-obligations over regular-proof-obligations, we try to reuse lemmas that already exist. In addition, as may-proof-obligations usually consist of significantly fewer literals than regular proof obligations, we effectively try to avoid detecting lemmas that are too weak, while by computing and using the set of reachable states for generalization, we also try to avoid detecting lemmas that are too strong. Finally, we can now classify lemmas as *good*, *bad* and *unknown*, and thus gain some control on which lemmas we want to propagate and keep, and which lemmas we do not. In what follows, we show how to integrate the presented ideas into an efficient algorithm and experimentally demonstrate that this represents a significant performance improvement over classical IC3.

We believe that our work extends the IC3 framework with additional reasoning capabilities: computing maximal inductive invariants, considering may-proof-obligations and forward reachable states. These naturally lead to other optimizations and new techniques that can lead to further improvement in the future. Last but not least, the new framework can be used with all other known IC3 optimizations and can be adapted to more general domains.

The rest of the paper is structured as follows. In Section II, we review the necessary background about IC3. We present the Quip algorithm at high-level in Section III, and the details of our implementation in Section IV. Our empirical evaluation is reported in Section VI. Finally, we conclude the paper with an overview of related work in Section VII, and conclusion in Section VIII.

II. Background

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathit{Init}(\mathcal{V})$ and $\mathit{Bad}(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $\mathit{Tr}(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that Init and Tr are in CNF.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$\mathit{Init}(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1})\right) \wedge \mathit{Bad}(\vec{v}_N) \qquad (1)$$

P is UNSAT (or SAFE) iff there exists a formula $\mathit{Inv}(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$\mathit{Init}(\vec{v}) \to \mathit{Inv}(\vec{v}) \qquad \mathit{Inv}(\vec{v}) \wedge \mathit{Tr}(\vec{v}, \vec{v}') \to \mathit{Inv}(\vec{v}') \qquad (2)$$
$$\mathit{Inv}(\vec{v}) \to \neg\mathit{Bad}(\vec{v}) \qquad (3)$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*.

We give a brief description of IC3 that highlights some steps, but omits many crucial optimizations. We refer the reader to [8] for an overview of available optimizations and their possible implementations.

IC3 maintains a set of clauses $F_0, F_1, \ldots$ called a *trace*. Each F_i in a trace is called a *frame*, each clause $c \in F_i$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to Init and that $\mathit{Init} \to \neg\mathit{Bad}$. IC3 maintains the following invariant:

$$F_i \to \neg\mathit{Bad} \qquad F_{i+1} \subseteq F_i \qquad F_i \wedge \mathit{Tr} \to F'_{i+1}$$

That is, each element of the trace is safe, the trace is syntactically monotone, and each F_{i+1} is inductive relative to F_i.

Additionally, IC3 maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle m, i\rangle$ where m is a cube over state variables and i is a *level*. At each point of the execution, it considers a proof obligation $\langle m, i\rangle$ with the smallest level i, and attempts to prove that m is reachable in i steps. If $i = 0$ then there is a real counterexample. Otherwise, it makes a *predecessor* query $SAT?(\neg m \wedge F_{i-1} \wedge \mathit{Tr} \wedge m')$ that checks whether a state in m can be reached from a state in F_{i-1}. If the result is satisfiable, it adds a predecessor of m as a new proof obligation at level $i-1$. If the result is unsatisfiable, it learns a new lemma φ, such that $\mathit{Init} \to \varphi$, $\varphi \to \neg m$ and $\varphi \wedge F_{i-1} \wedge \mathit{Tr} \to \varphi'$, and adds φ to all F_j,

```
Data: A cex queue $Q$, where $c \in Q$ is a pair $\langle m, i\rangle$, $m$ is a cube over state variables, and $i \in \mathbb{N}$. A level $N$. A trace $F_0, F_1, \ldots$
Initially: $Q = \emptyset$, $N = 0$, $F_0 = Init$, $\forall i > 0 \cdot F_i = \top$.
repeat
    Unreachable If there is an $i < N$ s.t. $F_{i+1} \subseteq F_i$
        return Unreachable.
    Reachable If there is an $m$ s.t. $\langle m, 0\rangle \in Q$
        return Reachable.
    Unfold If $F_N \to \neg Bad$, then set $N \leftarrow N + 1$, and $Q \leftarrow \emptyset$.
    Candidate If for some $m$, $m \to F_N \wedge Bad$, then add
        $\langle m, N\rangle$ to $Q$.
    Predecessor If $\langle m, i+1\rangle \in Q$ and there are $m_0$ and $m_1$ s.t.
        $m_1 \to m$, $m_0 \wedge m_1'$ is satisfiable, and
        $m_0 \wedge m_1' \to F_i \wedge Tr \wedge m'$, then add $\langle m_0, i\rangle$ to $Q$.
    NewLemma For $0 \leq i < N$: given $\langle m, i+1\rangle \in Q$ and a
        clause $\varphi$, such that $\varphi \to \neg m$,
        if $Init \to \varphi$, and $\varphi \wedge F_i \wedge Tr \to \varphi'$, then
        add $\varphi$ to $F_j$, for $j \leq i + 1$.
    ReQueue If $\langle m, i\rangle \in Q$, $0 < i < N$ and $F_{i-1} \wedge Tr \wedge m'$ is
        unsatisfiable, then add $\langle m, i+1\rangle$ to $Q$.
    Push For $0 \leq i < N$ and a clause $(\varphi \vee \psi) \in F_i$,
        if $\varphi \notin F_{i+1}$, $Init \to \varphi$ and $\varphi \wedge F_i \wedge Tr \to \varphi'$, then
        add $\varphi$ to $F_j$, for each $j \leq i + 1$.
until $\infty$;
```

Fig. 1. Rule-based description of IC3/PDR.

for $j \leq i$. In other words, the lemma φ represents a new over-approximation, and in particular demonstrates why the state m cannot be reached in up to i steps from the initial states. An important optimization is to re-enqueue $\langle m, i+1\rangle$ as a new proof obligation. If at any point of the execution $F_{i-1} = F_i$ and $F_i \to \neg Bad$, then F_i represents an inductive invariant establishing the correctness of the property.

Fig. 1 shows a rule-based overview of IC3 (adapted from [9]). Initially, Q is empty, $N = 0$ and $F_0 = Init$. Then, the rules in Fig. 1 are applied (possibly in a non-deterministic order) until either **Unreachable** or **Reachable** rule is applicable. **Unfold** extends the current trace and increases the level at which counterexample is searched. **Candidate** picks a bad state. **Predecessor** extends a counterexample from the queue by one step. **NewLemma** blocks a counterexample and adds a new lemma. **ReQueue** moves the counterexample to the next level. Finally, **Push** pushes a lemma to the next level, optionally generalizing it inductively. A typical schedule of the rules is to first apply all applicable rules except for **Push** and **Unfold**, followed by **Push** at all levels, then **Unfold**, and then repeating the cycle.

III. QUIP: THE ALGORITHM

In this section, we give a high-level description of Quip as a set of rules. This description shows various reasoning capabilities of Quip and establishes its correctness. A practical implementation of these rules is described in Section IV.

The main data structures and rules for Quip are shown in Fig. 2. Similarly to IC3, Quip manages proof obligations using a priority queue Q. However each proof obligation is a triple $\langle m, i, t\rangle$, where m and i are as in IC3, and t is the type of the proof-obligation: either *may* or *must*. Must proof-obligations represent cubes that must be blocked for the problem to be SAFE. May-proof-obligations represent cubes that we would like to block, but the problem might be SAFE even if they are not blocked. As in IC3, Quip maintains a trace of clauses $F_0, F_1, \ldots$. However, the number of the non-empty frames in the trace can be larger than the current depth N. Intuitively, a non-empty frame F_i with $i > N$ contains clauses that are inductive up to a yet-to-be-explored level i. Additionally, as in PDR, Quip maintains a set F_∞ of absolute invariants. The unique feature of Quip is that it also maintains a set REACH of states reachable from *Init*. In practice, we keep REACH as a set of cubes. We say that a lemma $\varphi \in F_i$ is *good* if it is also in F_∞, *bad* if it excludes a state in REACH, and *unknown* otherwise. Note that the categories above are exclusive – a lemma cannot be both *good* and *bad* at the same time.

We now describe the rules.

```
Data: A cex queue $Q$, where $c \in Q$ is a triple $\langle m, i, t\rangle$, $m$ is a cube over state variables, $i \in \mathbb{N}$, and $t \in \{may, must\}$. A level $N$. A trace $F_0, F_1, \ldots$ An invariant $F_\infty$. A set of reachable states REACH.
Initially: $Q = \emptyset$, $N = 0$, $\textsc{Reach} = F_0 = Init$, $\forall i \geq 1 \cdot F_i = \top$, $F_\infty = \top$.
Require: $Init \to \neg Bad$
repeat
    Unreachable If $F_\infty \to \neg Bad$
        return Unreachable.
    Reachable If $\langle m, i, must\rangle \in Q$, $m \cap (\vee\textsc{Reach}) \neq \emptyset$
        return Reachable.
    Unfold If $F_N \to \neg Bad$, then set $N \leftarrow N + 1$.
    Candidate If for some $m$, $m \to F_N \wedge Bad$, then add
        $\langle m, N, must\rangle$ to $Q$.
    Predecessor If $\langle m, i+1, t\rangle \in Q$ and there are
        $m_0$ and $m_1$ s.t. $m_1 \to m$, $m_0 \wedge m_1'$ is satisfiable,
        and $m_0 \wedge m_1' \to F_i \wedge Tr \wedge m'$,
        then add $\langle m_0, i, t\rangle$ to $Q$.
    NewLemma For $0 \leq i < N$: given $\langle m, i+1\rangle \in Q$ and a
        clause $\varphi$, such that $\varphi \to \neg m$,
        if $(\vee\textsc{Reach}) \to \varphi$, and $\varphi \wedge F_i \wedge Tr \to \varphi'$, then
        add $\varphi$ to $F_j$, for $j \leq i + 1$.
    ReQueue If $\langle m, i, must\rangle \in Q$, and $F_{i-1} \wedge Tr \wedge m'$ is
        unsatisfiable, then add $\langle m, i+1, must\rangle$ to $Q$.
    Push For $1 \leq i$ and a clause $(\varphi \vee \psi) \in F_i \setminus F_{i+1}$,
        if $(\vee\textsc{Reach}) \to \varphi$ and $\varphi \wedge F_i \wedge Tr \to \varphi'$, then
        add $\varphi$ to $F_j$, for each $j \leq i + 1$.
    MaxIndSubset If there is $i > N$ s.t. $F_{i+1} \subseteq F_i$, then
        $F_\infty \leftarrow F_i$, and $\forall j \geq i \cdot F_j \leftarrow F_\infty$.
    Successor If $\langle m, i+1, t\rangle \in Q$ and exist $m_0$, $m_1$ s.t.
        $m_0 \wedge m_1'$ are satisfiable and
        $m_0 \wedge m_1' \to (\vee\textsc{Reach}) \wedge Tr \wedge m'$, then
        add $m_1$ to REACH.
    MayEnqueue For $i \geq 1$ and a clause $\varphi \in F_i \setminus F_{i+1}$,
        if $(\vee\textsc{Reach}) \to \varphi$, add $\langle \neg\varphi, i+1, may\rangle \in Q$.
    ResetQ $Q \leftarrow \emptyset$.
    ResetReach REACH $\leftarrow Init$.
until $\infty$;
```

Fig. 2. Rule-based description of Quip.

a) Termination: The rule **Unreachable** in `Quip` is even simpler than the corresponding one in `IC3`: the verification problem is deduced to be SAFE as soon as $F_\infty \Rightarrow \neg Bad$. Note that this formulation makes it extremely easy to handle designs with multiple properties. The rule **Reachable** in `Quip` states that the problem is UNSAFE if a must-proof-obligation includes a reachable state; that is, either an initial state or a new reachable state explicitly found by the algorithm.

b) Generating proof obligations: The rules **Candidate**, **Predecessor**, and **ReQueue** are similar to the corresponding rules of `IC3`. The rule **MayEnqueue** is new. **Candidate** picks a bad state and adds it as a must-proof-obligation. **Predecessor** adds a CTI m_0 for an already existing proof obligation m_1 as a new proof obligation, at the level one lower than that of m_1. The type of m_0 is the same as that of m_1, and so in particular m_0 is a must-proof-obligation whenever m_1 is. **ReQueue** moves a blocked must-proof-obligation to the next level. We explicitly limit this rule to must-proof-obligations only, as may-proof-obligations are handled by **MayEnqueue**. **MayEnqueue** picks a lemma $\varphi \in F_i \setminus F_{i+1}$ that is not yet established at level $i+1$ and adds its negation $\neg\varphi$ as a may-proof-obligation at level $i+1$. The rule is only applicable if the status of φ is *unknown*. Note that it is actually sound to take *any* clause ψ such that $Init \Rightarrow \psi$ and *any* level k, and add $\neg\psi$ at level k as a may-proof-obligation. However, we do not currently use this level of generality.

c) Managing lemmas: **Unfold** increases the level at which a counterexample is searched. **NewLemma** adds a new lemma that blocks a proof obligation. We explicitly disallow learning *bad* lemmas. For correctness, it is possible to take any clause ψ such that $\psi \wedge F_i \wedge Tr \rightarrow \psi'$ and add ψ to all F_j for $j \leq i+1$. **Push** pushes a lemma to the next level, optionally generalizing it inductively. As before, we limit pushing and generalization to *unknown* lemmas only. An important distinction from `IC3` is that in `Quip` **Push** is not limited to the current working depth N of the algorithm.

d) Inductive invariant: **MaxIndSubset** checks whether for some i there is $F_i = F_{i+1}$. In this case, F_i is an inductive invariant which is used to enlarge F_∞. In the case $i < N$, F_∞ is a safe inductive invariant and an immediate application of **Unreachable** finishes verification. Otherwise, it discovers new *good* lemmas. Correctness follows from the fact that $F_i = F_{i+1}$ indirectly implies that $\forall j \geq i \cdot F_j \cap \text{REACH} = \emptyset$. That is, there are no *bad* lemmas in any F_j for $j \geq i$. Note that a maximal inductive subset of current lemmas is computed by applying **Push** as much as possible, followed by **MaxIndSubset**.

e) Reachability: **Successor** adds new reachable states. Given a proof obligation m that can be reached in one transition from an already known reachable state (either an initial state or an explicitly found reachable state), it computes a new reachable state m_1 that is included in m and adds it to REACH.

f) Restarts: The final set of rules deals with various reset mechanisms. The rule **ResetQ** allows to empty the proof obligation queue. This rule can be though of as a "local reset" that may guide `Quip` in a different search place by examining different predecessors and learning new lemmas. Note that in `IC3`, **ResetQ** is implicitly included in **Unfold**. That is, `IC3` resets its queue every time a new depth is explored. On the other hand, in `Quip` this choice is flexible. The rule **ResetReach** resets the reachable states. In practice, we may remove only some (less useful) reachable states when their number becomes too large.

```
$F_0 = Init$
if $F_0 \wedge Bad$ then
    return CEX
$N \leftarrow 0$; $F_\infty \leftarrow \top$; REACH $= F_0$
while (true) do
    $N \leftarrow N + 1$
    if Quip_RecBlockCube($Bad, N$) = CEX then
        return CEX
    if Quip_Push() = PROOF then
        return PROOF
```

Fig. 3. Main Procedure (`Quip_Main`).

IV. QUIP: IMPLEMENTATION

In this section, we describe our implementation of the `Quip` rules.

The set of all reachable states handled by `Quip` is of the form REACH $= Init \cup R$, where $Init$ are the initial states and R are the reachable states dynamically discovered by the algorithm. In our current implementation, R consists only of *concrete* states. That is, each element of R is a complete assignment to *all* state variables. Each state in R is stored as a Boolean array. The main functionality required from R is checking whether a given cube s intersects (or equivalently subsumes) one of the states r in R. In the pseudocode below, the function `Intersect`(R, s) returns NULL if $R \cap s = \emptyset$, and returns a state $r \in R$ with $r \cap s \neq \emptyset$ otherwise.

In what follows, we require an additional bookkeeping mechanism. If a proof obligation $\langle s, f, p\rangle$ is added as a predecessor of another proof obligation $\langle \tilde{s}, \tilde{f}, \tilde{p}\rangle$ using the **Predecessor** rule, then we say that $\tilde{s}$ is a *parent* of s. On the other hand, if $\langle s, f, p\rangle$ is added using either **Candidate** or **MayEnqueue**, then we say that s has no parent. Finally, the rule **ReQueue** keeps the parent information. In the pseudocode, we let `Parent`(s) be the parent of s or NULL if none. To some extent this bookkeeping is already supported by most `IC3` implementations as it is required for reconstructing counterexamples.

A. The Main Loop

Our implementation of `Quip` is structured similarly to `PDR` [2]. For completeness, the main loop is shown in Fig. 3. The algorithm first checks for a counterexamples at level 0 ($N = 0$), and then incrementally increases the working level N until either a counterexample or a safe inductive invariant is found.

B. Recursive Block Cube

The central procedure, `Quip_RecBlockCube`, that recursively blocks a bad state, is shown in Fig. 4. On the surface, it looks similar to `Pdr_RecursiveBlockCube` from [2], but there are many important differences.

```
Input: (Cube $s_0$, Frame $f_0$)
Data: Priority queue $Q$ of triples $\langle c, f, t\rangle$, where $c$ is a
      cube, $f$ is a level and $t \in \{may, must\}$
Data: Map Parent from a proof obligation to its
      parent proof obligation (NULL if none)
Data: Array $R$ containing concrete reachable states
1  Add($Q, \langle s_0, f_0, must\rangle$)
2  Parent($s_0$) ← NULL
3  while ¬Empty($Q$) do
4      $\langle s, f, p\rangle$ ← Pop($Q$)
5      if $f = 0$ then
6          if $p = must$ then
               // Found Real Counterexample
7              return CEX
8          else
               // New reachable state
9              Find $r$ such that $Init \wedge Tr \rightarrow r'$ and
               $r \cap$ Parent($s$) $\neq \emptyset$; Add $r$ to $R$
10             continue
11     if ($r_0$ ← Intersect($R, s$)) $\neq$ NULL then
12         if $p = must$ then
               // Found Real Counterexample
13             return CEX
14         else
15             if Parent($s$) $\neq$ NULL then
                   // New reachable state
16                 Find $r$ such that $r_0 \wedge Tr \rightarrow r'$ and
                   $r' \cap$ Parent($s$) $\neq \emptyset$; Add $r$ to $R$
17             continue
18     $\langle t, g\rangle$ ← Block($s, f$)
19     if $g \neq f - 1$ then
           // Cube $s$ is successfully blocked
              by lemma $\neg t$
           // Lemma $\neg t$ holds until frame $g$
20         if ($g < N$) then
21             if $t \neq s$ then
22                 Add($Q, \langle t, g+1, may\rangle$)
23                 Parent($t$) ← NULL
24             else
25                 Add($Q, \langle t, g+1, p\rangle$)
26     else
           // $t$ is a predecessor of $s$
27         Add($Q, \langle t, f-1, p\rangle$)
28         Add($Q, \langle s, f, p\rangle$)
29         Parent($t$) ← $s$
30 return BLOCKED
```

Fig. 4. Recursive Block Cube (Quip_RecBlockCube).

Quip_RecBlockCube accepts a must-proof-obligation $\langle s_0, f_0, must\rangle$, and either succeeds to strengthen the trace so that s_0 is blocked at level f_0, or finds a concrete reachable state r that intersects s_0 (hence r is a witness that $\neg s_0$ is not an invariant).

Quip_RecBlockCube starts by adding the proof-obligation $\langle s_0, f_0, must\rangle$, with no parent, to Q (lines 1–2) and proceeds to the main loop. In each iteration of the loop, it retrieves the proof-obligation from Q with the lowest-level, and in case of a tie, with the smaller number of literals. In particular, the proposed tie-breaking condition means that when Q contains two proof-obligations s_1 and s_2 at the lowest level, with $s_1 \subseteq s_2$, the algorithm will select s_1 first – hence attempting to derive the strongest possible lemma (that would automatically block s_2 as well). Let $\langle s, f, p\rangle$ be this proof obligation (line 4).

Let us first assume that the level f of the proof obligation is 0 (lines 5–10). In particular, $s \cap Init \neq \emptyset$ and Parent(s) $\neq$ NULL (according to our rules, only **Predecessor** can add proof-obligations at level 0). If this is a *must*-proof-obligation (lines 6–7), then our property is deduced to be UNSAFE and Quip_RecBlockCube terminates. Moreover, a concrete counterexample can be reconstructed using the parent information. If this is a may-proof-obligation (lines 8–10), then we compute a new reachable state r that is one-step reachable from *Init* and that intersects Parent(s). Note that such a state r must always exist since s is a CTI for Parent(s). In our implementation, we use a dedicated SAT-solver for all the successor queries, including reconstruction of real counterexamples. However, by also saving for each predecessor the assignment to inputs, this task can be reduced to simulation. The new state r is then added to R. In particular, when on some future iteration the algorithm returns to examining the proof-obligation corresponding to Parent(s), Parent(s) already intersects R.

Next, let us assume that s intersects a state $r_0 \in R$ (lines 11–17). If this is a must-proof-obligation, then our property is deduced to be UNSAFE and the procedure terminates. By additionally storing for each state in R its predecessor (not explicitly shown in the pseudocode), we can again reconstruct a real counterexample. If this is a may-proof-obligation and $Parent(s) \neq$ NULL, then as before we compute a reachable state r that is one-step reachable from r_0 and that intersects Parent(s) – and so when the algorithm returns to examining Parent(s) the condition Intersect(R,Parent(s)) $\neq \emptyset$ is activated and the reachable state is further propagated. In other words, as soon as a recursive predecessor of a may-proof-obligation intersects an initial or an already existing reachable state in R, a sequence of additional reachable states is discovered, including a reachable state that intersects a given proof-obligation.

The helper procedure Block (line 18), adapted from PDR [2], hides some less relevant details. In our implementation, $Block(s, f)$ first *syntactically* checks whether s is already blocked in the frame f – i.e., whether there exists a lemma $\neg t \in F_g$ with $t \subseteq s$ and $f \leq g$ (the case $g = \infty$ is also allowed). If so, then (t, g) is returned. Otherwise, $Block(s, f)$ checks whether the formula $F_{f-1} \wedge Tr \wedge s'$ is satisfiable. If it is, a predecessor t of s is extracted and suitably generalized. In this case, $(t, f-1)$ is returned. If the formula is unsatisfiable, then using an inductive generalization procedure, we obtain a lemma $\neg t$ which holds at least up to the frame f (and possibly up to a larger frame g, including ∞). In this case, *Block* adds the lemma $\neg t$ to F_g and returns (t, g). Note that lemma generalization takes the reachable states R into account, and ensures that new lemmas always include all of R.

Let us first consider the case that the cube s was successfully blocked (lines 20–25), i.e., Block returns a lemma $\neg t \in F_g$ with $t \subseteq s$ and $f \leq g$. An important optimization in IC3 consists of reinserting the proof-obligation s at the level $g+1$, forcing the algorithm to block s in all higher frames as well. The unique feature of Quip is that $\neg t$ is inserted into

```
  for $k = 1, \ldots$ do
      for all lemmas $c \in F_k \setminus F_{k+1}$ do
          // Rule Push
1         if $\neg bad(c)$ then
2             if $F_k \wedge c \wedge Tr \Rightarrow c'$ then
3                 $F_{k+1} \leftarrow F_{k+1} \cup \{c\}$
      if $F_k \setminus F_{k+1} = \emptyset$ then
          // Rule MaxIndSubset
4         $F_\infty \leftarrow F_k$
5         for $j = k+1, \ldots$ do
6             $F_j \leftarrow F_\infty$
7         break;
  if $F_\infty \Rightarrow \neg Bad$ then
      // Found Safe Inductive Invariant
8     return PROOF
  return UNKNOWN
```

Fig. 5. Pushing lemmas (`Quip_Push`).

Q at the level $g+1$ instead of s. This forces the algorithm to concentrate on further pushing existing lemma t rather than discovering new lemmas to block s at a higher level. However, $\neg t$ can be only added as a may-obligation (with the only exception being that $s = t$ and s is a must-obligation). Finally, note that when $t \neq s$, the cube t has no parent, otherwise we keep the previous parent of s.

In the case that a predecessor t of s is found (lines 16–29), just as in `IC3`, `Quip` returns $\langle s, f, p\rangle$ to Q, as well as inserts a new proof obligation $\langle t, f-1, p\rangle$ with the same type of a proof obligation as that of s. The parent of t is set to s.

C. Pushing

Fig. 5 describes our pushing procedure `Quip_Push`. For each lemma c, we keep a Boolean flag $bad(c)$ that represents whether c is known to be *bad* (that is, whether c excludes some states in REACH). We say that a lemma is *unknown* if $bad(c) =$ FALSE and $c \notin F_\infty$. Each time that a new reachable state r is added in `Quip_RecBlockCube`, we check it against every *unknown* lemma in the system and mark as *bad* those lemmas that exclude r. Just as in `IC3`, in practice the sets F_i are delta-encoded: for any i, j, $F_i \cap F_j = \emptyset$. However, for this presentation, we are using the full sets F_i as defined in the introduction. The pushing stage proceeds as in `IC3`, with the following exceptions. First, *bad* lemmas are not pushed. This has two positive effects. The primary effect is conserving resources by not propagating lemmas that have no chance to be in the final invariant. A secondary effect is that as the new lemmas are learned, they are less dependent on the currently known bad lemmas. Second, the lemmas are pushed arbitrarily far past the current depth N. In particular, in the last iteration of the outer *for*-loop, all lemmas at level k are pushed to the next frame. In this case, the *if*-condition on line 3 is true, and all lemmas of F_{k+1} are added to F_∞. It is easy to see that after `Quip_Push`, F_∞ contains the maximal inductive subset of all lemmas in the system. If F_∞ implies $\neg Bad$, i.e., F_∞ represents a safe inductive invariant, then `Quip_Push` returns PROOF.

D. Managing reachable states

Efficiently handling reachable states poses additional challenges. Currently we represent reachable states explicitly, and as their number grows large, the time taken by `Intersect` and the memory required for their storage become significant. However, our experience shows that many of the reachable states can be removed without much sacrificing the number of may-proof-obligations pruned or the quality of lemmas discovered, and that the newly discovered states are more likely to be useful in the immediate future. Thus our solution mimics the clause deletion strategy as used in a SAT solver: for each reachable state we keep its *activity* representing how many times the state was a witness for intersection, and we periodically decay this activity and aggressively delete the less active states. Furthermore, as in our current implementation most of the time on managing lemmas is spent during the inductive generalization (making sure that a learned lemma includes all the states in REACH), we have found it further beneficial to consider even fewer reachable states during the generalization.

It may also be possible to compute partial states directly from the **Successor** query, or to represent reachable states symbolically by computing minimal DNF representation of R. An alternative way to take reachable states into account is to include them directly in F_0. Another optimization is to check whether a given may-obligation is one-step reachable from R. However, we have found both of these difficult to implement efficiently. Finally, it might also be useful to push reachable states forward more aggressively, for example, by running a simulation from already known reachable states.

V. ALTERNATIVES

In this section, we present two alternative implementations of `Quip`, which illustrate the variety of possibilities offered by our framework. Unfortunately, for the reasons discussed below, both of these variants do not perform consistently. We sketch how they could be improved in the future.

A. Reset-free approach

Both `IC3` and `Quip` as described previously implicitly reset the queue of proof obligations each time that a new depth is explored. An interesting alternative in `Quip` is as follows. (1) Allow to enqueue proof-obligations at any level (and not only up to N) by removing the *if*-condition on line 20 of `Quip_RecBlockCube`. (2) Check whether $F_k = F_{k+1}$ each time that a lemma is successfully pushed from F_k to a higher-frame (or simply each time that a proof obligation at level $k+1$ is successfully blocked); if $F_k = F_{k+1}$, then grow the set F_∞ to F_k, and check the termination condition $F_\infty \Rightarrow \neg Bad$. (3) Replace `Quip_Main` by a single call to `Quip_RecBlockCube`$(Bad, 1)$. In this way, the negation of *every* unknown lemma in the system is always present as a proof obligation at the corresponding frame and the external pushing stage can be avoided altogether. This alternative procedure takes to the extreme the idea of pushing every lemma in the system as far as possible, and arguably results in an even simpler overall algorithm. However, a preliminary experimental evaluation shows that this scheme performs worse in practice. One possible explanation is that

TABLE I. SUMMARY OF EXPERIMENTAL RESULTS

	UNSAFE solved	UNSAFE time	SAFE solved	SAFE time
IC3	22 (2)	52,302	76 (7)	137,244
Quip	32 (12)	20,302	99 (30)	69,590

Experimental results on the instances solved by either IC3 or Quip separated into unsafe and safe instances. The numbers in parentheses represent the unique solves. The times are in seconds.

periodically resetting the proof obligation queue keeps proof obligations more focused to proving the property, while the procedure above handles the "main" lemmas and the "supporting" lemmas (and the supporting lemmas for the supporting lemmas, and so on) equally. A possible solution would be to define some additional criteria for proof obligations reflecting their expected usefulness, and to take these into account when choosing the next proof obligation.

B. Garbage-collecting bad lemmas

We can use the classification of all lemmas into *good*, *bad* and *unknown* to periodically remove all the bad lemmas from the system. However, as bad lemmas may be supporting other unknown lemmas, we cannot simply remove *bad* lemmas from their corresponding frames. Instead, we can keep all the *good* lemmas in F_∞, put all the *unknown* lemmas into F_1, and use **Push** to push the *unknown* lemmas as far as possible. We have found that during this pushing stage it is important to preserve the set of lemmas as much as possible, which requires to disable both the additional generalizing capability of pushing and the built-in subsumption mechanism for storing lemmas. Note that we might also need to decrease the current bound N at which the property is proved. A preliminary experimental evaluation shows that this variant usually allows Quip to converge at a smaller depth, and in some cases leads to a significant speedup. However, it is also true that applying "garbage collection" too aggressively on average leads to a significant performance degradation, and an ongoing work is to find the a good heuristic for when to apply it and how to properly combine it with the resetting of reachable states described in Section IV.

VI. EXPERIMENTS

In this section, we present our experimental results[2]. We compare Quip with a custom variant of IC3, as implemented in the IBM formal verification tool *Rulebase-Sixthsense* [10]. All experiments were performed on a 2.13Ghz Linux-based machine with Intel Xeon E7-4830 processor, 16GB of RAM, and one hour time limit. We have used 300 single property designs from the HWMCC'13 and HWMCC'14 benchmark sets. These are obtained by removing duplicates and instances solved using standard logic synthesis (similar to the &dc2 command in ABC [11]).

The overall results are shown in Table I. The columns "UNSAFE solved" and "SAFE solved" show that number of unsafe and safe instances, respectively, solved by either IC3 or Quip. The numbers in parentheses represent the number of instances not solved by the other configuration. The columns "UNSAFE time" and "SAFE time" represent the cumulative time in seconds for unsafe and safe properties, respectively.

[2]See http://arieg.bitbucket.org/quip for more details.

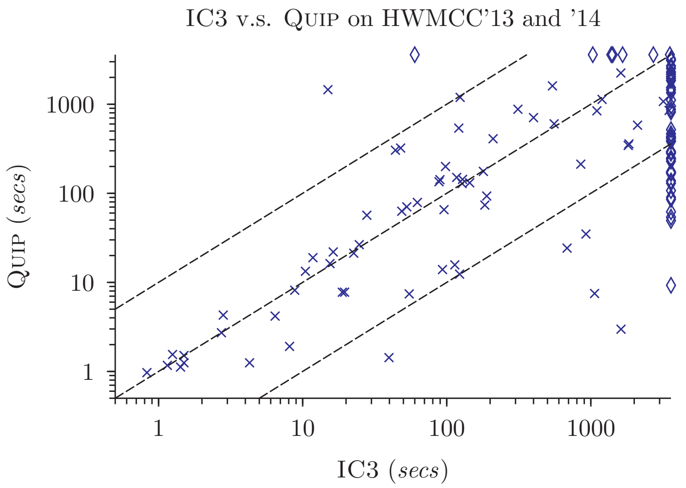


Fig. 6. Run-time comparison between IC3 and Quip. Points below the diagonal are in favor of Quip. The scale is logarithmic. Diagonals mark an order of magnitude. Timeout is 3,600 seconds.

TABLE II. DATA ON REACHABLE STATES DISCOVERED BY QUIP

# reach. states	0–10	11 – 100	101 – 1K	1K – 10K	10K – 50K
# instances	42	19	29	32	9
# unique solved	1	1	10	22	8

According to our experiments, either IC3 or Quip was successful on 34 unsafe instances and 106 safe instances. In the remaining 160 instances both IC3 and Quip timed out. We can see that Quip is clearly superior to IC3 on both safe and unsafe problems, solving more properties and running in roughly half of the time.

A more detailed comparison between IC3 and Quip is shown on the scatter plot in Fig. 6. Only the 140 instances solved by at least one tool are shown. For instances solved by both, the run time is similar, with an advantage for Quip Sometimes, the advantage is over an order of magnitude. Quip shines on harder instances and is able to solve significantly more of them than IC3.

Finally, we give some intuition on the total number of reachable states typically discovered by Quip and whether these states are useful for verification. Table II contains the data for the 131 instances solved by Quip, including the 42 instances not solved by IC3. In the table, the row "#reach. states" represents a range, the row "#instances" specifies the number of instances solved by Quip with the total number of reachable states in this range, and the row "#unique solves" further specifies the number of instances solved uniquely by Quip. For example, the third column means that 29 instances solved by Quip required between 101 and 10,000 reachable states, and 10 out of 29 are not solved by IC3. We draw two conclusions. First, even though we use concrete reachable states (i.e., complete assignment to all state variables), relatively few states had to be discovered. Second, the advantage of Quip over IC3 is especially pronounced as the number of learned reachable states increases. For example, from the 61 instances where Quip required less than 100 reachable states, only 2 are not solved by IC3. However, from the set of 9 instances where Quip finds more than $10,001$ reachable states, 8 (i.e., all but 1) are not solved by IC3.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

VII. Related Work

Computing Maximal Inductive Subset (MIS) is a well-known problem in both hardware and software verification (e.g., [12], [13]). Applying MIS to enlarge F_∞ in IC3/PDR is already suggested in [2], but it was not effective since the cost of computing an MIS out-weighted the gains. In Quip, the MIS computation is amortized by not limiting Quip/**Push** rule to the current bound N (for comparison, see IC3/**Push** in Fig. 1) and by discovering MIS opportunistically using Quip/**MaxIndSubset**. Thus, even if the MIS computation is unsuccessful and no new lemmas are added to F_∞, the trace is strengthened for the future runs of the algorithm. In our experience, extending IC3 in this way is beneficial regardless of the other Quip rules.

Blocking states that are not necessarily backward reachable from an error state and separating proof obligations into *may* and *must* was proposed in the context of IC3-based abstraction refinement [4]. The idea is also implicitly present in computation of minimal inductive clauses [3] and predicate-abstraction-based extensions of IC3 to software [14], [15]. In contrast to the above algorithms, Quip seamlessly integrates *must* and *may* reasoning into one algorithmic procedure without any specialized refinement steps. More significantly, Quip uses the reachable states that witness a failure of a *may*-proof obligation to improve future lemma generalization. Thus, both proving and *disproving* a may-proof-obligation is beneficial to the overall algorithm.

Extracting forward reachable states from spurious counterexamples also appears in NewItp [16] as *states to refinement* in the context of interpolation-based model checking. Similar to Quip, these states are used to guide future interpolants to avoid reachable states. In essence, Quip computes both an over-approximation (lemmas) and under-approximation (Reach) of reachable states. This can be seen as an extension of over- and under-approximations used in Spacer [6] from modular to monolithic-proofs. The key difference is that, Spacer under-approximates summaries of procedures and not states reachable from an initial state.

Interestingly, CTI's of Reverse IC3 [17] – a dual variant of IC3 that recursively enumerates states reachable from *Init* and that learns an over-approximation of states backwards reachable from *Bad* – are forward reachable states. Thus, it might be possible to combine IC3 and Reverse IC3 into an algorithm that computes both forward and backward reachable states and their over-approximations, somewhat akin to DAR [18]. Although DAR is restricted only to over-approximations.

VIII. Conclusions

In this paper, we present an improvement to the core of the IC3 algorithm. We propose an approach, called Quip, that is designed to propagate learned lemmas more aggressively, and whose implementation seamlessly integrates must and may proof-obligations and forward reachable states. The experimental results show that a naïve implementation of Quip significantly outperforms a highly-tuned implementation of IC3/PDR. We believe that the new reasoning capabilities introduced in Quip open up many opportunities for further improvements to SAT-based automated verification.

References

[1] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[2] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[3] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 157–164.

[4] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and sat-based reachability in hardware model checking," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, 2012, pp. 173–181.

[5] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, pp. 157–171.

[6] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 17–34.

[7] A. Cimatti and A. Griggio, "Software model checking via IC3," in *CAV*, 2012, pp. 277–293.

[8] A. Griggio and M. Roveri, "Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking," in *Proceedings of International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'14)*, Lausanne, Switzerland, October 2014.

[9] N. Bjørner and A. Gurfinkel, "Property directed polyhedral abstraction," in *VMCAI*, 2015, pp. 263–281.

[10] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[11] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.

[12] C. A. J. van Eijk, "Sequential Equivalence Checking without State Space Traversal," in *1998 Design, Automation and Test in Europe (DATE '98), February 23-26, 1998, Le Palais des Congrès de Paris, Paris, France*, 1998, pp. 618–623.

[13] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, 2001, pp. 500–517.

[14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *TACAS*, 2014, pp. 46–61.

[15] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to induction-guided abstraction-refinement (CTIGAR)," in *CAV*, 2014, pp. 831–848.

[16] C. Wu, C. Wu, C. Lai, and C. R. Haung, "A counterexample-guided interpolant generation algorithm for sat-based model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1846–1858, 2014.

[17] F. Somenzi and A. R. Bradley, "IC3: where monolithic and incremental meet," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, 2011, pp. 3–8.

[18] Y. Vizel, O. Grumberg, and S. Shoham, "Intertwined forward-backward reachability analysis using interpolants," in *TACAS*, 2013, pp. 308–323.

Skolem Functions for Factored Formulas

Ajith K. John, HBNI, BARC, India
Shetal Shah, IIT Bombay
Supratik Chakraborty, IIT Bombay
Ashutosh Trivedi, IIT Bombay
S. Akshay, IIT Bombay

***Abstract*—Given a propositional formula $F(x, y)$, a Skolem function for x is a function $\psi(y)$, such that substituting $\psi(y)$ for x in F gives a formula semantically equivalent to $\exists x\ F$. Automatically generating Skolem functions is of significant interest in several applications including certified QBF solving, finding strategies of players in games, synthesising circuits and bit-vector programs from specifications, disjunctive decomposition of sequential circuits etc. In many such applications, F is given as a conjunction of factors, each of which depends on a small subset of variables. Existing algorithms for Skolem function generation ignore any such factored form and treat F as a monolithic function. This presents scalability hurdles in medium to large problem instances. In this paper, we argue that exploiting the factored form of F can give significant performance improvements in practice when computing Skolem functions. We present a new CEGAR style algorithm for generating Skolem functions from factored propositional formulas. In contrast to earlier work, our algorithm neither requires a proof of QBF satisfiability nor uses composition of monolithic conjunctions of factors. We show experimentally that our algorithm generates smaller Skolem functions and outperforms state-of-the-art approaches on several large benchmarks.**

I. Introduction

Skolem functions, introduced by Thoraf Skolem in the 1920s, occupy a central role in mathematical logic. Formally, let $F(x, y)$ be a first-order logic formula, and let $\mathrm{dom}(x)$ and $\mathrm{dom}(y)$ denote the domains of x and y respectively. A *Skolem function* for x in F is a function $\psi : \mathrm{dom}(y) \to \mathrm{dom}(x)$ such that substituting $\psi(y)$ for x in F yields a formula semantically equivalent to $\exists x F(x, y)$, i.e. $F(\psi(y), y) \equiv \exists x F(x, y)$. In this paper, we focus on the case where the formula F is propositional and given as a conjunction of factors. Classically, Skolem functions have been used in proving theorems in logic. More recently, with the advent of fast SAT/SMT solvers, it has been shown that several practically relevant problems can be encoded as quantified formulas, and can be solved by constructing *realizers* of quantified variables. We identify these realizers as specific instances of Skolem functions, and focus on algorithms for constructing them in this paper.

We begin by listing some applications that illustrate the utility of constructing instances of Skolem functions in practice.

1) *Quantifier elimination.* Given a quantified formula $Qx\ F(x, y)$, where $Q \in \{\exists, \forall\}$, the quantifier elimination problem requires us to find a quantifier-free formula that is semantically equivalent to $Qx\ F(x, y)$. Quantifier elimination has important applications in diverse areas (see, e.g. [7], [15], [2] for a sampling). It follows from the definition of Skolem function that eliminating the quantifier from $\exists x F(x, y)$ can be achieved by substituting x with a Skolem function for x. Since $\forall x F(x, y)$ can be written as $\neg\exists x \neg F(x, y)$, the same idea applies in this case too. In fact, the process can be repeated in principle to eliminate quantifiers from a formula with arbitrary quantifier prefix.
2) *Controller Synthesis and Games.* Control-program synthesis in the Ramadge-Wonham [13] framework reduces to games between two players—environment and the controller—such that the optimal strategy of the controller corresponds to an optimal control program. The optimal (or winning) strategy of the controller corresponds to choosing values of variables controlled by it such that regardless of the way the environment fixes its variables, the resulting play satisfies the controller's objective. If the rules of the game are encoded as a propositional formula and if the strategy space for both players is finite, the optimal strategy of the controller corresponds to finding Skolem functions of variables controlled by it. In fact, for a number of two-player games—such as reachability games and safety games [2], tic-tac-toe [5] and chess-like games [3], [2]—the problem of deciding a winner can be reduced to checking satisfiability of a quantified Boolean formula (QBF), and the problem of finding winning or best-effort strategy reduces to Skolem function generation.
3) *Graph Decomposition.* Skolem functions can be used to compute disjunctive decompositions of implicitly specified state transition graphs of sequential circuits [17]. The disjunctive decomposition problem asks the following question: Given a sequential circuit, derive "component" sequential circuits, each of which has the same state space as the original circuit, but only a subset of transitions going out of every state. The components should be such that the complete set of state transitions of the original circuit is the union of the sets of state transitions of the components. Disjunctive decompositions have been shown to be useful in efficient reachability analysis [16].

There are several other practical applications where Skolem functions find use; see, e.g. [12], for a discussion. Hence, there is a growing need for practically efficient and scalable approaches for generating instances of Skolem functions. Large and complex representations of the formula F in $\exists x\ F$ often present scalability hurdles in generating Skolem functions in practice. Interestingly, for several problem instances, the specification of F is available in a *factored* form, i.e., as a conjunction of simpler sub-formulas, each of which depends

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

on a subset of variables appearing in F. Unfortunately, unlike in the case of disjunction, existential quantification does not distribute over conjunction of sub-formulas. Existing algorithms therefore ignore any factored form of F and treat the conjunction of factors as a single monolithic function. We show in this paper that exploiting the factored form can help significantly when generating Skolem functions.

Our main technical contribution is a SAT-based Counter-Example Guided Abstraction-Refinement (CEGAR) algorithm for generating Skolem functions from factored formulas. Unlike competing approaches, our algorithm exploits the factored representation of a formula and leverages advances made in SAT-solving technology. The factored representation is used to arrive at an initial abstraction of Skolem functions, while a SAT-solver is used as an oracle to identify counter-examples that are used to refine the Skolem functions until no counter-examples exist. We present a detailed experimental evaluation of our algorithm vis-a-vis state-of-the-art algorithms [7], [12] over a large class of benchmarks. We show that on several large problem instances, we outperform competing algorithms. Proofs that are omitted can be found in the long version at [9].

Related Work. We are not aware of other techniques for Skolem function generation that exploit the factored form of a formula. Earlier work on Skolem function generation broadly fall in one of four categories. The first category includes techniques that extract Skolem functions from a proof of validity of $\exists X\ F(X,Y)$ [12], [8], [4], [10]. In problem instances where $\exists X\ F(X,Y)$ is valid (and this forms an important sub-class of problems), these techniques can usually find succinct Skolem functions if there exists a short proof of validity. However, in several other important classes of problems, the formula $\exists X\ F(X,Y)$ does not evaluate to true for all values of Y, and techniques in the first category cannot be applied. The second category includes techniques that use templates for candidate Skolem functions [15]. These techniques are effective only when the set of candidate Skolem functions is known and small. While this is a reasonable assumption in some domains [15], it is not in most other domains. BDD-based techniques [14] are yet another way to compute Skolem functions. Unfortunately, these techniques are known not to scale well, unless custom-crafted variable orders are used. The last category includes techniques that use cofactors to obtain Skolem functions [7], [17]. These techniques do not exploit the factored representation of a formula and, as we show experimentally, do not scale well to large problem instances.

II. Preliminaries

We use lower case letters (possibly with subscripts) to denote propositional variables, and upper case letters to denote sequences of such variables. We use 0 and 1 to denote the propositional constants false and true, respectively. Let $F(X,Y)$ be a propositional formula, where X and Y denote the sequences of variables $(x_1,\ldots,x_n)$ and $(y_1,\ldots,y_m)$, respectively. We are interested in problem instances where $F(X,Y)$ is given as a conjunction of factors $f^1(X_1,Y_1),\ldots,f^r(X_r,Y_r)$, where each X_i (resp., Y_i) is a possibly empty sub-sequence of X (resp., Y). For notational convenience, we use F and $\bigwedge_{j=1}^{r} f^j$ interchangeably throughout this paper. The set of variables in F is called the *support* of F, and is denoted $\mathsf{Supp}(F)$. Given a propositional formula $F(X)$ and a propositional function $\Psi(X)$, we use $F[x_i/\Psi(X)]$, or simply $F[x_i/\Psi]$, to denote the formula obtained by substituting every occurrence of the variable x_i in F with $\Psi(X)$. Since the notions of formulas and functions coincide in propositional logic, the above is also conventionally called *function composition*. If X is a sequence of variables and x_i is a variable, we use $X \setminus x_i$ to denote the sub-sequence of X obtained by removing x_i (if present) from X. Abusing notation, we use X to also denote the set of elements in X, when there is no confusion. A *valuation* or *assignment* π of X is a mapping $\pi : X \to \{0,1\}$.

Definition 1. *Given a propositional formula $F(X,Y)$ and a variable $x_i \in X$, a* Skolem function *for x_i in $F(X,Y)$ is a function $\psi(X \setminus x_i, Y)$ such that $\exists x_i\ F \equiv F[x_i/\psi]$.*

A Skolem function for x_i in F need not be unique. The following proposition, which effectively follows from [7], [17], characterizes the space of all Skolem functions for x_i in F.

Proposition 1. *A function $\psi(X \setminus x_i, Y)$ is a Skolem function for x_i in $F(X,Y)$ iff $F[x_i/1] \wedge \neg F[x_i/0] \Rightarrow \psi$ and $\psi \Rightarrow F[x_i/1] \vee \neg F[x_i/0]$.*

The function $F[x_i/0]$ (resp., $F[x_i/1]$) is called the *positive* (resp., *negative*) *cofactor* of F with respect to x_i, and plays a central role in the study of Skolem functions for propositional formulas. In particular, it follows from Proposition 1 that $F[x_i/1]$ is a Skolem function for x_i in F. The above definition for a single variable can be naturally extended to a vector of variables. Given $F(X,Y)$, a *Skolem function vector* for $X = (x_1,\ldots,x_n)$ in F is a vector of functions $\mathbf{\Psi} = (\psi_1,\ldots,\psi_n)$ such that $\exists x_1 \ldots x_n\ F \equiv (\cdots(F[x_1/\psi_1])\cdots[x_n/\psi_n])$. A straightforward way to obtain a Skolem function vector $\mathbf{\Psi}$ is to first obtain a Skolem function ψ_1 for x_1 in F, then compute $F' \equiv \exists x_1\, F$ and obtain a Skolem function ψ_2 for x_2 in F', and so on until ψ_n has been obtained. More formally, ψ_i can be computed as a Skolem function for x_i in $\exists x_1 \ldots x_{i-1}\, F$, starting from ψ_1 and proceeding to ψ_n. Note that $\exists x_1 \ldots x_{i-1}\, F$ can itself be computed as $(\cdots(F[x_1/\psi_1])\cdots[x_{i-1}/\psi_{i-1}])$.

Definition 2. *The "*$\mathbf{C}$*an't-*$\mathbf{b}$*e-*$\mathbf{1}$*" function for x_i in F, denoted $\mathtt{Cb1}[x_i](F)$, is defined to be $(\neg\exists x_1 \ldots x_{i-1}\, F)\,[x_i/1]$. Similarly, the "*$\mathbf{C}$*an't-*$\mathbf{b}$*e-*$\mathbf{0}$*" function for x_i in F, denoted $\mathtt{Cb0}[x_i](F)$, is defined to be $(\neg\exists x_1 \ldots x_{i-1}\, F)\,[x_i/0]$. When X and F are clear from the context, we use $\mathtt{Cb1}[i]$ and $\mathtt{Cb0}[i]$ for $\mathtt{Cb1}[x_i](F)$ and $\mathtt{Cb0}[x_i](F)$, respectively.*

Intuitively, in order to make F evaluate to 1, we cannot set x_i to 1 (resp. 0) whenever the valuation of $\{x_{i+1},\ldots,x_n\} \cup Y$ satisfies $\mathtt{Cb1}[i]$ (resp., $\mathtt{Cb0}[i]$). The following proposition follows from Definition 2 and from our observation about computing a Skolem function vector one component at a time.

Proposition 2. $\mathbf{\Psi}$=($\neg$`Cb1`$[1], \ldots, \neg$`Cb1`$[n]$) *is a Skolem function vector for X in F.*

Note that the support of ψ_i in $\mathbf{\Psi}$, as given by Proposition 2, is $\{x_{i+1}, \ldots, x_n\} \cup Y$. If we want a Skolem function vector $\mathbf{\Psi}$ such that every component function has only Y (or a subset thereof) as support, this can be obtained by repeatedly substituting the Skolem function for every variable x_i in all other Skolem functions where x_i appears. We denote such a Skolem function vector as $\mathbf{\Psi}(Y)$.

III. A MONOLITHIC COMPOSITION BASED ALGORITHM

Our algorithm is motivated in part by cofactor-based techniques for computing Skolem functions, as proposed by Jiang et al [7] and Trivedi [17]. Given $F(X, Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, the techniques of [7], [17] essentially compute a Skolem function vector $\mathbf{\Psi}(Y)$ for X in F as shown in algorithm MonoSkolem (see Algorithm 1). In this algorithm, the variables in X are assumed to be ordered by their indices. While variable ordering is known to affect the difficulty of computing Skolem functions [7], we assume w.l.o.g. that the variables are indexed to represent a desirable order. We describe the variable order used in our study later in Section V.

MonoSkolem works in two phases. In the first phase, it implements a straightforward strategy for obtaining a Skolem function vector, as suggested by Proposition 2. Specifically, steps 3 and 4 of MonoSkolem build a monolithic conjunction F_i of all factors that have x_i in their support, before computing ψ_i. This restricts the scope of the quantifier for x_i to the conjunction of these factors. In Step 6, we use $\neg$`Cb1`$[i]$ as a specific choice for the Skolem function ψ_i. After computing ψ_i from F_i, step 7 discards the factors with x_i in their support, and introduces a single factor representing $\exists x_i\, F_i$ (computed as $F_i[x_i/\psi_i]$) in their place. Note that each ψ_i obtained in this manner has $\{x_{i+1}, \ldots, x_n\} \cup Y$ (or a subset thereof) as support. Since we want each Skolem function to have support Y, a second phase of "reverse" substitutions is needed. In this phase (see Algorithm 2), the Skolem function $\psi_n(Y)$ obtained above is substituted for x_n in $\psi_1, \ldots, \psi_{n-1}$. This effectively renders all Skolem functions independent of x_n. The process is then repeated with ψ_{n-1} substituted for x_{n-1} in $\psi_1, \ldots, \psi_{n-2}$ and so on, until all Skolem functions have been made independent of $x_1, \ldots, x_n$, and have only Y (or subsets thereof) as support.

MonoSkolem can be further refined by combining steps 6 and 7, and directly defining ψ_i in terms of F_i. However, we introduce the intermediate step using `Cb0`$[i]$ and `Cb1`$[i]$ to motivate their central role in our approach. Note that instead of $\neg$`Cb1`$[i]$, we could combine `Cb1`$[i]$ and `Cb0`$[i]$ in other ways (denoted by Combine(`Cb0`$[i]$, `Cb1`$[i]$) within comments in Algorithm 1) to get ψ_i in Step 6. In fact, Jiang et al [7] compute a Skolem function for x_i in F as an interpolant of $\neg$`Cb1`$[i] \wedge$ `Cb0`$[i]$ and `Cb1`$[i] \wedge \neg$`Cb0`$[i]$, while Trivedi [17] observes that the function ($\neg$`Cb1`$[i] \wedge ($`Cb0`$[i] \vee g)) \vee ($`Cb1`$[i] \wedge$ `Cb0`$[i] \wedge h)$ serves as a Skolem function for x_i in F where h and g are arbitrary propositional functions with support in $X \setminus \{x_i\} \cup Y$. Since computing interpolants using a SAT solver is often time-intensive and does not always lead to succinct Skolem functions [7], we simply use $\neg$`Cb1`$[i]$ as a Skolem function in Step 6. Proposition 2 guarantees the correctness of this choice.

Algorithm 1: MonoSkolem

```
Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$,
       where $X = (x_1, \ldots, x_n)$
Output: Skolem function vector $\mathbf{\Psi}(Y)$
// Phase 1 of algorithm
1 Factors := $\{f^j : 1 \le j \le r\}$;
2 for $i$ in 1 to $n$ do
3     FactorsWithXi := $\{f : f \in$ Factors$, x_i \in$ Supp$(f)\}$;
4     $F_i := \bigwedge_{f \in \text{FactorsWithXi}} f$;
5     Cb0$[i] := \neg F_i[x_i/0]$; Cb1$[i] := \neg F_i[x_i/1]$;
6     $\psi_i := \neg$Cb1$[i]$;
      // Generally, $\psi_i$:=Combine(Cb0$[i]$,Cb1$[i]$);
7     Factors := (Factors \ FactorsWithXi) $\cup \{F_i[x_i/\psi_i]\}$;
// Phase 2 of algorithm
8 return ReverseSubstitute($\psi_1, \ldots, \psi_n$);
```

Algorithm 2: ReverseSubstitute

```
Input: Functions
       $\psi_1(x_2, \ldots, x_n, Y), \psi_2(x_3, \ldots, x_n, Y), \ldots, \psi_n(Y)$
Output: Function vector $\mathbf{\Psi}(Y)$
1 for $i = n$ downto 2 do
2     for $k = i - 1$ downto 1 do $\psi_k = \psi_k[x_i/\psi_i]$;
3 return $\mathbf{\Psi}(Y) = (\psi_1(Y), \ldots, \psi_n(Y))$;
```

Observe that MonoSkolem works with a *monolithic* conjunction (F_i) of factors that have x_i in their support. Specifically, it composes each such monolithic conjunction F_i with a cofactor of F_i in Step 7 to eliminate quantifiers sequentially. This can lead to large memory footprints and more time-outs when used with medium to large benchmarks, as confirmed by our experiments. This motivates us to ask if we can develop a cofactor-based algorithm that does not suffer from the above drawbacks of MonoSkolem.

IV. CEGAR FOR GENERATING SKOLEM FUNCTIONS

We now present a new CEGAR [6] algorithm for generating Skolem function vectors, that exploits the factored form of $F(X, Y)$. Like MonoSkolem, our new algorithm, named CegarSkolem, works in two phases, and assumes that the variables in X are ordered by their indices. The first phase of the algorithm consists of the core abstraction-refinement part, and computes a Skolem function vector $(\psi_1, \ldots, \psi_n)$, where ψ_i has $\{x_{i+1}, \ldots, x_n\} \cup Y$, or a subset thereof, as support. Unlike in MonoSkolem, this phase avoids composing monolithic conjunctions of factors, yielding simpler Skolem functions. The second phase of the algorithm performs reverse substitutions, similar to that in MonoSkolem.

Before describing the details of CEGARSKOLEM, we introduce some additional notation and terminology. Given propositional functions (or formulas) f and g, we say that f *refines* g and g *abstracts* f iff f logically implies g. Given $F(X,Y)$ and a vector of functions $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$, we say that $\mathbf{\Psi^A}$ is an *abstract Skolem function vector* for X in F iff there exists a Skolem function vector $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$ for X in F such that ψ_i^A abstracts ψ_i, for every $i \in \{1, \ldots, n\}$. Instead of using $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ to compute Skolem functions, as was done in MONOSKOLEM, we now use their *refinements*, denoted $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ respectively, to compute abstract Skolem functions. For convenience, we represent $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ as sets of implicitly disjoined functions. Thus, if $\mathtt{r1}[i]$, viewed as a set, is $\{g_1, g_2\}$, then it is $g_1 \vee g_2$ when viewed as a function. We abuse notation and use $\mathtt{r1}[i]$ (resp., $\mathtt{r0}[i]$) to denote a set of functions or their disjunction, as needed.

A. Overview of our CEGAR algorithm

Algorithm CEGARSKOLEM has two phases. The first phase consists of a CEGAR loop, while the second does reverse substitutions. The CEGAR loop has the following steps.

- **Initial abstraction and refinement.** This step involves constructing refinements of $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ for every x_i in X. Using Proposition 2, we can then construct an initial abstract Skolem function vector $\mathbf{\Psi^A}$. This step is implemented in Algorithm 3 (INITABSREF), which processes individual factors of $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$ separately, without considering their conjunction. As a result, this step is time and memory efficient if the individual factors are simple with small representations.
- **Termination Condition.** Once INITABSREF has computed $\mathbf{\Psi^A}$, we check whether $\mathbf{\Psi^A}$ is already a Skolem function vector. This is achieved by constructing an appropriate propositional formula ε, called the "error formula" for $\mathbf{\Psi^A}$ (details in Subsection IV-C), and checking for its satisfiability. An unsatisfiable formula implies that $\mathbf{\Psi^A}$ is a Skolem function vector. Otherwise, a satisfying assignment π of ε is used to improve the current refinements of $\mathtt{Cb1}[i]$ and $\mathtt{Cb0}[i]$ for suitable variables x_i.
- **Counterexample guided abstraction and refinement.** This step is implemented in Algorithm 4: UPDATEABSREF, and computes an improved (i.e., more abstract) refinement of $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ for some $x_i \in X$. This, in turn, leads to a refinement of the abstract Skolem function vector $\mathbf{\Psi^A}$.

The overall CEGAR loop starts with the first step and repeats the second and third steps until a Skolem function vector is obtained. We now discuss the three steps in detail.

B. Initial Abstraction and Refinement

Algorithm INITABSREF (see Algorithm 3) starts by initializing each $\mathtt{r1}[i]$ and $\mathtt{r0}[i]$, viewed as sets, to the empty set. Subsequently, it considers each factor f in $\bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, and determines the contribution of f to $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$, for every x_i in the support of f. Specifically, if $x_i \in \mathsf{Supp}(f)$, the contribution of f to $\mathtt{Cb0}[i]$ is $(\neg\exists x_1 \ldots x_{i-1}\, f)\,[x_i/0]$, and

Algorithm 3: INITABSREF

```
   Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$,
          where $X = (x_1, \ldots, x_n)$
   Output: Abstract Skolem function vector
          $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$, and refinements $\mathtt{r0}[i]$ and
          $\mathtt{r1}[i]$ for each $x_i$ in $X$
1  for $i$ in $1$ to $n$ do
2      $\mathtt{r0}[i] := \emptyset$; $\mathtt{r1}[i] := \emptyset$; // Initializing
3  for $j$ in $1$ to $r$ do
4      $f := f^j$; // for each factor
5      for $i$ in $1$ to $n$ do
6          if $x_i \in \mathsf{Supp}(f)$ then
7              $\mathtt{r0}[i] := \mathtt{r0}[i] \cup \{\neg f[x_i/0]\}$;
8              $\mathtt{r1}[i] := \mathtt{r1}[i] \cup \{\neg f[x_i/1]\}$;
               // Skolem function for $x_i$ in $f$
9              $\psi_{i,f} := f[x_i/1]$;
10             $f := f[x_i/\psi_{i,f}]$; // $\because f[x_i/\psi_{i,f}] \equiv \exists x_i\, f$
11 for $i$ in $1$ to $n$ do
12     $\psi_i^A := \neg\mathtt{r1}[i]$;
       // Interpreting $\mathtt{r1}[i]$ as a function
13 return $\mathbf{\Psi^A}=(\psi_1^A, \ldots, \psi_n^A)$ and $\mathtt{r0}[i], \mathtt{r1}[i]$ $\forall x_i \in X$
```

its contribution to $\mathtt{Cb1}[i]$ is $(\neg\exists x_1 \ldots x_{i-1}\, f)\,[x_i/1]$. These contributions are accumulated in the sets $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$, respectively, and x_i is existentially quantified from f. The process is then repeated with the next variable in the support of f. Once the contributions from all factors are accumulated in $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for each x_i in X, INITABSREF computes an abstract Skolem function ψ_i^A for each x_i in F by complementing $\mathtt{r1}[i]$, interpreted as a disjunction of functions. Note that executing steps 4 through 10 of INITABSREF for a specific factor f is operationally similar to executing steps 1 through 7 of MONOSKOLEM with a singleton set of factors, i.e., $\mathsf{Factors} = \{f\}$. This highlights the key difference between INITABSREF and MONOSKOLEM: while MONOSKOLEM works with monolithic conjunctions of factors and their compositions, INITABSREF works with individual factors, without ever considering their conjunctions. Lemma 1 asserts the correctness of INITABSREF.

Lemma 1. *The vector* $\mathbf{\Psi^A}$ *computed by* INITABSREF *is an abstract Skolem function vector for X in $F(X,Y)$. In addition,* $\mathtt{r0}[i]$ *and* $\mathtt{r1}[i]$ *computed by* INITABSREF *are refinements of* $\mathtt{Cb0}[i](F)$ *and* $\mathtt{Cb1}[i](F)$ *for every x_i in X.*

C. Termination condition

Given $F(X,Y)$ and an abstract Skolem function vector $\mathbf{\Psi^A}$, it may happen that $\mathbf{\Psi^A}$ is already a Skolem function vector for X in F. We therefore check if $\mathbf{\Psi^A}$ is a Skolem function vector before refinement. Towards this end, we define the *error formula* for $\mathbf{\Psi^A}$ as $F(X', Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$, where $X' = (x_1', \ldots, x_n')$ is a sequence of fresh variables with no variable in common with X. The first term in the error formula checks if there exists some valuation of X that renders $\exists Y F(X,Y)$ true. The second term assigns variables in X to the values given by the abstract Skolem functions, and the third term checks if this assignment falsifies the formula F.

Lemma 2. *The error formula for $\mathbf{\Psi^A}$ is unsatisfiable iff $\mathbf{\Psi^A}$ is a Skolem function vector of X in F.*

The following example illustrates the role of the error formula.

Example 1. *Let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2, y_3\}$ in $\exists x_1 x_2 F(X,Y)$ where $F \equiv (f_1 \wedge f_2 \wedge f_3)$, with $f_1 = (\neg x_1 \vee \neg x_2 \vee \neg y_1)$, $f_2 = (x_2 \vee \neg y_3 \vee \neg y_2)$, $f_3 = (x_1 \vee \neg x_2 \vee y_3)$.*

Algorithm InitAbsRef *gives $\texttt{r1}[1] = (x_2 \wedge y_1)$, $\texttt{r0}[1] = (x_2 \wedge \neg y_3)$, $\texttt{r1}[2] =$ false, $\texttt{r0}[2] = y_3 \wedge y_2$. This yields $\psi_1^A = (\neg x_2 \vee \neg y_1)$, $\psi_2^A =$ true. Now, while ψ_1^A is a correct Skolem function for x_1 in F, ψ_2^A is not for x_2. This is detected by the satisfiability of the error formula $\varepsilon = F(x_1', x_2', Y) \wedge (x_1 = \neg x_2 \vee \neg y_1) \wedge (x_2 = 1) \wedge \neg F(x_1, x_2, Y)$. Note that $\neg F(\neg x_2 \vee \neg y_1, 1, Y)$ simplifies to $(y_1 \wedge \neg y_3)$, and $y_1 = 1, y_2 = 1, y_3 = 0, x_1 = 0, x_2 = 1, x_1' = 0, x_2' = 0$ is a satisfying assignment for ε.*

D. Counterexample-guided abstraction and refinement

Let ε be the error formula for $\mathbf{\Psi^A}$, and let π be a satisfying assignment of ε. We call π a *counterexample* of the claim that $\mathbf{\Psi^A}$ is a Skolem function vector. For every variable $v \in X' \cup X \cup Y$, we use $\pi(v)$ to denote the value of v in π. Satisfiability of ε implies that we need to refine at least one abstract Skolem function ψ_i^A in $\mathbf{\Psi^A}$ to make it a Skolem function vector. Since ψ_i^A is $\neg\texttt{r1}[i]$ in our approach, refining ψ_i^A can be achieved by computing an improved (i.e., more abstract) version of $\texttt{r1}[i]$. Algorithm UpdateAbsRef implements this idea by using π to determine which $\texttt{r1}[i]$ should be rendered abstract by adding appropriate functions to $\texttt{r1}[i]$, viewed as a set.

Before delving into the details of UpdateAbsRef, we state some key results. In the following, we use $\pi \models f$ to denote that the formula f evaluates to 1 when the variables in $\mathsf{Supp}(f)$ are set to values given by π. If $\pi \models f$, we also say f evaluates to 1 under π. We use $\texttt{r0}[i]_{init}$ and $\texttt{r1}[i]_{init}$ to refer to $\texttt{r0}[i]$ and $\texttt{r1}[i]$, as computed by algorithm InitAbsRef. Since UpdateAbsRef only adds to $\texttt{r1}[i]$ and $\texttt{r0}[i]$ viewed as sets, it is easy to see that $\texttt{r0}[i]_{init} \Rightarrow \texttt{r0}[i]$ and $\texttt{r1}[i]_{init} \Rightarrow \texttt{r1}[i]$ viewed as functions (recall these functions are simply disjunctions of elements in the corresponding sets).

Lemma 3. *Let π be a satisfying assignment of the error formula ε for $\mathbf{\Psi^A}$. Then the following hold.*

(a) $\pi \models \neg\texttt{Cb0}[n] \vee \neg\texttt{Cb1}[n]$.
(b) *There exists $k \in \{1, \ldots, n-1\}$ s.t., $\pi \models \texttt{r1}[k] \wedge \texttt{r0}[k]$.*
(c) *There exists no Skolem function vector $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$ such that $\psi_j \Leftrightarrow \psi_j^A$ for all j in $\{k+1, \ldots, n\}$.*
(d) *There exists $l \in \{k+1, \ldots, n\}$ such that $x_l = 1$ in π, and $\pi \models \texttt{Cb1}[l] \wedge \neg\texttt{r0}[l]$.*

Algorithm 4 (UpdateAbsRef) uses Lemma 3 to compute abstract versions of $\texttt{r0}[i]$ and $\texttt{r1}[i]$, and a refined version of $\mathbf{\Psi^A}$, when $\mathbf{\Psi^A}$ is not a Skolem function vector. It takes as input the current versions of $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all x_i in X, and a satisfying assignment π of the error formula for the current version of $\mathbf{\Psi^A}$. Since $\pi \models F(X', Y)$ and $\pi \models \neg F(X, Y)$, and since the value of every x_i in π is given by ψ_i^A, there exists at least one ψ_l^A, for $l \in \{1, \ldots, n\}$, that fails to generate the right value of x_l when the value of Y is as given by π. UpdateAbsRef works by identifying such an index l and refining ψ_l^A. Since $\psi_i^A = \neg\texttt{r1}[i]$, ψ_l^A is refined by updating (abstracting) the corresponding $\texttt{r1}[l]$ set. In fact, the algorithm may, in general, end up abstracting not only $\texttt{r1}[l]$, but several $\texttt{r0}[i]$ and $\texttt{r1}[i]$ as well in a sound manner.

As shown in Algorithm 4, UpdateAbsRef first finds the largest index k such that $\pi \models \texttt{r0}[k] \wedge \texttt{r1}[k]$. Lemma 3b guarantees the existence of such an index in $\{1, \ldots, n\}$. We assume access to a function called Generalize that takes as arguments an assignment π and a function φ such that $\pi \models \varphi$, and returns a function ξ that generalizes π while satisfying φ. More formally, if $\xi =$ Generalize(π, φ), then $\mathsf{Supp}(\xi) \subseteq \mathsf{Supp}(\varphi)$, $\pi \models \xi$ and $\xi \Rightarrow \varphi$ (details of

Algorithm 4: UpdateAbsRef

```
Input: $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all $x_i$ in $X$,
       Satisfying assignment $\pi$ of error formula, i.e.,
       $F(X',Y) \wedge \bigwedge_{i=1}^{n} (x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$
Output: Improved (i.e., refined) $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$,
        Improved (i.e., abstracted) $\texttt{r0}[i]$ & $\texttt{r1}[i]$, $\forall x_i \in X$
1  $k :=$ largest $m$ such that $\pi$ satisfies $\texttt{r0}[m] \wedge \texttt{r1}[m]$;
2  $\mu_0 :=$ Generalize($\pi$, $\texttt{r0}[k]$);
3  $\mu_1 :=$ Generalize($\pi$, $\texttt{r1}[k]$);
4  $\mu := \mu_0 \wedge \mu_1$;
   // Search for Skolem function among
        $\{\psi_{k+1}^A, \ldots, \psi_n^A\}$ to be refined
5  $l := k + 1$;
6  while true do    // current guess: refine $\psi_l^A$
7      if $x_l \in \mathsf{Supp}(\mu)$ then
8          if $x_l = 1$ in $\pi$ then
9              $\mu_1 := \mu[x_l/1]$;
10             $\texttt{r1}[l] := \texttt{r1}[l] \cup \{\mu_1\}$;
11             if $\pi$ satisfies $\texttt{r0}[l]$ then
12                 $\mu_0 :=$ Generalize($\pi$, $\texttt{r0}[l]$);
13                 $\mu := \mu_0 \wedge \mu_1$;
14             else
15                 break;
16         else
17             $\mu_0 := \mu[x_l/0]$;
18             $\texttt{r0}[l] := \texttt{r0}[l] \cup \{\mu_0\}$;
19             $\mu_1 :=$ Generalize($\pi$, $\texttt{r1}[l]$);
20             $\mu := \mu_0 \wedge \mu_1$;
21     $l := l + 1$ ;
22 $\mathbf{\Psi^A} = (\neg\texttt{r1}[1], \ldots, \neg\texttt{r1}[n])$;
23 return $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all $x_i$ in $X$, and $\mathbf{\Psi^A}$
```

GENERALIZE used in our implementation are discussed later). Thus, in steps 2 and 3 of UPDATEABSREF, we compute generalizations of π that satisfy $\mathtt{r0}[k]$ and $\mathtt{r1}[k]$, respectively. The function μ computed in step 4 is therefore such that $\pi \models \mu$ and $\mu \Rightarrow \mathtt{r0}[k] \wedge \mathtt{r1}[k]$. Since $\mathtt{r0}[k] \wedge \mathtt{r1}[k] \Rightarrow \neg\exists x_1 \ldots x_k F$, any abstract Skolem function vector that produces values of $x_1, \ldots, x_n$ (given the valuation of Y as in π) for which μ evaluates to 1, cannot be a Skolem function vector. Since the support of μ is $\{x_{k+1}, \ldots, x_n\} \cup Y$, one of the abstract Skolem functions $\psi_{k+1}^A, \ldots, \psi_n^A$ must be refined.

The loop in steps 6–21 of UPDATEABSREF tries to identify an abstract Skolem function ψ_l^A to be refined, by iterating l from $k+1$ to n. Clearly, if $x_l \notin \mathsf{Supp}(\mu)$, the value of ψ_l^A under π is of no consequence in evaluating μ, and we ignore such variables. If $x_l \in \mathsf{Supp}(\mu)$ and if $x_l = 1$ in π, then $\pi \models \mu[x_l/1]$ and $\mu[x_l/1] \Rightarrow (\neg\exists x_1 \ldots x_{l-1} F)[x_l/1]$. Recalling the definition of $\mathtt{Cb1}[l]$, we have $\mu[x_l/1] \Rightarrow \mathtt{Cb1}[l]$, and therefore $\mu[x_l/1]$ can be added to $\mathtt{r1}[l]$ (viewed as a set) yielding a more abstract version of $\mathtt{r1}[l]$. Steps 8–10 of UPDATEABSREF implement this update of $\mathtt{r1}[l]$. Note that since $\pi \models \mu[x_l/1]$, we have $\pi \models \mathtt{r1}[l]$ after step 10. If it so happens that $\pi \models \mathtt{r0}[l]$ as well, then we have $\pi \models \mathtt{r0}[l] \wedge \mathtt{r1}[l]$, where $\mathtt{r1}[l]$ refers to the updated refinement of $\mathtt{Cb1}[l]$. In this case, we have effectively found an index $l > k$ such that $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$. We can therefore repeat our algorithm starting with l instead of k. Steps 11–13 followed by step 21 of algorithm UPDATEABSREF effectively implement this. If, on the other hand, $\pi \not\models \mathtt{r0}[k]$, then we have found an l that satisfies the conditions in Lemma 3d. We exit the search for an abstract Skolem function in this case (see steps 14–15).

If $x_l = 0$ in π, a similar argument as above shows that $\mu[x_l/0]$ can be added to $\mathtt{r0}[l]$. Steps 17–18 of UPDATEABSREF implement this update. As before, it is easy to see that $\pi \models \mathtt{r0}[l]$ after step 18. Moreover, since $\pi \models \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi_i^A)$ and $\psi_i^A \equiv \neg\mathtt{r1}[l]$, in order to have $x_l = 0$ in π, we must have $\pi \models \mathtt{r1}[l]$. Therefore, we have once again found an index $l > k$ such that $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$, and can repeat our algorithm starting with l instead of k. Steps 19–21 of algorithm UPDATEABSREF effectively implement this.

Once we exit the loop in steps 6–21 of UPDATEABSREF, we compute the refined Skolem function vector $\mathbf{\Psi^A}$ as $(\neg\mathtt{r1}[1], \ldots \neg\mathtt{r1}[n])$ in step 22 and return the updated $\mathtt{r0}[i]$, $\mathtt{r1}[i]$ for all x_i in X, and also $\mathbf{\Psi^A}$.

Example 1 (Continued). *Continuing with our earlier example, the error formula after the first step has a satisfying assignment* $y_1 = 1, y_2 = 1, y_3 = 0, x_1 = 0, x_2 = 1, x_1' = 0, x_2' = 0$. *Using this for* π *in* UPDATEABSREF, *we find that* ψ_1^A *is left unchanged at* $(\neg x_2 \vee \neg y_1)$, *while* ψ_2^A, *which was* true *earlier, is refined to* $(\neg y_1 \vee y_3)$. *With these refined Skolem functions,* $F(\psi_1^A, \psi_2^A, Y)$ *evaluates to* true *for all valuations of* Y. *As a result, the (new) error formula becomes unsatisfiable, confirming the correctness of the Skolem functions.*

It can be shown that Algorithm UPDATEABSREF always terminates, and renders at least one $\mathtt{r1}[i]$ strictly abstract, and at least one ψ_i^A strictly refined, for $i \in \{1, \ldots, n\}$ (see [9] for the proof). The overall CEGARSKOLEM algorithm can now be implemented as depicted in Algorithm 5. From the above discussion and Lemmas 1 and 2, we obtain our main result.

Algorithm 5: CEGARSKOLEM

```
Input: Propositional formula
       $F(X,Y) = \bigwedge_{j=1}^r f^j(X_j, Y_j)$, $X = (x_1, \ldots, x_n)$
Output: Skolem function vector $\mathbf{\Psi}(Y)$ for $X$ in $F$
1 $(\mathbf{\Psi^A}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \le i \le n\}) :=$
  INITABSREF($\bigwedge_{j=1}^r f^j$);
2 $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X, Y)$;
3 while $\varepsilon$ is satisfiable do
4     Let $\pi$ be a satisfying assignment of $\varepsilon$;
5     $(\mathbf{\Psi^A}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \le i \le n\}) :=$
      UPDATEABSREF($\{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \le i \le n\}$, $\pi$);
6     $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X, Y)$;
7 $\mathbf{\Psi}(Y) :=$ REVERSESUBSTITUTE($\neg\mathtt{r1}[1], \ldots, \neg\mathtt{r1}[n]$);
8 return $\mathbf{\Psi}(Y)$;
```

Theorem 1. CEGARSKOLEM*(*$F(X,Y)$*) terminates and computes a Skolem function vector for* X *in* F.

The function GENERALIZE(π, φ) used in UPDATEABSREF can be implemented in several ways. Since $\pi \models \varphi$, we may return a conjunction of literals corresponding to the assignment π, or the function φ itself. From our experiments, it appears that the first option leads to low memory requirements and increased run-time (due to large number of invocations of UPDATEABSREF). The other option requires more memory and less run-time due to fewer invocations of UPDATEABSREF. For our study, we let GENERALIZE(π, $\mathtt{r1}[k]$) return one element in $\mathtt{r1}[k]$ (viewed as a set) amongst all those that evaluate to 1 under π, such that the support of μ computed in Algorithm UPDATEABSREF is minimized (we had to allow GENERALIZE($\cdot, \cdot$) access to μ for this purpose). We follow a similar strategy for GENERALIZE(π, $\mathtt{r0}[k]$). This gives us a reasonable tradeoff between time and space requirements.

V. EXPERIMENTAL RESULTS

A. Experimental Methodology

We compared CEGARSKOLEM with (a) MONOSKOLEM (the algorithm based on the cofactoring approach of [7], [17]) and with (b) Bloqqer (a QRAT-based Skolem function generation tool reported in [12]). As described in [12], Bloqqer generates Skolem functions by first generating QRAT proofs using a remarkably efficient (albeit incomplete) preprocessor, and then generates Skolem functions from these proofs.

The Skolem function generation benchmarks were obtained by considering sequential circuits from the HWMCC10 benchmark suite, and by reducing the problem of disjunctively decomposing a circuit into components to the problem of generating Skolem function vectors. Details of how these benchmarks were generated are described in [1]. Each benchmark is of the form $\exists X F(X,Y)$, where $F(X,Y)$ is a conjunction of factors and $\exists Y(\exists X F(X,Y))$ is true. However, for some benchmarks, $\forall Y(\exists X F(X,Y))$ does not evaluate to true. Since Bloqqer can generate Skolem functions only when $\forall Y(\exists X F(X,Y))$ is true, we divided the benchmarks into two categories: a) TYPE-1 where $\forall Y \exists X F(X,Y)$ is true, and b) TYPE-2 where $\forall Y \exists X F(X,Y)$ is false (although $\exists Y \exists X F(X,Y)$ is true). While we ran CEGARSKOLEM and MONOSKOLEM on all benchmarks, we ran Bloqqer only on TYPE-1 benchmarks. Further, since Bloqqer required the input to be in `qdimacs` format, we converted each TYPE-1 benchmark into `qdimacs` format using Tseitin encoding [18]. All our benchmarks can be downloaded from [1].

Our implementations of MONOSKOLEM and CEGARSKOLEM make use of the ABC [11] library to represent and manipulate functions as AIGs. For CEGARSKOLEM, we used the default SAT solver provided by ABC, which is a variant of MiniSAT. We used a simple heuristic to order the variables, and used the same ordering for both MONOSKOLEM and CEGARSKOLEM. In our ordering, variables that occur in fewer factors are indexed lower than those that occur in more factors.

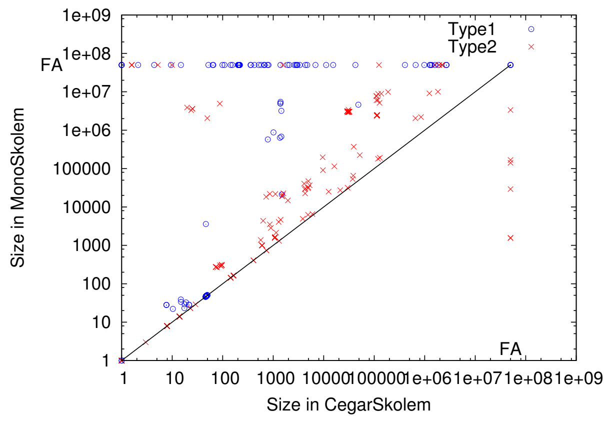
We used the following metrics to compare the performance of the algorithms: (i) average/maximum size of the generated Skolem functions in a Skolem function vector, where the size is the number of nodes in the AIG representation of a function, and ii) total time taken to generate the Skolem function vector (excluding any input format conversion time). The experiments were performed on a 1.87 GHz Intel(R) Xeon machine with 128GB memory running Ubuntu 12.04.4. The maximum time and main memory usage was restricted to 2 hours and 32GB, although we noticed that for most benchmarks, all three algorithms used less than 2 GB memory.

B. Results and Discussion

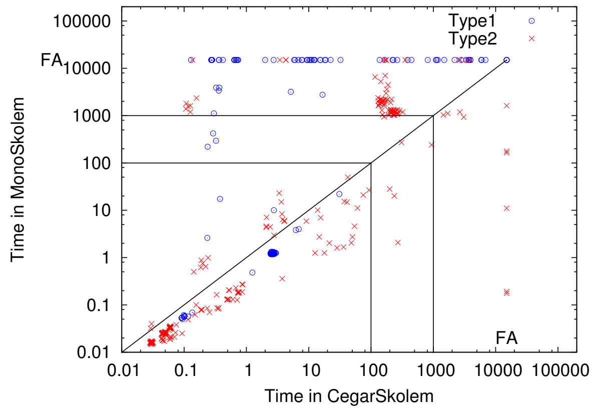
We conducted our experiments with 424 benchmarks, of which 160 were TYPE-1 benchmarks and 264 were TYPE-2 benchmarks. The 424 benchmarks covered a wide spectrum in terms of number of factors, total number of variables, and number of quantified variables (see [9] for details).

1) CEGARSKOLEM vs MONOSKOLEM: The performance of these two algorithms on all the benchmarks (TYPE-1 and TYPE-2) is shown in the scatter plots of Figure 1, where Figure 1a shows the average sizes of Skolem functions generated in a Skolem function vector and Figure 1b shows the total time taken in seconds. From Figure 1a, it is clear that the Skolem functions generated by CEGARSKOLEM in a Skolem function vector are on average *smaller* than those generated by MONOSKOLEM. *There is no instance on which* CEGARSKOLEM *generates Skolem function vectors with larger functions on average vis-a-vis* MONOSKOLEM.

Due to repeated calls to the SAT-solver, CEGARSKOLEM takes more time than MONOSKOLEM on some benchmarks, but on most of them the total time taken by both algorithms is *less than* 100 seconds (Figure 1b). Indeed, on profiling we found that CEGARSKOLEM spent most of its time on SAT solving. On 38 benchmarks where CEGARSKOLEM took greater than 100 but less than 300 seconds, MONOSKOLEM



(a) Average Skolem function sizes



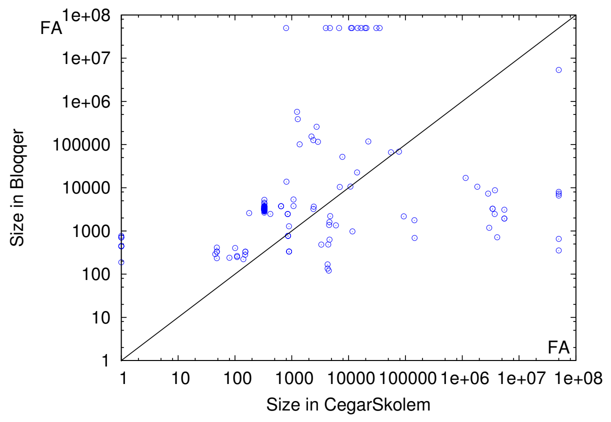
(b) Time taken (in seconds)

Fig. 1: CEGARSKOLEM vs MONOSKOLEM on TYPE-1 & TYPE-2 benchmarks. Topmost (rightmost) points indicate benchmarks where MONOSKOLEM (CEGARSKOLEM) was unsuccessful.

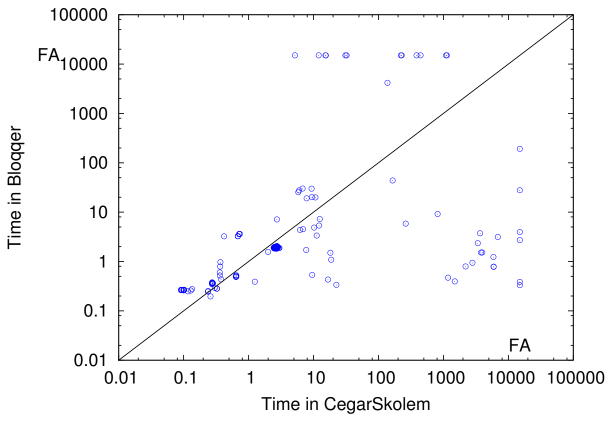
performed significantly worse, taking more than 1000 seconds. We found the degradation of MONOSKOLEM was due to the large sizes of Skolem functions generated (of the order of 1 million AIG nodes) compared to those generated by CEGARSKOLEM ($<$ 8000 AIG nodes). *Large Skolem function sizes clearly imply more time spent in function composition and reverse-substitution.*

For benchmarks where the sizes of Skolem functions generated were even larger (of the order of 10^7 AIG nodes), MONOSKOLEM could not complete generation of all Skolem functions: for 8 benchmarks, the memory consumed by MONOSKOLEM increased rapidly, resulting in memory outs; for 10 benchmarks, it ran out of time; for an overwhelming 83 benchmarks, it encountered integer overflows (and hence assertion failures) in the underlying ABC library. These are indicated by the topmost points (see label “FA” on the axes) in Figure 1. In contrast, CEGARSKOLEM *generated Skolem functions for almost all* (412/424) *benchmarks*. The rightmost points indicate the 12 cases where CEGARSKOLEM failed, of which 10 were time-outs and 2 were memory outs.

2) CEGARSKOLEM vs Bloqqer: Of the 160 TYPE-1 benchmarks, Bloqqer successfully generated Skolem function



(a) Maximum size of Skolem functions



(b) Time taken (in seconds)

Fig. 2: CEGARSKOLEM vs Bloqqer on TYPE-1 benchmarks. Topmost (rightmost) points indicate benchmarks for which Bloqqer (CEGARSKOLEM) was unsuccessful.

vectors in 148 cases. It gave a `NOT VERIFIED` message for the remaining 12 benchmarks (in less than 30 minutes). These benchmarks are indicated by the topmost points (see label "FA" on the axes) in the scatter plots of Figure 2. Of these, 8 are large benchmarks with 1000+ factors and variables to eliminate (overall, there are 9 such large benchmarks). On the other hand, CEGARSKOLEM was able to successfully generate Skolem functions on 154 benchmarks, including the 9 large benchmarks, on each of which it took less than 20 minutes.

For the 142 benchmarks for which both algorithms succeeded, we compared the times taken in Figure 2b. As earlier, CEGARSKOLEM took more time on many benchmarks, but there were several benchmarks, including the large benchmarks, on which Bloqqer was out-performed. We also compared the maximum sizes of Skolem functions generated in a Skolem function vector (see Figure 2a). We used the maximum (instead of average) size, since Tseitin encoding was needed to convert the benchmarks to `qdimacs` format, and this introduces many variables whose Skolem function sizes are very small, skewing the average. For a majority (108/142) of the benchmarks where both algorithms succeeded, the maximum sizes of Skolem functions obtained by CEGARSKOLEM were *smaller* than those generated by Bloqqer. Hence, *not only does* CEGARSKOLEM *run faster on the large benchmarks, it also generates smaller Skolem functions on most of them.*

3) Discussion: For all benchmarks on which CEGARSKOLEM timed out, we noticed that there were large subsets of factors that shared many variables in their supports. As a result, CEGARSKOLEM could not exploit the factored representation effectively, requiring many refinements. We also noticed that for many benchmarks (197/424), the initial abstract Skolem functions were correct, and most of the time was spent in the SAT solver. In fact, on averaging over all benchmarks, we found that around 33% of the time spent by CEGARSKOLEM was for SAT-solving. This shows that we can leverage improvements in SAT solving technology to improve the performance of CEGARSKOLEM.

VI. CONCLUSION AND FUTURE WORK

We presented a CEGAR algorithm for generating Skolem functions from factored propositional formulas. Our experiments show that for complex functions, our algorithm outperforms two state-of-the-art algorithms. As part of future work, we will explore integration with more efficient SAT-solvers and refinement using multiple counter-examples.

REFERENCES

[1] A. John et al. Disjunctive Decomposition Benchmarks. http://www.cse.iitb.ac.in/~supratik/tools/fmcad_2015_experiments/.
[2] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *STTT*, 7(2):118–128, 2005.
[3] Carlos Ansotegui, Carla P Gomes, and Bart Selman. The Achilles' heel of QBF. In *Proc. of AAAI*, volume 2, pages 275–281, 2005.
[4] Marco Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proc. of CADE*, pages 369–376. Springer-Verlag, 2005.
[5] Christian Bessière and Guillaume Verger. Strategic constraint satisfaction problems. In *Proc. of CP*, pages 17–29, 2006.
[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003.
[7] J.-H. R. Jiang. Quantifier elimination via functional composition. In *Proc. of CAV*, pages 383–397. Springer, 2009.
[8] J.-H. R. Jiang and V Balabanov. Resolution proofs and Skolem functions in QBF evaluation and applications. In *Proc. of CAV*, pages 149–164. Springer, 2011.
[9] A. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay. Skolem functions for factored formulas. *CoRR*, Identifier: submit/1333056, 2015. (https://arxiv.org/submit/1333056).
[10] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. Wintersteiger. A First Step Towards a Unified Proof Checker for QBF. In *Proc. of SAT*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.
[11] Berkeley Logic and Verification Group. ABC: A System for Sequential Synthesis and Verification . http://www.eecs.berkeley.edu/~alanmi/abc/.
[12] Martina Seidl Marijn Heule and Armin Biere. Efficient Extraction of Skolem Functions from QRAT Proofs. In *Proc. of FMCAD*, 2014.
[13] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
[14] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design, vol. 173 of NATO Science Series F*, pages 303–366. IOS Press, 1999.
[15] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
[16] D. Thomas, S. Chakraborty, and P.K. Pandya. Efficient guided symbolic reachability using reachability expressions. *STTT*, 10(2):113–129, 2008.
[17] A. Trivedi. Techniques in symbolic model checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India, 2003.
[18] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics*, pages 115–125, 1968.

Theory-Aided Model Checking of Concurrent Transition Systems

Guy Katz
Weizmann Institute of Science
guy.katz@weizmann.ac.il

Clark Barrett
New York University
barrett@cs.nyu.edu

David Harel
Weizmann Institute of Science
david.harel@weizmann.ac.il

***Abstract*—We present a method for the automatic compositional verification of certain classes of concurrent programs. Our approach is based on the casting of the model checking problem into a theory of transition systems within CVC4, a DPLL(*T*) based SMT solver. Our transition system theory then cooperates with other theories supported by the solver (e.g., arithmetic, arrays), which can help accelerate the verification process. More specifically, our theory solver looks for known patterns within the input programs and uses them to generate lemmas in the languages of other theories. When applicable, these lemmas can often steer the search away from safe parts of the search space, reducing the number of states to be explored and expediting the model checking procedure. We demonstrate the potential of our technique on a number of broad classes of programs.**

I. Introduction

In concurrent programming, the size of the composite program is typically exponential in the number of its constituent threads. This phenomenon, an instance of the *state explosion* problem, is a major hindrance to the verification of concurrent software. In recent decades, a prominent approach to tackling this difficulty has been that of compositional verification [13]: properties of threads are derived/verified in isolation, and are used to deduce global system correctness, without exploring the entire composite state space. When applicable, compositional verification can often significantly outperform direct verification techniques.

A key challenge in compositional verification is how to *automatically* come up with "good" thread properties — those whose verification is considerably cheaper than the verification of the global property on the one hand, but which are sufficiently meaningful to imply the desired system properties on the other. Automatic property generation is essential in rendering a compositional verification scheme scalable [11].

Since the compositional verification of arbitrary programs is difficult (and often impossible [9]), one reasonable approach is to trade generality for effectiveness — i.e., to limit the scope of programs that a scheme handles, in exchange for better performance on programs that remain within that scope. Here, we adopt this approach and propose an automatic compositional verification scheme for certain kinds of concurrent software.

The paper has two main contributions. The first is the rigorous formalization and implementation of a solver for a *theory of transition systems* ($\mathcal{TS}$) within the context of CVC4 [1] — a lazy, DPLL(T) based SMT solver [28]. The $\mathcal{TS}$ solver takes as input formulas describing a program's concurrent threads (given as transition systems) and the assertion that a certain safety property is violated; and it answers UNSAT if the program is safe, or SAT if it is not. As a standalone module, the $\mathcal{TS}$ solver explores the space of reachable states in order to determine a system's safety — an exploration that is driven by the SMT solver's underlying SAT engine.

Several existing approaches utilize SMT solvers in model checking (e.g., Lazy Annotation [26] and PDR [8]), but typically the process is driven by a model checker that uses an SMT solver as a black-box tool. In our approach the roles are reversed, and the SMT engine, via the $\mathcal{TS}$ solver, can be regarded as invoking a model checker. This design allows other theories within CVC4 to be seamlessly used in analyzing the input program at hand, determining which parts of the state space should be explored and which may safely be ignored. These theories may then influence the search conducted by the $\mathcal{TS}$ solver by asserting lemmas to the underlying DPLL(T) core, sometimes pruning significant portions of the search space and greatly improving performance. We term this process *theory-aided model checking*: the $\mathcal{TS}$ solver explores the state space while also looking for opportunities in which other theories may aid and direct the search.

The second contribution of the paper is in the way other theories determine which parts of the state space may be ignored during model checking. We perform this by having the $\mathcal{TS}$ solver analyze the input threads and look for pre-supplied *patterns*: structural properties of the threads that may be expressed as assertions in the languages of other theories, such as arithmetic or arrays. It is through these assertions that other theories can "understand" the program and efficiently discover, e.g., that a certain branch of the search space cannot lead to a violation. A key fact here is that each thread/transition system is analyzed separately — and hence the compositionality of our approach: the analysis complexity is proportional to the size of the program and not to that of its state space. We thoroughly describe three of the currently implemented patterns.

While our proposed technique is compositional and completely automatic, it is useful only when the input programs match one of the pre-supplied patterns. This is in line with our approach of trading generality for effectiveness, and, as we demonstrate in later sections, our approach is capable of effectively handling broad classes of programs even with just a few stored patterns.

The type of software that we target here is a family of discrete event systems. In particular, we focus on a computational model that has three fundamental concurrency idioms — requesting events, waiting-for events and blocking events — which we term the $\mathcal{RWB}$ model. The $\mathcal{RWB}$ concurrency idioms are widespread and appear, sometimes

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

in related forms, in various formalisms such as *publish-subscribe* architectures [12], *supervisory control* [29] and *live sequence charts* (LSC) [10]. Together, these three idioms also form the *behavioral programming* (*BP*) model [20]. Thus, by focusing on the $\mathcal{RWB}$ model, we hope to make our technique applicable (with appropriate adjustments) to a variety of programming formalisms. Further, we believe that the technique can be extended to cater to additional concurrency idioms and models.

The rest of the paper is organized as follows. In Section II we recap the definitions of the DPLL(T) framework for SMT solvers and of the $\mathcal{RWB}$ model. Next, in Section III we introduce the theory of transition systems ($\mathcal{TS}$) and describe a theory solver aimed at model checking $\mathcal{RWB}$ programs. In Section IV we demonstrate how the $\mathcal{TS}$ solver can cooperate with other theory solvers in order to expedite model checking. Subsequently, we apply our technique to two broad classes of problems: *periodic problems* in Section V, and programs with shared arrays in Section VI. Experimental results appear in Section VII, and we conclude with a discussion and related work in Section VIII.

II. Definitions

The DPLL(T) Framework. DPLL(T) [28] is an extensible framework used by modern SMT solvers. It employs multiple specialized theory solvers that interact with a SAT solver. The SAT solver maintains an input formula F and a partial assignment M for F. Periodically, a theory solver is asked whether M is satisfiable in its theory; and, if it is not, the theory solver generates a conflict clause, the negation of an unsatisfiable subset of M, that is added to F. The theory solver may request case splitting by means of the splitting-on-demand paradigm [2], which allows the solver to add theory lemmas to F consisting of clauses possibly with literals not occurring in F.

The $\mathcal{RWB}$ Model. In this work we focus on the $\mathcal{RWB}$ model for concurrent discrete event systems. An $\mathcal{RWB}$ program is comprised of a set of events and set of threads that communicate via the requesting, waiting-for and blocking of events. More specifically, the threads repeatedly synchronize with each other at predetermined synchronization points, and at each such point they each declare events that they request and events that they block. Then, an event that is requested by at least one thread and not blocked is selected for triggering, and all the threads that requested or waited-for this event proceed with their execution. Whenever a new synchronization point is reached, the process is repeated.

The $\mathcal{RWB}$ model is not intended to be programmed in directly. Rather, it is used to describe the underlying transition systems of threads written in higher level languages, for the purpose of analysis and verification. Actual programming in $\mathcal{RWB}$ is performed, e.g., using the *behavioral programming* (*BP*) framework [20], which is implemented in various high level languages such as C++ or Java (see http://www.b-prog.org). Thus, while inter-thread communication in BP is performed solely through the $\mathcal{RWB}$ idioms, threads may internally use any construct provided by the underlying programming language (e.g., C++). Indeed, the tool and examples described in this paper were prepared using a C++ version of BP (termed *BPC* [16]), and CVC4. It should further be noted that the $\mathcal{RWB}$ definitions as given here entail global lockstep synchronization between components, which may cause unwanted overhead. There exist extensions to $\mathcal{RWB}$ that mitigate this difficulty without altering the model's semantics [14], and our technique is applicable to these extensions as well.

Formally, an $\mathcal{RWB}$-thread T over event set E is a tuple $T = \langle Q, \delta, q_0, R, B\rangle$, where Q is a set of states (one for each synchronization point), q_0 is the initial state, $R : Q \to 2^E$ and $B : Q \to 2^E$ map states to events requested and blocked at these states (respectively), and $\delta : Q \times E \to 2^Q$ is a transition function (the definition is adopted from [19]).

$\mathcal{RWB}$ programs are created by *composing* $\mathcal{RWB}$-threads. The parallel composition of threads $T^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1\rangle$ and $T^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2\rangle$, both over the same event set E, yields the $\mathcal{RWB}$-thread defined by $T^1 \parallel T^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2\rangle, R^1 \cup R^2, B^1 \cup B^2\rangle$, where $\langle \tilde{q}^1, \tilde{q}^2\rangle \in \delta(\langle q^1, q^2\rangle, e)$ iff $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$. The union of the labeling functions is defined in the natural way, i.e. $e \in (R^1 \cup R^2)(\langle q^1, q^2\rangle)$ iff $e \in R^1(q^1) \cup R^2(q^2)$. An $\mathcal{RWB}$ *program* P comprised of $\mathcal{RWB}$-threads $T^1, T^2, \ldots, T^n$ is the composite thread $P = T^1 \parallel \ldots \parallel T^n$. Denoting $P = \langle Q, \delta, q_0, R, B\rangle$, an execution of P starts from q_0, and in each state q along the run an enabled event is chosen for triggering, if one exists (i.e., an event $e \in R(q) - B(q)$). Then, the execution moves to state $\tilde{q} \in \delta(q, e)$, and so on. An execution can either be infinite, or finite if it ends in a state with no successors (a *deadlock* state). An illustration of a simple $\mathcal{RWB}$ program appears in Fig. 1.

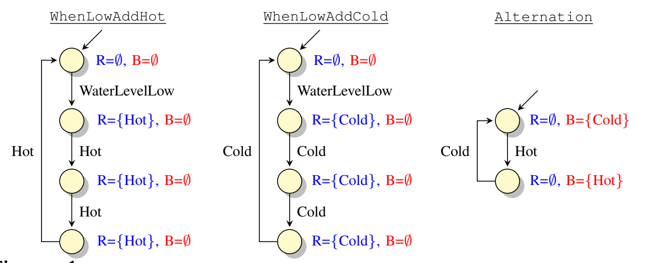


Figure 1: An $\mathcal{RWB}$ program for controlling the water level in a tank with hot and cold water sources. Each node corresponds to a synchronization point in a thread, labeled with its requested (R) and blocked (B) events. Waited-for events are not labeled, and are represented by transitions. If an event that a thread did not wait for is triggered, the thread does not change states. In the program depicted, the $\mathcal{RWB}$-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events (requested by a sensor thread, not shown) and requests three times the event `Hot`. `WhenLowAddCold` performs a similar action with the event `Cold`. In order to keep the water temperature stable, the `Alternation` thread enforces the interleaving of `Hot` and `Cold` events by using event blocking.

From a software-engineering perspective, the motivation for using the $\mathcal{RWB}$ idioms for inter-thread communication lies in the model's strict and simple synchronization mechanism. Studies show that this form of inter-thread interaction — i.e., through repeated synchronization and declaration of requested, waited-for and blocked events — facilitates incremental, non-intrusive development, and the resulting systems often have threads that are aligned with the specification [20].

Verifying $\mathcal{RWB}$ Programs. In [19], [22], the authors demonstrate how the transition systems underlying $\mathcal{RWB}$-

threads can be automatically extracted from high level code and then, using abstraction techniques, be symbolically traversed in order to verify safety properties. Safety properties are themselves expressed by *marker* $\mathcal{RWB}$-threads, marking that a violation has occurred with a special API call [19]. For simplicity, we assume that marker threads signal that a violation has occurred by blocking all events, causing a deadlock. Thus, safety checking is reduced to checking for deadlock freedom.

The manual compositional verification of $\mathcal{RWB}$ programs is discussed in [15]. There, it is shown how the simple $\mathcal{RWB}$ synchronization mechanism facilitates the generation of individual thread properties, which are then used for proving the system property at hand. The beneficial effect that simple concurrency idioms have on verification is also discussed in [18]. Indeed, the simplicity of the $\mathcal{RWB}$ idioms plays a key role in the pattern matching algorithm that we discuss later.

III. The Theory of Transition Systems

We now cast the model checking of $\mathcal{RWB}$ into a DPLL(T) setting, by defining a dedicated *theory of transition systems* ($\mathcal{TS}$). We assume familiarity with the definitions of many-sorted first order logic (see, e.g., [3]). The theory is parameterized by a set $\bar{Q} = \{Q_1, \dots, Q_n\}$ of *state sorts* used to represent the state sets of the program's constituent threads. Let $\bar{Q}^+$ denote the composite state sorts obtained by taking the Cartesian product of one or more elements in Q. Every element $Q \in \bar{Q}^+$ is a sort in $\mathcal{TS}$. Further, every such Q is also associated with a matching transition system sort, S_Q. Finally, $\mathcal{TS}$ has an event sort, E.

For every $Q \in \bar{Q}^+$ the signature includes: the predicate $I_Q : S_Q \times Q$, indicating initial states; the predicates $R_Q, B_Q : S_Q \times Q \times E$ to indicate whether an event is requested (R_Q) or blocked (B_Q) at a given state; and the predicate $Tr_Q : S_Q \times Q \times E \times Q$ to indicate the state transition rules.

In order to reason about composite transition systems, the signature includes the following functions and predicates. For every $Q^1, Q^2 \in \bar{Q}^+$ we have the transition system composition function $\|_{Q^1Q^2}$: $S_{Q^1} \times S_{Q^2} \to S_{Q^1 \times Q^2}$ (Recall that $(Q^1 \times Q^2)$ is itself a sort in $\bar{Q}^+$); and also the $pair_{Q^1Q^2} : Q^1 \times Q^2 \to (Q^1 \times Q^2)$ function for composing states, which, per the $\mathcal{TS}$ semantics, is a bijection. Later we often omit the Q subscripts when clear from the context.

For each $Q^1, Q^2 \in \bar{Q}^+$, $\mathcal{TS}$ has the following axioms which enforce the $\mathcal{RWB}$ composition rules. A composite state is initial iff its components are initial states:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies\\ &\quad \forall q : Q_1 \times Q_2.\ (I(s,q) \iff\\ &\quad \exists q_1 : Q_1, q_2 : Q_2.\ (I(s_1,q_1) \wedge I(s_2,q_2) \wedge q = pair(q_1,q_2))).\end{aligned}$$

Composite transitions are performed component-wise:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies\\ &\quad \forall q, q' : Q^1 \times Q^2, e : E.\ (Tr(s,q,e,q') \iff\\ &\quad \exists q_1, q_1' : Q^1, q_2, q_2' : Q^2.\ (q = pair(q_1,q_2) \wedge\\ &\qquad q' = pair(q_1',q_2') \wedge Tr(s_1,q_1,e,q_1') \wedge Tr(s_2,q_2,e,q_2'))).\end{aligned}$$

Requested and blocked events in a composite state are the union of those in the component states:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies\\ &\quad \forall q : Q^1 \times Q^2, e : E.(R(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2.\\ &\quad q = pair(q_1,q_2) \wedge (R(s_1,q_1,e) \vee R(s_2,q_2,e))) \wedge\\ &\quad (B(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2.\\ &\quad q = pair(q_1,q_2) \wedge (B(s_1,q_1,e) \vee B(s_2,q_2,e))).\end{aligned}$$

As previously discussed, by encoding safety properties as threads of the program to be checked, safety is reduced to deadlock freedom. For each $Q \in \bar{Q}^+$, the signature includes a $deadlock_Q : S_Q \times Q$ predicate, such that:

$$\begin{aligned}&\forall s : S_Q, q : Q.\ (deadlock(s,q) \iff\\ &\quad \neg\exists q' : Q, e : E.\ Tr(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e)),\end{aligned}$$

and the $safe\_state_Q : S_Q \times Q$ predicate, with:

$$\begin{aligned}&\forall s : S_Q, q : Q.\ \neg safe\_state(s,q) \implies deadlock(s,q) \vee\\ &\quad \exists q' : S_Q, e : E.\ (Tr(s,q,e,q') \wedge R(s,q,e) \wedge\\ &\qquad \neg B(s,q,e) \wedge \neg safe\_state(s,q')).\end{aligned}$$

$\neg safe\_state_Q(s,q)$ indicates that state q is unsafe, because it is (or can lead to) a deadlock state. Finally, for each $Q \in \bar{Q}^+$, $safe_Q : S_Q$ indicates that a transition system is safe:

$$\forall s : S_Q.\ \neg safe(s) \iff \exists q : Q.\ I(s,q) \wedge \neg safe\_state(s,q).$$

The Theory Solver. Inputs for the $\mathcal{TS}$ solver start with a *preamble* $\mathfrak{P}$ that contains assertions that describe the program's threads. Specifically, $\mathfrak{P}$ includes variables $s_1 \dots, s_n$, each of sort S_Q for some basic state sort $Q \in \bar{Q}$; and for every s_i it includes assertions describing its initial states, its transitions and its requested and blocked events. After $\mathfrak{P}$, the solver expects an assertion Φ about the system's safety: $s = s_1 \parallel s_2 \parallel s_3 \parallel \dots \parallel s_n \wedge \neg safe(s)$. The solver then returns SAT iff s is determined to be unsafe.

Fig. 2 shows derivation rules used to implement a simple explicit-state model checker.[1] Intuitively, $\mathcal{TS}$ traverses the state graph in a DFS-like manner, looking for bad states. The underlying SAT solver manages the splits by deciding which successor state to check at every point. The process ends when a deadlock state is found or when the state space has been exhausted and no derivation rules apply; an example appears in Fig. 3. As demonstrated in the next section, this implementation allows us to seamlessly leverage other theory solvers in curtailing the state space, which may reduce the overall runtime. Additional details and proofs of correctness and termination appear in Section A of the supplementary material [23].

IV. Automatic Analysis of Transition Systems

The calculus in Section III captures the basic proof strategy of our theory solver: a forward reachability search. We next enrich this basic strategy with additional derivation rules, aimed at narrowing down the state space that needs to be explored. The idea is to include within the $\mathcal{TS}$ solver a database of *structural patterns* that characterize common/useful threads and alongside each pattern also to keep lemmas that describe these threads' behavior in the language of some other theory

[1]While we do not assume the system is finite-state, we do assume that the initial states and the successors for each state are finite and decidable.

$$\textsc{Start}\ \frac{\Gamma[\neg safe(s)]}{\Gamma, \neg safe\_state(s,q_1)\quad \ldots\quad \Gamma, \neg safe\_state(s,q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1,\ldots,q_n \text{ ARE THE INITIAL STATES OF } s\\ \text{AND } \forall_{1\le i\le n}.\ \neg safe\_state(s,q_i) \notin \Gamma\end{array}$$

$$\textsc{Decide}\ \frac{\Gamma[\neg safe\_state(s,q)]}{\Gamma, \neg safe\_state(s,q_1)\quad \ldots\quad \Gamma, \neg safe\_state(s,q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1,\ldots,q_n \text{ ARE THE SUCCESSORS OF } q\ (n \ge 1)\\ \text{AND } \neg deadlock(s,q) \notin \Gamma\end{array}$$

$$\textsc{Unsat}\ \frac{\Gamma[\neg safe\_state(s,q)]}{\bot}\ \text{IF } \forall q'.\ \neg safe\_state(s,q') \in \Gamma \implies \neg deadlock(s,q') \in \Gamma$$

$$\textsc{Deadlock-Lemma} : \mathfrak{P} \wedge \Phi \implies \neg deadlock(s,q)\quad \text{IF}\quad q \text{ HAS A SUCCESSOR IN } s$$

Figure 2: Γ represents an arbitrary set of assertions that the solver has gathered at a given state, and $\Gamma[\phi]$ indicates that ϕ appears in Γ. The *Start* rule starts the traversal of the graph: the solver initiates a forward reachability search for bad states by nondeterministically guessing an initial state that is unsafe. When a state with unvisited successors is asserted to be unsafe, the *Decide* rule is used to nondeterministically assert that one of its successors is unsafe. Splitting is handled through the splitting-on-demand feature of the DPLL(T) framework. The UNSAT rule closes branches that fail to reach a deadlock state. If all branches terminate with $\bot$, UNSAT is returned; otherwise, if a branch terminates with a state other than $\bot$ where no rule is applicable, we return SAT. The last rule, *Deadlock-Lemma*, is a lemma generation rule: the resulting lemma is theory-valid, i.e. does not depend on the context in which it was generated. These lemmas mark that a non-deadlock state has been visited, and that it does not need to be revisited in the future. As part of the proof strategy, the $\mathcal{TS}$ solver invokes the lemma generation rule for (s,q) immediately after the *Decide* rule is invoked for $\neg safe\_state(s,q)$, and only then (provided that q is not a deadlock state). This strategy, together with the side-conditions on the derivation rules, ensures that no state is visited more than once. See Section A of the supplementary material [23] for more details.

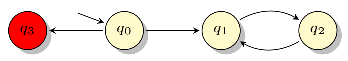


Figure 3: The depicted program has a reachable deadlock state, q_3. After reading the preamble, the solver uses the *Start* rule to assert $\neg safe\_state(s,q_0)$. Then, it invokes the *Decide* rule for state q_0, nondeterministically asserting $\neg safe\_state(s,q_1)$. This invocation of *Decide* is followed by the generation of the lemma $\neg deadlock(s,q_0)$. Next, *Decide* is invoked for state q_1, generating the assertion $\neg safe\_state(s,q_2)$ — followed by the lemma $\neg deadlock(s,q_1)$. *Decide* is then invoked for state q_2, generating $\neg safe\_state(s,q_1)$ and the lemma $\neg deadlock(s,q_2)$. At this point, the conditions for the UNSAT rule are met, and the solver closes this branch of the tree. The solver backtracks to the last nondeterministic split and generates the assertion $\neg safe\_state(s,q_3)$. State q_3 is deadlocked, and so the *Deadlock-Lemma* rule is not invoked. No additional derivation rules apply, and so the process terminates with a SAT result, indicating that the system is unsafe.

in CVC4. As the $\mathcal{TS}$ solver traverses the state space, it also repeatedly checks to see if any of the patterns apply to the threads at hand. When a match is found, the solver asserts the matching lemmas to the SMT framework. Sometimes, these lemmas may be contradictory to the assertion that the safety property is violated along the current search path, and another theory solver will raise a conflict: this will cause the $\mathcal{TS}$ solver to backtrack and check other areas of the state space.

We demonstrate the method on a simple example, adopted from [15]. Observe an $\mathcal{RWB}$ program over event set $E = \{0,1\}$ that generates the event sequence $(0^5 \cdot (0+1))^\omega$. The program has three threads, depicted in Fig. 4. The safety property to be verified is that event 1 is never triggered (and so, the program is unsafe). Observe that direct model checking of this system requires visiting 6 composite states.

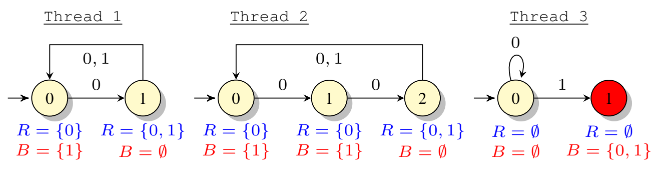


Figure 4: An $\mathcal{RWB}$ program adopted from [15]. Thread 1 counts the number of events triggered so far, modulo 2. Every second event it requests both events 0 and 1; otherwise, it requests 0 but blocks 1. Thread 2 does the same, but counts modulo 3. Thread 3 is a *bad marker*: it waits for a violation to occur, i.e. for 1 to be triggered, and then goes into a "bad" state that blocks all the events, forcing a deadlock. This $\mathcal{RWB}$ program can have 0 triggered at every index, and can have 1 triggered precisely every 6 events. Consequently, it is unsafe.

For this program as input, the $\mathcal{TS}$ solver performs the following automatic compositional proof. First, it compares the transition systems to its pattern bank, and recognizes that they match the *looped* thread mold — a thread whose state is determined uniquely by the step index in the run (assuming a violation has not occurred). This is a structural property of each thread, that is checked locally and in isolation from its siblings. After determining that all threads are looped, the solver finds all individual thread states in which 1 is not blocked. In our case, this is state 1 for thread 1, state 2 for thread 2, and state 0 for thread 3. Denoting composite states as triplets, this is state $\langle 1,2,0\rangle$. Finally, the solver uses the gathered information to generate the following lemma in order to curtail the state space:

$$\begin{aligned}&\mathfrak{P} \wedge \Phi \implies ((\neg safe\_state(s,\langle 0,0,0\rangle) \implies\\ &\quad \neg safe\_state(s,\langle 1,2,0\rangle)) \ \wedge\ \ \exists t:\mathbb{N}.\\ &\quad (t \equiv 1 \pmod 2) \wedge (t \equiv 2 \pmod 3) \wedge (t \equiv 0 \pmod 1))).\end{aligned}$$

This lemma connects the safety of the initial state $\langle 0,0,0\rangle$ with that of the only state in which 1 is not blocked, state $\langle 1,2,0\rangle$ — provided that there exists an integer t for which $t \equiv 1 \pmod 2$, $t \equiv 2 \pmod 3$ and $t \equiv 0 \pmod 1$. Because, in looped threads, the step index determines the state, this last part captures the fact that state $\langle 1,2,0\rangle$ is reachable.

Upon generation of this lemma, CVC4 asserts the lemma's arithmetical clauses to the arithmetic solver. If the latter determines that there is no solution for t, CVC4 answers UNSAT on the entire query. This signifies that the system is safe, which is indeed the case if state $\langle 1,2,0\rangle$ cannot be reached. However, if the arithmetic solver manages to solve for t, as is the case here, the $\mathcal{TS}$ solver continues exploring the successors of state $\langle 1,2,0\rangle$ and discovers that it has a bad successor. Then, SAT is returned for the query.

The key observation is that through the automatically generated lemma, the 4 intermediate states between state $\langle 0,0,0\rangle$ and $\langle 1,2,0\rangle$ did not need to be explored. Because the threads matched the looped pattern, CVC4 was able to deduce that these intermediate states would be safe iff state $\langle 1,2,0\rangle$ was safe. Further, because the arithmetic solver can solve for t more quickly than the intermediate states can be traversed (especially when generalizing to $(0^n \cdot (0+1))^\omega$ for a large n), the solver's performance is improved.

Pattern Matching. The $\mathcal{TS}$ solver's pattern database consists of *pattern matchers*. A pattern matcher P is comprised of a family of *recognizer predicates* $\{R_n\}_{n\ge 1}$, where R_n is defined over n transition system variables $s_1,\ldots,s_n$, and a lemma generating function f (described later). For input system $s = s_1 \parallel \ldots \parallel s_n$, we say that pattern P applies to s if $R_n(s_1, \ldots, s_n)$ evaluates to true. The R_n predicates can encode various facts about the transition systems: e.g., that threads always or never block certain events, that they have a certain state that must always be revisited, that certain events always send threads into a deadlock state, etc. For example, in the previously discussed looped pattern, R_n evaluates to true iff each of the threads' states has precisely one successor state.

In our proof-of-concept C++ implementation, recognizer predicates are coded as Boolean methods that take as input a list (of arbitrary length) of transition systems. Upon receiving a query, the $\mathcal{TS}$ solver passes the input program's threads to the recognizer predicates of each of the patterns, to determine which patterns apply in this case. Recognizer implementations may traverse the given transition systems, compute strongly connected components, etc. The only restriction, needed for the method to be efficient, is that recognizers do not compute the composite transition systems of the system; they are restricted to (polynomial) operations on the individual threads. Thus, the complexity of pattern matching is polynomial in the size of the individual threads — and because these threads are typically exponentially smaller than the composite program [17], we can quickly test multiple patterns.

The second component in a pattern matcher is the lemma generating function, f. When pattern P applies to an input program, its lemma generating function is invoked repeatedly during state space traversal, in order to allow P to generate lemmas that affect the search. Specifically, f is invoked whenever $\mathcal{TS}$ visits a new state q (i.e., after the *Decide* rule generates the assertion $\neg \mathit{safe\_state}(s, q)$), and returns a (possibly empty) list of lemmas concerning the safety of state q. The $\mathcal{TS}$ solver then asserts these lemmas to the underlying SAT engine, and other theories may use them in trimming the search space. In practice, the generated lemmas may depend on parameters extracted from the input threads by the pattern recognizers. For example, in the looped pattern, the size of the loop is extracted by the recognizer and is then used in generated lemmas.

Limitations. The above example demonstrates our method's potential advantages, but also raises a question regarding its generality: can the pattern database be sufficiently extensive, i.e. apply to a sufficient range of programs, so as to make our approach worthwhile? Indeed, if one needed to "teach" the solver new patterns for every new input program, the method would boil down to a manual compositional proof.

We believe that the answer to this question is affirmative: our findings show that even a small set of patterns included within the $\mathcal{TS}$ solver may already apply to broad classes of interesting programs. We demonstrate two such cases, *periodic programs* and *programs with arrays*, in Sections V and VI. Still, adding new patterns is not a trivial task, and so we store them in a central repository — amortizing the cost of adding additional patterns over future applications.

The $\mathcal{TS}$ Solver vs. Model Checking. In the simple example given above, our theory-aided approach could also be implemented by a more standard design: a model checker that issues queries to a black-box SMT solver. Our motivation for conducting model checking within the $\mathcal{TS}$ solver is in handling more elaborate examples, in which SMT theories partake in directing the state space traversal (see, e.g., Section V). While such cases can still be accommodated by a model checker that is "running the show" and an SMT solver that exposes proper callbacks, we feel that a DPLL(T)-based solution is cleaner, and also more extensible and robust. By encoding the state traversal engine as a few axioms and lemma generation rules, and by having the pattern matching mechanism likewise generate lemmas, the complexity of integrating and synchronizing the two is automatically and seamlessly handled by CVC4's DPLL(T) core — simplifying the implementation of the $\mathcal{TS}$ solver. Further, this enables the $\mathcal{TS}$ solver to be plugged into any other SMT solver that adheres to the DPLL(T) framework.

V. Verifying Periodic Programs

In this section, we discuss the theory-aided verification of *periodic programs* [25] — a class of single processor scheduling problems that have been widely studied over the last decades. A periodic program consists of a finite set of *tasks* $T_1, \ldots, T_n$, which are processes that repeatedly need to be scheduled for execution on a single processor. Each task T_i is characterized by its period time P_i and an execution time C_i (for simplicity, we ignore here other parameters such as relative deadlines and initial offsets). From task T_i's point of view, the execution of the program is divided into time cycles of length P_i each, and in each such cycle the task must be alloted C_i time slots on the processor. The least common multiple of the tasks' period times is called the program's *hyper-period*. Tasks may have *priorities*: a task with a higher priority will preempt another if both need to be scheduled at a specific point in time. A periodic program is said to be *schedulable* if there exists a task scheduling in which no deadlines are violated. See Section C of the supplementary material [23] for an example.

Here, we study the verification of safety properties in periodic programs: we assume that the input program is schedulable, and check whether it can violate a given property. This is typically done by transforming the periodic program into an equivalent sequential program and then verifying it using standard model checking [7]. Our approach is similar, but we seek to leverage the program's special structure in order to explore only a portion of its state space.

In $\mathcal{RWB}$, periodic programs may be programmed by expressing each task as a thread that requests an event whenever the task needs to be scheduled [15]. Priorities are expressed using blocking: a thread (task) may block events belonging to other threads with lesser priorities. Fig. 5 illustrates the structure of task threads and describes their pattern matcher.

Whenever all input threads are identified as *tasks*, the pattern recognizer reports that the program is periodic. This causes the pattern's lemma generation function to be repeatedly invoked during state space traversal, so that it may generate lemmas aimed at curtailing the search space. For this pur-

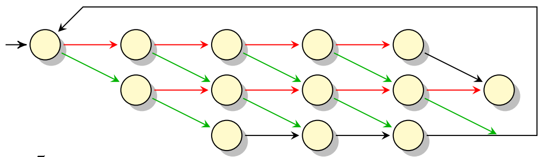


Figure 5: An $\mathcal{RWB}$ implementation of a *task* thread with period time $P = 5$ and execution time $C = 2$. The thread's underlying transition system can be regarded as a $(C+1) \times P$ matrix, where the columns represent the time passed since the beginning of the period and the rows represent the number of times the task has been scheduled so far. Green edges in the figure represent the task being successfully scheduled (i.e., its requested event was triggered) and red edges represent the task not being scheduled (an event requested by some other task was triggered). Thus, with every time unit the state moves to the right, and if the task was scheduled it also moves one row down. If the task's deadline is violated, it enters a deadlock state (the rightmost state in the figure). The task pattern matcher traverses the state graph of each input thread and checks whether it has these structural properties. If so, it also extracts the task's P and C parameters and its sequence of requested events (not illustrated). If blocking is used to prioritize tasks, the matcher also extracts the prioritization hierarchy.

pose, we extend the signature of $\mathcal{TS}$ to include a sort $\mathbb{Z}^+$ for non-negative integers, and the predicate $deadlock_Q : S_Q \times \mathbb{Z}^+$. Intuitively, $deadlock_Q(s,t)$ indicates that a deadlock state in s is reachable in t steps from an initial state. Further, we extend $\mathcal{TS}$ to support backward reachability analysis, in addition to the forward reachability analysis afforded by the *safe_state* predicate. To this end, we add the $reachable_Q : S_Q \times Q$ predicate, with the following semantics:

$$\forall s : S_Q, q : Q.reachable(s,q) \implies I(s,q) \vee \exists q' : S_Q, e : E.$$
$$(Tr(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e) \wedge reachable(s,q'))$$

Intuitively, a state is reachable if it is initial or has a reachable predecessor. For more details, see Section B of the supplementary material [23].

The lemmas generated by the pattern matcher assert that there must be a time t within the hyper-period in which a violation occurs. They also limit the possible values of t based on the information gathered about the individual tasks. Specifically, the pattern matcher generates the lemma:

$$\mathfrak{P} \wedge \Phi \implies \exists t : \mathbb{Z}^+.\ deadlock(s,t) \wedge \Psi_t$$

where Ψ_t describes constraints on t that are deduced from the structure of the task threads. If the arithmetic solver finds a solution t_0 for Ψ_t it assigns it to t, and the $\mathcal{TS}$ solver then translates it, by analyzing the task threads' possible locations in time t, into candidate reachable bad states $q_1, \ldots, q_\ell$:

$$\mathfrak{P}\ \wedge\ \Phi \wedge deadlock(s,t_0) \implies \vee_{i=1}^{\ell} reachable(s,q_i)$$

$\mathcal{TS}$ then performs backward reachability checks on candidates $q_1, \ldots, q_\ell$. If a path to an initial state is found, the system is unsafe and we are done. Otherwise, the contradiction forces the arithmetic solver to propose another solution $t = t_1$, which corresponds to additional candidate bad states. The process is repeated until the system is proven unsafe, or until all possible solutions are exhausted. Other bad states, which do not correspond to any of the proposed values of t, are guaranteed to be unreachable and are ignored.

In order to generate the constraints in Ψ_t, the pattern matcher identifies tasks *participating* in the violation: these are the threads whose requested events are part of a violating sequence. Then, it uses information about these threads, and about threads with higher priority, to put constraints on t.

We demonstrate this on a schedulable periodic program with 4 tasks: task T_1 with parameters $P_1 = 5, C_1 = 1$; T_2 with $P_2 = 6, C_2 = 1$; T_3 with $P_3 = 9, C_3 = 3$; and task T_4 with parameters $P_4 = 11, C_4 = 2$. Task 1 has the highest priority, task 2 has the second highest priority, and tasks 3 and 4 both share the lowest priority. The safety property in question is that it is impossible for task T_4 to be scheduled for three consecutive time slots. Here, direct model checking requires visiting 55000 states in the composite program.

By intersecting the violating event sequence with the events requested by each thread, the pattern matcher determines that T_4 is the only participating task. By the information extracted regarding task priorities, it deduces that tasks T_1 and T_2 supersede it. Then, it generates the Ψ_t constraint as follows. One conjunct in Ψ_t is $0 \leq t \leq 990$, as the hyper-period is $lcm(5,6,9,11) = 990$. Another conjunct is $((t \geq 3 \pmod 5) \wedge (t \geq 3 \pmod 6))$: if it did not hold, T_1 or T_2 would preempt T_4, preventing it from being scheduled 3 consecutive times. Yet another conjunct is $(t \leq 1 \pmod{11})$; it holds because in order for T_4 to be scheduled 3 consecutive times (with execution time $C_4 = 2$), a fresh period must start at time t or $t-1$. A few additional conjuncts are omitted. The complete lemma reduces the number of possible values for t from 990 to just 15, and the query as a whole entails exploring only 700 states out of 55000 reachable states in order to prove the system's safety.

VI. Verifying Programs With Shared Arrays

Next we demonstrate the theory-aided verification of programs with shared arrays — a widespread construct in concurrent programming. In the $\mathcal{RWB}$ model, a shared m-ary array with n cells may be implemented using n b-threads, each of size m. Each thread represents a single array cell and has a clique-like structure, where each state s_i is associated with a write event w_i and a read event r_i. Intuitively, each state s_i corresponds to a value v_i that is stored in the array cell. Whenever event w_i is triggered, the thread moves to state s_i; and whenever not in state s_i, the thread blocks r_i. Thus, other threads can request r_i in order to check if the thread is in state s_i (i.e., to check if the array cell has value v_i). See Fig. 6 for an illustration. Note that this implementation is only needed for shared arrays; internally, threads may use any construct available in the underlying programming language.

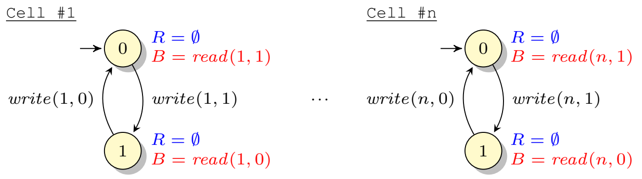


Figure 6: An $\mathcal{RWB}$ implementation of a binary array with n cells. Each cell is represented by a thread with two states, signifying the stored value in that cell, 0 or 1. Each thread/cell is associated with two `write` events, for 0 and 1; when they occur, the thread changes states to indicate the new stored value. Other threads in the program may read from a cell by requesting the two `read` events associated with it, one for 0 and one for 1; the read event that does not match the value in the cell will be blocked by the cell thread, and so only the "correct" read event may be triggered.

The $\mathcal{TS}$ solver has a pattern matcher that looks for threads that match this *array cell* pattern. If an array is found, the pattern matcher checks whether deadlocks are possible only in certain array configurations (e.g., when certain array cells hold certain values; an example appears later in this section). If such constraints are found, it generates a lemma that conditions the system's unsafety on the array threads reaching an unsafe configuration.

We demonstrate with an example. Observe a program with a shared array of size n and an initial state q_0. The array pattern matcher creates an array expression arr_{q_0} whose value at each index i is set to some fresh constant c_i. This expression is used to represent the value of the array in various states of the program. The matcher also creates a *target* array, arr_{target}, and asserts constraints on arr_{target} signifying the state that the array has to be in for a violation to occur. Then, it generates the lemma $\mathfrak{P} \wedge \Phi \implies (arr_{q_0} = arr_{target})$.

The bulk of the work is then performed as $\mathcal{TS}$ traverses the state space. Whenever a new state q is visited, the pattern matcher analyzes the threads (each of them separately), looking for array entries that have become fixed. This can be determined, e.g., when additional write events to a cell are never requested or are always blocked. Suppose that it is discovered that the first cell's value has been fixed to e_0; then the lemma $\mathfrak{P} \wedge \Phi \wedge \neg safe\_state(s, q) \implies (c_0 = e_0)$ is generated. If this is consistent with the earlier assertion $arr_{q_0} = arr_{target}$, the solver continues traversing the successors of q; otherwise, the array theory solver will raise a conflict, resulting in q's successor states not being traversed.

A more detailed example and an evaluation of applying the shared array pattern to a web-server application appears in Section VII. An additional detailed example regarding the verification of an $\mathcal{RWB}$ application for playing Tic-Tac-Toe appears in Section D of the supplementary material [23].

VII. Experimental Results

We evaluated our proof-of-concept tool, implemented as an extension to CVC4, by comparing it to BPMC — a symbolic model checker specifically designed for $\mathcal{RWB}$ programs [19], [22] (the tool and experiments are available online [23]). Our tool uses a portfolio approach: if the input program does not match any of the known patterns, the tool simply invokes BPMC (or any other model checker, for that matter). The decision of whether or not to invoke BPMC is made within seconds, rendering the performance of both tools effectively the same in these cases. Hence, for the remainder of this section we focus on inputs in which a pattern did apply and theory-aided model checking was indeed attempted.

We first compared the tools using a benchmark suite of over 120 hand-crafted $\mathcal{RWB}$ programs — some periodic, and some containing shared arrays. The benchmarks' sizes ranged from a few hundred to over 10 million reachable states, and contained both SAT and UNSAT instances. The results are depicted and disucssed in Fig. 7.

Next, we set out to test our tool's applicability to a large, real-world system by using it to verify safety properties on a web-server (implementing TCP and HTTP stacks) written in BPC [16]. We were very curious to see whether our pattern recognition mechanism would pick up any matching threads.

As it turns out, the shared array pattern proved useful in verifying this application. Per the TCP protocol, the web-server only accepts TCP *push* segments on active connections. Slightly simplified, a connection to a client is active if the client sent a *syn* segment but not a *fin* segment. This functionality is implemented using blocking: for every connection, a dedicated thread, named *EnsureActiveConnection*, blocks *push* events while the connection is inactive. This blocking is removed when a *syn* segment is received, and is restored when a *fin* segment is received. Thus, the *EnsureActiveConnection* threads were picked up as shared array cells by our tool: they each had two states, labeled *active* and *inactive*, with respective read events *push* and *reject* and write events *syn* and *fin*. Interestingly, the programmers of the web-server did not seem to have had this design pattern in mind [16].

We tested 10 safety properties on the web-server (see Fig. 8). These properties included the proper rejection of messages on inactive connections, proper usage of allotted sequence numbers for outgoing segments, and the detection and blocking of unstable clients, who quickly and repeatedly opened and closed connections. The theory-aided approach did better on 7 of 10 instances (4 SATs and 3 UNSATs), demonstrating an average speedup of 16% over all instances. BPMC did better on 2 SAT and 1 UNSAT instances, where the property in question and the discovered patterns were disparate (e.g., properties involving proper usage of sequence numbers, that had nothing to do with the *EnsureActiveConnection* threads).

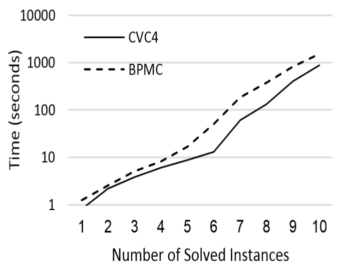


Figure 8: Experiments on the web-server.

These initial results are encouraging. We conclude that (i) the theory-aided approach is viable, in the sense that the stored patterns apply to real programs, sometimes significantly reducing verification times; and (ii) that performance may be further improved by enhancing the portfolio approach; i.e., if we were able to more accurately characterize cases in which, despite matching a stored pattern, a thread does not affect the property in question, we could delegate those cases to BPMC and achieve faster running times. This is left for future work.

VIII. Related Work and Discussion

In this work, we proposed a framework for the automated compositional verification of concurrent software. Our technique was based on casting the model checking problem into the DPLL(T) framework used by the CVC4 SMT solver, and then utilizing other theory solvers to prune the search space in order to improve performance. Other theories were able to affect the search through lemmas in their respective languages that were generated by matching the input program's threads to presupplied patterns.

SMT solving has been used for various verification-related tasks such as lemma dispatching [8], [26], reachability analysis [4] and model-checking concurrent programs [6], [27]. Our technique shares some of these aspects, but differs in that the state exploration is driven by an SMT solver and

		# Instances	Avg. # States Explored			Avg. Time (milliseconds)		
			CVC4	BPMC	Change	CVC4	BPMC	Change
Periodic Programs	SAT	11	9994	9236	+8%	18791	15894	+18%
	UNSAT	50	35299	184388	-80%	10247	15041	-31%
	UNSAT[†]	6	59816	8195666	N/A	170673	809946	N/A
	Timeout	2						
Shared Arrays	SAT	35	24416	293525	-91%	24882	168755	-85%
	UNSAT	15	121133	511292	-76%	124911	292779	-57%
	UNSAT[†]	6	267000	1989666	N/A	359324	1510028	N/A
Total		111	190842	998441	-80%	178831	492469	-63%

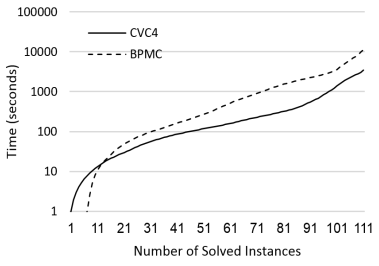


Figure 7: Experiments on a benchmark suite, conducted using an X230 Lenovo laptop with 16GB memory. The suite contained SAT and UNSAT instances of periodic $\mathcal{RWB}$ programs and programs with shared arrays. The table compares our tool (*CVC4* columns) to the *BPMC* tool, measuring the average number of explored states and average solving time for each category. The *Change* columns measure the effectiveness of CVC4 in comparison to BPMC. The UNSAT[†] row indicates UNSAT instances on which CVC4 answered correctly but on which BPMC ran out of space (but listing the number of states it was able to explore). The *Timeout* row indicates instances on which both tools ran out of space/time. We did not encounter examples on which BPMC returned and CVC4 did not. The table reveals that for SAT queries on periodic programs, BPMC was able to outperform CVC4. This is not surprising; indeed, the pattern for periodic programs is designed to quickly show that bad states are unreachable, which is not the case for SAT instances. In all other categories, i.e. UNSAT queries on periodic programs and both types of queries on programs with shared arrays, CVC4 typically outperformed BPMC. Instances where BPMC did better were either very small (the cost of thread analysis and pattern matching exceeded the cost of the actual model checking), or instances where the property in question had nothing to do with the recognized patterns, making it impossible for our tool to trim the search space. The UNSAT[†] instances had too many states for BPMC to cover, but with the theory-aided approach we were able to trim the search space down to a manageable size. Finally, the *Timeout* instances were too large to handle, even with theory-aided pruning. The *Total* row sums up the instances solved by both tools, demonstrating an encouraging average speedup of 63%; these 111 instances are also the ones described in the graph.

in that lemmas are derived using stored patterns. A related approach for circuit verification appears in [5], where the input is analyzed to find unreachable states in advance. Our framework follows a similar spirit, but extends the technique to concurrent software and utilizies a modern SMT solver.

In [30], the authors extend the Z3 solver with an automaton sort for symbolic automata over infinite alphabets. It would be interesting to combine this technique with ours, enabling it to reason about $\mathcal{RWB}$ programs with infinite event sets.

We evaluated our technique on two broad classes of $\mathcal{RWB}$ programs: periodic programs and programs with shared arrays. Specifically, we showed how the $\mathcal{TS}$ solver may leverage CVC4's arithmetic and array theory solvers in order to expedite the model checking process. Others have explored SMT-based techniques for similar models; e.g., the validation of guessed invariants in Lustre programs [21]. We consider this as encouragement that applying SMT-based techniques to synchronous, discrete event models may prove fruitful, and intend to extend our technique to Lustre as well.

We find our initial results encouraging, and plan to continue extending our pattern database. One direction that we are presently pursuing is the addition of a new pattern matcher that leverages CVC4's *string* theory solver [24], by translating constraints imposed by certain types of input threads into regular expressions. Indeed, a prototype implementation we have created shows interesting potential.

Acknowledgements. The research of Katz and Harel was supported by a grant from the Israel Science Foundation, and by the Philip M. Klutznick Fund for Research, the Joachimowicz Fund and the Benoziyo Fund for the Advancement of Science at the Weizmann Institute of Science.

References

[1] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. *CAV*, 2011.
[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. *LPAR*, 2006.
[3] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. *Handbook of Satisfiability*, 2009.
[4] J. Berdine, N. Bjørner, S. Ishtiaq, J. Kriener, and C. Wintersteiger. Resourceful Reachability as HORN-LA. *LPAR*, 2013.
[5] P. Bjesse and K. Claessen. SAT-Based Verification without State Space Traversal. *FMCAD*, 2000.
[6] R. Bryant, S. Lahiri, and S. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. *CAV*, 2002.
[7] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman. Compositional Seqentialization of Periodic Programs. *VMCAI*, 2013.
[8] A. Cimatti and A. Griggio. Software Model Checking via IC3. *CAV*, 2012.
[9] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking Up is Hard to do: an Investigation of Decomposition for Assume-Guarantee Reasoning. *ISSTA*, 2006.
[10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.
[11] M. Dwyer, J. Hatcliff, R. Robby, C. Pasareanu, and W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. *ICSE*, 2007.
[12] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *CSUR*, 2003.
[13] O. Grumberg and D. Long. Model Checking and Modular Verification. *TOPLAS*, 1994.
[14] D. Harel, A. Kantor, and G. Katz. Relaxing Synchronization Constraints in Behavioral Programs. *LPAR*, 2013.
[15] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. *EMSOFT*, 2013.
[16] D. Harel and G. Katz. Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. *AGERE*, 2014.
[17] D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming. *CONCUR*, 2015.
[18] D. Harel, G. Katz, A. Marron, and G. Weiss. The Effect of Concurrent Programming Idioms on Verification. *MODELSWARD*, 2015.
[19] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. *EMSOFT*, 2011.
[20] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *C. ACM*, 2012.
[21] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-Based Invariant Discovery. *NFM*, 2011.
[22] G. Katz. On Module-Based Abstraction and Repair of Behavioral Programs. *LPAR*, 2013.
[23] G. Katz, C. Barrett, and D. Harel. Theory-Aided Model Checking of Concurrent Transition Systems: Supplementary Material. https://sites.google.com/site/guykatzhomepage/fmcad15.
[24] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. *CAV*, 2014.
[25] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 1973.
[26] K. McMillan. Lazy Annotation Revisited. *CAV*, 2014.
[27] L. D. Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. *CAV*, 2004.
[28] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 2006.
[29] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. on Control and Optimization*, 1987.
[30] M. Veanes and N. Bjørner. Symbolic Automata: The Toolkit. *TACAS*, 2012.

Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays

Anvesh Komuravelli
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Nikolaj Bjørner
Microsoft Research
Redmond, WA, USA

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

Kenneth L. McMillan
Microsoft Research
Redmond, WA, USA

***Abstract*—We present a compositional SMT-based algorithm for safety of procedural C programs that takes the heap into consideration as well. Existing SMT-based approaches are either largely restricted to handling linear arithmetic operations and properties, or are non-compositional. We use Constrained Horn Clauses (CHCs) to represent the verification conditions where the memory operations are modeled using the extensional theory of arrays (ARR). First, we describe an exponential time quantifier elimination (QE) algorithm for ARR which can introduce new quantifiers of the index and value sorts. Second, we adapt the QE algorithm to efficiently obtain under-approximations using models, resulting in a polynomial time Model Based Projection (MBP) algorithm. Third, we integrate the MBP algorithm into the framework of compositional reasoning of procedural programs using may and must summaries recently proposed by us. Our solutions to the CHCs are currently restricted to quantifier-free formulas. Finally, we describe our practical experience over SV-COMP'15 benchmarks using an implementation in the tool SPACER.**

I. INTRODUCTION

Under-approximating a projection (i.e., existential quantification), for example in computing an image, is a key aspect of many techniques of symbolic model checking. A typical (though not ubiquitous) approach to this is what we call *Model-based Projection* (MBP) [17]: we generalize a particular point in the space of the image (obtained using a model) to a subset of the image that contains it. In some cases, the purpose is to compute the exact image by a series of under-approximations [12]. In other cases, such as IC3 [6], the purpose of MBP is to produce a relevant proof sub-goal. When the number of possible generalizations is finite, we say that we have a *finite MBP* which allows us to compute the exact image by iterative sampling, or to guarantee that the branching in our proof search is finite.

The feasibility of a finite MBP depends on the underlying logical theory. Finite MBPs exist for propositional logic [12], [16] and Linear Integer Arithmetic (LIA) with a divisibility predicate [17], and have been applied in both hardware and software model checking. LIA is often adequate for software verification, provided that heap and array accesses can be eliminated. This can be done by abstraction, or by inlining all procedures and performing compiler optimizations to lower memory into registers (e.g., [2], [15]). However, the inlining approach has many drawbacks. It can expand the program size exponentially, it cannot handle recursion, and it is not always feasible to eliminate heap and array accesses.

We address this issue here by considering the problem of MBP for the extensional theory of arrays (ARR). We find that a finite MBP exists that can be computed in polynomial time when only array-valued variables are projected. Projecting variables of index and value sorts is not always possible, since the quantifier-free fragments of the theory combinations are not guaranteed to be closed under projection. We therefore take a pragmatic approach to MBP that may not always converge to the exact projection. This allows us to handle, for example, the combination of ARR and LIA.

We test the effectiveness of this approach using the model checking framework of SPACER [17]. This SMT-based framework makes use of MBP to produce proof sub-goals for Hoare-style procedure-modular proofs of recursive programs. The ability to reason with ARR makes it possible to handle heap-allocating programs without inlining procedures, as the heap can be faithfully modeled using ARR [14]. This leads to significant improvements in scalability, when compared to the use of LIA alone with inlining, as measured using benchmark programs from the 2015 Software Verification Competition (SVCOMP 2015) [4]. Not inlining the programs also has the advantage that we generate procedure-modular proofs (containing procedure summaries) that might be reusable in various ways (e.g., [11]).

In summary, we (a) describe an exponential rewriting procedure for projecting array variables (Sec. III-A), (b) adapt this procedure to obtain a polynomial-time (per model) finite MBP for projecting array variables (Sec. III-B), (c) integrate this with existing MBP procedures for Linear Arithmetic (Sec. III-C) in the SPACER framework obtaining a new compositional proof search algorithm (Sec. IV), and (d) evaluate the algorithm experimentally using SVCOMP benchmarks (Sec. V).

II. PRELIMINARIES

We consider a first-order language with equality whose signature $\mathcal{S}$ contains basic sorts (e.g., `bool` of Booleans, `int` of integers, etc.) and array sorts. An array sort $\texttt{arr}(I, V)$ is parameterized by a sort of indices I and a sort of values V.

This material is based upon work funded and supported by by NASA Contract No. NNX14AI09G and the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002442.

We assume that I is always a basic sort. For every array sort $\texttt{arr}(I, V)$, the language has the usual function symbols $rd : \texttt{arr}(I, V) \times I \to V$ and $wr : \texttt{arr}(I, V) \times I \times V \to \texttt{arr}(I, V)$ for reading from and writing to the array. Intuitively, $rd(a, i)$ denotes the value stored in the array a at the index i and $wr(a, i, v)$ denotes the array obtained from a by replacing the value at the index i by v. We use the following axioms for the extensional theory of arrays (ARR):

Read-after-write

$\forall a : \texttt{arr}(I, V)\ \forall i, j : I\ \forall v : V$

$$\begin{aligned}&(i = j \implies rd(wr(a, i, v), j) = v) \wedge\\ &(i \neq j \implies rd(wr(a, i, v), j) = rd(a, j))\end{aligned}$$

Extensionality

$\forall a, b : \texttt{arr}(I, V) \cdot (\forall i : I \cdot rd(a, i) = rd(b, i)) \implies a = b$

Intuitively, the first schema says that after modifying an array a at index i, a read results in the new value at index i and $rd(a, j)$ at every other index j. The second schema says that if two arrays agree on the values at every index location, the arrays are equal. We use an over-bar to denote a vector. We write $\overline{x} : S$ to denote that every term in vector $\overline{x}$ has sort S, $\overline{x}(k)$ to denote the kth component of $\overline{x}$, and $y \in \overline{x}$ to denote that y is equal to some component of $\overline{x}$, i.e., $\bigvee_{k=1}^{|\overline{x}|} y = \overline{x}(k)$. Let $\overline{i} : I$ and $\overline{v} : V$ be vectors of index and value terms of the same length m. We write $wr(a, \overline{i}, \overline{v})$ to denote $wr(wr(\ldots wr(a, \overline{i}(0), \overline{v}(0)) \ldots), \overline{i}(m), \overline{v}(m))$. Unless specified otherwise, $\mathcal{S}$ contains no other symbols.

For arrays a and b of sort $\texttt{arr}(I, V)$, and a (possibly empty) vector of index terms $\overline{i}$, we write $a =_{\overline{i}} b$ to denote $\forall j : I \cdot \big(j \notin \overline{i} \implies rd(a, j) = rd(b, j)\big)$ and call such formulas *partial equalities* [20]. Using extensionality, one can easily show the following

$$a =_{\emptyset} b \equiv a = b \tag{1}$$

$$wr(a, j, v) =_{\overline{i}} b \equiv \begin{array}{l}\big(j \in \overline{i} \wedge a =_{\overline{i}} b\big) \vee \\ \Big(j \notin \overline{i} \wedge a =_{\overline{i}, j} b \wedge rd(b, j) = v\Big)\end{array} \tag{2}$$

$$a =_{\overline{i}} b \equiv \exists \overline{v} : V \cdot a = wr(b, \overline{i}, \overline{v}) \tag{3}$$

We write $\varphi(\overline{x})$ for a formula φ with free variables $\overline{x}$, and we treat ϕ as a predicate over $\overline{x}$. We also write $\varphi[t]$ to to indicate that a term or formula t occurs in φ at some syntactic position.

Given formulas $\varphi_A(\overline{x}, \overline{z})$ and $\varphi_B(\overline{y}, \overline{z})$ with $\overline{x} \cap \overline{y} = \emptyset$ and $\varphi_A \implies \varphi_B$, a Craig Interpolant [7], denoted $\textsc{Itp}(\varphi_A, \varphi_B)$, is a formula $\varphi_I(\overline{z})$ such that $\varphi_A \implies \varphi_I$ and $\varphi_I \implies \varphi_B$.

III. QE and MBP for the theory ARR

By projection of a variable we mean elimination of an existential quantifier. Consider a formula φ of the form $\exists \overline{x} \cdot \varphi_{qf}(\overline{x}, \overline{y})$ where φ_{qf} is quantifier-free. The problem of *quantifier elimination* (QE) in φ is to find a logically equivalent quantifier-free formula $\psi(\overline{y})$. In this case, we say that ψ is the result of projecting $\overline{x}$ in φ_{qf}.

A *model-based projection* (MBP) for φ is an operator *Proj* that takes a model M of φ_{qf} and returns a quantifier-free formula $\psi_M(\overline{y})$ such that $M \models \psi_M$ and ψ_M entails φ. The operator *Proj* is a *finite MBP* if its image is finite up to logical equivalence (that is, over all models we obtain only finitely many semantically distinct formulas).[1] In this case, we obtain the exact projection as the disjunction of the image of *Proj*. We will refer to *Proj*(M) as a *generalization* of M.

In some cases, there is a trivial approach to MBP that we will call the *substitution approach*. We simply substitute for each variable x in φ a constant that is equal to x in the given model M (for example, a numeric literal). This approach was taken for propositional logic by Ganai et al. [12]. For theories that admit models of unbounded size (e.g., LIA), however, this does not yield a finite MBP, as the number of distinct generalizations we obtain can be infinite.

Instead, we can take the approach used for Linear Real Arithmetic and LIA in our earlier work [17]. Suppose that for the given theory we have a QE procedure that produces a formula with an exponential (or higher) number of disjunctions. We can adapt this procedure to an MBP by always choosing just one disjunct that is true in the given model M. The result may be a procedure that is polynomial for any given model, though the number of distinct generalizations is exponential. We will show how to apply this idea for the projection of array-valued variables in the theory of arrays ARR. When combining this theory with LIA, we will find that some variables of index and value sorts must be eliminated by the substitution method, which gives us a useful MBP but not necessarily a *finite* MBP.

A. *Quantifier elimination for ARR*

Consider an existentially quantified formula $\exists a : \texttt{arr}(I, V) \cdot \varphi$ where φ is quantifier-free. We restrict our discussion to infinite interpretations of I. While we cannot always obtain an equivalent quantifier-free formula, our objective here is to obtain an equivalent existentially quantified formula where every quantifier (if any) is of the sort V.

```
ArrayQE($\exists a \cdot \varphi$)
1   $\varphi_1 \leftarrow (\textsc{ElimWr}^*)(\exists a \cdot \varphi)$
2   $\varphi_2 \leftarrow (\textsc{CaseSplitEq}^*; \textsc{FactorRd}^*)(\varphi_1)$
3   $(\bigvee_{k=1}^{n} \delta_k) \leftarrow \textsc{LiftEqDiseqRd}(\varphi_2)$
4   for $k \in [1, n]$ do
5       $\psi_k \leftarrow (\textsc{ElimEq}; \textsc{ElimDiseq}; \textsc{Ackermann})(\delta_k)$
6   return $\bigvee_{k=1}^{n} \psi_k$
```

Algorithm 1: QE for $\exists a \cdot \varphi$, where a is an array variable.

Our algorithm is inspired by the decision procedure for the quantifier-free fragment of ARR by Stump *et al.* [20]. At a high level, the QE algorithm proceeds in 3 steps: (i) eliminate write terms using the read-after-write axiom schema and partial equalities over arrays, (ii) eliminate (partial) equalities and disequalities over arrays, and (iii) eliminate read terms over arrays. Alg. 1 shows the pseudo-code for our QE algorithm ArrayQE using the rewrite rules in Fig. 1, 2, and 3. Each rule rewrites the formula above the line to the logically equivalent formula below the line. We use regular expression notation

[1] MBP as defined in [17] corresponds to *finite MBP* here.

$$\textsc{ElimWrRd}\ \frac{\varphi[rd(wr(t,i,v),j)]}{(i=j\wedge\varphi[v])\vee(i\neq j\wedge\varphi[rd(t,j)])}\qquad \textsc{ElimWrEq}\ \frac{\varphi[wr(t_1,j,v)=_{\bar{i}}t_2]}{\begin{array}{c}(j\in\bar{i}\wedge\varphi[t_1=_{\bar{i}}t_2])\ \vee\\ \left(j\notin\bar{i}\wedge\varphi[t_1=_{\bar{i},j}t_2\wedge v=rd(t_2,j)]\right)\end{array}}$$

$$\textsc{PartialEq}\ \frac{\varphi[t_1=t_2]}{\varphi[t_1=_{\emptyset}t_2]}\ t_i\text{'s have array sort}\qquad \textsc{TrivEq}\ \frac{\varphi[t=_{\bar{i}}t]}{\varphi[\top]}\qquad \textsc{Symm}\ \frac{\varphi[t_1=_{\bar{i}}t_2]}{\varphi[t_2=_{\bar{i}}t_1]}\ \begin{array}{l}t_2\text{ is a write term}\\ \text{but }t_1\text{ is not}\end{array}$$

$$\textsc{ElimWr} = (\textsc{ElimWrRd}\mid\textsc{ElimWrEq}\mid\textsc{PartialEq}\mid\textsc{TrivEq}\mid\textsc{Symm})$$

Fig. 1: Rewriting rules to eliminate write terms. ElimWr denotes one of the rules chosen non-deterministically.

$$\textsc{CaseSplitEq}\ \frac{\exists a\cdot\varphi[a=_{\bar{i}}t]}{\exists a\cdot((a=_{\bar{i}}t\wedge\varphi[\top])\vee(\neg(a=_{\bar{i}}t)\wedge\varphi[\bot]))}\qquad \textsc{FactorRd}\ \frac{\exists a\cdot\varphi[rd(a,t)]}{\exists a,s\cdot(\varphi[s]\wedge s=rd(a,t))}\ \begin{array}{l}s\text{ is fresh, }t\text{ does not}\\ \text{contain array terms}\end{array}$$

Fig. 2: Rewriting rules to factor out equalities and read terms on the quantified array variable.

$$\textsc{ElimEq}\ \frac{\exists a\cdot(a=_{\bar{i}}t\wedge\varphi)}{\exists\bar{v}\cdot\varphi[wr(t,\bar{i},\bar{v})/a]}$$

where a does not appear in t and $\bar{v}$ denotes fresh variables

$$\textsc{ElimDiseq}\ \frac{\exists a\cdot\left(\varphi\wedge\bigwedge_{k=1}^{m}\neg(a=_{\bar{i}_k}t_k)\right)}{\exists a\cdot\varphi}$$

where $m\in\mathbb{N}$, a does not appear in any t_k, and a appears in φ only in read terms over a

$$\textsc{Ackermann}\ \frac{\exists a\cdot\left(\varphi\wedge\bigwedge_{k=1}^{m}s_k=rd(a,t_k)\right)}{\varphi\wedge\bigwedge_{1\le k<\ell\le m}(t_k=t_\ell\implies s_k=s_\ell)}$$

where $m\in\mathbb{N}$ and a does not appear in φ, s_k's, or t_k's

Fig. 3: Rewriting rules for QE of arrays.

to express sequences of rewrites. In particular, Kleene star applied to a rule denotes the rule's application to a fixed point.

Line 1 of ArrayQE eliminates write terms using the rewrite rules in Fig. 1. Here ElimWr denotes a rule in Fig. 1 chosen non-deterministically. ElimWrRd rewrites terms using the read-after-write axiom and ElimWrEq rewrites partial equalities using Eq. (2). PartialEq converts equalities into partial equalities using Eq. (1). TrivEq eliminates trivial partial equalities with identical arguments and Symm ensures that write terms on the r.h.s. of equalities are also eliminated.

Line 2 of ArrayQE rewrites the formula by case-splitting on partial equalities on the array quantifier a (via CaseSplitEq) followed by factoring out read terms over a by introducing new quantifiers of sort V (via FactorRd). Note that, as presented, these two rules are not terminating as the partial equalities and read terms are preserved in the conclusion of the rules. However, one can easily ensure that a given partial equality or read term is considered exactly once by first computing the set of all partial equalities and read terms in the formula and processing them in a sequential order. The details are straightforward and are left to the reader.

LiftEqDiseqRd on line 3 of ArrayQE performs Boolean rewriting and returns an equivalent disjunction such that in every disjunct, the partial equalities, array disequalities, and equalities over read terms appear at the end as conjuncts, in that order. For each disjunct, line 5 applies the rules in Fig. 3 to eliminate the array quantifier a. ElimEq obtains a substitution term for a using the equivalence in Eq. (3). ElimDiseq is applicable when the disjunct contains no partial equalities and given that the domain of interpretation of I is infinite, one can always satisfy the disequalities and hence, they can simply be dropped. Ackermann performs the Ackermann reduction [1] to eliminate the read terms.

Note that while the rewrite rules are applicable to all array terms and equalities in the original formula, in practice, we only need to apply them to eliminate the relevant terms containing the array quantifier a. See Fig. 4 for an illustration of ArrayQE on an example.

Correctness and Complexity. We can show the following properties of ArrayQE.

Theorem *1:* ArrayQE$(\exists a:\texttt{arr}(I,V)\cdot\varphi)$ returns $\exists\bar{v}:V\cdot\rho$, where ρ is quantifier-free and $\exists\bar{v}\cdot\rho\equiv\exists a\cdot\varphi$.

Theorem *2:* ArrayQE$(\exists a\cdot\varphi)$ terminates in time exponential in the size of φ.

$$\begin{aligned}
&\exists a \cdot (b = wr(a,i_1,v_1) \vee (rd(wr(a,i_2,v_2),i_3) > 5 \wedge rd(a,i_4) > 0))\\
&\equiv \exists a \cdot \begin{array}{l}(\boldsymbol{i_2 = i_3} \wedge (b = wr(a,i_1,v_1) \vee (\boldsymbol{v_2 > 5} \wedge rd(a,i_4) > 0))) \vee\\ (\boldsymbol{i_2 \neq i_3} \wedge (b = wr(a,i_1,v_1) \vee (\boldsymbol{rd(a,i_3) > 5} \wedge rd(a,i_4) > 0)))\end{array} && \{\textsc{ElimWrRd}\}\\
&\equiv \exists a \cdot \begin{array}{l}(i_2 = i_3 \wedge ((\boldsymbol{a =_{i_1} b \wedge rd(b,i_1) = v_1}) \vee (v_2 > 5 \wedge rd(a,i_4) > 0))) \vee\\ (i_2 \neq i_3 \wedge ((\boldsymbol{a =_{i_1} b \wedge rd(b,i_1) = v_1}) \vee (rd(a,i_3) > 5 \wedge rd(a,i_4) > 0)))\end{array} && \{\textsc{PartialEq}; \textsc{ElimWrEq}\}\\
&\equiv \exists a \cdot \begin{array}{l}\left(\boldsymbol{a =_{i_1} b} \wedge \begin{array}{l}(i_2 = i_3 \wedge (rd(b,i_1) = v_1 \vee (v_2 > 5 \wedge rd(a,i_4) > 0))) \vee\\ (i_2 \neq i_3 \wedge (rd(b,i_1) = v_1 \vee (rd(a,i_3) > 5 \wedge rd(a,i_4) > 0)))\end{array}\right) \vee\\ \left(\boldsymbol{\neg(a =_{i_1} b)} \wedge \begin{array}{l}(i_2 = i_3 \wedge (v_2 > 5 \wedge rd(a,i_4) > 0)) \vee\\ (i_2 \neq i_3 \wedge (rd(a,i_3) > 5 \wedge rd(a,i_4) > 0))\end{array}\right)\end{array} && \{\textsc{CaseSplitEq}\}\\
&\equiv \exists a, s_3, s_4 \cdot \begin{array}{l}\left(\left(a =_{i_1} b \wedge \underbrace{\begin{array}{l}(i_2 = i_3 \wedge (rd(b,i_1) = v_1 \vee (v_2 > 5 \wedge \boldsymbol{s_4} > 0))) \vee\\ (i_2 \neq i_3 \wedge (rd(b,i_1) = v_1 \vee (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0)))\end{array}}_{\varphi_1}\right) \vee\right.\\ \left.\left(\neg(a =_{i_1} b) \wedge \underbrace{\begin{array}{l}(i_2 = i_3 \wedge (v_2 > 5 \wedge \boldsymbol{s_4} > 0)) \vee\\ (i_2 \neq i_3 \wedge (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0))\end{array}}_{\varphi_2}\right)\right)\\ \wedge \boldsymbol{s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)}\end{array} && \{\textsc{FactorRd}\}\\
&\equiv \exists a, s_3, s_4 \cdot \begin{array}{l}(\varphi_1 \wedge \boldsymbol{a =_{i_1} b \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)}) \vee\\ (\varphi_2 \wedge \boldsymbol{\neg(a =_{i_1} b) \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)})\end{array} && \{\textsc{LiftEqDiseqRd}\}\\
&\equiv \exists v, s_3, s_4 \cdot (\varphi_1 \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4))\,[\boldsymbol{wr(b,i_1,v)/a}] \vee && \{\textsc{ElimEq}\}\\
&\quad\ \exists a, s_3, s_4 \cdot (\varphi_2 \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)) && \{\textsc{ElimDiseq}\}\\
&\equiv \exists v, s_3, s_4 \cdot (\varphi_1 \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4))\,[wr(b,i_1,v)/a] \vee\\
&\quad\ \exists s_3, s_4 \cdot (\varphi_2 \wedge \boldsymbol{(i_3 = i_4 \implies s_3 = s_4)}) && \{\textsc{Ackermann}\}
\end{aligned}$$

Fig. 4: Illustrating ArrayQE on an example.

B. Model Based Projection

In this section, we will assume that for a satisfiable formula we can obtain a finite representation of a model of the formula and that we can effectively evaluate the truth of any formula in this model. This is possible for ARR and its combinations with LIA and propositional logic. The ability to evaluate allows us to strengthen a formula in a way that preserves a given model. Suppose we have a formula $\varphi[\psi_1 \vee \psi_2]$ with model M, where the sub-formula $\psi_1 \vee \psi_2$ occurs positively (under an even number of negations) in φ. If we also have $M \models \psi_1$, then $M \models \varphi[\psi_1]$ and clearly, $\varphi[\psi_1]$ entails φ. This gives us a way to eliminate a disjunction while preserving a given model and maintaining an under-approximation. If neither ψ_1 nor ψ_2 is true in M, we can similarly replace φ with $\varphi[\bot]$. These transformations are expressed as MBP rewrite rules in Fig. 5.

For each QE rule R, we can produce a corresponding under-approximate rule R_M that preserves model M. This rule can be written R ; (MbpLeft | MbpRight | MbpVac)*. In practice, we can choose to only apply the MBP rules to disjunctions introduced by the QE rules and not to those originally occurring in φ. Correspondingly, we can convert our QE algorithm ArrayQE to ArrayQE$_M$ by replacing each rule R with R_M. We can then obtain an MBP ArrayMBP$(\varphi)(M)$ = ArrayQE$_M(\varphi)$ and we can show the following:

Theorem *3:* For any quantifier-free formula φ in ARR, ArrayMBP$(\exists a : \texttt{arr}(I,V).\ \varphi)$ is a finite MBP.

The fact that it is an MBP can be easily shown by induction on the number of rewrites applied. The fact that it is finite derives from the fact that there are only finitely many ways to resolve the disjunctions in the QE result.

Moreover, assuming that the evaluation of a formula in a model can be done in polynomial time, we can evaluate ArrayMBP$(\varphi)(M)$ in time that is polynomial in the size of M and the size of φ. This is because we can polynomially bound the number of times each rule R_M applies, and each rule can only expand the formula size by a constant amount. Fig. 6 shows an example of applying ArrayMBP.

C. MBP for ARR+LIA

We now consider the combination of the ARR and LIA theories. Assume that the only basic sorts are `bool` and `int`. Furthermore, we only consider linear functions over `int` along with a divisibility predicate (with constant divisors). We developed a finite MBP for LIA in a previous work [17] (call it LiaMBP). When the index sort I is `int`, one can obtain a more efficient MBP with a slight modification of Ackermann$_M$ (for eliminating array read terms) that utilizes the predicate symbol $<$. Given a model M of the formula,

$$\begin{aligned}
&\exists a \cdot (b = wr(a,i_1,v_1) \vee (rd(wr(a,i_2,v_2),i_3) > 5 \wedge rd(a,i_4) > 0)) \\
&\quad \Leftarrow\ \exists a \cdot (\boldsymbol{i_2 \neq i_3} \wedge (b = wr(a,i_1,v_1) \vee (\boldsymbol{rd(a,i_3) > 5} \wedge rd(a,i_4) > 0))) && \{\textsc{WrRd}_M, M \models i_2 \neq i_3\} \\
&\quad \Leftarrow\ \exists a \cdot (i_2 \neq i_3 \wedge ((\boldsymbol{a =_{i_1} b \wedge rd(b,i_1) = v_1}) \vee (rd(a,i_3) > 5 \wedge rd(a,i_4) > 0))) && \{\textsc{PartialEq}; \textsc{WrEq}_M\} \\
&\quad \Leftarrow\ \exists a \cdot \boldsymbol{\neg(a =_{i_1} b)} \wedge i_2 \neq i_3 \wedge (rd(a,i_3) > 5 \wedge rd(a,i_4) > 0) && \{\textsc{CaseEq}_M, M \not\models a =_{i_1} b\} \\
&\quad \Leftarrow\ \exists a, s_3, s_4 \cdot \left(\neg(a =_{i_1} b) \wedge \underbrace{i_2 \neq i_3 \wedge (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0)}_{\varphi_2} \right. && \{\textsc{FactorRd}\} \\
&\qquad\qquad \left. \wedge\, \boldsymbol{s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)} \right) \\
&\quad \Leftarrow\ \exists a, s_3, s_4 \cdot (\varphi_2 \wedge \boldsymbol{\neg(a =_{i_1} b) \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)}) && \{\textsc{LiftEqDiseqRd}\} \\
&\quad \Leftarrow\ \exists a, s_3, s_4 \cdot (\varphi_2 \wedge s_3 = rd(a,i_3) \wedge s_4 = rd(a,i_4)) && \{\textsc{ElimDiseq}\} \\
&\quad \Leftarrow\ \exists s_3, s_4 \cdot (\varphi_2 \wedge (\boldsymbol{i_3 = i_4 \wedge s_3 = s_4})) && \{\textsc{Ack}_M, M \models i_3 = i_4\}
\end{aligned}$$

Fig. 6: Illustrating ArrayMBP on the example of Fig. 4 with a given model M.

$$\textsc{MbpLeft}\ \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi, \psi_1}{\varphi[\psi_1]}$$

$$\textsc{MbpRight}\ \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi, \psi_2}{\varphi[\psi_2]}$$

$$\textsc{MbpVac}\ \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi \qquad M \not\models \psi_1, \psi_2}{\varphi[\bot]}$$

Fig. 5: MBP rules for formulas in negation-normal form.

one can first partition the set of index terms t_k's according to their interpretations in M and choose a representative for each equivalence class. Then, the conjunction in the result of the rule is modified as follows: (a) for every equivalence class, add the equality $t_k = t_\ell$ for every non-representative t_ℓ, where t_k is the representative, (b) linearly order the representatives and add the corresponding inequalities. The modified rule (and hence, the resulting MBP) is linear in time and space.

However, the combination of arrays and integers introduces terms over the combined signature which need to be handled as well. For example, there is no equivalent quantifier-free formula for $\exists i : \texttt{int} \cdot rd(a, i) > 0$. This implies that there does not exist a finite MBP for the combination of LIA and ARR. In the example, the only way to under-approximate the quantification is to use the substitution method, replacing i with its interpretation in a model $M \models rd(a, i) > 0$ as a numeric literal.

Based on the above observations, we obtain an MBP for ARR+LIA as follows. First, we apply ArrayMBP, using the modified Ackermann$_M$ above, to eliminate array quantifiers. Then, we use LiaMBP to eliminate integer quantifiers that do not appear in any array term. Finally, we use the substitution method to eliminate any remaining integer quantifiers. When the last step of substitution method is not necessary, the resulting MBP will be finite.

IV. The Compositional Verification Framework

MBP plays a crucial role in enabling the search for compositional proofs. In this section, we will consider the role played by MBP in a model checking framework called Spacer [17]. In this framework, MBP is used to create succinct localized proof sub-goals that make it possible to reason about only one procedure at a time. The proof goals take the form of under-approximate summaries, either of the calling context of a procedure or of the procedure itself. Without some form of projection, Spacer would not be compositional, as it would build up formulas of exponential size, in effect inlining procedures to create bounded model checking formulas.

A. *Modeling programs with CHCs*

Spacer checks safety of procedural programs by reducing the problem to SMT of a special kind of formulas known as *Constrained Horn Clauses* (CHCs) [5], [17], [14]. We augment the signature $\mathcal{S}$ with a set of fresh predicate symbols $\mathcal{P}$. A *Constrained Horn Clause* (CHC) is a formula of the form

$$\forall \overline{x} \cdot \underbrace{\bigwedge_{k=1}^{m} P_k(\overline{x}_k) \wedge \varphi(\overline{x})}_{body} \implies head$$

where for each k, P_k is a symbol in $\mathcal{P}$, $\overline{x}_k \subseteq \overline{x}$ and $|\overline{x}_k|$ is equal to the arity of P_k. The constraint φ is a formula over $\mathcal{S}$, and *head* is either an application of a predicate in $\mathcal{P}$ or another formula over $\mathcal{S}$. We use *body* to refer to the antecedent of the CHC, as shown above. A CHC is called a *query* if *head* is a formula over $\mathcal{S}$ and otherwise, it is called a *rule*. If $m \leq 1$ in the body, the CHC is *linear* and is *non-linear* otherwise. Following the convention of logic programming literature, we also write the above CHC as $head \leftarrow P_1(\overline{x}_1), \ldots, P_m(\overline{x}_m), \varphi(\overline{x})$.

Intuitively, each predicate symbol P_k represents an unknown partial correctness specification of a procedure (that is, an over-approximate summary). A query defines a property to be proved, while each rule gives modular verification condition for one procedure. A satisfying assignment to the symbols P_k

is thus a certificate that the program satisfies its specification and corresponds to the annotations in a Floyd/Hoare style proof. In this work, we are interested in finding annotations that can be expressed in the *quantifier-free* fragment of our first-order language, to avoid the difficulty of reasoning with quantifiers.

Any given set of CHCs encoding safety of procedural programs can be transformed to an equisatisfiable set of just three CHCs with a single predicate symbol (encoding the program location using a variable). These CHCs have the following form:

$$\begin{array}{cc} Inv(\overline{x}) \leftarrow init(\overline{x}) & \neg bad(\overline{x}) \leftarrow Inv(\overline{x}) \\ \multicolumn{2}{c}{Inv(\overline{x}') \leftarrow Inv(\overline{x}), Inv(\overline{x}^o), tr(\overline{x}, \overline{x}^o, \overline{x}')} \end{array} \tag{4}$$

Intuitively, Inv is the program invariant, $\overline{x}$ denotes the pre-state of a program transition, $\overline{x}'$ denotes the post-state, and $\overline{x}^o$ denotes the summary of a procedure call (if one is made). If there are no procedure calls, tr is independent of $\overline{x}^o$ and $Inv(\overline{x}^o)$ can be dropped: in this case Inv denotes an inductive invariant of an ordinary transition system. In the sequel, we restrict to this normal form and consider only quantifier-free interpretations of the predicate Inv.

It is useful to rewrite the above rules using a function $\mathcal{F}$ that substitutes given predicates $\phi_A(\overline{x})$ and $\phi_B(\overline{x})$ for the occurrences of Inv in the rule bodies. That is, let

$$\begin{aligned} \mathcal{F}(\varphi_A, \varphi_B) \equiv & (\varphi_A(\overline{x}) \wedge \varphi_B(\overline{x}^o) \wedge tr(\overline{x}, \overline{x}^o, \overline{x}')) \\ & \vee init(\overline{x}') \end{aligned}$$

The rules are thus equivalent to $\mathcal{F}(Inv, Inv) \Rightarrow Inv(\overline{x})$. Abusing notation, we will also write $\mathcal{F}(\varphi_A)$ for $\mathcal{F}(\varphi_A, \varphi_A)$.

B. The SPACER framework

SPACER is a general framework that can be instantiated for a given logical theory T by supplying three elements: *(a)* a model-generating SMT solver for T, *(b)* an MBP procedure MBP for T and *(c)* in interpolation procedure ITP for T. Compared to other SMT-based algorithms (e.g., [3], [13], [10], [18]), the key distinguishing feature of SPACER is compositional reasoning. That is, instead of checking satisfiability of large formulas generated by program unwinding, SPACER iteratively creates and checks local reachability queries for individual procedures. In this way it is similar to IC3 [6], [9], a SAT-based algorithm for safety of finite-state transition systems, and GPDR [16], its extension to Linear Real Arithmetic. Like these methods, SPACER maintains a sequence of over-approximations of procedure behaviors, called *may summaries*, corresponding to program unwindings. However, unlike other approaches, SPACER also maintains under-approximations of procedure behaviors, called *must summaries*, to avoid redundant reachability queries. Another distinguishing feature of SPACER is the use of MBP for efficiently handling existentially quantified formulas to create a new query or a must summary. We note, however, that MBP is a general technique and can be exploited in IC3/PDR as well.[2]

[2]Arguably sub-goal creation in IC3 is a simple MBP for propositional logic.

Alg. 2 gives a simplified description of SPACER as a solver for CHCs in the form of (4) (though SPACER handles general CHCs). It is described using a set of rules that can be applied non-deterministically. Each rule is presented as a guarded command "[*grd*] *cmd*", where *cmd* can be executed only if *grd* holds.

```
Input: Formulas $init(\overline{x})$, $tr(\overline{x}, \overline{x}^o, \overline{x}')$, $bad(\overline{x})$
Output: Inductive invariant (FO interpretation of $Inv$
        satisfying (4)) or UNSAFE

if $(init \wedge bad)$ satisfiable then return UNSAFE
// initialize data structures
$Q := \emptyset$                                  // set of pairs $\langle\varphi, i\rangle, i \in \mathbb{N}$
$N := 0$                                  // max level, or recursion depth
$\mathcal{O}_0 = init, \mathcal{O}_i = \top, \forall i > 0$        // may summary sequence
$\mathcal{U} = init$                                  // must summary
forever non-deterministically do
    (Candidate) [ $(\mathcal{O}_N \wedge bad)$ satisfiable ]
        $Q := Q \cup \langle\varphi, N\rangle$, for some $\varphi \implies \mathcal{O}_N \wedge bad$
    (DecideMust) [ $\langle\varphi, i+1\rangle \in Q$, $M \models \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi'$ ]
        $Q := Q \cup \langle \text{MBP}(\exists \overline{x}^o, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi', M), i\rangle$
    (DecideMay) [ $(\varphi, i+1) \in Q$, $M \models \mathcal{F}(\mathcal{O}_i) \wedge \varphi'$ ]
        $Q := Q \cup \langle \text{MBP}(\exists \overline{x}, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i) \wedge \varphi', M)[\overline{x}/\overline{x}^o], i\rangle$
    (Leaf) [ $(\varphi, i) \in Q$, $\mathcal{F}(\mathcal{O}_{i-1}) \implies \neg\varphi'$, $i < N$ ]
        $Q := Q \cup \langle\varphi, i+1\rangle$
    (Successor) [ $\langle\varphi, i+1\rangle \in Q$, $M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'$ ]
        $\mathcal{U} := \mathcal{U} \vee \text{MBP}(\exists \overline{x}, \overline{x}^o \cdot \mathcal{F}(\mathcal{U}) \wedge \varphi', M)[\overline{x}/\overline{x}']$
    (Conflict) [ $\langle\varphi, i+1\rangle \in Q$, $\mathcal{F}(\mathcal{O}_i) \implies \neg\varphi'$ ]
        $\mathcal{O}_j := \mathcal{O}_j \wedge \text{ITP}(\mathcal{F}(\mathcal{O}_i), \neg\varphi')[\overline{x}/\overline{x}']$, $\forall j \leq i+1$
    (Induction) [ $(\varphi \vee \psi) \in \mathcal{O}_i$, $\mathcal{F}(\varphi \wedge \mathcal{O}_i) \implies \varphi'$ ]
        $\mathcal{O}_j := \mathcal{O}_j \wedge \varphi$, $\forall j \leq i+1$
    (Unfold) [ $\mathcal{O}_N \implies \neg bad$ ]  $N := N + 1$
    (Safe) [ $\mathcal{O}_{i+1} \implies \mathcal{O}_i$ ]  return invariant $\mathcal{O}_i$
    (Unsafe) [ $(\mathcal{U} \wedge bad)$ satisfiable ]  return UNSAFE
```

Algorithm 2: Rule-based description of SPACER.

As shown in Alg. 2, SPACER maintains a set of reachability queries Q, a sequence of may summaries $\{\mathcal{O}_i\}_{i \in \mathbb{N}}$, and a must summary $\mathcal{U}$. Intuitively, a query $\langle\varphi, i\rangle$ corresponds to checking if φ is reachable for recursion depth i, $\mathcal{O}_i$ over-approximates the reachable states for recursion depth i, and $\mathcal{U}$ under-approximates the reachable states. N denotes the current bound on recursion depth. The sequence of may summaries and N correspond to the *trace of approximations* and the maximum *level* in IC3/PDR, respectively. For convenience, let $\mathcal{O}_{-1}$ be $\bot$. MBP(φ, M), for a formula $\varphi = \exists \overline{v} \cdot \varphi_{qf}$ and model $M \models \varphi_{qf}$, denotes the result of some MBP function associated with φ for the model M.

Alg. 2 initializes N to 0 and, $\mathcal{O}_0$ and $\mathcal{U}$ to $init$. **Candidate** initiates a backward search for a counterexample beginning with a set of states in bad. The potential counterexample is expanded using either **DecideMust** or **DecideMay**. **DecideMust** *jumps over* the call $Inv(\overline{x}^o)$, in the last CHC of (4), utilizing the must summary $\mathcal{U}$. **DecideMay**, on the other hand, creates a query for the call using the may summary of its calling context.

Successor updates $\mathcal{U}$ when a query is known to be reachable. The other rules are similar to IC3 [6] and GPDR [16] and we skip their explanation in the interest of space. SPACER is sound and if MBP utilizes finite MBP functions, SPACER also terminates for a fixed N [17].

C. Instantiation for ARR+LIA

In instantiating this framework for ARR+LIA, the key ingredient is the MBP procedure of the previous section. An interpolation procedure ITP can be trivially obtained by using literal-dropping approach based on UNSAT cores, or a more sophisticated approach can be taken (e.g., see [16], [18]).

Because we do not have a *finite* MBP, SPACER is not guaranteed to terminate even for a fixed bound on the recursion depth N. That is, it can generate an infinite sequence of queries and must summaries. Note that MBP is used in 3 rules: **DecideMay**, **DecideMust**, and **Successor**. The elimination of quantifiers in **Successor** is only an optimization and can be avoided. This is not the case with **DecideMay** or **DecideMust** without changing the structure of the queries, the considerations of which are outside the scope of this paper. In the following, we identify restrictions on the CHCs where termination is still guaranteed and for the other cases, we propose some heuristic modifications to MBP and ITP to help avoid divergence.

1) Equality resolution in MBP*:* There are several cases where terms over combined signatures appear in conjunction with equality terms over the index quantifier, e.g., $\exists i : \mathtt{int} \cdot i = t \wedge rd(a, i) > 0$ for a term t independent of i. In these cases, the quantifier can be eliminated using equality resolution, e.g., $rd(a, t) > 0$ in the above example. Such cases seem to be natural in the case of a single procedure, i.e., when tr in (4) is independent of $\overline{x}^o$. Consider a disjunct δ in a DNF representation of tr. Now, δ represents a path in the procedure and typically, index terms (in reads and writes) in δ can be ordered such that every index term is a function of the previous index terms or the current-state variables $\overline{x}$. This makes it possible to eliminate any index variables in $\overline{x}'$ using equality resolution as mentioned above.

2) Privileging array equalities: Here is a simple example that exhibits non-termination:

$$Inv(a, b) \leftarrow a = b$$
$$\bot \leftarrow Inv(a, b), rd(a, j) < 0, rd(b, j) > 0$$

Here, intuitively, $Inv(a, b)$ denotes the summary of a procedure which takes an array a as input and produces b as output and we are interested in checking if there is sign change in the value at an index j as a result of the procedure call. For this example, **DecideMay** creates queries of the form $rd(a, k) < 0 \wedge rd(b, k) > 0$ where k is a specific integer constant. If ITP returns interpolants of the form $rd(a, k) = rd(b, k)$, it is easy to see that SPACER would not terminate even for $N = 0$, even though there is a trivial solution: $a = b$.

To alleviate this problem, we modify MBP and ITP to promote the use of array equalities in interpolants. Let ψ be the result of MBP for a given model M. For every pair of array terms a, b in ψ, we strengthen ψ with the array equality $a = b$ or disequality $a \neq b$, depending on whether $M \models a = b$ holds or not. In the above example, the queries will now be of the form $rd(a, k) < 0 \wedge rd(b, k) > 0 \wedge a \neq b$. However, $rd(a, k) = rd(b, k)$ continues to be an interpolant whereas the desired interpolant is $a = b$. To reduce the dependence on specific integer constants in the learned interpolants, and hence in the may summaries, we modify ITP as follows. Suppose we are computing an interpolant for $\psi \implies \neg\varphi'$ (as occurs in **Conflict**). We let $\varphi = \varphi_1 \wedge \varphi_2$ where φ_2 contains all the literals where an integer quantifier is substituted using its interpretation in a model. Using a *minimal unsatisfiable subset* (MUS) algorithm, we can generalize φ_2 to $\hat{\varphi}_2$ such that $\psi \wedge (\varphi_1 \wedge \hat{\varphi}_2)'$ is unsatisfiable and then obtain $\text{ITP}(\psi, \neg(\varphi_1 \wedge \hat{\varphi}_2)')$. In the above example, for $N = 0$ we have $\psi = (a = b)$, $\varphi_1 = (a \neq b)$, and $\varphi_2 = rd(a, k) < 0 \wedge rd(b, k) > 0$. One can show that $\hat{\varphi}_2$ is simply $\top$ and the only possible interpolant is $a = b$. In our implementation, we add such (dis-)equalities on-demand in a lazy fashion. Note that adding such (dis-)equalities to the queries is only a heuristic and may not always help with termination.

V. EXPERIMENTAL RESULTS

As noted in the introduction, the array theory allows us to model heap references accurately. This eliminates the need to inline procedures so that heap-allocated objects are reduced to local variables. We hypothesize that the resulting increase in modularity will allow SPACER to more efficiently verify procedural programs using ARRAYMBP, in spite of the potential for divergence due to non-finiteness of the MBP.

We test this hypothesis using a prototype implementation of SPACER with ARRAYMBP.[3] To verify C programs, we use SEAHORN [14], which uses the LLVM infrastructure to compile and optimize the input program, then encodes the verification conditions as CHCs in the SMT-LIB2 format. SEAHORN can optionally inline procedure calls before encoding, allowing us to test our hypothesis regarding modularity.

For reference, we also compare SPACER to the implementation of GPDR [16] in Z3 [8]. A key difference between SPACER and GPDR is that the latter does not use must summaries. Z3 also uses MBP, but is limited to equality resolution and the substitution method. As a result Z3 GPDR is effective only for inlined programs.

We use benchmarks from the software verification competition SVCOMP'15 [4]. We considered the 215 benchmarks from the *Device Drivers* category where Z3 GPDR (with inlining) needed more than a minute of runtime or did not terminate within the resource limits of SVCOMP [15]. All experiments have been carried out using a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM, running Ubuntu Linux. Our resource limits are 30 minutes and 15GB for each verification task. In the scatter plots that follow, a diamond indicates a *time-out*, a star indicates a *mem-out*, and a box indicates an anomaly in the implementation.

[3] https://bitbucket.org/spacer/code

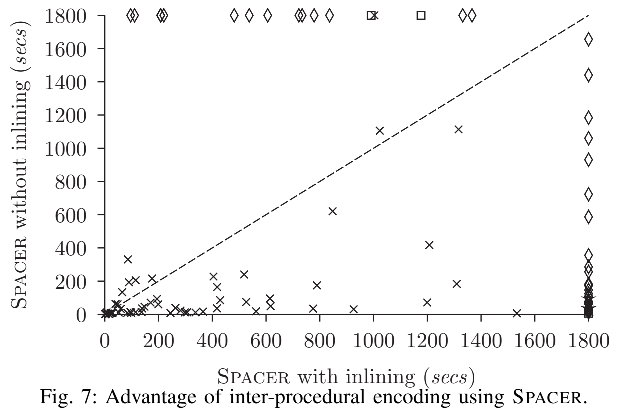


Fig. 7: Advantage of inter-procedural encoding using SPACER.

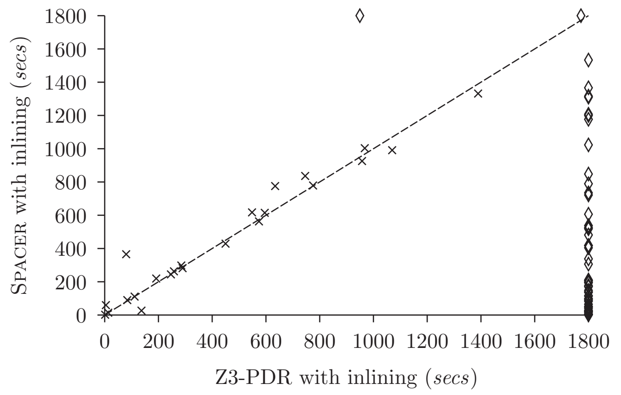


Fig. 8: SPACER vs. Z3 on hard SVCOMP benchmarks with inlining.

The scatter plot in Fig. 7 compares the combined run time for the CHC encoding and verification, when inlining is turned on and off. A clear advantage is seen in the non-inlining case. This shows that SPACER is able to effectively exploit the additional modularity that is made possible by ARRAYMBP, and that this advantage outweighs any occurrences of divergence due to non-finite MBP.[4] We note that SPACER with only LIA is able to handle only a small fraction of the non-inlined benchmarks. This result confirms our hypothesis.

For reference, we also compare to the performance of Z3 GPDR. We observed that without ARRAYMBP, Z3 is very ineffective in the non-inlined case. We should mention, however, that of the 7 unsafe programs verified by Z3, 5 could not be verified by SPACER. Fig. 8 compares SPACER and Z3 with inlining on. This shows an overwhelming advantage for SPACER, which is due to its more effective MBP approach.

VI. Related Work

There are several SMT-based approaches for sequential program verification that iteratively check satisfiability of formulas corresponding to safety of various unwindings of the program [3], [13], [10], [18]. However, these monolithic SMT formulas can grow exponentially. In contrast, the SPACER framework [17] we use allows us to do a compositional proof search for safety. Such local proof search is also found in the IC3 algorithm for hardware model checking [6] and its extensions to software model checking (e.g., [16]), although SPACER is the first to use under-approximate summaries of procedures for avoiding redundant proof sub-goals. Model-based generalizations have also been used to obtain projections efficiently in decision procedures for quantified formulas [19].

[4]Unfortunately, we have no way to distinguish divergence from timeouts.

VII. Conclusion and Future Work

We have presented a procedure for existentially projecting array variables from formulas over combined theories of ARR, LIA, and propositional logic. We have adapted the procedure to a finite MBP for array variables. While existential projection is worst-case exponential, the corresponding MBP is polynomial. However, projecting arrays might introduce new existentially quantified variables (whose sort is the same as the index- or value-sort of the eliminated array). For projecting these variables, a finite MBP need not exist. We described heuristics for obtaining a practical (but not necessarily finite) MBP procedure, obtaining an instantiation of the SPACER framework for verification of safety of sequential heap-manipulating programs. We show that the new variant of SPACER is effective for constructing compositional proofs of Linux Device Drivers. In the future, we plan to extend these ideas for handling more complex heap-manipulating programs that require universal quantifiers in the program invariants.

References

[1] W. Ackermann, *Solvable Cases of The Decision Problem.* North-Holland, Amsterdam, 1954.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From Under-Approximations to Over-Approximations and Back," in *TACAS*, 2012.
[3] ——, "Whale: An Interpolation-Based Algorithm for Inter-procedural Verification," in *VMCAI*, 2012.
[4] D. Beyer, "Software Verification and Verifiable Witnesses – (Report on SV-COMP 2015)," in *TACAS*, 2015.
[5] N. Bjørner, K. McMillan, and A. Rybalchenko, "Program Verification as Satisfiability Modulo Theories," in *SMT*, 2012.
[6] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011.
[7] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *Symbolic Logic*, vol. 22(3), 1957.
[8] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008.
[9] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient Implementation of Property Directed Reachability," in *FMCAD*, 2011.
[10] M. H. et al., "Ultimate Automizer with SMTInterpol - (Competition Contribution)," in *TACAS*, 2013.
[11] G. Fedyukovich, O. Sery, and N. Sharygina, "eVolCheck: Incremental Upgrade Checker for C," in *TACAS*, 2013.
[12] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring," in *ICCAD*, 2004.
[13] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing Software Verifiers from Proof Rules," in *PLDI*, 2012.
[14] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. Navas, "The SeaHorn Verification Framework," in *CAV*, 2015.
[15] A. Gurfinkel, T. Kahsai, and J. A. Navas, "SeaHorn: A Framework For Verifying C Programs - (Competition Contribution)," in *TACAS*, 2015.
[16] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT*, 2012.
[17] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV*, 2014.
[18] K. L. McMillan and A. Rybalchenko, "Solving Constrained Horn Clauses using Interpolation," Tech. Rep. MSR-TR-2013-6, 2013.
[19] D. Monniaux, "Quantifier Elimination by Lazy Model Enumeration," in *CAV*, 2010.
[20] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for an Extensional Theory of Arrays," in *LICS*, 2001.

IC3 Software Model Checking on Control Flow Automata

Tim Lange
RWTH Aachen University, Germany
tim.lange@cs.rwth-aachen.de

Martin R. Neuhäußer
Siemens AG, Germany
martin.neuhaeusser@siemens.com

Thomas Noll
RWTH Aachen University, Germany
noll@cs.rwth-aachen.de

Abstract—**In recent years, the inductive, incremental verification algorithm IC3 had a major impact on hardware model checking. Also with respect to software model checking, a number of adaptations of Boolean IC3 and combinations with CEGAR and ART-based techniques have been developed. However, most of them exploit the peculiarities of software programs, such as the explicit representation of control flow, only to a limited extent. In this paper, we propose a technique that supports this explicit representation in the form of control flow automata, and integrates it with symbolic reasoning about the data state space of the program. It thus provides a true lifting of IC3 from hardware to software model checking. By evaluating the approach on a number of case studies using a prototypical implementation, we demonstrate that our method shows promising results.**

I. Introduction

IC3 [1] is an incremental algorithm that has originally been designed for verifying invariant properties of finite transition systems. It constructs an over-approximation of the reachable state space by generating Boolean clauses that are inductive relative to stepwise reachability information. During this construction, candidate counterexamples are being disproved using Boolean SAT-solving techniques. This approach has turned out to be highly effective; in fact, it is considered to be one of (if not *the*) most important contribution of bit-level formal verification of hardware systems for the last decade.

There have been several attempts to lift Boolean IC3 to the domain of software model checking. As this setting usually induces infinite-state systems, more advanced symbolic reasoning techniques are required. The most prominent one is Satisfiability Modulo Theories (SMT). Here, sets of states are symbolically specified by first-order formulas over constraints from the respective theory, and SMT-solving techniques are employed to rule out spurious counterexamples.

One of the first integrations of SMT into IC3 has been presented in [2]. In addition to the generalisation of SAT to SMT solving, it exploits the partitioning of the program's state space as induced by its control flow graph. This is achieved by unwinding the latter into an Abstract Reachability Tree (ART) in which each node is associated with a control location and a formula, resulting in an "explicit-symbolic" approach named Tree-IC3. Candidate counterexamples are handled by computing under-approximations of pre-images.

The advantages in comparison to Boolean IC3 are twofold. First, Tree-IC3 eliminates the possible redundancy of subformulas partitioning of the control state space, the solver is exposed to simpler and smaller formulas.

On the downside, the key idea underlying IC3, relative inductiveness, cannot be directly applied in this setting due to the partitioned representation that leads to a path-wise unwinding of the transition system. The follow-up publication [3] therefore reverts to a monolithic transition relation, replacing the pre-image computation by (implicit) predicate abstraction. The latter is a standard abstraction technique [4] that partitions the state space according to the equivalence relation induced by a set of predicates. Its implicit variant [5] allows to express abstract transitions without explicitly computing the abstract system. In the IC3 setting, this avoids theory-specific generalisation techniques.

In [6], Horn clauses are employed to represent recursive predicate transformers. Proof obligations are generalised using a specialised interpolation procedure for linear arithmetic. However, the latter again does not exploit relative induction.

In summary, existing work exploits the peculiarities of software programs only to a limited extent to support IC3-style verification. In this paper, we develop an approach that combines the advantage of explicitly handling the control flow of a program, employing a corresponding automata model, with relative inductive reasoning over a symbolic representation of its data space. It thus provides a true lifting of IC3 from hardware to software model checking. We demonstrate the applicability and efficiency of our method by evaluating it on a number of case studies using a prototypical implementation.

The remainder of this paper is organised as follows. We start by introducing some general concepts in Section II. Section III sets the stage for our contribution with a description of the original IC3 algorithm, which is then extended in Section IV by taking control flow automata into account. Results of the experimental evaluation are given in Section V. Section VI concludes the paper with a summary and a description of future work.

II. Preliminaries

A *control flow automaton* (CFA) $A = (L, G)$ consists of a finite set of locations $L = \{0, \dots, n\}$, modeling the program counter of a corresponding sequential code, and edges in $G \subseteq L \times FO \times L$ labeled with quantifier-free first-order formulas over the set Var of program variables and their next-state primed forms, Var' [7]. Priming of a formula, φ', is

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

the same as priming every variable in φ. Such a formula can either encode a variable assignment, containing primed and unprimed variables, or an assume statement in which case it will only contain unprimed variables. We assume that for any two locations there exists at most one edge between them.

Given a subset of variables $X \subseteq Var$, a *cube* over X is defined as a conjunction of *literals*, each literal being a variable or its negation in the propositional case and a theory atom or its negation in the quantifier-free first-order case. The negation of a cube, i.e. a disjunction of literals, is called a *clause*.

A *program* $\mathcal{P} = (A, l_0, l_E)$ consists of a CFA A representing the control flow, as well as an initial location $l_0 \in L$ and an error location $l_E \in L$. This representation allows to encode arbitrary programs and assertions for safety verification. For every assertion that has to be verified at location l, split l and introduce an edge with the negated assertion to l_E and an edge with the positive assertion between the split nodes. This way, checking the violation of assertions becomes checking reachability of l_E in A.

Given two locations $l_1, l_2 \in L$, we define the *transition formula*

$$T_{l_1 \to l_2} = \begin{cases} (pc = l_1) \wedge t \wedge (pc' = l_2) & \text{, if } (l_1, t, l_2) \in G \\ false & \text{, otherwise.} \end{cases} \tag{1}$$

This yields the global transition formula as

$$T = \bigvee_{(l_1,t,l_2) \in G} T_{l_1 \to l_2}. \tag{2}$$

Definition 1 (Relative inductiveness [1]): Given a transition formula T, a formula φ is inductive relative to another formula ψ if

$$\psi \wedge \varphi \wedge T \Rightarrow \varphi' \tag{3}$$

is valid.

Starting from Def. 1 we can refine relative inductiveness to consider only a single transition rather than the whole transition relation.

Definition 2 (Edge-relative inductiveness): Given a CFA A and locations $l_1, l_2 \in L$, a formula φ is edge-relative inductive to another formula ψ if

$$\psi \wedge \varphi \wedge T_{l_1 \to l_2} \Rightarrow \varphi' \tag{4}$$

is valid.

Note that edge-relative inductiveness does also hold if $(l_1, t, l_2) \notin G$ for every t. In this case, $T_{l_1 \to l_2} = false$, which makes (4) hold trivially, i.e. if we are in a state satisfying φ and we cannot leave it via the considered edge, we remain in a state satisfying φ.

To handle the possibly infinite state space over Var, we use a symbolic representation through quantifier-free first-order formulas.

Definition 3 (Data region): A data region is represented by a quantifier-free FO formula s over Var and consists of all variable assignments σ satisfying s, i.e., $\{\sigma \mid \sigma \models s\}$.

Based on Def. 3 we can augment a data region with a control flow location $l \in L$. This information, sometimes referred to as atomic region [8] is called *region* in the following.

Definition 4 (Region): We define a region $r = (l, s)$ as a pair consisting of location $l \in L$ and data region s. Given such a region $r = (l, s)$, the corresponding formulas are defined as $\{\phi \mid \phi \equiv (pc = l \wedge s)\}$. Analogously the corresponding formulas for a negated region $\neg r$ are defined as $\{\phi \mid \phi \equiv \neg(pc = l \wedge s)\}$.

Given two regions r_1, r_2 and representatives of their corresponding propositional formulas φ_1, φ_2, then r_1 is inductive relative to r_2 iff φ_1 is inductive relative to φ_2. The analogue can be defined for the special case of edge-relative inductiveness.

Using their corresponding formula, we can use relative and edge-relative inductiveness for regions very similar to [1].

Even though it works for two non-negated regions as well, the IC3 algorithm only makes use of the case where we check whether a negated region $\neg r_2$ is inductive edge-relative to a non-negated region r_1. Therefore we will only consider this case in the following and inspect it in detail. We found two different cases whose premise can be statically determined and that simplify the SMT queries that we have to use.

Lemma 1 (Relative inductive regions): Assuming two regions $r_1 = (l_1, s_1)$, $\neg r_2 = \neg(l_2, s_2)$, we can reduce edge-relative inductiveness of $\neg r_2$ to r_1 to

$$s_1 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2' \qquad \text{, if } l_2 \neq l_1 \tag{5}$$

$$s_1 \wedge \neg s_2 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2' \qquad \text{, if } l_2 = l_1 \tag{6}$$

Proof 1: Given two regions $r_1 = (l_1, s_1)$ and $r_2 = (l_2, s_2)$ with corresponding formulas φ_1 and φ_2, we have:

$$\varphi_1 \equiv (pc = l_1 \wedge s_1) \qquad \neg\varphi_2 \equiv \neg(pc = l_2 \wedge s_2)$$

Def. 2 yields:

$$\begin{aligned} & (pc = l_1 \wedge s_1) \wedge \neg(pc = l_2 \wedge s_2) \wedge T_{l_1 \to l_2} \\ & \Rightarrow \neg(pc' = l_2 \wedge s_2') \\ \equiv\; & (pc = l_1 \wedge s_1) \wedge (pc \neq l_2 \vee \neg s_2) \wedge T_{l_1 \to l_2} \\ & \Rightarrow (pc' \neq l_2 \vee \neg s_2') \end{aligned}$$

If $l_1 \neq l_2$, this is equisatisfiable to

$$\begin{aligned} & (true \wedge s_1) \wedge (true \vee \neg s_2) \wedge T_{l_1 \to l_2} \Rightarrow (false \vee \neg s_2') \\ \equiv\; & s_1 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2' \end{aligned}$$

Otherwise, we obtain

$$\begin{aligned} & (true \wedge s_1) \wedge (false \vee \neg s_2) \wedge T_{l_1 \to l_2} \Rightarrow (false \vee \neg s_2') \\ \equiv\; & s_1 \wedge \neg s_2 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2' \end{aligned}$$

□

In Proof 1, we can use the presented equisatisfiable transformations because the transition from l_1 and l_2 implicitly contains the atoms $(pc = l_1)$ and $(pc' = l_2)$. Therefore in the first case, where $l_1 \neq l_2$, the atoms $(pc = l_1)$ and $(pc' \neq l_2)$ can be rewritten to $true$, while the atom $(pc' \neq l_2)$ can be rewritten to *false*. The analogous holds for the case where $l_1 = l_2$.

Given a program $\mathcal{P}$, we can define a finite *path* π of length n as a sequence $l_0, l_1, \dots, l_n$, s.t. for every $0 \leq i < n$ there exists an edge $(l_i, t_i, l_{i+1}) \in G$ in $\mathcal{P}$. A path π is called *feasible* iff for every l_j in π we can construct a region (l_j, s_j) that is non-empty, i.e. $s_j \not\equiv false$, s.t. $s_i \wedge T_{l_i \to l_{i+1}} \Rightarrow s_{i+1}$ for $0 \leq i < n$.

III. Original IC3 Algorithm

Let $S = (X, I, T)$ be a transition system over a set X of Boolean variables, and $I(X)$ and $T(X, X')$ two propositional formulas respectively describing the initial condition and the transition relation over variables in X and next-state primed successors X'. Given a propositional property $P(X)$, we want to verify that every state in S that is reachable from a state in I satisfies P. Sometimes also an inverted formulation is used like in [9] where $\neg P$ states are *bad states* and we want to show that no bad state is reachable from the initial states. The main idea of the IC3 algorithm [1] and the earlier *finite state inductive strengthening* (FSIS [10]) is that if P is *inductive*, i.e. $I \Rightarrow P$ and $P \wedge T \Rightarrow P'$, then P is also an invariant on S. However, even if P is an invariant on S it may not be inductive. Therefore the goal of IC3 and FSIS is to produce a so called *inductive strengthening* F of property P, s.t. $F \wedge P$ is inductive. This means that we can restrict the set of states in P to a smaller set of states in the intersection of F and P, which still contains all states reachable from I, but excludes unreachable states that will lead to a violation of induction. While FSIS tries to come up with such a strengthening in one step, IC3 proceeds in a more relaxed approach and constructs F *incrementally*.

This incremental construction is based on a sequence of *frames* $F_0, \dots, F_k$ for which

$$\begin{aligned} I &\Rightarrow F_0 & \\ F_i &\Rightarrow F_{i+1} & \text{, for } 0 \leq i < k \\ F_i &\Rightarrow P & \text{, for } 0 \leq i \leq k \\ F_i \wedge T &\Rightarrow F'_{i+1} & \text{, for } 0 \leq i < k \end{aligned}$$

has to hold in order to produce an inductive invariant. The algorithm starts with two initial checks for 0- and 1-step reachable states in $\neg P$ and afterwards initializes the first frame F_0 to I. The rest of the algorithm can be divided into an inner and an outer loop, sometimes also referred to as *blocking* and *propagation* phases, respectively.

The outer loop iterates over the maximal frame index k, looking for states in F_k that can reach $\neg P$, so called *counterexamples to induction* (CTI). If such a CTI exists, it is analyzed in the inner loop, the blocking phase. If no such CTI exists, IC3 tries to propagate clauses learned in frame F_i forward to F_{i+1}. In the end it checks for termination, which is given if $F_i = F_{i+1}$ for some $0 \leq i < k$.

The objective of the blocking phase is to decide whether a CTI is reachable from I or not. For this purpose, it maintains a set of pairs of frame indices and states, called *proof obligations*. From this set it picks the pair (i, s) with the smallest frame index i. If there is more than one pair with this frame index, the choice between those is arbitrary. For the chosen state, IC3 checks whether $\neg s$ is relative inductive to F_{i-1}, using (3). If it is relative inductive, we can block s in frames F_j for $0 \leq j \leq i+1$. But rather than just adding $\neg s$, IC3 first tries to obtain a clause that is a subset of $\neg s$ and therefore excludes more states. This clause, called a *generalization* of $\neg s$, is then added to the frames and afterwards the pair (i, s) in the set of proof obligations is replaced by $(i+1, s)$. If s is not relative inductive to F_{i-1} this means that there exists an F_{i-1} predecessor p that can reach s. IC3 therefore adds $(i-1, p)$ to the set of proof obligations. The blocking phase terminates if either there exists an s in the set of proof obligations that is relative inductive to an initial state at index 0, in which case there exists a counterexample path, or for every proof obligation the frame index $i > k$, i.e. there exists a $j \geq 0$, s.t. every predecessor of the original CTI is inductive relative to F_j.

IV. IC3 on Control Flow Automata

In this section we will present our IC3 algorithm for control flow automata as annotated pseudocode, give a short explanation and a proof of partial correctness, followed by an example showing the benefits of our method.

The most straight-forward way to lift IC3 to software model checking is to encode the control flow in an additional pc variable representing the program location, as presented in [2]. However, this approach introduces some tedious handling of the implicit pc variable and is not very competitive. One reason is that the control flow of the input program already gives a very clear structure to the system, which is completely disregarded when encoded inside a global transition formula. Thus our approach tries to exploit as much of the structure given in the input program as possible.

In [1] Bradley draws an analogy between the way IC3 proves properties on a transition system and how a human analyzes a system - by producing a set of lemmas s.t. each holds relative to a previous one and that all together imply the property. It is that stepwise approach that makes IC3 so competitive and which motivated us to apply our version of IC3 directly to a control flow automaton as an explicit representation of a program's possible execution steps.

With respect to the definition of programs we follow the notion of [9] and reason about error states, rather than property states.

The explicit representation of edges leads to a situation where we can reduce the possible transitions for a region $r = (l, s)$ to those that are available from l in the program $\mathcal{P}$, which allows us to formulate significantly smaller solver queries. Explicit initial and error states enable us to statically check 0-step and (potential) 1-step reachability, as well as to avoid initial and error conditions, which in turn reduces the size of the solver queries even further.

In analogy to bit-level IC3, we construct frame sequences $F_0, \dots, F_k$, but instead of using global frames, we use *location-local* frames $F_{(i,l)}$ in every $l \in G$. We interpret those $F_{(i,l)}$ as the set of, possibly overapproximating, data regions reachable in at most i steps at location l.

Algorithm 1 Outer loop

```
Ensure: ret. value iff $l_E$ is reachable
  function BOOL PROVE
    if $l_0 = l_E$ or $((l_0, t, l_E) \in G$ and $sat(t))$ then
      return false
    initialize frames
    for $k = 1$ to $\dots$ do
      if not STRENGTHEN($k$) then
        return false
      propagate
      if termination then
        return true
```

Alg. 1 works more or less like the original function *prove* in [1]. In the initial checks, we can reformulate the 0-step reachability query $I \wedge \neg P$ to the simple check whether $l_0 = l_E$. Also for 1-step counterexamples, originally $I \wedge T \wedge \neg P'$, we can still check the necessary syntactic reachability condition statically. If there exists an edge $e = (l_0, t, l_E)$, then we have to use the solver to check satisfiability of t. After those initial checks are completed we initialize frames F_0 and F_1. Exploiting the fact that only l_0 is initial, we can set $F_{(0,l_0)}$ to *true* and $F_{(0,l)}$ to *false* for every $l \neq l_0$. After the initial phase, the algorithm starts the main loop with frame limit k and tries to strengthen the new frame set. If the blocking phase succeeds and finds a strengthening for k, the propagation phase starts and tries to push learned data regions forward. The inner loop ends with checking termination. Here we have to modify the original termination condition $F_i = F_{i+1}$, for some i, to $F_{(i,l)} = F_{(i+1,l)}$ for some i and every $l \in L \setminus \{l_E\}$.

Algorithm 2 Strengthening

```
Require: (a) $k \geq 1$
Require: (b) $\forall i \geq 0, l \in L, F_{(i,l)} \Rightarrow F_{(i+1,l)}$
Require: (c) $\forall 0 \leq i < k, l, l' \in L$, s.t. $(l, t, l') \in G$, $F_{(i,l)} \wedge T_{l \to l'} \Rightarrow F'_{(i+1,l')}$
Ensure: $\forall i \geq 0, l \in L, F_{(i,l)} \Rightarrow F_{(i+1,l)}$
Ensure: if ret. value then $\forall 0 \leq i < k, l, l' \in L$, s.t.$(l, t, l') \in G$, $F_{(i,l)} \wedge T_{l \to l'} \Rightarrow F'_{(i+1,l')}$
Ensure: if $\neg$ret. value, there exists a counterexample path
  function BOOL STRENGTHEN($k$: int)
    while $\exists l$, s.t. $sat(F_{(k,l)} \wedge T_{l \to l_E})$ do
      @assert (b),(c)
      s := predecessor data region
      if not BACKWARDBLOCK($k, l, s$) then
        return false
      @assert $s \not\models F_{(k,l)}$
    return true
```

The function STRENGTHEN in Alg. 2 also works similarly to the original IC3. The main difference here is in the condition of the while loop which checks whether there exists $l \in L$ s.t. $e = (l, t, l_E) \in G$ and t is satisfiable under $F_{(k,l)}$. If this is the case, there exists a CTI. Note that in this paper we do not tackle the whole topic of generalization and restrict ourselves to computing weakest preconditions (WP) of the error state w.r.t. to t. While this might not offer the best performance possible, it is a safe approximation, as the WP is the smallest overapproximation of CTI states. The so extracted predecessor s is then analyzed in the function BACKWARDBLOCK, shown in Alg. 3.

Algorithm 3 Inner loop

```
Require: (b),(c)
Require: $sat(F_{(\hat{i},\hat{l}')} \wedge \hat{s} \wedge T_{\hat{l}' \to l_E})$
Ensure: if ret. value, then $\neg\hat{s}$ is inductive relative to $F_{(\hat{i}-1,l)}$, $\forall l$, s.t. $(l, t, \hat{l}') \in G$
Ensure: if ret. value, then (b),(c)
Ensure: if $\neg$ret. value, there exists a feasible path $l_0 \rightsquigarrow \hat{l}'$
  function BOOL BACKWARDBLOCK($\hat{i}$: int, $\hat{l}'$: location, $\hat{s}$: data region)
    $Q.add(\hat{i}, \hat{l}', \hat{s})$
    while $|Q| > 0$ do
      @assert $\forall (i, l', s) \in Q. 0 \leq i \leq k$
      @assert $\forall (i, l', s) \in Q. \exists$ path $(l', s) \rightsquigarrow (l_E, true)$
      $(i, l', s) = Q.pop$
      if $i = 0$ then
        return false
      else
        @assert $(l', \neg s)$ is inductive relative to $F_{(j,l)}$, $\forall 0 \leq j < i$, $l \in L \backslash \{l_E\}$
        for each $l$, s.t. $(l, t, l') \in G$ do
          if $l = l'$ and $sat(F_{(i-1,l)} \wedge \neg s \wedge T_{l \to l'} \wedge s')$ then
            generate predecessor $c$ of $s$
            @assert $\forall (i, l', s) \in Q, c \neq s$
            add $(i-1, l, c)$ and $(i, l', s)$ to $Q$
          else if $l \neq l'$ and $sat(F_{(i-1,l)} \wedge T_{l \to l'} \wedge s')$ then
            generate predecessor $c$ of $s$
            @assert $\forall (i, l', s) \in Q, c \neq s$
            add $(i-1, l, c)$ and $(i, l', s)$ to $Q$
          else
            block $s$ in frames $F_{(j,l')}$ for $0 \leq j \leq i$
      return $true$
```

While Alg. 1 and 2 are based on [1], we decided to present the inner loop similarly to the representation in [9] as we found it easier to comprehend. The function BACKWARDBLOCK gets as parameter a frame index $\hat{i}$, a location $\hat{l}' \in L$ and a data region $\hat{s}$. Following [9] we add this initial proof obligation to a priority queue Q, s.t. the obligation with the lowest i will get popped first. While the queue is non-empty, we start the inner loop by picking the obligation with the smallest i. If $i = 0$, we can immediately stop with a counterexample because l' has to be initial. If it were not initial, the previous proof obligation at level 1 would have included the frame $F_{(0,l)}$, which is *false* for every non-initial location l. If $i \neq 0$ we have to check whether the region $\neg(l', s)$ is inductive edge-relative to $F_{(i-1,l)}$ of any predecessor l by solving the query (5) or (6), depending on whether $l = l'$ or not. If $\neg(l', s)$ is inductive edge-relative to $F_{(i-1,l)}$ for every predecessor l, we can block s in all $F_{(j,l')}, 0 \leq j \leq i$. If, on the other hand, $\neg(l', s)$ is not inductive edge-relative to $F_{(i-1,l)}$ for some l, then there must exist a predecessor that can reach (l', s). We therefore take the WP c of s w.r.t. the transition formula and add the new proof obligation $(i-1, l, c)$ to the obligation queue. We also add the old obligation (i, l', s) to the queue for future re-inspection. The inner loop terminates with *true* in case that Q is empty or with *false* in case there exists an obligation at frame 0.

Note that Alg. 3 slightly differs from the idea of blocking a region r iff r is edge-relative inductive to all incoming edges. However, the presented algorithm behaves correctly: Due to the ordering of the obligation queue, the algorithm proceeds in a kind of depth-first search manner. Even if r has been blocked at level i via one edge, another obligation r', that is a predecessor of r, at level $i - 1$ will be chosen. Here we can distinguish two cases: Either r' is the last step on a counterexample path from l_0 to r at level i or it is not, in which case all regions explored will be blocked. In both cases r at level i will not be reconsidered before backtracking to an obligation at level $i + 1$, in which case r at level i has been blocked.

In the following we will show that our algorithm is partially correct, i.e. it is correct given its termination. We construct our proof bottom-up by first proving that Alg. 3 is correct. As we use weakest preconditions, we can actually show full correctness, because termination is achieved for the following reason: Given an initial obligation (i, l, s) we have to construct at most n^i proof obligations, where n is the maximal in-degree of locations in L. This upper bound can be established because a WP of s covers all, possibly infinitely many, predecessor data regions that can reach s w.r.t. the transition. Therefore we cover all data regions in one step and only have to construct predecessor regions until we reach frame level 0, i.e., after doing this i times.

Lemma 2: Function BACKWARDBLOCK returns true iff $\neg\hat{s}$ is inductive relative to $F_{(\hat{i}-1,l)}$ for all locations l that are a predecessor of $\hat{l}'$.

Proof 2: Function BACKWARDBLOCK starts the while loop by examining the proof obligation in the queue that has the lowest frame index i. If the frame index is zero, then $\hat{l}'$ must be l_0, because $F_{(0,I)}$ is the only region with frame index 0 to which any other region is relatively inductive, by construction. This way, there must exist a feasible path from l_0 to $\hat{l}'$.

If there exists no feasible path from l_0 to $\hat{l}'$ given $F_0, ..., F_k$, then every path of length j ending in $\hat{l}'$ starts in a location l, s.t. the region $(l, \neg s)$ is inductive relative to $F_{(k-j-1,l^x)}$ for all l^x, s.t.$(l^x, t, l) \in G$. This means that every proof obligation added to Q is ultimately inductive relative to its predecessors and thus also $\neg\hat{s}$ is inductive relative to $F_{(\hat{i}-1,l)}$.

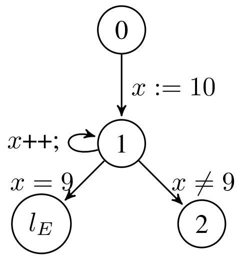


Fig. 1. Example for non-termination of Alg. 1

We continue by proving correctness of Alg. 2. Here we can guarantee termination for the same reason as for BACKWARDBLOCK. Because we only search for CTIs, we have at most n_E of them, where n_E is the number of predecessor locations of l_E in $\mathcal{P}$. As we compute exact pre-images by weakest preconditions, every data region has exactly one predecessor data region per edge.

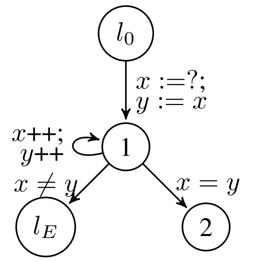
Lemma 3: Function STRENGTHEN terminates with result true iff there exists an inductive strengthening for the frames $F_{(k,l)}$ in all locations l in the CFA.

Proof 3: Assume a call of function STRENGTHEN returns false, then there must have been a call to BACKWARDBLOCK with some (k, l, s), such that BACKWARDBLOCK returned false. From Lemma 2 we know that in this case, there exists a feasible path of length k from l_0 to l that ends up in data region s. Because l is a predecessor of l_E and s is a precondition under $T_{l \to l_E}$, there exists a counterexample path of length $k + 1$. Otherwise every call of BACKWARDBLOCK returned true, which means that every predecessor location (and data region) of l_E is unreachable in the current frame sequence. Thus every predecessor of l_E was excluded from their frames at level k which yields an inductive strengthening for F_k.

After proving correctness of Alg. 2 and 3, we have to drop termination for Alg. 1. The reason is that there might exist infinite ascending or descending chains that we cannot generalize. This is exemplified in the simple program in Fig. 1. While there exist an inductive strengthening, e.g., $(x \geq 10)$, for every maximal frame index k our algorithm will block a region $(1, 9 - (k - 1))$ of which there can be infinitely many for unbounded integers, such that there will never exist an i for which $F_{(i,1)} = F_{(i+1,1)}$. Note that in the following, as all statements over $F_{(i,l)}$ always concern all non-error locations, we will slightly abuse notation and use F_i in place of $F_{(i,l)}, \forall l \in L \backslash \{l_E\}$.

Lemma 4: In case function PROVE terminates, it returns *true* iff there exists an inductive strengthening F for P, s.t. $F \wedge P$ is inductive.

Proof 4: Assume PROVE terminates with true, then every call of STRENGTHEN for every $j < k$ must have returned true and there must exist a frame with index $i < k$, s.t. $F_i = F_{i+1}$, i.e. the frame F_i is inductive, because $F_i \wedge T \Rightarrow F'_i$. Therefore there cannot exist a counterexample path of length k (more precise of length i) or less and there cannot exist one of length greater than k, because F_i is inductive.



l: \ i:	0	1	2
l_0	true	true	true
1	false	$x = y$	$x = y$

Fig. 2. Example program and resulting frames

Now assume that PROVE returns false: Then there must exist a k, s.t. for no $i < k$, F_i is inductive and STRENGTHEN for k returns false, i.e. there exists a path of length k from the initial to the error state.

Theorem 1: If the algorithm terminates, it returns true iff P is an invariant on S.

Proof 5: By Lem. 2-4 the theorem holds. □

Example

In the remainder of this section we show by example how our lifting of IC3 to control flow automata works and what its benefits are. We start with the program from Fig. 2 as input. Note that in our presentation we omit the computation steps for $l = 2$, as l_E is not reachable from that location. As shown in Alg. 1 we start by the two static checks for 0- and 1-step counterexamples. As they are both obviously not satisfied, we proceed to initializing $F_{(0,l_0)}$ to $true$ and $F_{(0,1)}$ to $false$. $F_{(1,l)}$ is set to $true$ for both locations $l \in \{l_0, 1\}$.

We now start our algorithm with $k = 1$ and try to construct a strengthening. There exists exactly one l s.t. $(l, t, l_E) \in G$, namely $l = 1$, which yields the initial proof obligation $(1, 1, x \neq y)$ for the priority queue Q in Alg. 3. As i is not 0, we start the blocking phase by searching for a predecessor of 1 and find location l_0, which means we have to apply query (5) and check $sat(F_{(0,l_0)} \wedge y' = x' \wedge x' \neq y')$, which is obviously not satisfiable. Next we check $l = 1$. As this is a self-loop we can use the stronger query (6) to check $sat(F_{(0,1)} \wedge x = y \wedge x' = x + 1 \wedge y' = y + 1 \wedge x' \neq y')$.

This query shows two improvements over existing techniques: First, we initialized $F_{(0,1)}$ to $false$ because it is not initial, which allows us to block the obligation one step earlier and also make the query non-satisfiable immediately. Second, even without any information learned in $F_{(0,1)}$ the query is not satisfiable due to the stronger edge-relative inductiveness of (6).

We continue the execution by blocking $x \neq y$ in $F_{(1,1)}$, i.e. add $\neg(x \neq y)$ to $F_{(1,1)}$. Afterwards there is no entry in Q and we leave BACKWARDBLOCK. As by the blocking there is no more CTI at index 1, we also leave STRENGTHEN and continue with major iteraton $k = 2$.

The function call STRENGTHEN(2) enters BACKWARD-BLOCK with the initial obligation $(2, 1, x \neq y)$. For predecessor l_0, we check $sat(F_{(1,l_0)} \wedge y' = x' \wedge x' \neq y')$, which is again unsatisfiable, as well as $sat(F_{(1,1)} \wedge x = y \wedge x' = x+1 \wedge y' = y + 1 \wedge x' \neq y')$ for location 1, which is not satisfiable either. We can therefore block $x \neq y$ in frame $F_{(2,1)}$, too.

Again we have an empty Q and no more CTI at level 2. We check for termination and find $F_{(1,l_0)} = F_{(2,l_0)}$ as well as $F_{(1,1)} = F_{(2,1)}$, which means that we found an inductive strengthening $F = (1, x = y)$ s.t. l_E is not reachable in the CFA.

V. BENCHMARKS

In this section we start with details of our implementation, followed by an evaluation and end with a discussion of the presented results.

Implementation

We implemented our IC3CFA algorithm on top of an existing proprietary model checking framework. A flow chart of the framework is shown in Fig. 3. The framework makes use of the LLVM project enable parsing a wide range of input languages and translate them into the LLVM intermediate representation (IR) [11]. To close the gap between compiler oriented semantics of C as assumed by LLVM and verification semantics that was encountered in the development of the UFO model checker we use the approach of initializing variables with a call to an external function as presented in [12]. We translate this IR into our own intermediate verification language (IVL) that is more suitable for verification, e.g. no SSA form, no three-adress code. On the IVL code we execute some optimization stages that are also widely used in other model checkers, e.g. the Kratos software model checker [13], like program slicing, expression propagation and bisimulation minimization, as well as Steensgaard's pointer analysis [14] to rewrite simple pointer expressions. The bit-precise memory model (similar to the one of CBMC [15]) supports limited pointer operations, including array-element and record-field addressing. After reaching a fixpoint of these optimizations, we construct the program's control flow graph, labeled with instructions of a guarded command language similar to that of [16]. This allows for efficient construction of weakest preconditions [16], [17]. The results of our preprocessing are in line with the CFAs produced by Kratos and only differ by one or two locations. While the presented approach is fully theory unaware and can be used for infinite-domain theories such as linear real arithmetic (LRA) we use the finite-domain theory of bit vectors. Our tool supports the Z3 and MathSAT SMT solvers; all following benchmarks were executed using the Z3 solver. To minimize overhead and reduce unnecessary pushing attempts we implemented the efficient pushing strategy of [18]. For storing frames efficiently we implemented the delta encoding approach of [9] which, in our experiments, reduced memory consumption as well as runtime significantly. For some benchmarks the reduction

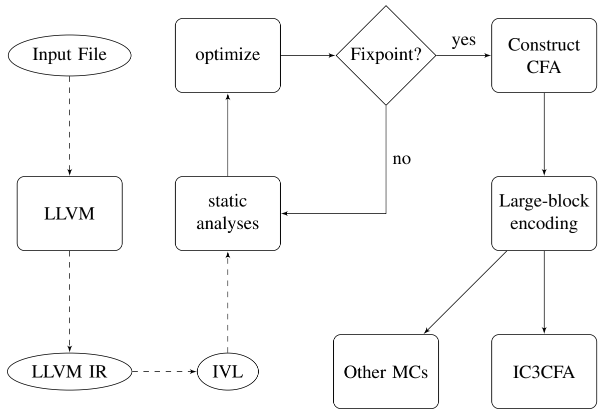


Fig. 3. Tool flow

in runtime was almost 20 times. Generalization of computed preimages is not implemented, yet.

Evaluation

For evaluation of our algorithm and to compare it to others we used a subset[1] of 28 programs from the set of benchmarks used in [2], originating from different domains such as device drivers, communication protocols, SystemC designs and textbook algorithms, some of which are contained in the set of benchmarks of the software verification competition (http://sv-comp.sosy-lab.org/2015/). One out of four programs contains a bug.

All presented results can be reproduced by our tool via the web-interface at http://www-i2.informatik.rwth-aachen.de/mctools/vplc/fmcad15/ .

All experiments have been executed on a cluster using a single core per instance, running at 2.1 GHz with a memory limit of 4GB per file and a timeout of 1200 seconds.

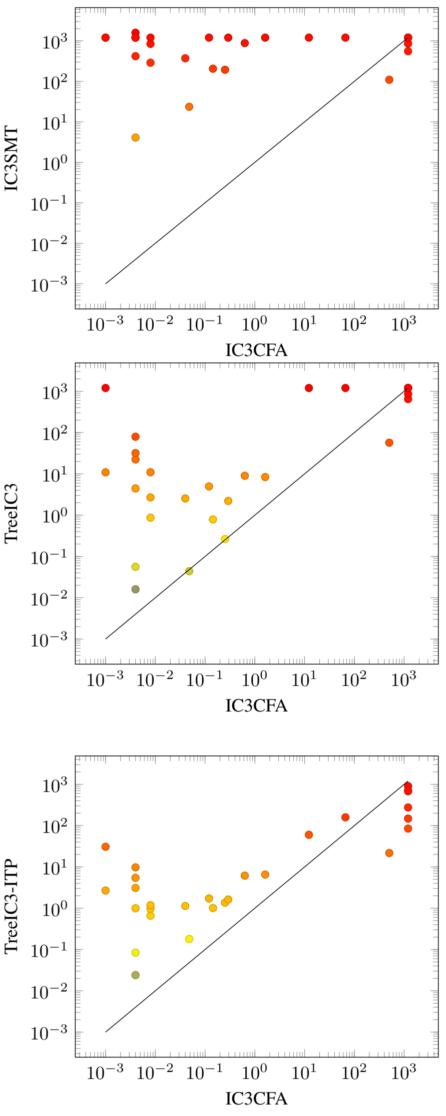
We briefly compare our implementation to the IC3SMT, Tree-IC3 and Tree-IC3-ITP algorithms of [2]. Scatter plots in Fig. 4 give a graphical idea of how IC3CFA compares time-wise against each of these algorithms while the appended table gives statistics of the overall performance of the four algorithms. The second column highlights the number of solved instances of the 28 benchmark programs. The third column shows the time in seconds for solved instances only, while the last column also takes timeouts into account.

IC3SMT, the implementation of the straight-forward lifting of IC3 to SMT as described in the beginning of Sec. IV is clearly the one that performs worst from all four. This comparison is in line with the observations from [2] and shows that IC3CFA benefits from using the explicit representation of control flow over the implicit representation using a dedicated *pc* variable. IC3CFA is able to solve nine more instances than IC3SMT and does so in almost 9% of the time.

The comparison of IC3CFA with the Tree-IC3 implementation shows the effect of constructing frames for control flow locations, rather than unrolling the ART, as both algorithms are based on a very similar representation of control flow automata and both use weakest preconditions to construct the preimage of a state. IC3CFA is able to solve one more instance than Tree-IC3 in a third of the time.

Besides these two direct comparisons we also evaluated our implementation against the Tree-IC3-ITP implementation that almost resembles Tree-IC3 with the only difference in the computation of preimages, where Tree-IC3-ITP uses interpolants. This comparisons shows the advantage of using interpolants over weakest preconditions, as Tree-IC3-ITP is able to solve six more instances than IC3CFA.

[1]Currently we do not support assignment of nondeterministic values inside a loop, due to limitations in our current memory model.



Algorithm	# solve	solve t	total t
IC3SMT	13/28	6328s	24328s
Tree-IC3	21/28	1752s	10152s
Tree-IC-ITP	28/28	3107s	3107s
IC3CFA	22/28	584s	7784s

Fig. 4. Comparison of Tree-IC3 and IC3CFA

Results

From the results in Fig. 4 we can come to three conclusions. First, as already discovered in [2], the IC3SMT approach, while easy to lift, is very inefficient and is not competitive to control flow based techniques. Second, while the TreeIC3 approach is, in very rare cases, slightly more efficient on very small examples, the absence of overhead in ART unrolling makes our approach much faster on medium to large scale examples. Third, the effect of how predecessors are constructed, i.e. weakest preconditions against interpolants, is another important factor that has to be considered when evaluating the performance of IC3 based software model checking algorithms.

VI. Conclusion

In this paper we have presented an approach to lift the application domain of the inductive, incremental verification algorithm IC3 from hardware to software model checking. In resemblance to other adaptations of IC3 to software verification, it supports the handling of infinite-state systems by generalising SAT to SMT solving. Its distinguishing feature, however, is the explicit consideration of a program's control flow by maintaining location-specific frames for storing approximate reachability information. In comparison to more implicit approaches such as Tree-IC3, which is based on an unrolling of the ART, this allows to apply a stronger form of relative inductiveness than obtained in comparable algorithms. The gain in efficiency is demonstrated by experimental evaluations against different implementations of existing, control flow based IC3 algorithms. Due to the novelty of IC3 there is a wide range of future research on this field. One particularly interesting perspective that we intend to investigate are the potentialities of using generalization as a way to overapproximate the exact preimages that we use up to now.

References

[1] Bradley, A.R.: SAT-based model checking without unrolling. In Jhala, R., Schmidt, D.A., eds.: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 70–87

[2] Cimatti, A., Griggio, A.: Software model checking via IC3. In Madhusudan, P., Seshia, S.A., eds.: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 277–293

[3] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In Ábrahám, E., Havelund, K., eds.: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Volume 8413 of Lecture Notes in Computer Science., Springer (2014) 46–61

[4] Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. Volume 1254 of Lecture Notes in Computer Science., Springer (1997) 72–83

[5] Tonetta, S.: Abstract model checking without computing the abstraction. In Cavalcanti, A., Dams, D., eds.: FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Volume 5850 of Lecture Notes in Computer Science., Springer (2009) 89–105

[6] Hoder, K., Bjørner, N.: Generalized property directed reachability. In Cimatti, A., Sebastiani, R., eds.: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Volume 7317 of Lecture Notes in Computer Science., Springer (2012) 157–171

[7] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)

[8] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In Launchbury, J., Mitchell, J.C., eds.: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002, ACM (2002) 58–70

[9] Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In Bjesse, P., Slobodová, A., eds.: International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc. (2011) 125–134

[10] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In Baumgartner, J., Sheeran, M., eds.: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings, IEEE (2007) 173–180

[11] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA, IEEE Computer Society (2004) 75–88

[12] Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 672–678

[13] Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos - a software model checker for SystemC. In Gopalakrishnan, G., Qadeer, S., eds.: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Volume 6806 of Lecture Notes in Computer Science., Springer (2011) 310–316

[14] Steensgaard, B.: Points-to analysis in almost linear time. In Boehm, H., Jr., G.L.S., eds.: Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, ACM Press (1996) 32–41

[15] Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In Ábrahám, E., Havelund, K., eds.: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Volume 8413 of Lecture Notes in Computer Science., Springer (2014) 389–391

[16] Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In Hankin, C., Schmidt, D., eds.: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001,, ACM (2001) 193–205

[17] Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93**(6) (2005) 281–288

[18] Suda, M.: Triggered clause pushing for IC3. CoRR **abs/1307.4966** (2013)

Accelerating Invariant Generation

Kumar Madhukar*[†], Björn Wachter[‡], Daniel Kroening[‡], Matt Lewis[‡] and Mandayam Srivas[†]

*Tata Research Development and Design Center, Pune, Maharashtra, India
Email: kumar.madhukar@tcs.com
[†]Chennai Mathematical Institute, Chennai, Tamil Nadu, India
Email: mksrivas@hotmail.com
[‡]Department of Computer Science, University of Oxford, United Kingdom
Email: bjoern.wachter@cs.ox.ac.uk, kroening@cs.ox.ac.uk, matt@improbable.io

***Abstract*—Acceleration is a technique for summarising loops by computing a closed-form representation of the loop behaviour. The closed form can be turned into an *accelerator*, which is a code snippet that skips over intermediate states of the loop to the end of the loop in a single step.**

Program analysers rely on invariant generation techniques to reason about loops. The state-of-the-art invariant generation techniques, in practice, often struggle to find concise loop invariants, and, instead, degrade into unrolling loops, which is ineffective for non-trivial programs. In this paper, we evaluate experimentally whether loop accelerators enable *existing* program analysis algorithm to discover loop invariants more reliably and more efficiently. This paper is the first comprehensive study on the synergies between acceleration and invariant generation. We report our experience with a collection of safe and unsafe programs drawn from the Software Verification Competition and the literature.

I. Introduction

Consider the program in Fig. 1. It contains a simple assertion, which follows the while loop. An automated proof of safety for this assertion requires a technique that is able to discover the loop invariant $sn = sn + (n - i) * a$. State-of-the-art software model checkers either fail to prove the program or even if they do (for a bounded value of n), they do so by completely unwinding the loop, which does not scale for large n.

```
#define a 2

int main() {
  unsigned int i, j, n, sn = 0;
  j = i;
  while(i < n){
    sn = sn + a;
    i++;
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 1: Sample Safe Program

The simple recurrent nature of the assignments in the loop of program makes it amenable to *acceleration* [1]–[4]. Acceleration is a technique used to compute the effect of repeated iteration of statements. Specifically, the effect of k loop iterations in the example program is that the variable sn is increased by $k * a$. The idea is to replace, wherever possible, a loop with its closed form to obtain an equivalent accelerated program that is hopefully easier to verify.

Acceleration in the general case is, of course, as difficult as the original verification problem. Practical applications of acceleration are therefore typically restricted to particular special cases. For instance, Jeannet et al. [4] consider the case of deterministic linear loops over continuous variables. As there are very few cases in which the transitive closure is effectively computable, it is frequently not possible to obtain an accelerator that captures the behavior of the loop precisely. Thus, acceleration can be over-approximative (most references) or under-approximative (e.g. [5]). Acceleration frequently specialises in particular application domains, e.g., control software. Furthermore, acceleration techniques are frequently tuned to a particular analysis technique (e.g., abstract interpretation or predicate abstraction) that is applied subsequently.

The conjectures of this paper are: 1) accelerators support the invariant synthesis that is performed by program analysers, irrespective of the underlying analysis approach, and 2) analysers supported by acceleration not only do better than the original ones, they also outperform other state-of-the-art tools performing similar analysis. We aim to test these hypotheses by performing an evaluation over an extensive set of benchmarks and a variety of tools. Since all our benchmarks are C programs, we require an acceleration technique that is applicable to C programs and the fixed-width machine integers that they use. We use a template-based method published at CAV 2013 [5] to obtain the accelerators, and add them to the programs as additional paths. This transformation preserves safety i.e., the acceleration neither over- nor under-approximates. We are unable to pass the accelerated programs to all common off-the-shelf analysers, but we nevertheless compare with other tools in our experiments to quantify the advantage that acceleration provides over the state of the art.

Recall our example program. The program with accelerator added is given as Fig. 2. The instrumented code in Fig. 2 can be used instead of the original code for model checking state properties, as they have equivalent sets of reachable states.

This research was supported by ERC project 280053 (CPROVER).

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2

int main(){
  unsigned int i, j, n, sn, k = 0;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
      k = nondet_unsigned(); sn = sn + k*a;
      i = i + k;
      assume(i <= n); } // no overflow
    else{ // original body
      sn = sn + a; i++; }
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 2: Program from Fig. 1 with accelerator

We observe that several model checkers that failing on the original program are able to verify the accelerated program successfully.

The core contribution of this paper is an *experimental study*, with the goal to validate our conjectures stated earlier. We quantify the benefit of accelerators when using commodity program analysers. We use two analysers in our experiments to substantiate the first claim (that accelerators aid existing analyzers). CBMC [6] is the model checker used in [7]; as a bounded analyser, it makes no attempt to infer invariants and is only able to conclude correctness if the program is shallow. IMPARA [8] is a C program verifier based on the LAWI-paradigm. IMPARA generates invariants using a very basic approach that relies on weakest preconditions, and does not employ a powerful interpolation engine.

Both IMPARA and CBMC are characterised by very weak invariant inference, and are thus expected to benefit substantially from acceleration. To relate the outcome to the best invariant generation techniques, towards validating our second claim, we include two other analysers: CPAchecker [9] and UFO [10]. These tools implement a broad range of invariant generation methods, including various abstract domains and interpolation. The comparison is performed on over 200 benchmarks, including those used in the Software Verification Competition 2015.

Although acceleration has successfully been combined with interpolation-based invariant construction [11], to the best of our knowledge, there has not been a thorough experimental study that quantifies the benefits of using it in tools that aim to prove correctness. While [5] did integrate acceleration within a framework where paths in the CFG were explored lazily with refinement, the emphasis of their experiments was to accelerate bug detection for unsafe programs. Recently, a loop over-approximation technique based on acceleration was proposed in [12] but this technique is not applicable to unsafe programs. Moreover, there is no refinement to eliminate spurious counterexamples arising from the over-approximation in [12]. The experiments in [7] focus on bounded model checking and do not include state-of-the-art interpolation-based tools.

The rest of the paper is organized as follows. The next section gives an overview of each of the tools used in our experiments and of the acceleration method from [5], [7] and its scope and restrictions. Section III contains experimental data and a discussion of the results.

II. OVERVIEW OF THE ANALYSIS TOOLS

We start this section with a brief informal introduction of the different tools used for our experiments.

UFO [10] combines the efficiency of abstract interpretation with numerical domains with the ability to generalize by means of interpolation in an abstraction refinement loop. UFO starts by computing an inductive invariant for the given program and checks if the invariant implies the given property. If the implication does not hold, UFO employs SMT solvers to check the feasibility of counterexample produced. If the error path is found to be infeasible, an interpolation technique guided by the results of an abstract interpretation is used to strengthen the invariant.

CPAchecker [9] is a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, implements the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. The framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient way. CPAchecker uses MATHSAT as an SMT solver, and CSISAT and MATHSAT as interpolation procedures. It uses CBMC as a bit-precise checker for the feasibility of error paths, JAVABDD as the BDD package and provides an interface to an Octagon representation as well.

CBMC [6] is a bounded model checker for ANSI-C programs. It works by jointly unwinding the transition relation encoded in the given program and its specification, to obtain a first-order formula that is satisfiable if there exists an error trace. The formula is then checked using a SAT or SMT procedure. If the formula is satisfiable, a counterexample is extracted from the satisfying assignment provided by the SAT procedure. The tool also checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions*. This enables CBMC to prove correctness if the program is shallow.

IMPARA [8] extends the IMPACT algorithm to support asynchronous concurrent processes using an interleaved semantics. IMPARA, which analyses concurrent C programs with POSIX or Win32 threads, efficiently combines partial-order-reduction with the IMPACT algorithm. This paper highlights the benefits of combining IMPARA with acceleration for sequential programs.

The IMPARA algorithm returns either a safety invariant for a given program, finds a counterexample or diverges. To this end, it constructs an abstraction of the program execution in the form of an *Abstract Reachability Tree* (ART), which corresponds to an unwinding of the control-flow graph of the program, annotated with invariants. To prove a program correct for unbounded executions, a criterion is needed to prune the ART without missing any error paths. A covering relation assumes this role.

The tool constructs an ART by alternating three different operations on nodes: EXPAND, REFINE, and CLOSE. EXPAND takes an uncovered leaf node and computes its successors along a randomly chosen thread. REFINE takes an error node v, detects whether the error path is feasible and, if not, restores a safe tree labeling. First, it determines whether the unique path π from the initial node to v is feasible by checking satisfiability of the transition constraints along π. If it is satisfiable, the solution gives a counterexample in the form of a concrete error trace, showing that the program is unsafe. Otherwise, an interpolant is obtained, which is used to refine the labels and update the cover relation. CLOSE takes a node v and checks if v can be added to the covering relation. As potential candidates for pairs to be a part of the covering relation, it only considers nodes created before v. This is to ensure a stable behavior, as covering in arbitrary order may uncover other nodes, which may not terminate.

A. Overview

The acceleration procedure used in this paper is based on the method described in [5]. This method relies on a constraint solver to compute the accelerators. We first provide an overview of the steps of the acceleration procedure, and subsequently provide additional detail. From a high-level perspective, the procedure implements the following steps:

1) Choose a path π through the loop body to be accelerated.
2) Construct a path $\overset{\leadsto}{\pi}$ whose behaviour under-approximates the effect of repeatedly executing π an arbitrary number of times.
3) The construction also generates conditions under which the acceleration is an under-approximation. These conditions are given in the form of two constraints – a *feasibility constraint*, which denotes the condition under which $\overset{\leadsto}{\pi}$ can be applied, and a *range constraint*, which constraints the number of iterations. These constraints are included as *assume* statements in $\overset{\leadsto}{\pi}$.
4) By construction, the assumptions and constraints in $\overset{\leadsto}{\pi}$ may contain universal quantifiers ranging over an auxiliary variable that encodes the number of loop iterations. The procedure uses a few simple techniques to eliminate these quantifiers that work under certain restrictions. The path is not accelerated if it is not able to eliminate the quantifiers.
5) Augment the control flow graph of the original loop body with an additional branch corresponding to $\overset{\leadsto}{\pi}$ with a non-deterministic choice in the branch.
6) The accelerated paths subsume some (or sometimes all) paths in the original program. The augmented loop structure generated in the previous step is analyzed to build a trace automaton that filters some of the redundant paths. The result of this step is used to generate a final program with fewer paths.

The acceleration procedure, after executing the above steps, produces an instrumented code with the modifications described in the last two steps. For a program with several loops, possibly nested, the acceleration procedure processes the loops one at a time, inside-out for nested loops. In our experiments we analyse the instrumented code that is produced, without further modifications. This process of acceleration may succeed, fail or time out. The last two outcomes imply that either a closed form solution with a given template does not exist or acceleration was unable to find one.

In the following, we give a few more details of the procedure, the form of the accelerated paths produced and explain the conditions under which the procedure works.

B. Accelerating Scalar Variables in a Path

For scalar variables, the acceleration is generated by fitting a particular polynomial template. If $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ is the vector of variables in π, then the accelerated assignment generated for each variable is represented by the following polynomial function:

$$\begin{aligned} f_{\mathbf{x}}(\mathbf{X}^{\langle 0\rangle}, n) \stackrel{\text{def}}{=} & \sum_{i=1}^{k} \alpha_i \cdot \mathbf{x}_i^{\langle 0\rangle} \\ & + \left(\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathbf{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)} \right) \cdot n \\ & + \alpha_{(2\cdot k+2)} \cdot n^2 \end{aligned}$$

Here, n is the number of loop iterations that are summarized, $\mathbf{x}_1^{\langle 0\rangle}, \ldots, \mathbf{x}_k^{\langle 0\rangle}$ are the initial values for the variables and the α_i with $0 \leq i \leq 2k + 2$ are the unknown coefficients.

The acceleration for a path is performed in two steps. In the first step, the procedure solves for the coefficients α_i. This is done by considering only the assignments in the path π, i.e., by ignoring all the conditions, including the loop condition. This employs a combination of linear algebra techniques to first uniquely solve for the coefficients and then makes queries to SMT solver to inductively check that the generated polynomial for each variable is consistent with loop execution for an arbitrary number of iterations. If, for some $\mathbf{x}_i$, the inductive check fails, then it means there is no acceleration possible that fits the template.

In the second step the procedure considers the path with all the conditions, and generates the feasibility constraint, i.e., the condition under which the path is feasible. The feasibility constraint is essentially the negation of $wlp(\pi^n; false)$, where wlp is the weakest liberal precondition. Intuitively, a cumulative path π^n would be infeasible iff any intervening path π in the n-iteration cycle, starting from the state given by the accelerator, is infeasible. That is, π^n is infeasible if for any $j < n$ the first time frame of the suffix $\pi^{(n-j)}$ is infeasible (time frame refers to an instance of π in π^n). Thus, checking whether $wlp(\pi^n, false)$ holds is equivalent to checking if, for some j between 0 and n, $wlp(\pi, \mathit{false})$ holds (after substituting every variable in π by its accelerated closed form expressions). Thus, the feasibility constraint for π^n will, in general, contain a universal quantifier ranging over the number of loop iterations. This can be eliminated if the predicate in the body of the formula is monotonic over the quantified parameter. The procedure reduces the monotonicity check in a conservative fashion to a SMT query by defining a *representing function* that returns the size of the set of states for which a predicate is false. No acceleration is performed if the monotonicity check fails.

C. Range Constraints

Since closed-form expressions and the derived feasibility constraints usually contain the number of iterations n in them, an overflow is likely to break the monotonicity requirement when bit-vectors or modular arithmetic are used. Also, since the behaviour of arithmetic over- or under-flow in C is not specified for signed arithmetic, we conservatively rule out all occurrences thereof in the accelerated path. This is done by adding range constraints in the form *assume* statements, which enforce that none of the arithmetic expressions that involve n overflow.

D. Accelerating Array Assignments

Acceleration of array assignments is challenging, as under-approximating closed-form solutions for them can often only be expressed by formulas that contain quantifier alternation (existential inside universal) ranging over the number of loop iterations and the domain (index) of the array. It has been shown in [5] that for array assignments of the form $a[x] := e$ such a quantifier pattern can be eliminated under the following sufficient conditions.

- There exist accelerated closed-form expressions for the index variable x and the expression e.
- The function f_x defining the closed-form solution for the index variable is linear in the number of loop iterations.

Under the above conditions one can derive a closed form representing an under-approximation of the array assignments.

E. Eliminating Redundant Paths using Trace Automata

The instrumentation of the accelerators described in the introduction preserves the unaccelerated paths in the program along with the newly added accelerated paths – for instance, the *else* branch in Fig. 2. Note that the added paths subsume some of the previously existing program paths.

The idea presented in [7] is to eliminate executions that are subsumed by some other execution of the program. For instance, taking the same accelerated path twice in a row is equivalent to taking it just once (for instance, in Fig. 2, executing the *if* block twice for values k_1 and k_2 is the same as executing it once with the value of k equal to $k_1 + k_2$ – which is possible because k is chosen non-deterministically in each iteration).

Similarly, taking the unaccelerated path immediately after taking the accelerated path is subsumed by taking the accelerated path just once (with the value of k being one more its previously chosen value, in Fig. 2). The elimination of these redundant paths is done by encoding the redundancies as a regular expression, which is then translated into a *trace automaton* [13]. When the accelerated program executes, the states in this automaton are also updated and it is ensured that this automata never reaches a *reject* state. An optimized version of the accelerated code for the running example is given in Fig. 3. This is achieved by introducing an auxiliary variable g that determines whether the accelerator was traversed in the previous iteration of the loop. This flag is reset in the non-accelerated branch, which, however, in our example is infeasible.

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2
int main(){
  unsigned int i,j, n, sn, k = 0;
  bool g = *;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
       assume(!g);
       k = nondet_unsigned(); sn = sn + k*a;
       i = i + k;
       assume( i <= n); // no overflow
       g = true;}
    else{ // original body
       sn = sn + a; i++;
       g = false;}
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 3: Program from Fig. 2 with instrumented trace automaton

III. Experimental Results

A. Experimental Setup

We ran our experiments on a set of 201 benchmarks (138 safe, 63 unsafe) collected from the sources listed in [14] (published at CAV 2014) and SV-COMP 2015. We have eliminated examples that had syntax errors and the ones that were not supported by the accelerator (array examples, for instance). We compare the performance of UFO, CPAchecker, CBMC (with and without acceleration) and IMPARA (with and without acceleration). The unwinding depth used for experiments with CBMC was 100 for unaccelerated programs and 3 for accelerated programs. All experiments were run on a dual-core machine running at 2.73 GHz with 2 GB RAM, with a timeout limit of 60 seconds.

We elaborate on the benchmarks and the tools used to aid reproducibility. The benchmarks were collected from [15]–[17], the loops category in SV-COMP 2015 and the acceleration examples in the regression suite of CBMC (revision 4503). The tools used in the experiment were UFO (the SV-COMP 2014 binary), CPAchecker (release 1.3.4, with sv-comp14.properties as the configuration file), CBMC (built from revision 4503, used with Z3 as the decision procedure) and IMPARA (version

```
int main(void) {
  unsigned int x = 0;
  while (x < 268435454) {
    if(x < 65520){
      x++;
    } else {
      x += 2;
    }
  }
  assert(!(x % 2));
}
```

Fig. 4: A safe benchmark showing the need for acceleration.

0.2, used with MiniSat). The benchmarks, the exact commands used to invoke the tools, and the full results are available at http://www.cmi.ac.in/~madhukar/fmcad15.

Table I summarizes the performance of each of the tools. We record the number of safe instances reported as safe (*correct proofs*), the number of safe instances reported as unsafe (*wrong alarms*), the number of unsafe instances reported as unsafe (*correct alarms*), the number of unsafe instances reported as safe (*wrong proofs*), the number of instances which could not be decided by the tool (*no result*), the number of instances on which the tool reported the correct result in the least amount of time (*fastest*), the number of instances on which the tool was the only one to report the correct result (*unique*) and a score for each tool, calculated using the scoring scheme of SV-COMP 2015.[1]

B. Example

Before we discuss the results, we present an example to demonstrate the effectiveness of acceleration. Consider the safe example shown in Figure 4. All the tools involved in our experiments fail to prove this example safe. Even when the timeout is increased to 15 minutes, the tools still timeout. In general, one needs a loop invariant strong enough to prove the assertion outside the loop, to avoid unwinding the loop to the full. None of the tools were able to find such a loop invariant. Upon acceleration, a closed form for the variable x is generated: $x = 1 * k + 2 * l$. The additional constraint generated for k, that $k = 65520$, along with the closed form for x (and negation of the loop termination condition) is strong enough an invariant to prove the property.

In some circumstances, acceleration uses quantifiers in the accelerated programs. These are not the ones arising from the feasibility or range constraints that we discussed in Section II (those get eliminated during the acceleration). These quantifiers appear while encoding the overflow constraints in the accelerated program. Suppose we want to construct a closed form for a variable being modified in a loop, by assuming that the loop executed i times. In this case, we need to assure that there is no overflow that was caused during any of these i iterations. In some cases, it is sufficient to assume that i^{th} iteration does not lead to an overflow. An instance is example 4, as the loop condition is $(x < 268435454)$. Thus, if the i^{th} iteration does not lead to an overflow, none of the previous iterations do. However, if we change the loop condition to $(x \neq 268435454)$ this does not hold any more. Therefore, it must be ensured separately for every $k \in [0, \ldots, i]$ that there is no overflow after k iterations. In our experiments, there were 40 benchmarks (roughly 25 %) that use quantifiers in their corresponding accelerated programs. The presence of quantifiers makes the verification task difficult as none of the tools is able to instantiate the quantifiers correctly. More effective quantifier handling will yield further results in favor of acceleration.

[1]Score $= (2 \cdot \textit{correct proofs}) - (12 \cdot \textit{wrong proofs}) + \textit{correct alarms} - (6 \cdot \textit{wrong alarms})$

C. Discussion of Results

IMPARA + Acceleration clearly outperforms IMPARA without acceleration, UFO and CPAchecker. This underlines the benefit of acceleration as an auxiliary method for invariant generation. Note that we see an increase in the number of *correct proofs* as well as *correct alarms*. CPAchecker comes close in terms of the *correct proofs*, which we credit to its broad portfolio of techniques for generating invariants, including interpolation, abstract interpretation and predicate abstraction. The *wrong proofs* CPAchecker generates are partly caused by missing overflow situations.

When compared to CBMC + Acceleration, IMPARA + Acceleration does better for the following reason: The accelerators themselves are not helpful to CBMC for generating proofs – it simply unwinds the program CFG and makes a single decisive query to the solver. A large number of our benchmarks are safe, and CBMC only benefits from accelerators if the trace automaton is able to prune the original paths. By contrast, even without trace automata, acceleration may improve convergence of IMPARA, as acceleration can lead to “better” interpolants. Without acceleration an interpolation procedure is presented an unwinding of the loop body. It is well-known, see e.g. [18], that this can lead to overly specific interpolants that rule out only this particular unwinding. By contrast, in the accelerated program, the interpolation procedure is presented with the transitive closure of the loop; it thus is forced to compute an interpolant for a much larger number of unwindings. For instance, IMPARA without acceleration fails to generate a loop invariants for Figure 5, and thus falls back to loop unwinding, whereas, on the accelerated program, unwinding is avoided, and the tool generates the invariant $x + y = n$.

The overall score drops when combining CBMC with acceleration. This is due to the wrong alarms generated by the combination, which is heavily penalized according to the scoring rules at SV-COMP. There is a substantial increase in the number of correct proofs and correct alarms, however. The advantages of combining acceleration with CBMC and IMPARA (note that CBMC and IMPARA are very different tools) strongly suggests that a similar advantage could be obtained with other tools as well. An investigation of the cause for the increase in

TABLE I: Comparison of tools

Tools	Number of instances							Score
	correct proofs	wrong proofs	correct alarms	wrong alarms	no results	fastest	unique	
CPAchecker 1.3.4	83	16	35	14	53	18	11	−75
UFO SV-COMP 2014	52	2	18	2	127	4	2	86
CBMC r4503	32	0	35	0	134	16	1	99
+ Acceleration	53	0	45	12	91	28	9	79
IMPARA 0.2	78	1	36	15	71	73	0	90
+ Acceleration	86	0	47	12	56	36	6	147

```
int main(){
  unsigned int n = nondet_uint();
  int x = n;
  int y = 0;

  // loop invariant: x + y == n
  while(x > 0){
    x = x - 1;
    y = y + 1;
  }
  assert(y == n);
}
```

Fig. 5: Acceleration can improve generalisation in LAWI.

number of wrong alarms for CBMC and a precise quantification of the benefit of combining other tools is future work.

The fact that acceleration helps CBMC and IMPARA on unsafe instances is unsurprising; the technique we use was designed to aid counterexample detection [5]. The experimental results confirm that in addition, acceleration helps to generate invariants. Invariant generation techniques, in practice, often struggle to find concise loop invariants, and, instead, degrade into unrolling loops completely, which leads to poor performance and defeats the purpose of invariant generation. Our experiments demonstrate that there is a synergy between the two techniques, i.e., acceleration leads to better invariants, and invariant generation also helps finding bugs faster. We conjecture that the invariants steer the search for the bug away from irrelevant parts of the state space.

While CPAchecker employs a bit-accurate tool – by default CBMC – to verify counterexamples, its invariant-generation engine works over mathematical integers, i.e., invariants may hold over mathematical integers but are not checked with respect to integer overflow. Wrong proofs observed with CPAchecker mainly arise from deriving mathematical-integer invariants that do not hold in presence of overflow. In such situations, acceleration cannot help. This can be explained as follows. The accelerator represents a transitive closure of the loop body. It follows easily by induction that the result of CPAchecker, if the tool terminates, must be the same as for the unaccelerated program, since both programs are semantically equivalent.

IV. CONCLUSION AND FUTURE WORK

In this paper we have quantified the benefit of acceleration for checking safety properties. We report the results of a comprehensive comparison over a number of benchmarks, which shows that the combination of acceleration and a safety checker indeed outperforms existing techniques. The performance enhancement is visible for both safe and unsafe benchmarks, shown by an increase in the number of correct alarms as well as the correct proofs reported by the tool.

The source-level transformation of programs enables integration with futher invariant generation techniques. As a future work, we plan to investigate the interplay between acceleration and invariant generation to minimize the number of wrong alarms and to handle more cases correctly, including those that involve arrays. We also believe it would be worthwhile to investigate whether the accelerator can be assisted with additional invariants generated using some other technique. Our initial experiments suggest that some of these invariants, even over the interval domain, may help us rule out the possibility of overflows, thereby increasing the precision of the accelerator.

REFERENCES

[1] B. Boigelot, *Symbolic Methods for Exploring Infinite State Spaces*, ser. Collection des publications. Université de Liège, Faculté des sciences appliquées, 1999.

[2] M. Bozga, R. Iosif, and F. Konečný, "Fast acceleration of ultimately periodic relations," in *Computer Aided Verification*, ser. LNCS. Springer, 2010, vol. 6174, pp. 227–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_23

[3] A. Finkel and J. Leroux, "How to compose Presburger-accelerations: Applications to broadcast protocols," in *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, ser. LNCS. Springer, 2002, vol. 2556, pp. 145–156. [Online]. Available: http://dx.doi.org/10.1007/3-540-36206-1_14

[4] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," in *Principles of Programming Languages (POPL)*. ACM, 2014, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535843

[5] D. Kroening, M. Lewis, and G. Weissenbacher, "Under-approximating loops in C programs for fast counterexample detection," in *Computer Aided Verification (CAV)*. Springer, 2013, pp. 381–396. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_26

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2004, vol. 2988, pp. 168–176. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24730-2_15

[7] D. Kroening, M. Lewis, and G. Weissenbacher, "Proving safety with trace automata and bounded model checking," in *Formal Methods (FM)*, ser. LNCS, vol. 9109. Springer, 2015.

[8] B. Wachter, D. Kroening, and J. Ouaknine, “Verifying multi-threaded software with Impact,” in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 210–217. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679412

[9] D. Beyer and M. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *Computer Aided Verification (CAV)*, ser. LNCS. Springer, 2011, vol. 6806, pp. 184–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_16

[10] A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik, “UFO: Verification with interpolants and abstract interpretation,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer, 2013, vol. 7795, pp. 637–640. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_52

[11] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer, “Accelerating interpolants,” in *Automated Technology for Verification and Analysis (ATVA)*, ser. LNCS. Springer, 2012, pp. 187–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33386-6_16

[12] P. Darke, B. Chimdyalwar, R. Venkatesh, U. Shrotri, and R. Metta, “Over-approximating loops to prove properties using bounded model checking,” in *Design, Automation & Test in Europe (DATE)*. EDA Consortium, 2015, pp. 1407–1412. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757139

[13] M. Heizmann, J. Hoenicke, and A. Podelski, “Refinement of trace abstraction,” in *Static Analysis (SAS)*, ser. LNCS. Springer, 2009, vol. 5673, pp. 69–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03237-0_7

[14] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, “Counterexample to induction-guided abstraction-refinement (CTIGAR),” in *Computer Aided Verification (CAV)*, ser. LNCS. Springer International Publishing, 2014, vol. 8559, pp. 831–848. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_55

[15] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, “Dagger Benchmarks Suite,” http://www.cfdvs.iitb.ac.in/∼bhargav/dagger.php, 2014.

[16] A. Gupta and A. Rybalchenko, “InvGen Benchmarks Suite,” http://pub.ist.ac.at/∼agupta/invgen/, 2014.

[17] A. Albarghouthi and K. McMillan, “Beautiful interpolants,” in *Computer Aided Verification*, ser. LNCS, 2013, vol. 8044, pp. 313–329. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_22

[18] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, “Path invariants,” in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 300–309. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250769

Comparing Different Functional Allocations in Automated Air Traffic Control Design

Cristian Mattarei, Alessandro Cimatti, Marco Gario, and Stefano Tonetta
Fondazione Bruno Kessler - Trento, Italy
Email: {mattarei, cimatti, gario, tonettas}@fbk.eu

Kristin Y. Rozier
University of Cincinnati - Ohio, USA
Email: rozierky@uc.edu

***Abstract*—In the early phases of the design of safety-critical systems, we need the ability to analyze the safety of different design solutions, comparing how different functional allocations impact the overall reliability of the system. To achieve this goal, we can apply formal techniques ranging from model checking to model-based fault-tree analysis. Using the results of the verification and safety analysis, we can compare different solutions and provide the domain experts with information on the strengths and weaknesses of each solution.**

In this paper, we consider NASA's early designs and functional allocation hypotheses for the next air traffic control system for the United States. In particular, we consider how the allocation of separation assurance capabilities and the required communication between agents affects the safety of the overall system. Due to the high level of details, we need to abstract the domain while retaining all of the key properties of NASA's designs. We present the modeling approach and verification process that we adopted. Finally, we discuss the results of the analysis when comparing different configurations including both new, self-separating and traditional, ground-separated aircraft.

I. Introduction

By 2025 the airspace will be full [1]; demand for flights will exceed the maximum number of planes that can fly at one time. This problem is not due to a space limitation; there is room for more planes in the air. We will instead exceed the ability of our current system to safely separate commercial aircraft and provide on-the-fly conflict detection and resolution. This is because our current system relies heavily on human air traffic controllers and there is a limit to the number of planes humans can reason about simultaneously. We can solve this problem by adding automation and enabling computers to compute routes and resolutions to maintain safe separation between planes; we already have implementations of optimized algorithms for doing this [2]. But human controllers do much more than 3D geometric reasoning; the entire web of communications between agents in the system, the distributed control structure, and the logical system design all play integral roles in making air traffic control so safe and reliable. If we design a new, more automated system, how do we allocate all of the functions it must perform in a way that upholds at least the current level of safety? This is the *functional allocation* question.

Thanks to the Flight Trajectory Dynamics and Controls Branch of NASA Ames Research Center and NASA's Functional Allocation Project for supporting this work. All models and specifications are available at https://es-static.fbk.eu/projects/nasa-aac/.

The functional allocation question is first and foremost about safety: our goal is to create a partial order on the set of ways to allocate system functions such that system designers can choose a *most safe* configuration and then optimize for secondary goals, such as cost, scheduling, fuel efficiency, ease of use, and environmental impact.

To this purpose, we considered the requirements specification described in the NASA research plan [3] and interacted with NASA engineers to formalize the functions in different allocation configurations, as well as several system requirements to be analyzed. Model checking these properties on different configurations gives us a first means for comparing and ranking the design choices. As a second step, we analyzed functional allocation by extending the model with faults and analyzed the resulting fault trees [4]. Fault trees are commonly used [5] in safety critical contexts, such as aerospace [6], in order to understand which combination of faults can lead to a violation of a safety property. On top of the fault trees, we compute multiple metrics (including probabilistic ones) to compare the different allocation configurations.

We use the NUXMV [7] for model checking and XSAP [8] for fault-tree analysis. Together with the models, the artifacts produced by these tools provide information that goes beyond a mere pass/fail result, providing a rich characterization of the conditions under which properties pass or fail. This enables design choices that minimize the impact of faults on the overall system. To our knowledge, this is the first time that such artifacts have been utilized in the conceptual phase of a real design, when requirements are still blurred and there is no existing concrete design solution to compare with.

Related Work

The complexity of safety-critical systems is continuously increasing. Yet, the current state-of-the-practice is largely characterized by manual approaches, which are error prone, and may ultimately increase the costs of certification. This has motivated, in recent years, a growing interest in techniques for Model-Based Safety Assessment [9]. The perspective of model-based safety assessment is to represent the system by means of a formal model and perform safety analysis, both for the preliminary architecture and at system level, using formal verification techniques. The integration of model-based techniques allows safety analysis to be more tractable in terms of time consumption and costs. Such techniques must be able to verify functional correctness and assess system behavior in the presence of faults [10], [11], [12].

Formal analysis techniques have been applied in the context of NASA's Automated Airspace Concept (AAC) in [13]. That work focuses on analyzing the design proposed in [14] in which the current techniques for Air Traffic Control are extended with automated on-ground support (i.e., TSAFE and Autoresolver). [13] opens the way to the application of symbolic model checking techniques in this context; the analysis is then applied in a probabilistic setting in [15]. In this paper, we start from a more preliminary design proposal (described in NASA research plan [3]) in which we consider the distributed nature of separation assurance in systems were both ground- and self-separated aircraft coexist. To capture the interaction between the different agents, we develop a different modeling abstraction. Moreover, to provide interesting comparative information related to the safety of the designs, we apply safety assessment techniques, such as fault tree analysis.

Other works, e.g., [16], [17] focused on the formal verification of specific functions such as collision avoidance. In this paper, we assume that such functions are correct and we focus on the safety analysis of the overall system.

Another case study using the same tools is presented in [18]. In that case, an avionic wheel braking system is modeled and analyzed according to the standard AIR6110. Different architectures are considered following the process described in the standard. The main differences are that first it is applied on a well-established architecture and not on a design still in the conceptual phase; second, it describes an architecture where the controller is fixed and localized to a specific component instead of relying on a distributed and variable system; finally, the focus on functional allocation addressed in this paper gives more emphasis on some techniques such as the functional analysis of the system reliability in different configurations with respect to the failure probability of a function.

Contributions

The main contribution of this paper is the adaptation of model checking and model-based safety analysis to formally compare different scenarios of an early system design, as required by NASA's functional allocation question. This required a careful definition of the methodology for modeling and fault-tree analysis with the aim of comparing different configurations keeping an abstract view of the single functions. In particular, we handle modeling subtleties in creating a realistic model, such as receptiveness of faults, shadowing, and multiple disjoint communications. Moreover, we verified a list of functional safety features, and computed artifacts describing the reliability of the system with respect to function failures. The outcome of this work allowed us and NASA to reach a better understanding of the design space. Thus, helping NASA shape the work of research groups.

Outline

The rest of the paper is organized as follows: Section II lists the functions and agents that define the functional allocation question and the steps that we followed in the modeling and analysis process; Section III describes the formal model, including the architecture and the abstraction of the real system based on conflict areas and time windows; Section IV describes the properties used to validate the model; Section V describes the formal properties used to characterize the configurations; Section VI explains the use of fault-tree analysis to compare the configurations; Section VII covers modeling subtleties and lessons learned while Section VIII concludes.

II. Functional Allocation for the Automated Air Traffic Control System

A. *Problem description*

NASA is tasked with designing the next, more automated, air traffic control system for the United States. A major safety goal is to minimize Loss of Separation (LoS), resolve any such situations immediately, and never call upon collision avoidance. LoS occurs when two or more aircraft become too close to each other, i.e., they are below a defined safe distance of 1000 feet vertical and 5 nautical mile horizontal separation. If LoS is not resolved immediately, collision avoidance is necessary. The *functional allocation question* asks which separation assurance (SA) capabilities to require and how to distribute the functions of the design in combination with a subset of these capabilities on top of a set of agents, in order to minimize the number of LoS and the use of collision avoidance techniques [3]. We consider the following agents, functions, and capabilities:

Functions:

- **Strategic Separation** addresses short-term conflicts from 20 minutes in the future down to 3 minutes out from a predicted LoS. Strategic separation is implemented in software and can be running on a central computer on the ground, on-board individual aircraft, or some combination thereof. It uses the trajectories of each known aircraft in the airspace, detecting any conflicts, and outputting resolution maneuvers for any aircraft involved in conflicts.
- **Tactical Separation** addresses near-term conflicts predicted to occur less than 3 minutes in the future. It is also implemented in software running on either a ground computer, an on-board computer, or a combination thereof. Tactical separation must employ a different algorithm from strategic separation because the conflicts it addresses are more imminent and different details must be considered when generating resolution maneuvers.
- **Collision Avoidance** addresses possible collisions less than 30 seconds in the future. Its presence is required by Federal Aviation Administration (FAA) mandate, therefore, TCAS (and in the future ACAS-X), software runs on-board every aircraft, detects possible collisions using a transponder installed in the aircraft, and must operate

totally independently from on-ground systems. A system safety objective is to never trigger collision avoidance.

Agents:

- **Self-Separating Aircraft (SSEP)** carry a separation assurance software on-board.
- **Ground-Separated Aircraft (GSEP)** rely on SA software running on a central on-ground computer transmitting to the aircraft.
- **Air Traffic Control (ATC)** Provides on-ground separation of GSEPs and, when needed, of SSEPs.

Capabilities:

- **ADS-B Out** (Automatic Dependent Surveillance-Broadcast Out) is required on-board all aircraft by FAA mandate by 2020; it broadcasts position information to ADS-B ground stations and other aircraft within transmission range.
- **ADS-B In** is optional by FAA regulations; it receives ADS-B broadcasts from ground stations and other aircraft.

Depending on who is in charge of what, and the available resources, we can describe different designs. Different designs will have different characteristics. Our goal is to provide some qualitative measure of the goodness of each solution along different dimensions. For example, in a scenario in which both GSEP and SSEP aircraft are involved, we might want to know whether a solution in which SSEPs perform both tactical and strategic separation on-board is "better" than a solution in which tactical separation is handled on-ground.

B. Overview of the process based on formal techniques

We approach the problem described above using formal methods. We adopted the following four steps process.

1) Modeling (Sec. III): we formalized the system scenarios described in [3]; the informal specification is very abstract and includes only the aspects related to the interaction among the agents; therefore our formalization must choose the right level of abstraction capturing the relevant aspects.
2) Validation (Sec. IV): we performed sanity checks to validate that the formalization of model and properties captures their informal descriptions; in particular, the formal model describing aircraft and controllers are analyzed separately to validate that certain behaviors are allowed.
3) Verification (Sec. V): we formalized the requirements into temporal properties and verified them in the different configurations; some properties must be satisfied by all configurations, but others are used to distinguish and compare different configurations.
4) Safety analysis (Sec. VI): in the previous step, we evaluated the models under nominal conditions, i.e., each component behaves correctly; in the fourth step, each component was extended by adding faulty behaviors, and we evaluated the safety and reliability of the configurations using fault-tree analysis. To compare the different configurations, we analyzed under which failure conditions the system can violate the system requirements.

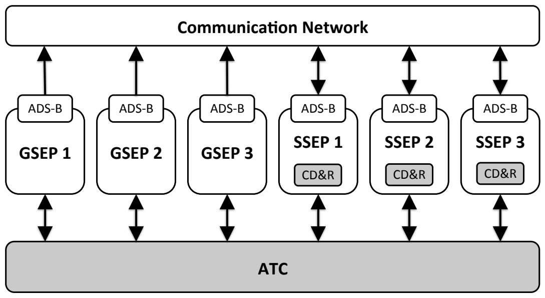


Fig. 1: Scenario instances

III. Formal Modeling for Comparative Analysis

A. System Architecture

In this section, we described the model used to analyze and compare different configurations of the functional allocation. The model describes different possible configurations on the number of aircraft. It does not consider the whole airspace, but only the set of aircraft that can be in a conflict on intended trajectories. Both SSEPs and GSEPs aircraft types are taken into account, in addition to ATC and a communication network at airborne level. Figure 1 provides an overview of our model, which allows us to describe a variety of scenarios by enabling or disabling some specific aircraft: only GSEPs or only SSEPs operations (by disabling respectively all SSEPs or all GSEPs), or mixed GSEPs/SSEPs scenario. All SSEP aircraft perform self-separation for the strategic separation with a Conflict Detection and Resolution (CD&R) onboard function, while they rely on the ATC for tactical separation. GSEPs always rely on ground ATC for both tactical and strategic separation. In case an SSEP experiences problems, it is able to ask the ground for strategic separation, thus being treated as a GSEP. The aircraft communicate directly with the ATC while they broadcast messages to other aircraft using the ADS-B. The broadcast is handled by the communication network.

It is important to define the right level of abstraction in order to guarantee that all the relevant aspects are taken into account. In the following sections, we detail what variables define the state of the system, how time passes, and how this influences the change in the state. Note that our analysis focuses on the protocol level and thus, in absence of faults, we assume each component implementation to be correct.

B. Trajectory Intentions and Conflict Areas

The basic information that is relevant for our analysis are the trajectories that the aircraft intend to follow, and more specifically if their intentions are in conflict with each other. The actual detail of the trajectories (i.e., the 3D position as a function of time) is not part of our model. In fact, we reason about the system at the architectural level, focusing on the interaction between the components rather than on the precise behavior of the components. We are not interested in which specific trajectory an aircraft should follow to avoid a collision,

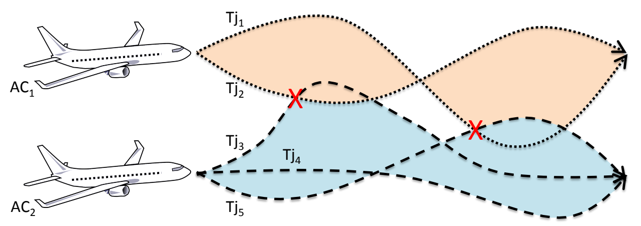


Fig. 2: Conflict Areas abstraction

but only in whether their intentions are in conflict or not. Therefore, we abstract away the detailed trajectory information by introducing *Conflict Areas* (CA). Intuitively, two aircraft are in the same CA, if their trajectories intersect in a given interval of time. In this way, we can abstract the problem of separation into the simpler problem of checking that two aircraft are not in the same conflict area. Figure 2 shows an example when two aircraft have to reach two separate destinations. In this example we consider Tj_1 and Tj_2 for AC_1, and Tj_3, Tj_4, Tj_5 for AC_2. Figure 2 shows that AC_1 and AC_2 are in the same CA if their intended trajectories are respectively Tj_1 and Tj_5, or Tj_2 and Tj_3. In all other cases, they are into different CAs, representing the absence of conflicts. CAs are used throughout our models anytime we talk about aircraft intentions and resolutions sent by controllers.

C. Time windows

Most scenarios in [3] divide the responsibility of the separation-assurance agents based on *time windows*. In particular, we consider four time windows: *Current*, *Near*, *Mid* and *Far*. They represent symbolically consecutive time intervals. Therefore, the trajectory intention of the aircraft define which aircraft are in the same CA in each time window, as defined in the previous section.

The Current window represents the immediate intention of the aircraft, i.e., within 30 seconds. This window is managed by *Conflict Avoidance* algorithms, e.g., TCAS, and is therefore the key to the definition of LoS: two aircraft are currently in LoS if they share the same conflict area in the Current window. The tasks of tactical and strategic separation are then mapped into the Near- and Mid-window (Tactical) and the Far-window (Strategic). If two aircraft share the same conflict area in the same window, we say that we have a *predicted* LoS.

In our model, the intention of aircraft is represented by assigning each airplane with a CA for each window. Figure 3 shows an example with two aircraft. In this example, the aircraft are in different CAs apart from the Far window. So, we have a predicted LoS in that time window.

Intuitively, the windows shift with the passage of time: the old Near information will became the new Current information (Figure 3), while the intention for the other time windows change according to the interaction among the agents. Therefore, if we manage to resolve all predicted LoS, e.g., in the Mid window, we will not have LoS. In order for conflicts to be detected and resolved, we need to take into account

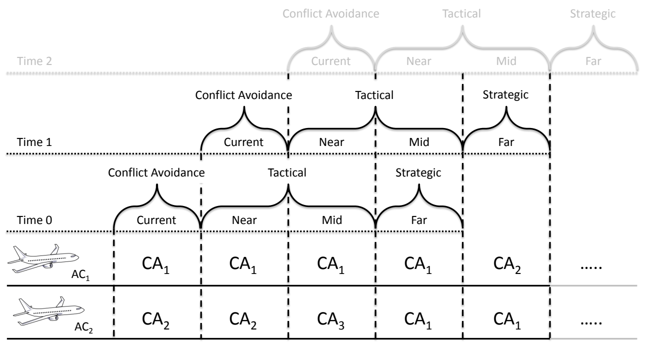


Fig. 3: Near, Mid, Far windows, and their shifting

the communication between aircraft and the ATC and when it occurs. In the model, passing of time is divided into two main phases that alternate constantly: *communication* and *maneuvering*.

During the maneuvering phase, windows are shifted (Figure 3). During the communication phase, the different agents are able to exchange intentions and resolutions. For example, the aircraft is able to provide its intention to the ATC, and receive a suggestion for a new trajectory. We introduce a bound on the number of communications during this phase, in order to better understand whether multiple iterations between agents can improve the reliability of the system. This interleaving model may seem unintuitive. However, this choice is justified by reality since we can only apply a maneuver after deciding it, and it simplifies the modeling.

D. Scenarios Instantiation

As described in Sec. II, our exemplary scenario allows self-separating aircraft (SSEP), which defines three sub-scenarios: i) non-mixed operations with only GSEPs (current approach); ii) mixed operations with both GSEPs and SSEPs; iii) non-mixed operations with only SSEPs.

We consider a “four aircraft” scenario, where at most four aircraft can be involved in a single conflict at one time. This realistically covers the actual system since conflicts involving more than two aircraft are exceedingly rare [14].

All possible configurations are represented by relying on a single formal model, where each configuration is modeled by enabling or disabling a subset of the components. For instance, considering the model representation shown in Fig. 1, the mixed operation scenario with 2 GSEPs and 1 SSEP is obtained by disabling SSEP 2 and 3.

On top of that, our exemplary scenario describes different possible implementation choices at the communication level that can be enabled (E) or disabled (D):

- GSEP-far: GSEPs send far intentions over ADS-B Out;
- SSEP-far: SSEPs send far intentions to ATC.

Table I shows the size of the different scenarios in term of Boolean variables and AND gates using And-Inverter Graphs (AIG). The last row contains the biggest configuration, which is composed of 353 bits and 4110 AND gates. The first column defines the code used in the rest of the paper to refer to specific scenarios.

Scenario code	Components			# Bool. vars	# AND gates
	GSEPs	SSEPs	ATC		
PA	3	3	✗	283	2226
G	3	0	✓	122	1119
M1	3	1	✓	185	1767
M2	2	2	✓	193	1908
M3	1	3	✓	201	2050
S	0	3	✓	146	1413
ALL	3	3	✓	353	4110

TABLE I: Scenario instances (AIG format)

IV. Validation

The scenarios that are taken into account in this work represent the interaction between a *controller* (the Air Traffic Control and the CD&R on-board of the SSEPs, i.e., the gray components in Fig. 1), and the *controlled system* (the set of aircraft). The objective of this work is to analyze how the Separation Assurance agents control the aircraft. In order to avoid a vacuous verification, we first need to validate separately *controllers* and *system*.

A. Validation Properties Formalization

In order to validate the *system* we identified the following requirements:

VAS-1: It is always possible to reach a LoS.
VAS-2: It is always possible to have no LoS.
VAS-3: It is always possible for an aircraft to maintain the same current intention.
VAS-4: It is always possible for an aircraft to change its intention.

The CTL formalization of the validation requirement VAS-1 is exemplified in (1). More specifically, we want to define that for every state of the system it is always possible to reach a state where Loss of Separation (LoS) holds. The LoS condition applies when at least two aircraft are in the same current conflict area.

$$\begin{aligned} \text{VAS-1} &:= AG(EF(\text{LoS})) \\ \text{LoS} &:= \bigvee_{i \neq j} ac_i.\text{current} = ac_j.\text{current} \end{aligned} \tag{1}$$

The expected behavior of the *controllers* is then defined by the following requirements:

VAC-1: The controller should accept any possible trajectory intent from every aircraft.
VAC-2: The controller should always send a correct resolution.

We validated the model using nuXmv [7], and the results were positive for all 37 properties that formalize the validation requirements.

V. Verification

The next phase starts with the formalization of a set of properties gathered from the requirements document [3]. We then check whether different configurations satisfy the formal properties. The results of these checks provide us with additional information on the difference between the configurations.

A. Requirements Formalization

We consider the following requirements:

VE-1 It is never possible to reach a Loss of Separation (LoS).
VE-2 It is never possible to have a predicted LoS in the Near-, Mid-, or Far-Window.
VE-3 Every predicted LoS in the Near-, Mid-, or Far-Window is detected by at least one SA agent.
VE-4 Every predicted LoS is detected by at least one SA agent.
VE-5 Resolutions sent by the agent resolve the predicted LoS.
VE-6 Each aircraft must correctly apply the resolution.

The requirement formalization required 93 LTL properties, and their consistency has been validated using the Requirement Analysis Tool [19].

The formal interpretation of the verification property VE-2 is shown in (2), as a composition of the constraints on the near, mid, and far window.

$$\begin{aligned} \text{VE-2} &:= \text{VE-2}_{\text{near}} \wedge \text{VE-2}_{\text{mid}} \wedge \text{VE-2}_{\text{far}} \\ \text{VE-2}_{\{\text{near,mid,far}\}} &:= \bigwedge_{i \neq j} ac_i.\{\text{near,mid,far}\} \neq ac_j.\{\text{near,mid,far}\} \end{aligned} \tag{2}$$

B. Formal Property Verification

All properties are evaluated against all models using nuXmv [7]. The outcome of this evaluation is a table where each cell expresses whether a scenario configuration satisfies a specific property. This allows for a classification of the different possible configurations, and distinguish between the different main aspects that characterize them.

An interesting result is obtained when considering different amount of information that are exchanged between agents. In particular, taking into account the scenario M2, and comparing the configurations E/E and D/D for GSEP-far/SSEP-far implementation choices. In the first case the verification of the requirement VE-2 is fully satisfied, while the latter does not satisfies the sub-requirement over the far window. The motivation of this fact is that each SSEP has the responsibility for the strategic separation, and it requests an ATC support only if it is not able to resolve the conflict. In addition to that, each SSEP computes its intent according to the information provided by other aircraft far intents, but in this case the GSEPs are not providing this information. The result is that each SSEP has not enough information to resolve the conflicts (with GSEPs) in the far window, and the ATC will not provide a backup support because the SSEPs are not requesting the ground support.

TABLE II: Fault descriptions

Comp.	Fault	Description
GSEP/ SSEP	fault_apply_near	Impossibility to apply the suggested trajectory
	fault_apply_mid	
	fault_apply_far	
	fault_comm_atc_par	Communication failure with ATC (partial or total)
	fault_comm_atc_tot	
	fault_comm_adsb	ADS-B In and Out not functional
ATC	fault_near_res	Failure on providing a correct resolution
	fault_mid_res	
	fault_far_res	
SSEP. CD&R	fault_resolve	Failure on generating the resolution
	fault_resolve_detection	Failure on detecting a resolution problem

VI. Safety Analysis

Performing safety analysis of a formal model requires extending the nominal behavior case, i.e., when everything goes as expected, by allowing undesirable behaviors, i.e., failures. The formal model is a representation of a set of requirements, so the occurrence of a fault describes a violation of a system requirement. For instance the constraint describing that each SSEP shall send its trajectory intentions to the ATC holds under nominal conditions, but not in case of a failure (triggered by a specific fault). The safety analysis will then evaluate which faults combinations can lead to an unwanted condition, represented by the negation of a system property, such as Loss of Separation between two aircraft. In safety analysis, such undesired condition is called Top Level Event (TLE). The set of all faults combinations, namely the Cutsets, are usually represented with a tree where leaf nodes are failures, intermediate nodes are AND/OR boolean operators, and the root is the TLE. This artifact is called Fault Tree [4]. A general application of the Fault Tree Analysis considers only the Minimal Cutsets (MCS), and more specifically, a cutset is called minimal if every additional failure will not prevent such undesired behavior.

A. Faults Definition

In this case study, we added several faults for each subcomponent (Table II), in order to check the robustness of each system. For example, we consider different types of communication failure. Aircraft equipped with ADS-B can permanently lose the ability of sending (ADS-B Out) and receiving (ADS-B In) messages (fault_comm_adsb). Similarly, aircraft might lose the ability of communicating with the ATC. In this case, however, we study two different ways in which we can lose communication: permanently (fault_comm_atc_tot) or temporarily (fault_comm_atc_par).

As defined in the requirements documentation, we assume that the components may have the ability to detect the occurrence of some specific faults. For example, a communication link might provide some sort of heartbeat. In these cases, it makes sense to consider some built-in resilience capabilities for the system. For example, if an SSEP realizes that it cannot communicate with the other SSEPs, it will request support from the ATC. The ATC also assumes that if an aircraft does not provide any new intentions due to a failure, then it will follow the most recent ones.

B. Formal Fault Tree Analysis

The analysis of an artifact like a fault tree is important to understand the dependencies between each single component, how they interact, and what is necessary to go wrong in order to not guarantee a necessary behavior of the system.

For instance, we expect that the communication failure of the radio transmission of the ATC can cause a loss of separation between two GSEP aircraft (even if it would be highly improbable). However, we may expect that the loss of communication of a single GSEP cannot cause a LoS, because the ATC would be able to maneuver all the other aircraft in order to avoid him.

For each combination of scenario configuration we computed the associated Fault Tree using xSAP [8], [20]. In the case of property verification, the comparison between two different system configurations is simple, because the property is either satisfied or not. Differently, the fault tree analysis provides very rich artifacts, and there are several techniques that allow us to compare them and define a partial order over the different configurations.

1) Minimal Cutsets Comparison: A common practice in Fault Tree Analysis consists in comparing the size of cutsets of the same cardinality. This approach is based on the intuition that the fewer the single point of failures in the system the higher is the overall reliability. This approach can be extended also to the cutsets of higher cardinality e.g., double failures. This approach provides an intuitive understanding of the relation between different fault trees, however, it is not always precise, since a single failure might be less probable than a double failure.

An example of this analysis is presented in Table III, which compares the results of the FTA on the model instances M1, M2, and M3 (see Table I) when varying the ability to share far intention on the GSEPs (configuration GSEP-far), with the negation of VE-1 as TLE. In this example, the number of single point of failures does not vary for every configurations (i.e., 5), while the number of double failures decreases when the GSEPs share their far intentions with SSEPs aircraft. Important fact, however, is that the number of triple failures increases when GSEP-far is enabled. This behavior in the fault tree analysis results is typical when adding redundant components. In fact the idea behind redundancy is to increase the fault tolerance, and essentially what is a single point of failure becomes a double (or higher) failure.

Further analysis on fault trees can be performed by evaluating the minimal cutsets that are not in common. An example of this analysis can be done by considering the configuration M2 in Table I, and comparing the fault trees obtained with the TLE “there is a LoS between SSEP1 and GSEP1”, when varying GSEP-far.

TABLE III: MCS, ¬VE-1 as TLE, and GSEP-far (E/D)

Card.	**3G-1S (M1)**		**2G-2S (M2)**		**1G-3S (M3)**	
	E	D	E	D	E	D
1	5	5	5	5	5	5
2	12	15	12	16	12	15
3	33	24	35	23	36	27
…	…		…		…	

The results of this evaluation shows that if GSEP-far is disabled then the fault configuration FC $= \{\mathrm{G1}.F\_comm\_\mathrm{ATC}\_tot,\ \mathrm{S1}.F\_comm\_\mathrm{ATC}\_tot\}$ can cause the occurrence of the TLE. Differently, when GSEP-far is enabled, FC is no more a necessary condition to reach the TLE because the CD&R on the SSEP is able to react to that situation. In fact, if GSEP-far is enabled then FC requires to be combined respectively with $\{\mathrm{ATC}.F\_far\_res\}$,$\{\mathrm{ATC}.F\_future\_res\}$, $\{\mathrm{G1}.F\_comm\_adsb\}$, and $\{\mathrm{S1}.cdr.F\_future\_resolve$, $\mathrm{S1}.cdr.F\_resolve\_detection\}$ to cause the occurrence of the TLE. Thus, the enabling of GSEP-far turned a minimal cutset of cardinality 2 into 3 cutsets of cardinality 3 and 1 of cardinality 4.

2) Reliability Function Evaluation: A Fault Tree represents all the faults configurations that are necessary to cause the occurrence of the TLE. Assigning a probability of failure to each fault event, and assuming that they are independent, it is then possible to compute the overall probability to reach the undesirable event. More specifically, this approach is based on the generation of the closed form of the reliability function presented in [21]. Such function $P_{\mathrm{TLE}}(f_1, f_2, \ldots, f_n)$ relates the probability of occurrence of the TLE with the failure probability of each fault f_i.

We formally analyze the set of possible AAC designs early in the system design phase, before specific module implementations or probabilities of failures are fully defined. However, we can evaluate how the reliability functions compare to each other by analyzing different possible probability values. For instance, if we take into account the probability of reaching a LoS between two aircraft of the same type (for instance GSEP1/2 and SSEP1/2 in the scenario M2), then we expect that the failure of the ATC will affect more the GSEPs than the SSEPs. This can be assumed considering that SSEP aircraft rely on ATC for strategic separation only as a backup, while they are self-separating otherwise. However, the CD&R onboard SSEPs highly depends on the ADS-B system and its possible failure. Fig. 4 compares the probability of having a LOS (y-axis) between two GSEP (G-G) and two SSEP (S-S), by varying the probability of failure of the ATC (x-axis). We plot the probability functions for different values of the ADS-B failure probability. For the GSEP case, not influenced by ADS-B failures, we obtain a single line, while for the SSEP case we show three different functions. By looking at the intersection points (vertical dashed lines in Fig. 4) this analysis shows us when one solution dominates the others.

The aim of this evaluation is to provide the functions that relate the probability of the TLE occurrence to the probability of failures of each component, and not the actual values of failure probability. In fact, the outcome of the reliability evaluation is a set of functions in Matlab format that can be analyzed using common numerical analysis tools. The remarkable aspect of such type of artifacts is that they do not need to be recomputed when the real component implementation will be defined.

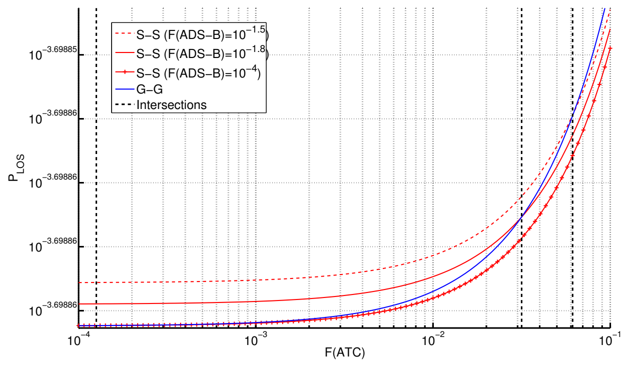


Fig. 4: Reliability comparison between different aircraft types

VII. Lessons Learned

Several subtle technical challenges must be surmounted to complete a realistic, comparative formal analysis on a set of scenarios.

a) Receptiveness of faults: During the fault tree analysis we noticed that some fault configurations were not necessary to cause the reachability of the unwanted condition e.g., LoS. The problem was caused by a chain of relational dependencies through the model, and under some conditions a set of faults $f_1, \ldots, f_n$ imposed f_k to be true. Essentially, we expected both cutsets $cs = \{f_1, \ldots, f_n\}$ and $cs' = \{f_k\}$ to belong to the fault tree FT, thus being minimal cutsets. However, the model implicitly defined the formula $f_1 \wedge \ldots \wedge f_n \rightarrow f_k$, meaning that if cs can cause the TLE then $cs \cup cs' \in FT$. Clearly, cs' is a strict subset of $cs \cup cs'$ which means that even if cs' can cause the occurrence of the TLE then it will not belong to FT, because cs' is not minimal. The solution to this problem was to perform specific receptiveness checks that evaluate if some variables, in this case the fault variables, are always allowed to be assigned to every possible value.

b) Coarse faulty behaviors: Originally, communication faults between ATC and aircraft were not constrained to any specific behavior. This situation caused shadowing in the results of the fault tree analysis. For example, our modeling considers three different time windows: near, mid, and far. For each time window, every aircraft has a trajectory intention and the ATC (or other CD&R components) resolves every conflict in the intentions for every time window. Intuitively, the objective of this design aims to describe a design where a single communication failure in the far window will not cause a Loss of Separation, because it will be possible to resolve the conflict either in the mid or the near window. However, there exists a system execution where the communication failure causes a LoS i.e., when it is total and permanent.

In standard fault tree analysis, each extension of the cutset $\{fault\_communication\}$ will not be considered due to the fact it would be not minimal. The solution to this problem is to refine the model with an additional communication failure, called partial, constrained to a maximum number of occurrences thus providing a more realistic evaluation.

c) Management of multiple communication steps: This work significantly extends the modeling methodology in [13]; for validation, we also modeled the scenario in [13] using the modeling approach described in Sec. III in order to prove that the additional level of detail is able to preserve the previous model's expressiveness. This task was important to discover a weakness in the level of abstraction that defined communication aspects. In fact, in a previous version of the model, the aircraft were allowed to perform a maneuver at every step having a single possible communication step between each maneuver. However, in [13] there is a counterexample where multiple communications directed to an aircraft from different separation assurance agents cause the violation of a property. This system execution, however, is only possible if there are more than one communication steps between each maneuver. Thus, we explicitly allow multiple communication steps.

d) Coarse Top-Level Events: The standard fault tree analysis is strongly characterized by the assumption that each cutset is minimal. This assumption allows us to represent all possible configurations in a compact and intuitive way. However, the choice of a top-level event needs to pay particular attention to this aspect, because the results may not be informative enough. In our analysis we performed the FTA by providing as TLE the negation of a system requirement. For example, our analysis of the requirement that no LoS are allowed between any aircraft provided per se fair results, representing all fault configurations that may cause a LoS. However, the refinement of this top-level event, by expressing each pairwise LoS, provided additional results that were not taken into account previously. More specifically, we expected that the LoS between two aircraft AC_1 and AC_2 can be caused only faults that apply to AC_1, AC_2, and ATC. However, this assumption was not valid in mixed operations when each SSEP aircraft is aware fact that there exists an aircraft that is not able to send the trajectory intentions through ADS-B. In this situation a failure on the aircraft AC_3 can change the behavior of both aircraft AC_1 and AC_2 if they are of SSEP type. As part of this analysis we then decided to enable the possibility to choose if SSEPs aircraft are aware or not of ADS-B failures. The result of the FTA was shadowing important details due to a coarse definition of the TLE, and the solution to this issue was to define more fine-grained TLEs.

VIII. Conclusions and Future Work

This case study provides a first step towards the analysis of the Functional Allocation question. We highlighted a methodology and a series of tools that can be used to analyze and compare different design solutions. In particular, we base the comparison on a set of properties that pass in some configurations and fail in others, on the minimal cut sets obtained with fault-tree analysis, and on the functional dependency of system failure on the failure of single functions. Our approach is expressive, and sufficiently scalable to reason about NASA's full-scale preliminary design space. Important challenges for the future include considering more dimensions of the design space, additional types of faults, and more complex interactions between the agents of the system.

References

[1] MITRE CAASD, "Capacity Needs in the National Airspace System: An Analysis of Airports and Metropolitan Aera Demand and Operational Capacity in the Future," tech. rep., FAA, May 2007.

[2] H. Erzberger, T. A. Lauderdale, and Y.-C. Chu, "Automated conflict resolution, arrival management and weather avoidance for ATM," *Proceedings of the Institution of Mechanical Engineers*, 2011.

[3] T. Lauderdale, T. Lewis, T. Prevot, M. Ballin, A. Aweiss, and N. Guerreiro, "Function allocation for separation assurance: Research plan." NASA HQ Project Overview, Aug. 2014.

[4] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl, "Fault tree handbook," Tech. Rep. NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S., 1981.

[5] "ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, SAE," Dec. 1996.

[6] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback, "Fault Tree Handbook with Aerospace Applications," tech. rep., NASA, 2002.

[7] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *CAV*, pp. 334–342, 2014.

[8] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri, "The xsap safety analysis platform," *CoRR*, vol. abs/1504.07513, 2015.

[9] A. Joshi, M. Whalen, and M. P. Heimdahl, "Modelbased safety analysis: Final report," tech. rep., 2005.

[10] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, *et al.*, "ESACS: an integrated methodology for design and safety analysis of complex systems," *Proc. ESREL 2003*, pp. 237–245, 2003.

[11] O. Åkerlund, P. Bieber, E. Böde, M. Bozzano, M. Bretschneider, *et al.*, "ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects," *Proc. ERTS*, vol. 2006, 2006.

[12] M. Bozzano and A. Villafiorita, *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book, 2010.

[13] Y. Zhao and K. Y. Rozier, "Formal specification and verification of a coordination protocol for an automated air traffic control system," *SCP Journal*, vol. 96, pp. 337–353, December 2014.

[14] H. Erzberger and K. Heere, "Algorithm and operational concept for resolving short-range conflicts," *Proc. IMechE G J. Aerosp. Eng.*, vol. 224, no. 2, pp. 225–243, 2010.

[15] Y. Zhao and K. Y. Rozier, "Probabilistic Model Checking for Comparative Analysis of Automated Air Traffic Control Systems," in *Proc. of 33rd ICCAD conference, San Jose, CA, USA, November 3-6, 2014*.

[16] J. Lygeros and N. Lynch, "On the formal verification of the tcas conflict resolution algorithms," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, vol. 2, pp. 1829–1834, IEEE, 1997.

[17] S. M. Loos, D. W. Renshaw, and A. Platzer, "Formal verification of distributed aircraft controllers," in *Proceedings of HSCC conference 2013, April 8-11, 2013, Philadelphia, PA, USA*, pp. 125–130, 2013.

[18] M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, "Formal design and safety analysis of AIR6110 wheel brake system," in *Proc. of 27th CAV Conference, San Francisco, CA, USA, July 18-24, 2015, Part I*, pp. 518–535.

[19] R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev, "Rat: A tool for the formal analysis of requirements," in *Computer Aided Verification*, pp. 263–267, Springer, 2007.

[20] M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei, "Efficient Anytime Techniques for Model-Based Safety Analysis," in *Proc. of 27th CAV Conference, San Francisco, CA, USA, July 18-24, 2015*, pp. 603–621.

[21] M. Bozzano, A. Cimatti, and C. Mattarei, "Automated Analysis of Reliability Architectures," in *18th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 198–207, IEEE, july 2013.

Pattern-based Synthesis of Synchronization for the C++ Memory Model

Yuri Meshman
Technion

Noam Rinetzky
Tel Aviv University

Eran Yahav
Technion

***Abstract*—We address the problem of synthesizing efficient and correct synchronization for programs running under the C++ relaxed memory model. Given a finite-state program P and a safety property S such that P satisfies S under a sequentially consistent (SC) memory model, our approach automatically eliminates concurrency errors in P due to the relaxed memory model, by creating a new program P with additional synchronization. Our approach works by automatically exploring the space of programs that can be created from P by adding synchronization operations. To explore this (vast) space, our algorithm: (i) explores bounded error traces to detect memory access patterns that can occur under the C++ memory model but not under SC, and (ii) eliminates these error traces by adding appropriate synchronization operations.**

We implemented our approach using CDSChecker as an oracle for detecting error traces and Z3 to symbolically explore the space of possible solutions. Our tool successfully synthesized synchronization operations for several challenging concurrent algorithms, including a state of the art Read-Copy-Update (RCU) algorithm.

I. Introduction

We address the problem of synthesizing efficient and correct synchronization for programs running under the *C++ relaxed memory model* (C++ RMM) [13]. The crucial task of writing correct and efficient low-level concurrent programs in C++ under this model is known to be very challenging: the model's complexity is such that it eludes even veteran systems programmers and requires the attention of formal semantics experts [7], [8], [23], [26].

Under C++ RMM, each operation on an atomic object is annotated with a *memory order*. The memory order ranges from being fully *relaxed* to being fully *sequentially consistent* (for that atomic object), with a few more subtle modes between these two extremes. To maintain efficiency, the programmer wants the most relaxed synchronization required to preserve correctness, and nothing more (even when it simplifies reasoning). Unfortunately, manually finding the right synchronization is extremely difficult, as it requires the programmer to reason about subtle interactions of the memory model. Our goal is to assist the programmer by automatically synthesizing the required synchronization.

A. The Problem

Given a finite-state program P and a safety property S such that $P \models S$ under a sequentially consistent (SC) memory model, we aim to automatically synthesize a program P', whose behaviors are a subset of P's behaviors, s.t. $P' \models S$ under C++ RMM in bounded executions.

B. Our Approach: Pattern Based Synthesis of Synchronization

Our synthesis algorithm automatically explores the (vast) space of programs that can be created from P by modifying memory access synchronization. It does so by: (i) inspecting P's (bounded) error traces to detect memory access patterns that can occur under C++ RMM but not under SC, and (ii) eliminating these error traces by preventing the occurrence of the detected violation patterns using as little synchronization as possible.

More specifically, our algorithm exhaustively explores the traces of P under *C++ RMM*, and looks for *error traces*—traces which do not satisfy the specification S. If it finds an error trace, it searches it for instances of *violation patterns*, behaviors that may occur under C++ RMM but not under SC *and* that we know how to avoid. (Recall that P satisfies S under *SC*. Hence, violations of S must be due to behaviors introduced by the weak memory model.) The algorithm then constructs a constraint which encodes all possible *avoidance templates* that can be used to eliminate that particular error trace. (*Avoidance templates* are strategies to synthesize *memory order annotations* of memory instructions such as `load`, `store`, and `cas`.) The algorithm accumulates the constraints required to eliminate the error traces and passes them to a SAT solver in the form of a CNF formula φ. Every satisfying assignment of φ represents a different way to synthesize the desired *memory order synchronization*.

The algorithm then checks which of the resulting programs satisfies S. The check is required because our set of violation patterns and avoidance templates is not complete. (In fact, we believe that devising a complete set is nontrivial, if at all possible). This means that a program P' with no violation patterns may still violate the original specification S.

C. Main Contributions

The contributions of this paper are as follows:

- A novel approach for detecting missing synchronization using violation patterns, patterns of memory accesses that can occur under C++ RMM but not under SC.
- A technique for synthesizing synchronization by eliminating violation patterns using *avoidance patterns*, a set of predefined synchronization strategies.
- An algorithm which, given a program P and a specification S, synthesizes synchronization to ensure that P satisfies S in bounded executions.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

- An implementation of our approach and an empirical evaluation in which we successfully synthesized synchronizations for several challenging concurrent programs, including a program using a state of the art Read-Copy-Update (RCU) algorithm.

II. Overview

In this section, we provide an informal overview of our approach using our running example, Dekker's mutual exclusion algorithm for two threads [12].

A. Running example

Fig. 1 shows one of the many variants of Dekker's algorithm. The `load` (read) and `store` (write) commands are subscripted with *memory order annotations*. For now, these annotations can be ignored. The algorithm is comprised of: an *entry* section (lines 1–7) and an *exit* section (lines 9–10). The critical section itself (line 8) is irrelevant, and thus elided. The algorithm enforces mutual exclusion using variables $\texttt{flag}[0]$ and $\texttt{flag}[1]$, and ensures deadlock and starvation freedom using variable `turn`.

To enter the critical section, thread i, where i is either 0 or 1, needs to execute its entry section: First, it sets the value of variable $\texttt{flag}[\texttt{i}]$ to 1 (line 1), thus signaling its intentions to the other thread. Then, it inspects the value of $\texttt{flag}[\texttt{1}-\texttt{i}]$ to check whether the other thread is also trying to enter the critical section or is already in it (line 2). If not, it proceeds to the critical section. Otherwise, it sets its own flag to 0 (line 4), thus letting the other thread proceed, and waits for its turn to enter the critical section (line 5). Upon leaving the critical section, thread i executes the exit section, where it gallantly gives precedence to the other thread by setting `turn` to $1-i$ and signals that it left the critical section by setting the value of its flag to 0.

It is important to note that: (i) as long as a thread executes the critical section, its flag is set to 1; and (ii) a thread enters the critical section only after it ensures that the other thread's flag is set to 0 while its own flag is set to 1. The above observation suffices to ensure mutual exclusion under SC, since, in this memory model, there is a total order between all the `load` and `store` commands and reading the value of a variable x returns the last value written to x. Thus, if two threads compete on entering the critical section, at least one must notice in line 2 that the flag of the other is set to 1.

Unfortunately, under C++ RMM this is no longer the case. The reason for this unintuitive behavior can be understood from the following simple program involving only two `store` and two `load` commands.

Example 1: Consider the following program and assume that both $\texttt{flag}[0]$ and $\texttt{flag}[1]$ are initialized to 0, that $\texttt{r}_0$ and $\texttt{r}_1$ are initialized to 2, and that $\texttt{r}_0$ and $\texttt{r}_1$ are each local to the respective thread.

$$\texttt{store}_W(\texttt{flag}[0],1);\ \texttt{r}_0 = \texttt{load}_X(\texttt{flag}[1]) \parallel \\ \texttt{store}_Y(\texttt{flag}[1],1);\ \texttt{r}_1 = \texttt{load}_Z(\texttt{flag}[0])\ .$$

Under *SC*, at the end of the program the only possible values of $\texttt{r}_0$ and $\texttt{r}_1$ are 0 and 1. Furthermore, at most one of them can be 0. Under *C++ RMM*, $\texttt{r}_0$ and $\texttt{r}_1$ can be 0 simultaneously, for certain memory order annotations W, X, Y, and Z. This is because under C++ RMM a `store` operation can behave as if it writes its value to a *thread-local store buffer*, leaving the other threads to read the value stored in the global memory.(C++ RMM exhibits x86-TSO behaviors).

The above example shows that mutual exclusion can only be ensured by adding synchronization to the program. One way to do it in C++ RMM is to explicitly annotate the `load` and `store` operations with the required synchronization type. Using strong synchronization primitives (e.g., requiring all `load` and `store` operations to be sequentially consistent) is expensive. Using synchronization primitives that are too weak, however, leads to unexpected behaviors. Thus, determining correct and efficient annotations is challenging. In contrast, our tool was able to determine that the program shown in Fig. 1 is safe if the memory operations in lines 1, 2, 3, and 9 are sequentially consistent, and the `store` in line 10 is memory order release [1]. (See Section III.)

Note 1: The `load` in line 5 is not synchronized (i.e., it is annotated with RLX). However, as we show in Section V, our result is still verified by our underlying model checker.

B. Synthesizing synchronization

Our approach rests on the insight that we can turn a program that is safe under SC into one that is safe under a weak memory model (C++ RMM in our case) by removing behaviors that cannot occur under SC. We face three main challenges in implementing this approach: (i) detecting such behaviors, (ii) determining a (cheap) way to remove them, and (iii) verifying that the resulting program is safe.

Addressing the first challenge We overcome the first challenge by exhaustively searching the program state space for an error trace, developing all the concrete traces possible under C++ RMM. We allow safety properties to be specified as: (a) assertions on the final state, (b) properties of thread-local variables, and (c) races on non-atomic locations (see Section III). The search is guaranteed to terminate because we only follow bounded traces of finite state programs.

Addressing the second challenge If we find an error trace, we look for instances of violation patterns, memory behaviors involving a small number of `load` and `store` actions possible under C++ RMM but not under SC and which we know how to prevent. Once we discover such an instance, we add synchronization annotations to the relevant memory operations using a predefined avoidance template that blocks the violation pattern, thus eliminating the error trace.

We describe the inferred synchronization annotations using a propositional formula and ask a SAT solver to find the sets of minimal satisfying assignments. (Note that a trace might contain several instances of violation patterns and thus can be eliminated using different avoidance patterns.) From each assignment, we generate a program and repeat the process

[1]To the best of our knowledge, our solution is the only one to use memory order synchronizations and not fences.

```
i.1 store_SC(flag[0], 0);
i.2 store_SC(flag[1], 0);
i.3 store_SC(turn, 0);

     Thread 0:
1    store_SC(flag[0], 1);
2    while( load_SC(flag[1])==1 ){
3     if( load_SC(turn)==1 ){
4      store_RLX(flag[0], 0);
5      while( load_RLX(turn)==1 )yield();
6      store_RLX(flag[0], 1);
7    } }
8        ... // critical section
9    store_SC(turn, 1);
10   store_REL(flag[0], 0);

     Thread 1:
1    store_SC(flag[1], 1);
2    while( load_SC(flag[0])==1 ){
3     if( load_SC(turn)==0 ){
4      store_RLX(flag[1], 0);
5      while( load_RLX(turn)==0 )yield();
6      store_RLX(flag[1], 1);
7    } }
8        ... // critical section
9    store_SC(turn, 0);
10   store_REL(flag[1], 0);
```

Fig. 1. Dekker's mutual exclusion algorithm. Variables `flag[0]`, `flag[1]` and `turn` are declared as atomic locations and initialized to 0. The subscripts indicate the synchronization (consistency) annotations synthesized by our tool.

until no bad trace is found. We use the verified solutions as a starting point in a new round of synthesis in which we raise the bound on the explored traces.

The algorithm is guaranteed to terminate because we consider only finite state programs, the number of memory annotations is finite, and every change only increases the degree of synchronization (see Section III).

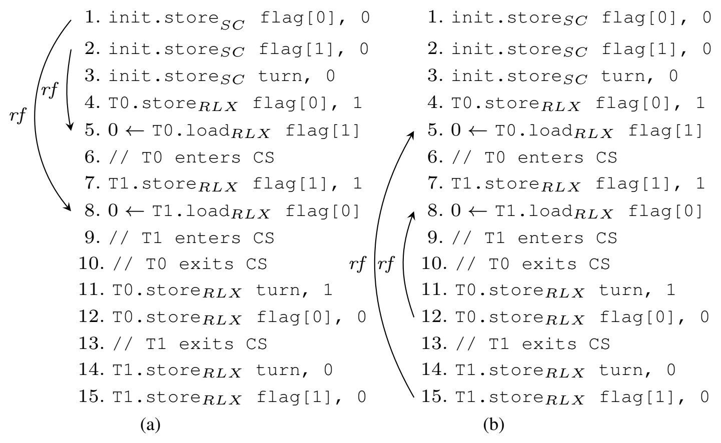
Example 2: Fig. 2 shows a trace of the Dekker algorithm that violates mutual exclusion. The trace contains two violation patterns, store buffering (SB) and load buffering (LB). The former, which we discussed in Example 1, is manifested here by the initialization `store` actions in lines 1 and 2, and the `load` actions in lines 5 and 8. (An rf-annotated arrow from a `store` action to a `load` action indicates that the latter read the value written by the former.) This instance of the SB pattern is blocked by synthesizing a SC annotation to the corresponding memory operations in the algorithms (lines $i.1$, $i.2$, 1, and 2 in Fig. 1.)

Note 2: The list of violation patterns and their corresponding synchronization templates is given as an input to the algorithm. Our algorithm is parametric in that list. The specific patterns and templates that we use in our implementation are given in Section IV-B.

Addressing the third challenge Our set of violation patterns and avoidance templates is not complete. Thus, after synthesizing the programs, we simply explore the state space again. The synthesis procedure terminates if the offered solution contains only sequentially consistent memory accesses and is thus correct by our assumption, or when no error trace is found. This ensures that the program satisfies the desired properties in executions in which every thread performs no more instructions than the explored bound.

III. C++ Relaxed Memory Model in a Nutshell

A memory model defines the possible behaviors of instructions such as `load` and `store` in the program. Arguably, the most intuitive (and restrictive) memory model is *Sequential Consistency* (SC) [19], in which there is a total order on the `load` and `store` instructions, and every `load` from location l reads the last value `stored` in l. (For simplicity, we treat



```
1. init.store_SC flag[0], 0
2. init.store_SC flag[1], 0
3. init.store_SC turn, 0
4. T0.store_RLX flag[0], 1
5. 0 ← T0.load_RLX flag[1]
6. // T0 enters CS
7. T1.store_RLX flag[1], 1
8. 0 ← T1.load_RLX flag[0]
9. // T1 enters CS
10. // T0 exits CS
11. T0.store_RLX turn, 1
12. T0.store_RLX flag[0], 0
13. // T1 exits CS
14. T1.store_RLX turn, 0
15. T1.store_RLX flag[1], 0
```

Fig. 2. An error trace containing two violation patterns: (a) store buffering (SB) and (b) load buffering (LB). These patterns were detected by our tool when analyzing Dekker's algorithm.

the initial state as if it were produced by explicit `store` operations.)

The C++ Relaxed Memory Model is relational: (i) without relations no order of executing instructions is guaranteed; and (ii) a load can read from arbitrary stores. In addition, the model distinguishes between *atomic* locations, where racy accesses are allowed, and *non-atomic* locations, where the behavior of races is undefined. The locations we discuss next will be *atomic*. Below, we provide a (greatly simplified) overview of the part of C++ RMM relevant to our work.

We shall use Fig. 3(SB) as a C++ Relaxed Memory execution trace example, though it was not intended as such and in Section IV-B will be referenced in a different context. Assume a two-threaded program where: variables `x` and `y` are initialized to zero; one thread sets the value of `x` to 1 and another sets the value of `y` to 1; finally, each thread reads the variable set by the other thread.

The first relation we consider is *read from* (rf), denoted by $\rightarrow_{rf}$, which relates `store` instructions to `load` instructions reading from them. The next relation we consider is *happens before* (hb), denoted by $\rightarrow_{hb}$. For our purpose it is a transitively-closed union of the following relations (in general, in C++RMM, hb can be non-transitive): (i) *sequence before* (sb), denoted by $\rightarrow_{sb}$, which places an irreflexive total order on the actions executed by the same thread; (ii) *additionally synchronized with* (asw), which relates instructions executed before thread creation to those executed by the thread, denoted by a dotted line separation; (iii) *synchronized with*(sw), which indicates instruction synchronizaion.

The model ensures that the only possible executions are ones in which these relations satisfy certain constraints. First, hb must be acyclic. Second, *rf* and hb should not contradict each other: a `load` *cannot* read from a `store` that (i) depends on it, i.e., follows it in the hb relation, or (ii) is masked by another write, i.e., there exists a $\texttt{store}_2$ operation such that $\texttt{store} \rightarrow_{hb} \texttt{store}_2 \rightarrow_{hb} \texttt{load}$. Third, the hb induced instruction order should not contradict the *modification order*, which defines a total order on all `store` operations to the *same* location.

Note 3: Note that in Fig. 3(SB) the aforementioned restrictions do not prevent reading values from initialization.

In addition, the hb relation should not contradict the *memory order annotation*. Every memory operation is annotated with a *memory order annotation* that specifies its consistency level: the level of synchronization and the ordering it requires. We consider three types of annotations:

(i) SC, whereby memory actions must be totally ordered;
(ii) ACQ/REL whereby a $\texttt{load}_{\text{ACQ}}$ that gets its value from a $\texttt{store}_{\text{REL}}$ imposes additional synchronization, and
(iii) RLX, whereby operations do not place additional restrictions on the hb relation.

Note 4: For item (ii) above, these annotations induce a sw relation, and for item (i) sc (total order) relation is induced.

IV. Synthesis of Synchronization

In this section we describe our synthesis algorithm (Section IV-A) and review the violation patterns and respective avoidance templates (Section IV-B) that we implemented and experimented with. We also present two *abstract violation patterns* that go beyond concrete litmus tests: we identify patterns involving a small number of memory operations on a *single* location, and describe how to block them by placing a *chain* of dependencies going through an unbounded number of accesses to (possibly) different locations (Section IV-C).

A. Atomic memory access synchronization synthesis

Our synthesis procedure is comprised of two nested loops. The inner one synthesizes synchronization for a given program and the outer one keeps refining the set of solutions by gradually increasing the bound on the length of the explored traces.

Algorithm 1 implements the inner loop of the synthesis procedure. It takes as input a program P and a specification S, and produces a set of programs P' which satisfy S under C++ RMM using different forms of synchronization.

The algorithm first checks whether P satisfies S, and if so returns it (line 2). Otherwise, it goes over the set of traces which violate the specification (line 5) and looks for violation

```
1  Procedure SynSync($P$, $S$)
2      if $P \models S$ then return $\{P\}$
3      $\varphi$ = true
4      $\mathcal{P}$ = $\emptyset$
5      foreach $e \in$ errorTraces($P$,$S$) do
6          $\beta$ = blockOccurr($e$, AcqRelFix())
7          if $\beta$ then continue
8          $\beta$ = blockOccurr($e$, SCFix())
9          if $\neg\beta$ then return allSC($P$)
10         $\varphi$ = $\varphi \wedge \beta$
11     $\varphi = \varphi \wedge \bigwedge impliedSync(\varphi)$
12     avoidance = SAT($\varphi$)
13     foreach $annotation \in avoidance$ do
14         $P'$ = addSync($P$,$annotation$)
15         $\mathcal{P}$ = $\mathcal{P} \cup$ SynSync($P'$, $S$)
16     return $\mathcal{P}$
17 blockOccurr($e$,$patterns$)
18     $\beta$ = false
19     foreach $(p,c) \in patterns$ do
20         foreach $i \in$ occurrence($p$, $e$) do
21             $\beta$ = $\beta \vee$ blockPattern($i$,$c$)
22     return $\beta$
23 $impliedSync(\varphi) = \{a \rightarrow b \mid a, b \in$ vars($\varphi$)
24                 $\wedge$ (SC $\in$ annot($a$))
25                 $\wedge$ (REL $\in$ annot($b$) $\vee$ ACQ $\in$ annot($b$))
26                 $\wedge$ (instr($a$) == instr($b$))$\}$
```

Algorithm 1: The inner loop of the synthesis procedure.

```
1  Procedure PSynSync($P$, $S$, $N$)
2      $C_{init}, C_0, \ldots, C_m$ = getCmds($P$)
3      $\mathcal{P}_1$ = SynSync($C_{\text{init}}$ ; ($C_0 \parallel \ldots \parallel C_m$) , $S$)
4      for $n = 2$ to $N$ do
5          $\mathcal{P}_n$ = $\emptyset$
6          foreach $P' \in \mathcal{P}_{n-1}$ do
7              $C_{init}, C_0, \ldots, C_m$ = getCmds($P'$)
8              $Loop_0$ = "for $i_0 = 1 \ldots n$ do $C_0$"
9              ...
10             $Loop_m$ = "for $i_m = 1 \ldots n$ do $C_m$"
11             $P''$ = "$C_{init}$;($Loop_0 \parallel \ldots \parallel Loop_m$)"
12             $\mathcal{P}_n$ = $\mathcal{P}_n \cup$ SynSync($P''$, $S$)
13     return $\mathcal{P}_N$
```

Algorithm 2: The synthesis procedure. Program P is comprised of an initialization command C_{init} followed by a parallel composition of $m+1$ threads, where thread i executes command C_i for N times.

patterns in each trace. First, it searches for patterns which can be prevented using Acquire-Release synchronization (line 6); and only if no such patterns are found in the trace does it search for patterns that can be prevented using the more expensive Sequential Consistency synchronization (line 8).

The search for instances of violation patterns and the corresponding avoidance template is done by the auxiliary procedure `blockOccur(·)` (Lines 19, 20). If there is an instance i of a pattern p in trace e, then the avoidance template is instantiated according to the instance i and recorded in β as one way to eliminate trace e (line 21). Technically, an instance of an avoidance-template is a conjunction of pairs (`instr`, `annot`), where `instr` is a `load` or a `store` in P and `annot` is the suggested synchronization for that instruction: either SC, REL, or ACQ. The conjunction records the memory order annotations pertaining to the actions forming the detected instance i, which suffice to prevent it. Formula β is constructed as a disjunction of ways to eliminate the trace e. The blocking formulae pertaining to all the error traces are accumulated as a conjunctive formula φ (line 10.)

Finally, we record in φ that every constraint enforced by a REL or ACQ synchronization is also enforced by an SC synchronization by adding the corresponding implications (line 11), thus increasing the set of possible solutions.

Every satisfying assignment to the program correction formula generates a different program, P', which has more restrictive synchronization than P (line 13). We determine whether P' complies with the specification S, or requires further synchronization, by calling `SynSync` recursively.

If `blockOccur(·)` does not find a way to eliminate an error trace, we annotate all memory operations as SC (line 9).

Algorithm 2 implements the outer loop of the synthesis procedure. For simplicity, we assume that the input program is comprised of an initialization command C_0 followed by a parallel composition of $m+1$ loops, where loop i repeats executing a sequential command C_i N times.

The algorithm takes the original program P, a specification S, and the loop bound N, and generates a set of programs $\mathcal{P}_N$ that restrict the synchronization in P so that it satisfies S. Because the number of behaviors rapidly grows as loop iteration is increased, we take an incremental approach: we iteratively construct a sequence of sets of programs $\mathcal{P}_n$, which satisfy the specification S when each loop performs only n iterations (lines 3 and 12). The programs in $\mathcal{P}_n$ are used as a starting point in synthesizing programs with $n+1$ iterations (line 6). Upon termination we return a set of different programs that refine P using different memory order synchronization such that P is compliant with S.

B. Patterns of weak memory behavior.

As mentioned previously, C++ RMM allows certain behaviors for a `load` that are not possible under SC. Below, we list some patterns of such behaviors and explain how they can be prevented using appropriate memory order annotations [7], [8]. The patterns can be seen in Fig. 3. We intuited the patterns from what are often referred to as *litmus tests* [8].

Store Buffering (SB): This is the pattern from Fig. 2(a). In this pattern, two threads first write to two different locations and then try to determine the value of the location written by the other one. It is possible that each thread will not observe the `store` executed by the other. This behavior can occur when the stores of one thread are not immediately visible to the other.
Pattern prevention. This pattern can be prevented only by making all the `load` and `store` instructions SC.

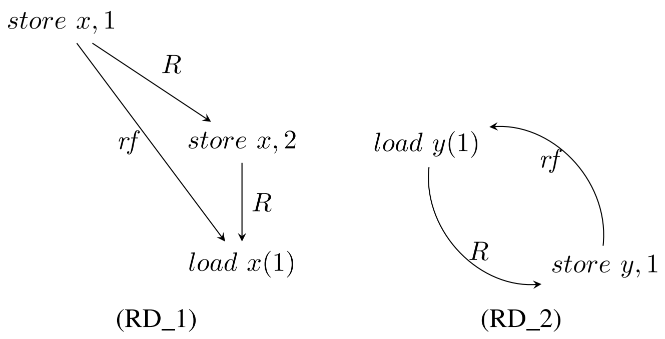


Fig. 4. Abstract patterns of behaviors possible under C++ RMM but not under SC.

Independent Reads of Independent Writes (IRIW): Here two threads write to two different locations and the other two threads see those writes in different orders.
Pattern prevention. The above pattern can be prevented only by making all the `load` and `store` instructions SC.

Load Buffering (LB): This is the pattern from Fig. 2(b). This pattern indicates that every thread can see later (according to the *sb* relation) writes of the other threads. Note that as the `store` might actually be dependent on the `load`, this pattern indicates that each thread can "magically" satisfy the needs of the other. Hence, this pattern is also called *satisfaction cycles* or reading values *out-of-thin-air*.
Pattern prevention. Adding one of the *rf* edges to *hb* would prevent this pattern. This can be done by annotating the `store` and `load` instructions of that edge with REL and ACQ, respectively.

Message Passing (MP): Here, one thread writes to two different locations, and the other thread sees the value written by the second `store` (to y), but misses the first `store` (to x).
Pattern prevention. Annotating the `store` to y with REL and the `load` from y with ACQ would add the *rf* edge to the *hb* relation and prevent the pattern.

Write-to-Read Causality (WRC): This pattern is similar to the message passing pattern, but involves three threads. Here, the value written to x by the first thread is read by the second thread, which then, according to the *sb* order, writes a value to y. The third thread sees the value written by the second thread but not by the first.
Pattern prevention. Annotating the `load` and `store` with REL and ACQ respectively would prevent this pattern.

C. Abstracting the patterns

The presented pattern list captures several behaviors of C++RMM. Instances of those patterns were observed in almost all of our benchmarks but there are still C++RMM behaviors not captured by the previous list. What's more, the patterns share some similarities. In an attempt to bring us closer to completeness, we drew on that resemblance and extracted the commonalities into abstract patterns.

Using the RD property in (RD_1, RD_2): The following patterns are motivated by the RD property defined in [7]. The relation R can be instantiated in two different ways: first as a transitive closure of *rf* and *hb* relations, and second

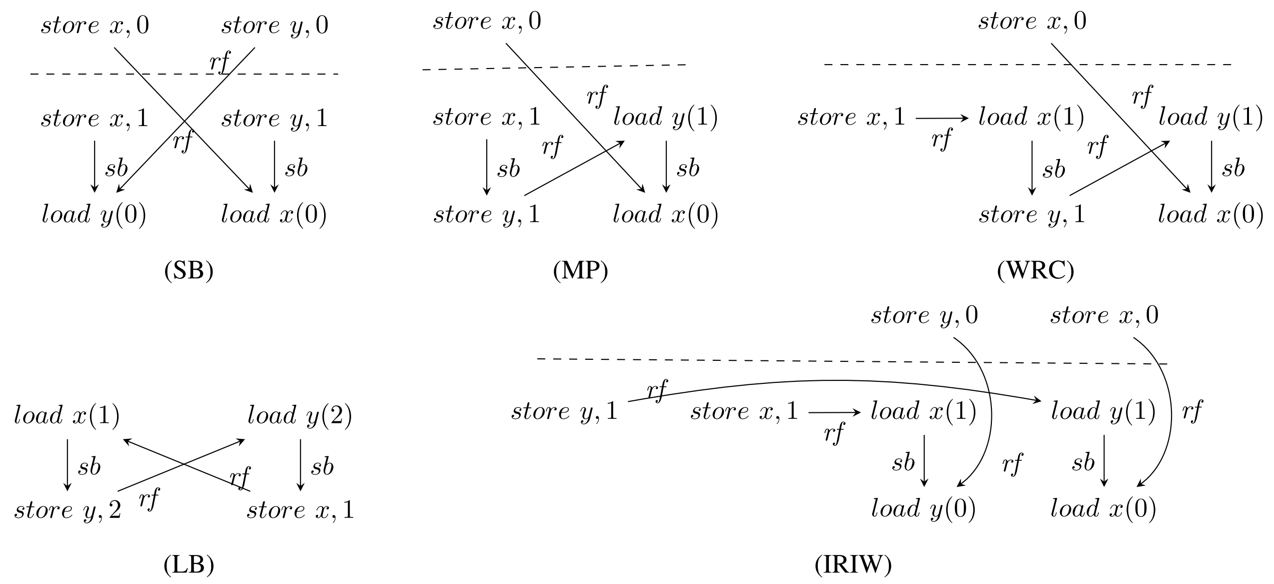


Fig. 3. Patterns of behaviors possible under C++ RMM but not under SC. Every column depicts the actions of one thread. We denote by *store x, 1* a write of value 1 to location x and by *load x(1)* a read of value 1 from x. We assume that the initial value of x and y is 0.

as a possible total order on the involved instructions.

For the first instantiation, the relation R is the transitive closure of $rf \cup hb$. Making all `load` instructions ACQ and all `store` instructions REL across the path will add all the rf edges along the path in R to hb, forming a sequence violating the RD property in [7] and preventing that behavior.

When we cannot find such instantiation of the relation R in the error trace, we try to instantiate it as a possible total order of instructions, and prevent the error trace using SC. In our implementation we chose to attempt instantiation of R as the scheduler choice made by CDSCHECKER. One such scheduling choice, exemplified in Fig. 2(a) as the index of instructions 1-15, is a possible total order which the SB pattern violates. Forcing total order of instructions involved in the pattern (making the memory order access SC) will cause the load to violate the RD property.

The following points should also be noted: 1) RD_1 with R as $rf \cup hb$ transitive closure is an abstraction of the message passing(MP) pattern. 2) RD_2 with R as $rf \cup hb$ transitive closure is an abstraction of the load buffering (LB) pattern. 3) RD_1 with R as a possible instruction total order is an abstraction of the store buffering (SB) pattern. 4) RD_2 with R as a possible instruction total order is a read from future C++ relaxed behavior.

V. Implementation and Experimental Evaluation

We implemented our approach in a tool called PSYNSYN, which is based on CDSCHECKER [20]. Our tool computes a symbolic formula that captures possible fixes, and uses Z3 to find minimal satisfying assignments. Then, we thoroughly evaluated the tool on a number of challenging concurrent algorithms. For all benchmarks our tool found a nontrivial solution with non-SC memory accesses. All experiments were conducted on an AMD Opteron Processor 6376 with 128GB RAM and 64 cores, but using only a single thread per benchmark execution. The synthesized solutions and visual tools that explain our work are available at [1].

The results of the experimental evaluation are summarized in Table V. Most table columns are self-explanatory, but we elaborate on the following:

- The *N* column shows the maximal number of iterations we attempted for each thread.
- The *patterns observed* column shows for each algorithm the instances of patterns described in Section IV-B and C.
- The *# solutions* column shows the number of solutions we found for the benchmark. Unless otherwise specified, the solutions are for maximal N attempted.
- The *# bad traces for N=1* column shows the number of bad traces CDSCHECKER found in the original benchmark with each process doing 1 iteration.
- The *inferred synch* column shows the number of memory access synchronizations of every type suggested by our tool in every solution. Due to space restrictions, we present synchronization of up to 3 solutions per benchmark and use "..." if more solutions exist.

All our benchmarks, when having an error trace, exhibited one of the patterns. For the RD_1 and RD_2 patterns, SC notation is used when the relation R was instantiated by a possible instruction scheduling and SC synchronization was required to prevent the error trace. In addition, the RD pattern occurrences are reported only if they could not be captured by patterns from Fig. 3.This is the case, for example, for pattern instances that are similar to MP but whose path from *store x,1* to *load x(0)* involved more than three $sb \cup rf$ edges and so can only be classified as RD_1 and cannot be classified as MP.

For *abp* we can see that the original algorithm was verified

Algorithm	N	time (s)	space (Mb)	# calls ToZ3	patterns observed	# solutions	# bad traces for N=1	inferred synch (SC, REL, ACQ, RLX)
Alternating Bit Protocol (abp)	5	20s.89	22	1	MP(SC), RD_1 RD_4	5	(N=1,2) 0, (N=3) 1	(5, 0, 0, 1) (4, 0, 0, 2) ...
dekker [12]	1	3m:22	22	3	MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_4	13	631	(10, 1, 0, 8) (13, 0, 1, 5) ...
d-prcu-v1 [6]	3	3m:14	19	20	LB, SB, RD_2, RD_1(SC), RD_2(SC),	7	5	(7, 2, 1, 0)10 (7, 1, 0, 2) ...
d-prcu-v2 [6]	3	3h:53m	22	88	MP, LB, RD_2, RD_1(SC), RD_2(SC),	17	8	(9, 2, 1, 4) (12, 1, 1, 2) ...
kessel [15]	3	57m:16	22	5	MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_2(SC)	2	85	(13, 1, 0, 0) (14, 0, 0, 0)
peterson [22]	3	26m:41	22	3	MP, LB,RD_2, RD_1(SC) , RD_2(SC)	(N=1) 2* (N=2,3) 2	37	(11, 1, 0, 1) *(12, 1, 0, 0) (13, 0, 0, 0)
bakery [18]	2	10m:21	33	3	MP, LB, RD_1,RD_2, RD_1(SC), RD_2(SC)	(N=1) 6 (N=2)4	974	(16, 1, 1, 0) (17, 0, 1, 0) ...
ticket [5]	4	1m:08	19	7	RD_1(SC), RD_2(SC)	4	8	(9, 0, 0, 1) (8, 0, 0, 2) ...
treiber stack [24]	1	1h:05	23	1	MP	1	160	(0, 5, 3, 4)

TABLE I
RESULTS OF SYNCHRONIZATION SYNTHESIS

when each process performed 1 iteration. It was not until each process performed 3 iterations that a violation of the checked property was encountered. At that point 1 error trace was found but it exhibited several patterns; therefore several ways of preventing it were found. We found 5 solutions that verified for 3 iterations of *abp*, and those solutions verified for 4 and 5 iterations as well.

In fact, for almost all our benchmarks, solutions once found, remained verified solutions when more iterations per thread were attempted. This was not the case for *peterson*, as indicated by the "*" in the last column. Here the solution (11, 1, 0, 1) (which are (SC, REL, ACQ, RLX) respectively) found in 1 iteration had a mutual exclusion breach when attempted with 2 iterations, and the solution was further restricted by making the one relaxed memory access SC, thus turning it into the solution marked with "*". That solution later verified for 3 iterations.

For *bakery*, the 6 solutions in iteration 1 reduced to a subset of 4. For all other benchmarks the solutions in the last column were found with 1 iteration per process and remained verified for the maximal number of iterations attempted.

Previous attempts (e.g. [26]) were made to verify RCU under C++RMM, but the version we verified was the first one where an update waits only for the reads whose consistency it affects, and does not wait for the completion of all existing reads.

For *Dekker*'s algorithm, we are not aware of any previous attempt to synthesize the correct version of it using memory accesses instead of fences (one such fence solution is a benchmark of CDSCHECKER). The solution found by our tool seems more restrictive than the fence based solution: for example, in our solution, the load of flags at the while condition creates fences (when translated to intermediate code) at the exit and at the entry of the loop; in the fenced version, however, a fence appears only after the loop exit and not at the entrance. What's more, where the fence placements do correlate, ours are still more restrictive, perhaps due to the incompleteness of our set of patterns and corrections.

For *Treiber's stack* algorithm, CDSCHECKER had a synchronized verified version. For it, our proposed solution was more restrictive than the manual one provided by CDSCHECKER.

VI. RELATED WORK

In this section, we review some closely related work, including synthesis of synchronization, automatic verification, bounded model checking, and dynamic analysis.

Fence Synthesis for x86-TSO and PSO Existing techniques for synthesizing synchronization for relaxed memory models have focused on hardware memory models. Kuperstein et al. [16] presented a framework for fence inference in hardware memory models such as PSO and TSO. Their framework is based on a simple operational semantics that explicitly tracks store buffers to capture effects of the relaxed memory model. They later [17] extended their technique using abstractions of unbounded store buffers. This allowed them to scale their technique and handle a larger set of algorithms. Abdulla et al. [3] infer memory fences for infinite-state programs under x86-TSO by combining predicate abstraction with abstractions of store buffers. Dan et al. [11] used an analysis based on numerical domains to synthesize minimal fence placements under PSO and TSO, utilizing various heuristic search optimizations to minimize the solution space. Our technique synthesizes synchronization for the C++ relaxed memory model. We note that the memory behavior under TSO and

PSO is captured by the SB violation pattern.

Formalizing C++ RMM Batty et al. [8], [9] formalized the C++ RMM and proved correctness of compilation onto TSO and Power [2]. These works inspired our definition of violation patterns and avoidance templates. We also intuited from the formal model when generalizing the concrete violation patterns into abstract ones. Their tool, *CPPMEM*, bears some similarity to CDSCHECKER, which we use in our implementation. Thus, we believe that it would be possible to incorporate *CPPMEM* in our synthesis procedure.

Program Logics for C++RMM Vafeiadis et al. [25], [27] developed a Hoare-style program logic verification technique that extends separation logic [21], [28] to C++ RMM. Batty et al. [7] provided an extension of linearizability and verified that an implementation of Treiber's stack [24] corresponds to an abstract stack under C++ RMM. These works allow for *manual* verification. Our synthesis procedure is based on Bounded Model Checking. However, if these works pan out to automatic verification techniques, it should be fairly straightforward to combine them with our technique as a final stage in which we verify the synthesized solutions.

Fence Synthesis for x86-TSO, PSO and IBM Power C++ RMM was developed with underlying hardware in mind. The following works should therefore shed some light on the behaviors it allows. Joshi et al. [14] introduced Reorder Bounded Model Checking. Their approach is based on instruction reordering, and their tool synthesizes minimal fence placement. We, on the other hand, synthesize memory order synchronization. It would be interesting to see whether our technique can be combined with theirs. *Musketeer*, developed by Algave et al. [4], provides a flexible scheme for fence synthesis to ensure robustness, i.e., that every concurrent execution be observationally equivalent to a serial execution. CheckFence of Burckhardt et al. [10] also ensures robustness by converting a program into a form that can be checked against an axiomatic model specification. Our technique makes it possible to verify user-provided safety properties.

CONCLUSION

We present the first synthesis procedure for inferring efficient memory order synchronizations for C++ RMM. Our procedure ensures that a program complies with a user-provided safety property in bounded executions. We introduce a novel approach for detecting missing synchronization by searching for violation patterns, behaviors possible under C++ RMM but not under SC. We generalize concrete patterns to abstract ones, thus significantly improving the applicability of our approach because the abstract patterns allow us to detect an infinite number of concrete patterns. We provide a technique to eliminate program executions that do not comply with the given safety property by blocking the violation patterns they contain using generic avoidance patterns. We successfully synthesized nontrivial memory order synchronization for several challenging concurrent algorithms, including a state of the art Read-Copy-Update (RCU) algorithm.

Our set of violation patterns and avoidance templates is not complete, and thus our algorithm might fail to find any solution except the trivial one, where all memory operations are sequentially consistent. In fact, we believe that coming up with a complete set is nontrivial, if at all possible. We plan to address this challenge in future work.

REFERENCES

[1] http://www.practicalsynthesis.org/PSynSyn.html.

[2] IBM Power ISA v.2.05. 2007.

[3] ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., LEONARDSSON, C., AND REZINE, A. Automatic fence insertion in integer programs via predicate abstraction. SAS'12.

[4] ALGLAVE, J., KROENING, D., NIMAL, V., AND POETZL, D. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV* (2014), pp. 508–524.

[5] ANDREWS, G. R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[6] ARBEL, M., AND MORRISON, A. Predicate RCU: an RCU for scalable concurrent updates. In *PPoPP* (2015), pp. 21–30.

[7] BATTY, M., DODDS, M., AND GOTSMAN, A. Library abstraction for C/C++ concurrency. In *POPL* (2013), pp. 235–248.

[8] BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. Mathematizing C++ concurrency. In *POPL* (2011), pp. 55–66.

[9] BATTY, M. J. *The C11 and C++11 Concurrency Model*. PhD thesis, Wolfson College University of Cambridge, November 2014.

[10] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007).

[11] DAN, A. M., MESHMAN, Y., VECHEV, M. T., AND YAHAV, E. Effective abstractions for verification under relaxed memory models. In *VMCAI* (2015), pp. 449–466.

[12] DIJKSTRA, E. Cooperating sequential processes, TR EWD-123. Tech. rep., 1965.

[13] ISO/IEC. Programming Languages – C, 9899:2011.

[14] JOSHI, S., AND KROENING., D. Property-driven fence insertion using reorder bounded model checking. In *FM* (2015).

[15] KESSELS, J. L. W. Arbitration without common modifiable variables. *Acta informatica 17*, 2 (1982), 135–141.

[16] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD* (2010).

[17] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Partial-coherence abstractions for relaxed memory models. PLDI '11.

[18] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* (1974).

[19] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.

[20] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with c/c++ atomics. In *OOPSLA* (2013).

[21] O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. Local reasoning about programs that alter data structures. In *CSL* (2001), pp. 1–19.

[22] PETERSON, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981).

[23] TASSAROTTI, J., DREYER, D., AND VAFEIADIS, V. Verifying read-copy-update in a logic for weak memory. In *PLDI* (2015).

[24] TREIBER, R. K. *Systems Programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[25] TURON, A., VAFEIADIS, V., AND DREYER, D. Gps: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA* (2014), pp. 691–707.

[26] VAFEIADIS, V., BALABONSKI, T., CHAKRABORTY, S., MORISSET, R., AND ZAPPA NARDELLI, F. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *POPL* (2015), pp. 209–220.

[27] VAFEIADIS, V., AND NARAYAN, C. Relaxed separation logic: a program logic for c11 concurrency. In *OOPSLA* (2013).

[28] VAFEIADIS, V., AND PARKINSON, M. J. A marriage of rely/guarantee and separation logic. In *CONCUR* (2007).

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Better Lemmas with Lambda Extraction

Mathias Preiner, Aina Niemetz, and Armin Biere

Johannes Kepler University, Linz, Austria

***Abstract*—In Satisfiability Modulo Theories (SMT), the theory of arrays provides operations to access and modify an array at a given index, e.g., *read* and *write*. However, common operations to modify multiple indices at once, e.g., *memset* or *memcpy* of the standard C library, are not supported. We describe algorithms to identify and extract array patterns representing such operations, including *memset* and *memcpy*. We represent these patterns in our SMT solver Boolector by means of compact and succinct lambda terms, which yields better lemmas and increases overall performance. We describe how extraction and merging of lambda terms affects lemma generation, and provide an extensive experimental evaluation of the presented techniques. It shows a considerable improvement in terms of solver performance, particularly on instances from symbolic execution.**

I. Introduction

The theory of arrays, which for instance has been axiomatized by McCarthy [7], enables reasoning about “memory” in both software and hardware verification. It provides two operations *read* and *write* for accessing and modifying arrays on *single* array indices. While these two operations can be used to capture many aspects of modeling memory, they are not sufficient to succinctly encode array operations over *multiple* indices or a range of indices, e.g., *memset* or *memcpy* from the standard C library. Such array operations can therefore only be represented verbosely by means of a constant number of read and write operations. It is further impossible to reason about a variable number of indices e.g., a memset operation of variable size (without introducing quantifiers).

To overcome these limitations, Seshia et. al. [1] introduced an approach to model arrays by means of restricted lambda terms. This also enabled their SMT solver UCLID [10] to reason about ordered data structures and partially interpreted functions. However, UCLID employs the eager SMT approach and thus eliminates all lambda terms as a rewriting step prior to bit-blasting the formula to SAT, which might result in an exponential blow-up in the size of the formula [10].

An extension to the theory of arrays by Sinz et.al. [5] uses lambda terms similarly to UCLID in order to model memset and memcpy operations as well as loop summarizations, which in essence are initialization loops for arrays. As UCLID, this approach suffers from the problem of exponential explosion through eager lambda elimination.

To avoid exponential lambda elimination, in [9] we introduced a new decision procedure, which lazily handles non-recursive and non-extensional lambda terms. That decision procedure enabled us to succinctly represent array operations such as memset and memcpy as well as other array initialization patterns by means of lambda terms within our SMT solver Boolector. Lambda terms also allow to reason about variable ranges of indices without the need for quantifiers.

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

In this paper, we continue this thread of research and describe various patterns of operations on arrays occurring in benchmarks from SMT-LIB (http://www.smtlib.org). We provide algorithms to identify these patterns, and to extract succinct lambda terms from them. Extraction leads to stronger, as well as fewer lemmas. This improves performance by orders of magnitude on certain benchmarks, particularly on instances from symbolic execution [2]. We further describe a technique called *lambda merging*. Our extensive experimental evaluation shows that both techniques considerably improve the performance of Boolector, the winner of the QF_ABV track of the SMT competition 2014.

II. Preliminaries

We assume the usual notions and terminology of first order logic and are mainly interested in many-sorted languages, where bit vectors of different bit width correspond to different sorts, and array sorts correspond to a mapping ($\tau_i \Rightarrow \tau_e$) from index sort τ_i to element sort τ_e. We primarily focus on the *quantifier-free* theories of *fixed size bit vectors* and *arrays*. However, our approach is not restricted to the above.

In general, we refer to 0-arity function symbols as *constant* symbols. Symbols a, b, i, j, and e denote constants, where a and b are used for array constants, i and j for array indices, and e for an array element. We denote an *if-then-else* over bit vector terms with condition c, then branch t_1, and else branch t_2 as $ite(c, t_1, t_2)$, which is interpreted as $ite(\top, t_1, t_2) = t_1$ and $ite(\bot, t_1, t_2) = t_2$. We identify operations *read* and *write* as basic array operations (cf. *select* and *store* in SMT-LIBv2 notation) for accessing and modifying arrays. A *read* operation $read(a, i)$ denotes the element of array a at index i, whereas a *write* operation $write(a, i, e)$ represents the modified array a with element e written to index i. The non-extensional theory of arrays is axiomatized by the following axioms originally introduced by McCarthy in [7]:

$$i = j \rightarrow read(a, i) = read(a, j) \tag{A1}$$
$$i = j \rightarrow read(write(a, i, e), j) = e \tag{A2}$$
$$i \neq j \rightarrow read(write(a, i, e), j) = read(a, j) \tag{A3}$$

Axiom (A1) asserts that accessing array a at two indices that are equal always yields the same element. Axiom (A2) asserts that accessing a modified array on the updated index i yields the written element e, whereas axiom (A3) ensures that the unmodified element of the original array a at index j is returned if the modified index i is not accessed.

A *write sequence* of n (consecutive) write operations of the form $a_1 = write(a_0, i_1, e_1), \ldots, a_n = write(a_{n-1}, i_n, e_n)$ is denoted as $(a_k := write(a_{k-1}, i_k, e_k))_{k=1}^{n}$ with array a_0 as the *base array* of the write sequence. In the following we use $a_n = write(a, \bar{i}, \bar{e})$ as shorthand for write sequences.

In [9] we use *uninterpreted functions* (UF) and *lambda* terms to represent array variables and array operations, respectively. Consequently, a read on an array of sort $\tau_i \Rightarrow \tau_e$ is represented as a function application $f(i)$ on either an UF f or a lambda term $f := \lambda j \,.\, t$, where function f maps terms of sort τ_i to terms of sort τ_e. Furthermore, write operations $write(a, i, e)$ are represented as lambda terms $\lambda j \,.\, ite(i = j, e, a(j))$, where given an array a, a function application yields element e if j is equal to the modified index i and the unchanged element $a(j)$, otherwise. Lambda terms allow us to succinctly model array operations such as *memset* and *memcpy* from the standard C library, or arrays initialized with a constant value. For example, memset with signature $memset\ (a, i, n, e)$, which sets each element of array a to e within the range $[i, i + n[$, can be represented as $\lambda j \,.\, ite(i \leq j < i + n, e, a(j))$. In this paper, we use read operations and function applications interchangeably.

III. Extracting Lambdas

Currently, the SMT-LIBv2 standard only supports write operations for modifying the contents of an array at one index at a time. Hence, quasi-parallel array operations like memset or memcpy usually have to be represented as a *fixed* sequence of consecutive write operations, where copying or setting n indices always requires n write operations. Further, modeling such array operations with a *variable* range is not possible (without quantifiers), since it would require a variable number of write operations. Lambda terms, however, provide means to succinctly represent parallel array operations, and further allow to model these operations with *variable* ranges. For example, modeling $memset(a, i, n, e)$ with a sequence of writes for some fixed n produces n nested write operations $write(write(\ldots(write(a, i, e), i{+}1, e)\ldots), i{+}n{-}1, e)$ which could be represented in a more compact way by means of a *single* lambda term $\lambda j \,.\, ite(i \leq j < i + n, e, a(j))$.

In the following, we describe several array operation patterns we identified by analyzing QF_ABV benchmarks in the SMT-LIB benchmark library. These patterns can not be captured compactly by means of write and read operations alone, but they can be succinctly represented using lambda terms. For each pattern identified in a formula, lambda terms are extracted and used instead of the original array operations, which are defined as follows.

A. Memset Pattern

The probably most common pattern is the *memset pattern* modeling the $memset\ (a, i, n, e)$ operation, which updates n elements of array a within range $[i, i{+}n[$ to a value e starting from address i. This is the pattern already described above, and it is represented by the lambda term

$$\lambda_{mset} := \lambda j \,.\, ite(i \leq j < i + n, e, a(j)).$$

Lambda term λ_{mset} yields value e if index j is within the range $[i, i + n[$, and the unmodified value from array a at position j otherwise. Note that in actual benchmarks, e.g., those from SMT-LIB, the upper bound n is constant, while indices, as well as values are usually symbolic.

B. Memcpy Pattern

The *memcpy pattern* models the $memcpy(a, b, i, k, n)$ operation, which copies n elements from source array a starting at address i to destination array b at address k. If arrays a and b are syntactically distinct, or if the source and destination addresses do not overlap, i.e., $(i + n < k)$ or $(k + n < i)$, *memcpy* can be represented as

$$\lambda_{mcpy} := \lambda j \,.\, ite(k \leq j < k + n, a(i + j - k), b(j)).$$

Lambda term λ_{mcpy} returns the value copied from source array a if it is accessed within the copied range $[k, k + n[$, and the value from destination array b at position j otherwise.

Assume arrays a and b are syntactically equal, then aliasing occurs. Writing to array b at overlapping memory regions modifies elements in a to be copied to the destination address. This is not captured by lambda term λ_{mcpy}, since λ_{mcpy} behaves like a *memmove* operation. It ensures that elements of a at the overlapping memory region are copied before being overwritten. The following lambda term λ_{mcpyo} can be used to model *memcpy* applied to potentially overlapping memory regions.

$$\begin{aligned}\lambda_{mcpyo} := \lambda j \,.\, ite(&k \leq j < k + n,\\ &ite(i \leq k < i + n,\\ &\quad a(i + ((j - k) \bmod (k - i))),\\ &\quad a(i + j - k)),\\ &b(j)).\end{aligned}$$

If condition $i \leq k < i + n$ holds, source and destination memory regions overlap and consequently, the elements of the overlapping memory region always contain the repeated sequence of the elements of array a in range $[i, k[$. This corresponds to the value $a(i + (j - k) \bmod (k - i))$, where $k - i$ represents the size of the non-overlapping memory region and thus, the number of elements that occur repeatedly. If the memory regions do not overlap, the behavior of lambda terms λ_{mcpyo} and λ_{mcpy} is equivalent. For the rest of this paper, we focus on *memcpy* with non-overlapping memory regions.

C. Loop Initialization Pattern

The *loop initialization pattern* models array initialization operations that can be expressed with the following loop

$$\textbf{for}\ (j = i;\ j < i + n;\ j = j + inc)\ \{a[j] = e;\},$$

where, starting from index i, the loop counter is incremented by a constant inc greater than one. Consequently, every inc-th

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

element of an array a is modified within the range $[i, i+n[$. The above loop pattern corresponds to the lambda term

$$\lambda_{i\to e} := \lambda j \,.\, ite(i \leq j \wedge j < i+n \ \wedge (inc \mid (j-i)), e, a(j)).$$

The memset pattern is actually a special case of this pattern with $inc = 1$. Further, the divisibility condition $inc \mid (j-i)$ makes sure that there exists a c such that index $j = i + c \cdot inc$ or equivalently $((j-i) \bmod inc = 0)$.

It is also possible that the value written on an index i depends on i itself. We found two such patterns in benchmarks. They can be expressed with the following loops

$$\begin{aligned}&\textbf{for } (j = i;\ j < i+n;\ j = j + inc)\ \{a[j] = j\},\\ &\textbf{for } (j = i;\ j < i+n;\ j = j + inc)\ \{a[j] = j+1\}\end{aligned}$$

or equivalently with the following lambda terms

$$\begin{aligned}\lambda_{i\to i} &:= \lambda j \,.\, ite(i \leq j \wedge j < i+n \ \wedge (inc \mid (j-i)),\\ &\qquad j,\ a(j))\\ \lambda_{i\to i+1} &:= \lambda j \,.\, ite(i \leq j \wedge j < i+n \ \wedge (inc \mid (j-i)),\\ &\qquad j+1,\ a(j)).\end{aligned}$$

Note that with $inc = 1$, the condition $inc \mid (j-i)$ is redundant and can be omitted. Further, this set of patterns is of course just a subset of all possible structures in benchmarks for which lambdas can be extracted. The ones discussed in this paper are those that we observed in actual benchmarks, and which turn out to be useful in our experiments.

D. Lemma Generation

Extracting lambda terms from write sequences does not only yield more compact array representations but improves the lemmas generated during search. As an example, consider a memset operation with range $[i, i+n[$ and value e, which is represented as a sequence of write operations

$$b := write(write(\ldots(write(a, i, e), i+1, e)\ldots), i+n-1, e).$$

A read operation on array b at index j may produce a conflict on index i, where $read(b, j) \neq e$. As a consequence, the following lemma is generated.

$$(\bigwedge_{k=1}^{n-1} j \neq i+k) \wedge j = i \rightarrow read(b, j) = e$$

In the worst case, this might be repeated for all the indices $i+k$ with $k \in [1, n[$, which also results in n lemmas of the above form. However, if we use a lambda term to represent memset, then a conflict produces a single lemma of the form

$$i \leq j \wedge j < i+n \rightarrow read(b, j) = e,$$

which is more succinct and stronger as it covers an index range instead of single indices. This effect can be observed in our experiments in Sect. IV-B as well. If applicable, the number of generated lemmas is reduced. This improves runtime and more instances are solved.

E. Algorithms

Figure 1 depicts the main *lambda extraction* algorithm `extract_lambdas`. The purpose of this procedure is to initially identify and extract array patterns from each sequence of write operations in formula ϕ (lines 5-7). The identified patterns are then used to create lambda terms on top of each other resulting in a new lambda term b, which is equisatisfiable to the original write sequence a_n (lines 8-16), and is used to substitute a_n in ϕ. Figures 2, 3, and 4 depict the algorithms for identifying and extracting the actual array patterns. In essence, they all can be split into the following three steps. Given a sequence of write operations,

1) group *write indices* w.r.t. the corresponding pattern,
2) identify *index sequences* in these grouped write indices,
3) and create new pattern for identified *index sequence*.

In the following we describe the algorithms for identifying and extracting array patterns in more detail.

A high level view of the main lambda extraction algorithm `extract_lambdas` is given in Fig. 1. Given a formula ϕ, for any write sequence $a_n = write(a, \bar{i}, \bar{e})$ with distinct indices $i_1, \ldots, i_n$, `extract_lambdas` initially generates a map $\rho_{i\to e}$, which maps indices $i_1, \ldots, i_n$ to values $e_1, \ldots, e_n$ (line 4), and is then used to extract memset (p_{set}), memcpy (p_{cpy}), and loop initialization patterns (p_{loop}) (lines 5-7). Note that procedures `find_mset_patterns`, `find_mcopy_patterns`, and `find_lp_patterns` remove all index/value pairs included in extracted patterns from $\rho_{i\to e}$. As a consequence, at line 8, map $\rho_{i\to e}$ contains all index/value pairs for which no pattern was extracted. The actual memset, memcpy, and loop initialization lambda terms are then created on top of each other with base array a_0 of write sequence a_n as the initial base array (lines 8-14). For the remaining index/value pairs in $\rho_{i\to e}$, lambda terms representing write operations are created on top of the previously generated lambda terms, and the resulting term b is then used to substitute the original write sequence a_n.

Note that indices $i_1, \ldots, i_n$ are required to be distinct constants (line 3) as otherwise, reordering write sequence a_n does not result in an equisatisfiable sequence. As an example, assume indices i and j are equal and values e_i and e_j are distinct. Accessing sequence $a_{ij} := write(write(a, i, e_i), j, e_j)$ at index j yields value e_j. However, accessing sequence $a_{ji} := write(write(a, j, e_j), i, e_i)$ at index j yields e_i since $i = j$. Thus, a_{ji} is not equisatisfiable to a_{ij}.

Figures 2, 3, and 4 illustrate the algorithms for the actual pattern extraction, which we describe in more detail in the following. Procedure `find_mset_patterns` as in Fig. 2 extracts *memset* patterns, i.e., in essence, it identifies index sequences that map to the same value. Given map $\rho_{i\to e}$, the procedure initially generates a reverse map $\rho_{e\to i}$, which maps values to indices and therefore groups indices that map to the same value (lines 3-4). For each index group *indices* in $\rho_{e\to i}$, `find_mset_patterns` sorts *indices* in ascending order (line 6) and identifies index sequences $s := (i_k)_{k=l}^{u}$ with $i_k := i_{k-1} + 1$ within lower bound l ($i_l := indices[l]$) and

```
1  procedure extract_lambdas($\phi$)
2    for write sequence $a_n := write(a, \bar{\imath}, \bar{e})$ in $\phi$ \
3     and $i_1, \ldots, i_n$ are distinct
4      $\rho_{i \to e}$ := index_value_map($a_n$)
5      $p_{set}$ := find_mset_patterns($\rho_{i \to e}$)
6      $p_{cpy}$ := find_mcopy_patterns($\rho_{i \to e}$)
7      $p_{loop}$ := find_lp_patterns($\rho_{i \to e}$)
8      $b$ := $a_0$
9      for $p$ in $p_{set}$
10       $b$ := mk_memset($b$, $p.i$, $p.n$, $p.e$)
11     for $p$ in $p_{cpy}$
12       $b$ := mk_memcopy($p.a$, $b$, $p.i$, $p.k$, $p.n$)
13     for $p$ in $p_{loop}$
14       $b$ := mk_loop_init($b$, $p.i$, $p.n$, $p.inc$)
15     for $i, e$ in $\rho_{i \to e}$
16       $b$ := mk_write($b$, $i$, $e$)
17     $\phi$ := $\phi[a_n/b]$
```

Fig. 1. Main lambda extraction algorithm in pseudo-code.

```
1  procedure find_mset_patterns($\rho_{i \to e}$)
2    $patterns$ := [], $\rho_{e \to i}$ := {}
3    for $index, value$ in $\rho_{i \to e}$
4      $\rho_{e \to i}$[$value$].add($index$)
5    for $value, indices$ in $\rho_{e \to i}$
6      $indices$ := sort($indices$)
7      $l$, $u$ := 0
8      while $u$ < len($indices$)
9        while $u$ + 1 < len($indices$) \
10        and $indices$[$u$ + 1] - $indices$[$u$] = 1
11         $u$ += 1
12       if $l \neq u$
13         Pattern $p$
14         $p.i$ := $indices$[$l$]
15         $p.n$ := $indices$[$u$] - $indices$[$l$] + 1
16         $p.e$ := $value$
17         $patterns$.add($p$)
18         $\rho_{i \to e}$ := $\rho_{i \to e} \setminus \{indices[i] \mid \forall i \in [l, u]\}$
19         $l$ := $u$ + 1  /* next sequence */
20       $u$ += 1
21   return $patterns$
```

Fig. 2. Memset pattern extraction algorithm in pseudo-code.

upper bound u ($i_u := indices[u]$) (lines 7-19). If sequence s includes at least two indices (i.e., $u \neq l$), a new memset pattern p with start address $p.i$, size $p.n$ and value $p.e$ is created and added to list $patterns$ (lines 13-17). All indices included in sequence s are removed from map $\rho_{i \to e}$ (line 18), since these indices are covered by a detected pattern. If all index groups have been processed, procedure `find_mset_patterns` returns the list of detected memset patterns $patterns$.

Figure 3 illustrates procedure `find_mcopy_patterns` for extracting *memcpy* patterns. Assume that write operation $write(b, dst + o, a(src + o))$ represents a single memcpy operation $memcpy(a, b, src, dst, n)$ with offset o and $src \leq o < src + n$, which copies one element from source address $src + o$ of array a to destination address $dst + o$ of array b. Consequently, $\rho_{i \to e}$ maps indices of the form $dst + o$ to values of the form $a(src + o)$. Initially, the procedure

```
1  procedure find_mcopy_patterns($\rho_{i \to e}$)
2    $patterns$ := [], $offset\_groups$ := {}
3    for $index, value$ in $\rho_{i \to e}$ \
4     and $index$ = $dst$ + $o$ \
5     and $value$ = $a(src$ + $o)$
6      $offset\_groups$[$dst, a, src$].add($o$)
7    for $dst, a, src$ in $offset\_groups$
8      $indices$ := sort($offset\_groups$[$dst, a, src$])
9      $u, l$ := 0
10     while $u$ < len($indices$)
11       while $u$ + 1 < len($indices$) \
12        and $indices$[$u$ + 1] - $indices$[$u$] = 1
13         $u$ += 1
14       if $l \neq u$
15         Pattern $p$
16         $p.a$ := $a$
17         $p.i$ := $src$ + $indices$[$l$]
18         $p.k$ := $dst$ + $indices$[$l$]
19         $p.n$ := $indices$[$u$] - $indices$[$l$] + 1
20         $patterns$.add($p$)
21         $\rho_{i \to e}$ := $\rho_{i \to e} \setminus \{indices[i] \mid \forall i \in [l, u]\}$
22         $l$ := $u$ + 1 /* next sequence */
23       $u$ += 1
24   return $patterns$
```

Fig. 3. Memcopy pattern extraction algorithm in pseudo-code.

collects all offsets o from the indices in $\rho_{i \to e}$ and groups them by destination address dst, source array a, and source address src (lines 3-6). Note that a group of offsets corresponds to the memory regions copied from source address of array a to destination address of array b. For each offset group $indices$ in $offset\_groups$, `find_mcopy_patterns` identifies index sequences $s := (i_k)_{k=l}^{u}$ similar to procedure `find_mset_patterns` (lines 11-13). If a sequence with at least two indices is found, a new memcpy pattern with source array $p.a$, source address $p.i$, destination address $p.k$, and size $p.n$ is created and added to the $patterns$ list (lines 15-20). As for `find_mset_patterns`, indices included in a sequence s are removed from $\rho_{i \to e}$ (line 21). If all offset groups have been processed, procedure `find_mcopy_patterns` returns the list of detected memcpy patterns $patterns$.

Figure 4 illustrates procedure `find_lp_patterns` for extracting *loop initialization* patterns. Initially, all indices in map $\rho_{i \to e}$ are categorized w.r.t. the three loop initialization patterns defined above, which correspond to the map $\rho_{e \to i}$, and the lists $\rho_{i \to i}$ and $\rho_{i \to i+1}$. Map $\rho_{e \to i}$ groups indices that map to the same value, list $\rho_{i \to i}$ contains indices that map to themselves, and list $\rho_{i \to i+1}$ contains all indices i that map to $i + 1$ (lines 4-9). For index groups $\rho_{i \to i+1}$ and $\rho_{i \to i+1}$, and for each index group in $\rho_{e \to i}$, procedure `find_lpp_aux` identifies sequences $s := (i_k)_{k=l}^{u}$ with $i_k := i_{k-1} + inc$ and $inc \geq 1$ within lower bound l ($i_l = indices[l]$) and upper bound u ($i_u := indices[u]$) (lines 10-13). Identifying index sequences in `find_lpp_aux` is similar to `find_mset_patterns`, except that increment inc can be greater than one. For each sequence, inc is initially set to $indices[u + 1] - indices[u]$ (lines 21-22), which

```
1  procedure find_lp_patterns($\rho_{i\to e}$)
2    $patterns$ := [], $\rho_{e\to i}$ := {}
3    $\rho_{i\to i}$ := [], $\rho_{i\to i+1}$ := []
4    for $index, value$ in $\rho_{i\to e}$
5      $\rho_{e\to i}$[$value$].add($index$)
6      if $value$ = $index$
7        $\rho_{i\to i}$.add($index$)
8      elif $value$ = $index$ + 1
9        $\rho_{i\to i+1}$.add($index$)
10   for $value, indices$ in $\rho_{e\to i}$
11     $patterns$.add(find_lpp_aux($\rho_{i\to e}$, $\rho_{e\to i}$))
12   $patterns$.add(find_lpp_aux($\rho_{i\to e}$, $\rho_{i\to i}$))
13   $patterns$.add(find_lpp_aux($\rho_{i\to e}$, $\rho_{i\to i+1}$))
14   return $patterns$
15
16 procedure find_lpp_aux($\rho_{i\to e}$, $indices$)
17     $patterns$ := []
18     $indices$ := sort($indices$)
19     $l$, $u$ := 0
20     while $u$ < len($indices$)
21       if $u$ + 1 < len($indices$)
22         $inc$ := $indices$[$u$ + 1] - $indices$[$u$]
23       while $u$ + 1 < len($indices$) \
24        and $indices$[$u$ + 1] - $indices$[$u$] = $inc$
25         $u$ += 1
26       if $l \neq u$
27         Pattern $p$
28         $p.i$ := $indices$[$l$]
29         $p.n$ := $indices$[$u$] - $indices$[$l$] + 1
30         $p.e$ := $value$
31         $p.inc$ := $inc$
32         $patterns$.add($p$)
33         $\rho_{i\to e}$ := $\rho_{i\to e} \setminus \{indices[i] \mid \forall i \in [l,u]\}$
34         $l$ := $u$ + 1  /* next sequence */
35       $u$ += 1
36   return $patterns$
```

Fig. 4. Loop initialization pattern extraction algorithm in pseudo-code.

defines the increment value between neighbouring indices, e.g., $(l, l + inc, l + 2 \cdot inc, l + 3 \cdot inc, \ldots, u)$. If a sequence with at least two indices is found, a new *loop initialization* pattern with lower bound $p.i$, size $p.n$, and increment $p.inc$ is created and added to the $patterns$ list. Index sequences found in $\rho_{e\to i}$ correspond to $\lambda_{i\to e}$ patterns. These require a $p.e$ value, which is saved in addition (but remains unused for sequences $\rho_{i\to i}$ and $\rho_{i\to i+1}$). As before, indices included in a detected sequence s are removed from map $\rho_{i\to e}$ (line 33). If index group $indices$ has been processed, procedure `find_lpp_aux` returns the list of detected loop initialization patterns.

In case that write expressions in a write sequence are *shared*, i.e., they also appear in the formula outside of the sequence, we still extract patterns for the whole sequence. This may duplicate parts, which is not a problem since the extracted lambda terms are succinct and the "duplication" only affects the index range check of a lambda and is therefore negligible.

There are two common approaches for representing the initialization of an array variable a with n concrete values: with (1) *write* sequences of size n with array a as base array, or (2) n *read* operations on array a by asserting for each index $i \in i_1, \ldots, i_n$ that $read(a, i) = e$. In case (1), we are able to directly represent such array initializations by means of lambda terms. However, in case (2), we first have to translate the read operations into sequences of write operations. For example, given an array a that is initialized with some values e on indices $1-4$, we could either represent this as a sequence of write operations with a fresh array variable b as base array

$$a := write(write(write(write(b, 1, e), 2, e), 3, e), 4, e),$$

or with the following four equalities asserted to be true

$$read(a, 1) = e, read(a, 2) = e,$$
$$read(a, 3) = e, read(a, 4) = e.$$

However, the initialization with read/value equalities can also be represented as lambda term

$$a := \lambda j \,.\, ite(1 \leq j \leq 4, e, b(j)),$$

where array b is a fresh array variable. In order to extract lambda terms from these equalities, we translate them into sequences of write operations and apply the lambda extraction algorithms to it. The only requirement is that, for the same reason as for the write sequence case, the read indices have to be distinct.

IV. Merging Lambdas

Lambda terms extracted from a sequence of write operations often do not cover all indices in the sequence. Some might be left over. In order to preserve equisatisfiability, we use the uncovered write operations to create a new write sequence on top of the extracted lambda terms (cf. lines 15-16 in Fig. 2). Note that as we represent write operations as lambda terms, we actually generate a sequence of lambda terms (representing write operations) on top of the extracted terms. Given a sequence of lambda terms of size n, however, we can apply a rewriting technique we refer to as *lambda merging*, which inlines the function bodies of lambda terms $\lambda_1, \ldots, \lambda_{n-1}$. The result is a single lambda term with a function body consisting of the function bodies of lambda terms $\lambda_1, \ldots, \lambda_n$. This technique may not yield representations as compact as lambda extraction, but merging function bodies of consecutive lambdas often enables additional simplifications. As an example consider write sequence $a_n := write(a, \bar{i}, \bar{e})$ of size n, where $e_1, \ldots, e_n$ are equal. It corresponds to the following lambda sequence.

$$\begin{aligned}
\lambda_n &:= \lambda j_n \,.\, ite(j_n = i_n, e, \lambda_{n-1}(j_n)),\\
&\vdots\\
\lambda_1 &:= \lambda j_1 \,.\, ite(j_1 = i_1, e, a_0(j_1))
\end{aligned}$$

If we apply lambda merging to $\lambda_n, \ldots, \lambda_1$ and inline function bodies, we obtain the following lambda term

$$\begin{aligned}
\lambda_{n'} := \lambda j_n \,.\, ite(j_n = i_n, e,&\\
&\ddots\\
&\quad ite(j_n = i_1, e, a_0(j_n)))\ldots)
\end{aligned}$$

```
1  procedure merge_lambdas($\phi$)
2    for write sequence $\lambda_n := write(a, \bar{i}, \bar{e})$ in $\phi$
3      $b$ := rec_merge($n$, $j_n$, $\lambda_n$)
4      $\phi$ := $\phi[\lambda_n/b]$
5
6  procedure rec_merge($n$, $j_n$, $\lambda_i$)
7    /* base array $a_0$ */
8    if $i$ = 0 return $a_0(j_n)$
9    /* $\lambda_i = \lambda j_i \,.\, ite(j_i = i_i, e_i, \lambda_{i-1}(j_i))$ */
10   $t_{i-1}$ := rec_merge($n$, $j_n$, $\lambda_{i-1}$)
11   if $i$ < $n$
12     $t_i$ := $ite(j_i = i_i, e_i, \lambda_{i-1}(j_i))[j_i/j_n]$
13     return $t_i[\lambda_{i-1}(j_n)/t_{i-1}]$
14   /* top-most lambda $a_n$ */
15   return $\lambda_n[\lambda_{i-1}(j_n)/t_{i-1}]$
```

Fig. 5. Merge lambdas algorithm in pseudo-code.

Note that $\lambda_{n'}$ can be further simplified by merging the if-then-else terms into one (since the if-branch of each if-then-else contains value e), which results in lambda term $\lambda_{n''}$.

$$\lambda_{n''} := \lambda j_n \,.\, ite(j_n = i_n \vee \ldots \vee j_n = i_1, e, a_0(j_n))$$

A. Lemma Generation

Merged lambdas can be more compact than write sequences and may even be beneficial for lemma generation. For example, a read operation on $\lambda_{n''}$ at index j may produce a conflict on index i_1, where $read(\lambda_{n''}, j) \neq e$. As a consequence, the following lemma is generated.

$$j = i_n \vee \ldots \vee j = i_1 \rightarrow read(\lambda_{n''}, j) = e.$$

The resulting lemma covers all cases where $read(\lambda_{n''}, j)$ could produce a conflict on indices $i_2, \ldots, i_n$. In the original write sequence version, however, it might need n lemmas.

B. Algorithm

Figure 5 illustrates procedures `merge_lambdas` and `rec_merge` for merging lambda sequences. Given formula ϕ, for every lambda sequence λ_n, procedure `merge_lambdas` recursively merges the lambda terms in λ_n into lambda term b, which is then used to substitute lambda sequence λ_n in formula ϕ (line 4). Procedure `rec_merge` recursively traverses the lambda sequence starting at the top most lambda term λ_n and substitutes every bound variable j_i by the variable j_n, which is bound by the top most lambda term λ_n. In the base case ($i = 0$), the procedure returns a fresh read operation on base array a_0 at index j_n (which substitutes variable j_1). Else, it performs a recursive call on λ_{i-1}, which yields term t_{i-1}. For every lambda term λ_i with $i < n$, `rec_merge` generates lambda term t_i by substituting all occurrences of variable j_i in the function body of λ_i by j_n, and returns the lambda term obtained by substituting all occurrences of read operation $read(\lambda_{i-1}, j_n,)$ in t_i with term t_{i-1} (line 13). For the top most lambda ($i = n$), procedure `rec_merge` returns the lambda term obtained by substituting all occurrences of read operation $read(\lambda_{i-1}, j_n,)$ in λ_n with term t_{i-1} (line 15).

V. Experimental Evaluation

We implemented lambda extraction and merging in our SMT solver Boolector and evaluated our techniques on all non-extensional benchmarks from the QF_ABV category of the SMT-LIBv2 benchmark library. Six configurations are considered: (1) Boolector$_{\text{Base}}$, (2) Boolector$_{\text{E}}$, (3) Boolector$_{\text{M}}$, (4) Boolector$_{\text{X}}$, (5) Boolector$_{\text{XME}}$, and (4) Boolector$_{\text{XM}}$. The base line Boolector$_{\text{Base}}$ is an improved version of Boolector that won the QF_ABV track of the SMT competition in 2014. For the other configurations, subscript X indicates that lambda extraction is enabled, and subscript M indicates that lambda merging is enabled. Subscript E indicates an eager solving approach by reducing the formula to QF_BV. It eliminates lambda terms with beta reduction, and the remaining read operations, i.e., applications of uninterpreted functions (UF), by Ackermann reduction. The Boolector$_{\text{E}}$ and Boolector$_{\text{XME}}$ configurations essentially simulate an eager approach similar to that of UCLID [10].

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.2 LTS. The memory and time limit for each solver/benchmark pair was set to 7GB and 1200 seconds CPU time, respectively. In case of a timeout or memory out, a penalty of 1200 seconds was added to the total CPU time. Note that the time and memory limits and the hardware used for our experiments differ from the setup used at the SMT competition 2014.

Table I depicts the overall results consisting of the number of solved benchmarks (Solved), number of timeouts (TO), number of memory outs (MO), and the CPU time (Time) of all four configurations on the QF_ABV benchmarks. Enabling either lambda extraction (Boolector$_{\text{X}}$) or lambda merging (Boolector$_{\text{M}}$) improves the number of solved benchmarks by up to 17 instances and the runtime by up to 19% compared to Boolector$_{\text{Base}}$. Combining both techniques (Boolector$_{\text{XM}}$) solves 21 more benchmarks and requires 30% less runtime compared to Boolector$_{\text{Base}}$. This suggests, that lambda extraction and merging have orthogonal effects. They complement each other and in combination improve solver performance further (most of the time). However, if the eager solving approach is employed, both configurations Boolector$_{\text{E}}$ and Boolector$_{\text{XME}}$ do not show a notable improvement in terms of solved instances (less timeouts, but more memory outs). This is due to the high memory consumption caused by eager elimination of lambda terms and UFs, where Boolector$_{\text{E}}$ in total consumes 2.6 times (397 GB), and Boolector$_{\text{XME}}$ 2.3 times (347 GB) more memory than Boolector$_{\text{Base}}$. The other four configurations require roughly the same amount of memory. Table II depicts the overall results and the number of extracted patterns grouped by QF_ABV benchmark families in more detail. On benchmark families *bmc*, *brubiere2*, *klee*, *platania*, and *stp* Boolector$_{\text{XM}}$ considerably improves in terms of runtime and number of solved instances compared to Boolector$_{\text{Base}}$. On the *brubiere2* and *platania* benchmark families, the combined use of lambda extraction and lambda

Solver	Solved	TO	MO	Time [s]
Boolector$_{\text{Base}}$	13242	68	7	122645
Boolector$_{\text{E}}$	13242	49	26	120659
Boolector$_{\text{M}}$	13259	50	8	105647
Boolector$_{\text{X}}$	13256	54	7	99834
Boolector$_{\text{XME}}$	13246	47	24	111114
Boolector$_{\text{XM}}$	**13263**	46	8	**84760**

TABLE I
OVERALL RESULTS ON QF_ABV BENCHMARKS (13317 IN TOTAL).

merging yields significantly better results than both Boolector$_{\text{X}}$ and Boolector$_{\text{M}}$ alone. The most notable improvement in terms of runtime is achieved on the *klee* benchmark family, where all three configurations with *lambda extraction* enabled improve by orders of magnitude compared to Boolector$_{\text{Base}}$. The *klee* benchmark family consists of symbolic execution benchmarks obtained from KLEE [2], a symbolic virtual machine built on top of the LLVM compiler infrastructure. Previous versions of Boolector were shown to have rather poor performance on these benchmarks [8], which is confirmed by our experiments. This is due to the extreme version of lazy SMT in Boolector, using lemmas on demand. In our experiments, Boolector$_{\text{Base}}$ requires almost 13000 seconds to solve the 622 *klee* benchmarks, while lambda extraction improved runtime by up to a factor of 500 compared to Boolector$_{\text{Base}}$. This effect is illustrated by the scatter plot in Fig. 6, which shows that the runtime on most of the benchmarks is improved by a factor of 10 to 100. The *klee* benchmarks contain many instances of the λ_{mset} and $\lambda_{i \to e}$ patterns, where Boolector$_{\text{XM}}$ was able to extract 9373 and 10049 lambda terms with an average size of 108 and 11, respectively. On most of the benchmarks where Boolector$_{\text{XM}}$ was able to extract lambda terms, the runtime improved. The only exceptions are the two benchmarks in the *jager* benchmark family, on which Boolector$_{\text{XM}}$ still timed out even though 14028 λ_{mset} and 239 $\lambda_{i \to e}$ patterns were extracted. In total, Boolector$_{\text{XM}}$ was able to extract 29377 λ_{mset}, 13 λ_{mcpy}, 10683 $\lambda_{i \to e}$, 58 $\lambda_{i \to i}$, and 120 $\lambda_{i \to i+1}$ patterns with an average size of 40, 7, 12, 39, and 38, respectively. The overall time required by Boolector$_{\text{XM}}$ for extracting and merging the lambda terms amounts to 41 and 24 seconds, which is less than 0.01% of the total runtime and therefore negligible.

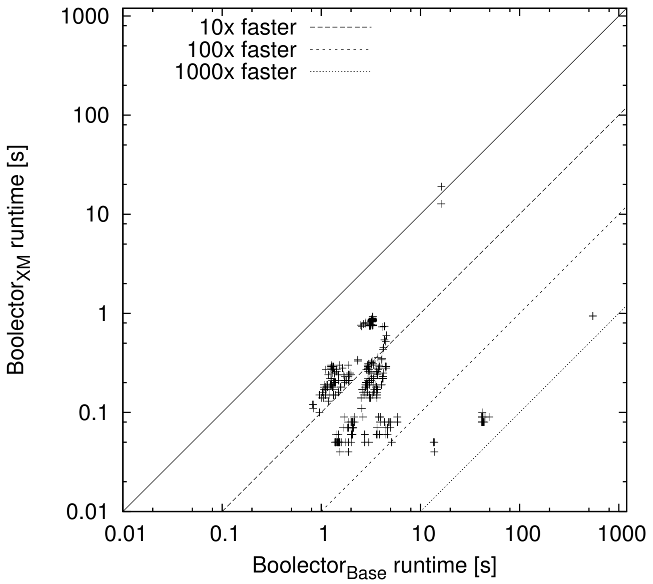


Fig. 6. Boolector$_{\text{Base}}$ vs. Boolector$_{\text{XM}}$ on *klee* benchmark family.

Benchmark family *brubiere* contains 11 benchmarks, which encode a *memcpy* operation on two non-overlapping memory regions and verify the correctness of the memcpy algorithm. The benchmarks are parameterized by the size of the copied memory region starting from size 2 up to size 12. We generated 21 additional benchmarks with size 2 to 2^{21} (i.e., 2^k with $1 \leq k \leq 21$) in order to evaluate how Boolector$_{\text{XM}}$ scales on these benchmarks. For comparison we additionally ran the top three solvers after Boolector at the SMT competition 2014, Yices [4] version 2.3.1, MathSAT [3] version 5.3.5, and SONOLAR [6] version 2014-12-04 on these benchmarks. Table III depicts the runtime of all solvers on the additional *memcpy* benchmarks of size 2 to 2^{21}, where T denotes out of time, and M denotes out of memory. Boolector$_{\text{Base}}$ and SONOLAR are able to solve these benchmarks up to size 2^5, MathSAT up to size 2^6, Yices up to size 2^9, and Boolector$_{\text{XM}}$ up to size 2^{20}. For the largest instance parsing consumes most of the runtime ($\sim$60%). For sizes greater than 2^{20}, Boolector is not able to fit the input formula into 7GB of memory, which results in a memory out.

Finally, we measured the impact of lambda extraction and lambda merging w.r.t. the number of generated lemmas. Since every lemma generated in Boolector entails an additional call to the underlying SAT solver, the number of generated lemmas usually correlates with the runtime of the solver. On the QF_ABV benchmarks commonly solved by Boolector$_{\text{Base}}$ and Boolector$_{\text{XM}}$ (13242 in total), Boolector$_{\text{Base}}$ generates 872913 lemmas, whereas Boolector$_{\text{XM}}$ generates 158175 lemmas, which is a reduction by a factor of 5.5. Consequently, the size of the CNF is reduced by 25% on average (no matter whether variables or clauses are counted). This is further illustrated in Fig. 7. On these benchmarks the reduction of the time spent in the underlying SAT solver is reduced from 59638 to 40101, i.e., an improvement of 33%.

VI. CONCLUSION

We discussed patterns of array operations occurring in actual benchmarks and presented a technique denoted as *lambda extraction*, which utilizes such patterns to extract compact and more succinct lambda terms. Another new complementary technique, called *lambda merging*, can still be exploited if lambda extraction is not applicable. These techniques allow to produce stronger and more succinct lemmas.

In the experimental analysis, based on our SMT solver Boolector, it was shown that these techniques reduce the number of generated lemmas by a factor of 5.5, and the overall size of the bit-blasted CNF by 25% on average. To summarize, we were able to considerably improve the overall performance

	Boolector$_{\text{Base}}$		Boolector$_{\text{E}}$		Boolector$_{\text{M}}$		Boolector$_{\text{X}}$		Boolector$_{\text{XME}}$		Boolector$_{\text{XM}}$		Extracted Patterns				
Family	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	**Slvd**	**[s]**	λ_{mset}	λ_{mcpy}	$\lambda_{i\to e}$	$\lambda_{i\to i}$	$\lambda_{i\to i+1}$
bench (119)	119	2	119	3	119	2	119	0.3	119	0.3	119	0.3	208	0	34	0	0
bmc (38)	38	1361	39	769	39	921	39	197	**39**	**88**	39	182	256	3	56	0	0
brubiere (98)	75	29455	75	30301	75	28944	75	29359	75	29167	**75**	**28854**	0	10	0	0	0
brubiere2 (22)	17	7299	21	2617	18	6927	18	7842	**22**	**2034**	20	3241	1392	0	8	0	0
brubiere3 (8)	0	9600	1	8464	1	8435	0	9600	1	8463	1	8435	0	0	0	0	0
btfnt (1)	1	134	1	144	1	134	1	134	1	146	1	134	0	0	0	0	0
calc2 (36)	36	862	36	1528	36	864	36	863	36	1527	36	863	0	0	0	0	0
dwp (4188)	4187	2668	**4188**	**2216**	4187	2090	4187	2666	4187	2235	4187	2089	42	0	0	0	0
ecc (55)	**54**	**1792**	54	1745	54	1792	54	1845	54	1808	54	1845	125	0	0	0	0
egt (7719)	7719	222	7719	544	7719	221	7719	225	7719	275	**7719**	**212**	3893	0	0	0	0
jager (2)	0	2400	0	2400	0	2400	0	2400	0	2400	0	2400	14028	0	239	0	0
klee (622)	622	12942	620	4408	622	12688	622	169	**622**	**126**	622	154	9373	0	10049	0	0
pipe (1)	1	10	1	14	1	10	1	10	1	14	1	10	0	0	0	0	0
platania (275)	247	42690	238	58807	256	35005	255	34993	240	56172	**258**	**31189**	0	0	0	58	120
sharing (40)	40	2460	40	2459	40	2459	40	2460	40	2458	40	2458	0	0	0	0	0
stp (40)	34	8749	38	4238	39	2755	38	7072	38	4200	**39**	**2695**	60	0	297	0	0
stp_sa (52)	52	0.7	52	0.5	52	0.6	52	0.6	**52**	**0.5**	52	0.7	0	0	0	0	0
totals (13317)	13242	122645	13242	120659	13259	105647	13256	99834	13246	111114	**13263**	**84760**	29377	13	10683	58	120

TABLE II
OVERALL RESULTS AND NUMBER OF EXTRACTED PATTERNS ON ALL QF_ABV BENCHMARKS GROUPED BY BENCHMARK FAMILY.

Solver	**k=1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
Boolector$_{\text{Base}}$	0.1	0.4	8	42	296	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	M
SONOLAR	0.1	0.2	2	15	201	T	T	T	T	T	T	T	T	T	T	T	T	M	M	M	M
MathSAT	0.1	0.3	2	9	70	709	T	M	T	T	T	T	T	T	T	T	T	T	M	M	M
Yices	**0.0**	**0.0**	0.1	0.6	2	8	23	93	371	T	T	T	T	T	T	T	T	M	M	M	T
Boolector$_{\text{XM}}$	0.1	0.1	0.1	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.2**	**0.2**	**0.3**	**0.5**	**1**	**2**	**6**	**14**	**44**	**140**	**463**	M

TABLE III
RUNTIME IN SECONDS ON *memcpy* BENCHMARKS OF SIZE 2^k COPIED ELEMENTS. T DENOTES OUT OF TIME, M DENOTES OUT OF MEMORY.

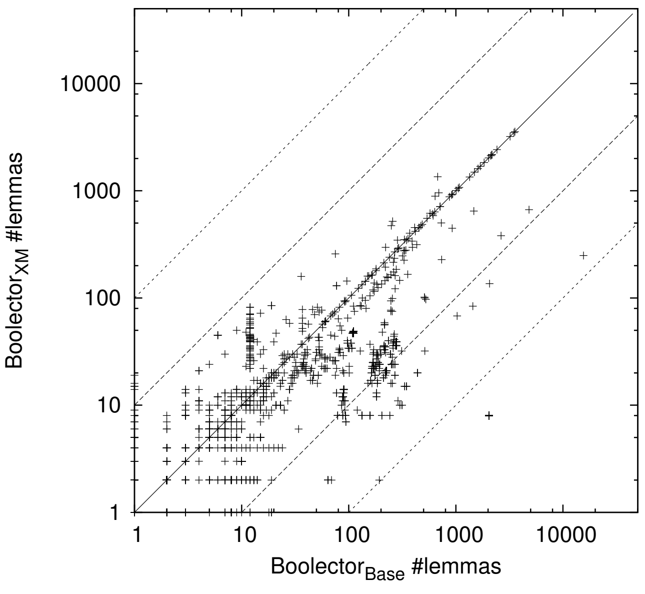


Fig. 7. Number of generated lemmas Boolector$_{\text{Base}}$ vs. Boolector$_{\text{XM}}$ on commonly solved QF_ABV benchmarks (13242 in total).

of Boolector and achieve speedups up to orders of magnitude, particularly on benchmarks from symbolic execution.

We believe, that there are additional patterns in software and hardware verification benchmarks, which can be extracted as lambdas and used to speed-up array reasoning further. Our results also suggest, that a more expressive theory of arrays might be desirable for users of SMT solvers, in order to allow more succinct encodings of common array operation patterns.

REFERENCES

[1] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proc. CAV*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX*, pages 209–224. USENIX Association, 2008.

[3] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[4] B. Dutertre. Yices 2.2. In *Proc. CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[5] S. Falke, F. Merz, and C. Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Proc. VSTTE, Selected Papers*, volume 8164 of *LNCS*, pages 108–128. Springer, 2013.

[6] F. Lapschies, J. Peleska, and E. Gorbachuk. System Description: SONOLAR SMT-COMP 2014. http://smtcomp.sourceforge.net/2014/systemDescriptions/sonolar-smtcomp2014.pdf, 2014.

[7] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.

[8] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proc. CAV*, volume 8044 of *LNCS*, pages 53–68. Springer, 2013.

[9] M. Preiner, A. Niemetz, and A. Biere. Lemmas on demand for lambdas. In *Proc. DIFTS*, volume 1130 of *CEUR Workshop Proceedings*, 2013.

[10] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification.* PhD thesis, CMU, 2005.

CAQE: A Certifying QBF Solver

Markus N. Rabe
University of California, Berkeley
rabe@berkeley.edu

Leander Tentrup
Saarland University
tentrup@cs.uni-saarland.de

***Abstract*—We present a new CEGAR-based algorithm for QBF. The algorithm builds on a decomposition of QBFs into a sequence of propositional formulas, which we call the clausal abstraction. Each of the propositional formulas contains the variables of just one quantifier level and additional variables describing the interaction with adjacent quantifier levels. This decomposition leads to a simpler notion of refinement compared to earlier approaches. We also show how to effectively construct Skolem and Herbrand functions from true, respectively false, QBFs; allowing us to certify the solver result.**

We implemented the algorithm in a solver called CAQE. The experimental evaluation shows that CAQE has competitive performance compared to current QBF solvers and outperforms previous certifying solvers.

I. Introduction

Efficient solving techniques for Boolean theories are an integral part of modern verification and synthesis methods. The ever growing complexity of verification and synthesis problems led to propositional problems of enormous size. To see further advances in these areas, we believe that is necessary to move to more compact representations of these requirements. Quantified Boolean formulas (QBFs) have repeatedly been considered as a candidate theory to compactly encode Boolean problems [1]–[7]. Recent advances in QBF solvers give raise to the hope that QBF may help to increase the scalability of verification and synthesis approaches.

The recent introduction of algorithms based on counterexample guided abstraction refinement (CEGAR) significantly improved the scalability of QBF solving [8], [9]. However, the CEGAR approach shows poor performance for instances with many quantifier alternations, as we show in this paper, and it currently lacks the ability to certify its results. In this work, we present a modification of the CEGAR approach for QBF that tackles these two problems.

Certifying the results is particularly important for QBF, as the pure yes/no answer is of little use. Like the propositional SAT problem [10], [11], QBF can be used to encode objects of interest, like error paths [3], [5] and implementations [4], [12], [13]. While the yes/no answer of a SAT or QBF solver then provides the information about the *existence* of this object, we often want to construct a concrete instance for further use. For the propositional SAT problem the object can typically be extracted as the assignment of the variables, but for QBF the object potentially consists of the Skolem functions for the existential quantifiers or the Herbrand functions for the universal quantifiers. Most current QBF solvers, however, are unable to provide Skolem or Herbrand functions or suffer performance penalties when they do [14], [15].

This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by NSF grants CCF-1139138 and CCF-1116993. The first author did a part of the work at Saarland University.

Our approach is based on the observation that the only information relevant for the processing of inner quantifier levels is which clauses are satisfied by the outer quantifier levels. Consider the following example:

$$\forall X \exists Y.\ (x_1 \lor \overline{x_2} \lor y_1) \land (x_2 \lor \overline{y_1} \lor \overline{y_2}) \land (y_1 \lor y_2)$$

where $x_i \in X$ and $y_i \in Y$ for all i.

To determine the truth value of this QBF, we need to show that for every assignment of the variables X, there is an assignment of the variables Y such that the propositional part of the formula above is true. Any assignment of X satisfies a certain set of clauses and thereby requires that the remaining set of clauses is satisfiable by the variables Y. For example the assignment $x_1\overline{x_2}$ satisfies exactly the first clause and any assignment of the variables Y that satisfies the remaining clauses is sufficient for this case. We can thus split the formula into two parts; one for the universally quantified variables X and one for the existentially quantified variables Y:

$$\begin{aligned}\varphi_X &:= ((x_1 \lor \overline{x_2}) \rightarrow \neg b_1) \land (x_2 \rightarrow \neg b_2) \land (\text{false} \rightarrow \neg b_3)\,,\\ \varphi_Y &:= (t_1 \lor y_1) \land (t_2 \lor \overline{y_1} \lor \overline{y_2}) \land (t_3 \lor y_1 \lor y_2)\,,\end{aligned}$$

where the variables $B = \{b_1, b_2, b_3\}$ (<u>b</u>ottom) indicate that the clause is satisfied by the lower quantifier level ($\exists Y$), and the variables $T = \{t_1, t_2, t_3\}$ (<u>t</u>op) indicate that the clause is satisfied by the upper quantifier level ($\forall X$). That is, for every clause φ_X requires that whenever the clause is satisfied by the X variables, it does not have to be satisfied by the Y variables. And φ_Y requires that when the clause is satisfied by some X, it does not have to be satisfied by the variables Y. The problem to determine the truth of the QBF is then equivalent to determining whether for each satisfying assignment of φ_X that includes the assignment $\mathbf{b}$ of B, the formula $\varphi_Y(\mathbf{t_b})$ is satisfiable, where $\mathbf{t_b}$ is the assignment of T that assigns t_i iff b_i is *not* assigned in $\mathbf{b}$ (for all $1 \leq i \leq 3$). We call this decomposition of the QBF the *clausal abstraction*.

Following the CEGAR approach to QBF, we would alternate between the two quantifier levels and determine satisfying assignments of φ_X and φ_Y. When there is no assignment of φ_X left, we conclude that the original formula is true, or when there is no satisfying assignment of φ_Y for a given assignment of X, we have found a counter-example. While the overall approach is similar to the existing CEGAR approaches to 2QBF [9], it lets us rephrase the refinement step in an interesting way: Every satisfying assignment α of φ_Y defines a single clause over the variables B that we can add to the formula φ_X. This excludes all assignments of X for which α satisfies the remaining formula.

The principle of clausal abstractions can be lifted to full QBF and we show that this leads to an algorithm with competitive performance. The evaluation of our implementation CAQE reveals the differences to the previous CEGAR-based QBF solver RAReQS: The algorithm we propose is particularly effective for QBFs with many quantifier alternations, while the previous CEGAR-based approach seems particularly effective for problems with few quantifier alternations.

Our approach can be used to certify the result of a QBF. From the sequence of T assignments and assignments of the quantified variables, we can effectively extract Skolem and Herbrand functions in the form of circuits. We describe a proof format and provide a tool chain for the certification process, which outperforms earlier certifying QBF solvers.

To summarize, the contributions of this paper are twofold:

- We develop a CEGAR algorithm for QBF based on clausal abstractions, and
- we give a method to effectively extract Skolem and Herbrand functions for the certification of the results.

II. Quantified Boolean Formulas

A quantified Boolean formula (QBF) is a propositional formula over a finite set of variables X with domain $\mathbb{B} = \{0, 1\}$ extended with quantification. The syntax is given by the following grammar:

$$\varphi := x \mid \neg\varphi \mid (\varphi) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x.\, \varphi \mid \forall x.\, \varphi \;\;,$$

where $x \in X$. For readability, we lift the quantification over variables to the quantification over sets of variables and denote $\forall x_1.\forall x_2.\ldots\forall x_n.\varphi$ with $\forall X.\varphi$ and $\exists x_1.\exists x_2.\ldots\exists x_n.\varphi$ with $\exists X.\varphi$, accordingly, for $X = \{x_1, \ldots, x_n\}$.

Given a subset of variables X, an *assignment* of X is a function $\alpha : X \to \mathbb{B}$ that maps each variable $x \in X$ to either true (1) or false (0). For simplicity we describe assignments also by the subset $\mathbf{x} \subseteq X$ of variables that are assigned 1 (or true). We denote *the set of assignments* of a set of variables X by $\mathcal{A}(X)$.

A quantifier $Q\, x.\, \varphi$ for $Q \in \{\exists, \forall\}$ *binds* the variable x in the *scope* φ. Variables that are not bound by a quantifier are called *free*. We assume the natural semantics of the satisfaction relation $\mathbf{x} \models \varphi$ for QBF φ and assignments $\mathbf{x} \subseteq X$ where X are the free variables of φ. *QBF satisfiability* is the problem to determine, for a given QBF φ, the existence of an assignment for the free variables of φ, such that the formula is true.

The dependency set of an existentially quantified variable y, denoted by $\mathit{dep}(y)$, is the set X of universally quantified variables x such that $\exists y.\, \varphi$ is in the scope of x. A *Skolem function* $f_y : \mathcal{A}(\mathit{dep}(y)) \to \mathbb{B}$ maps an assignment of the dependencies of y to an assignment of y. The truth of a QBF φ is equivalent to the existence of a Skolem function f_y for every variable y of the existentially quantified variables Y, such that $\{y \in Y \mid f_y(\mathbf{x} \cap \mathit{dep}(y))\} \models \varphi$ holds for every assignment $\mathbf{x}$ of the universal variables X.

A *closed* QBF is a formula without free variables. Closed QBFs are either true or false. A formula is in prenex normal form, if the formula consists of a quantifier prefix followed by a propositional formula. Every QBF can be transformed into a closed QBF and into prenex form while maintaining satisfiability. For a $k > 0$, a formula φ is in the kQBF fragment if it is closed, in prenex normal form, and has exactly k alternations between $\exists$ and $\forall$ quantifiers.

A *literal* l is either a variable $x \in X$, or its negation $\neg x$. Given a set of literals $\{l_1, \ldots, l_n\}$, the disjunctive combination $(l_1 \vee \ldots \vee l_n)$ is called a *clause* and the conjunctive combination $(l_1 \wedge \ldots \wedge l_n)$ is called a *cube*. Given a literal l, the polarity of l, $\mathit{sign}(l)$ for short, is 1 if l is positive and 0 otherwise. The variable corresponding to l is defined as $\mathit{var}(l) = x$ where $x = l$ if $\mathit{sign}(l) = 1$ and $x = \neg l$ otherwise.

A QBF is in prenex conjunctive normal form (PCNF) if its propositional formula is a conjunction over clauses, which is called a *matrix*. To simplify the notation, we treat a matrix ψ as a set of clauses $\psi = \{C_1, \ldots, C_n\}$ and a clause C as a set of literals $C = \{l_1, \ldots, l_m\}$ and use standard set operations like intersection and union for their manipulation. Every prenex QBF can be transformed into prenex CNF using the Tseitin transformation [16] with a linear increase in the size of the formula and number of existential variables.

III. Clausal Abstractions

In this section, we present clausal abstractions, a decomposition of QBFs into sequences of propositional formulas—one propositional formula for each quantifier level. Clausal abstractions provide us a new notion of refinement and thereby leads us to a variant of the CEGAR algorithm for QBF.

The example in the introduction intuitively explained how every clause in a QBF with one quantifier alternation can be split into two parts with additional variables that describe their interaction. The main observation was that every assignment for the variables quantified by the inner (existential) quantifier, corresponds to a single cube over the new variables T. In the following, we extend this principle to QBF with more than one quantifier alternation and we consider a closed QBF φ in prenex conjunctive normal form:

$$\varphi := Q_1 X_1.\, \ldots\, Q_n X_n.\, \psi\,,$$

where $\psi = C_1 \wedge \ldots \wedge C_k$ is the matrix of φ with k clauses and each $Q_i X_i$ is a *quantifier block*, i.e., a maximal list of consecutive quantifiers of the same type.

Let's first consider the case that Q_1 is an existential quantifier. Proving φ to be true means to find an assignment $\mathbf{x_1}$ for X_1 such that the remaining formula $Q_2 X_2.\, \ldots\, Q_n X_n.\, \psi(\mathbf{x_1})$ is true. Inspecting $\psi(\mathbf{x_1})$ reveals that the assignment of the variables X_1 eliminated certain clauses (by satisfying them) and removed all occurrences of X_1 literals from the remaining clauses. We can thus split each clause C_i into two parts $C_{i,1}$

and $C_{i,>1}$, where $C_{i,1}$ is the part that can be satisfied by some assignment to X_1 and $C_{i,>1}$ is the part that must be satisfied by some variable in X_i with $1 < i \leq n$. For each clause C_i, we introduce the variables b_i (bottom) in $C_{i,1}$ and t_i (top) $C_{i,>1}$ to indicate by which part C_i is satisfied.

$$\begin{aligned} C_{i,1} &= \left(\bigvee_{l \in C_i \wedge var(l) \in X_1} l\right) \vee b_i \\ C_{i,>1} &= \left(\bigvee_{l \in C_i \wedge var(l) \in \left(\bigcup_{i>1} X_i\right)} l\right) \vee t_i \end{aligned}$$

$C_{i,1}$ contains no variables in $\bigcup_{i>1} X_i$ and $C_{i,>1}$ is free of the variables X_1. We call the conjunction of clauses $\varphi_{\exists X_1} = \bigwedge_{j<k} C_{j,1}$ the *existential clausal abstraction* for X_1.

As the variables B occur only positively in $C_{i,1}$, the existential clausal abstraction is monotone in B. In particular, there is a *unique* minimal assignment $\mathbf{b}_{\min}(\mathbf{x_1})$ to B for every assignment $\mathbf{x_1}$ to X_1. The minimal assignment $\mathbf{b}_{\min}(\mathbf{x_1})$ contains exactly the set B variables whose clauses have to be satisfied by a variable from a quantifier block $i > 1$ when the first quantifier block chooses $\mathbf{x_1}$. Hence it is clear that the formula that results from fixing $\mathbf{x_1}$ in the matrix ψ is the same as the matrix that results from fixing $\mathbf{t} = \{t_i \mid b_i \notin \mathbf{b}_{\min},\ 1 \leq i \leq k\}$ in the remaining matrix $\psi_{>1} = \bigwedge_{i \leq k} C_{i,>1}$.

Now let's turn to the case that the outermost quantifier Q_1 is a universal quantifier. Analogue to the previous case, disproving φ means to show that there is an assignment $\mathbf{x_1}$ of X_1 such that the remaining formula $Q_2X_2. \ldots Q_nX_n.\ \psi(\mathbf{x_1})$ is false. Now, assignments of X_1 that satisfy *less* clauses are more desirable, as they set more of the variables B to true and therefore make it harder to satisfy the remaining formula. The existential clausal abstraction, however, always allows us to set more of the variables B to true and therefore fails to represent when a clause is *necessarily* fulfilled by an assignment $\mathbf{x_1}$. In other words, the existential clausal abstraction represents the lower bounds on B, while we need the upper bounds on B for the universal case. For the universal case, we therefore propose to use the following clausal abstraction:

$$\begin{aligned} C_{i,1} &= \bigwedge_{l \in C_i \wedge var(l) \in X_1} (\bar{l} \vee \bar{b}_i) \\ C_{i,>1} &= \left(\bigvee_{l \in C_i \wedge var(l) \in \left(\bigcup_{i>1} X_i\right)} l\right) \vee t_i \end{aligned}$$

The *universal clausal abstraction* for X_1 is then the conjunction $\varphi_{\forall X_1} = \bigwedge_{j<k} C_{j,1}$. The universal clausal abstraction guarantees that for every assignment $\mathbf{x_1}$ to X_1 there is a unique *maximal* assignment $\mathbf{b}_{\max}(\mathbf{x_1})$ to B, that is an assignment with a maximal number of variables set to true. The decomposition of each clause C_i into the conjunction $C_{i,1}$ ensures that whenever the clause C_i is satisfied by one of its literals, then the variable b_i must be set to false—representing that the other quantifier blocks do not have to satisfy this clause any more. Again, the formula that results from fixing $\mathbf{x_1}$ in $C_1 \wedge \ldots \wedge C_k$ is the same as the matrix that results from fixing $\mathbf{t} = \{t_i \mid b_i \notin \mathbf{b}_{\max},\ 1 \leq i \leq k\}$ in the remaining formula $\psi_{>1} = C_{1,>1} \wedge \ldots \wedge C_{k,>1}$.

We observe that each clausal abstraction only needs one copy of the T variables and another copy of the B variables. When we build the clausal abstractions $\varphi_{Q_iX_i}$ for all quantifier blocks Q_i, we can thus reuse the sets of variables for T and B and only need to introduce $2k$ fresh variables.

IV. CEGAR WITH CLAUSAL ABSTRACTIONS

In this section, we present a CEGAR algorithm for QBF based on clausal abstractions. To formulate the algorithm, we assume a method called SAT to solve propositional formulas. We assume that SAT returns whether the formula is satisfiable (SAT) or unsatisfiable (UNSAT). Further, in case the formula is satisfiable SAT returns a satisfying assignment—potentially only for a subset of the variables like in line 6. In our implementation these queries are solved by a SAT solver. In the following, we use $\Psi_{>1} = Q_2X_2 \ldots Q_nX_n.\psi_{>1}$ to denote the formula that remains when splitting the clausal abstraction $\varphi_{Q_1X_1}$ from the QBF Ψ. Expressions of the form $c\ ?\ e_1 : e_2$ denote abbreviated if-statements. If the conditional c evaluates to true we return e_1 and otherwise e_2.

The input to the algorithm SOLVE is a QBF $QX.\,\Psi$ and the output is either SAT or UNSAT.

```
1:  procedure SOLVE($QX.\,\Psi$)
2:      if $\Psi$ is propositional then
3:          return SAT($\Psi$)
4:      $\alpha \leftarrow (Q = \exists)\ ?\ \varphi_{\exists X} : \varphi_{\forall X}$
5:      while true do
6:          result, $\mathbf{b} \leftarrow$ SAT($\alpha$)
7:          if result = UNSAT then
8:              return $(Q = \exists)$ ? UNSAT : SAT
9:          $\mathbf{t} \leftarrow \{t_i \mid b_i \notin \mathbf{b},\ 1 \leq i \leq k\}$
10:         result $\leftarrow$ SOLVE($\Psi_{>1}(\mathbf{t})$)
11:         if $Q = \exists$ and result = UNSAT then
12:             $\alpha \leftarrow \alpha \wedge (\bigvee_{l \in \mathbf{b}} \bar{l})$
13:         else if $Q = \forall$ and result = SAT then
14:             $\alpha \leftarrow \alpha \wedge (\bigvee_{l \notin \mathbf{b}} l)$
15:         else
16:             return $(Q = \exists)$ ? SAT : UNSAT
```

The algorithm generates an assignment $\mathbf{b}$ for the variables B in the clausal abstraction, determines $\mathbf{t} = \{t_i \mid b_i \notin \mathbf{b},\ 1 \leq i \leq k\}$, and then goes into recursion for the remaining formula $Q_2X_2 \ldots Q_nX_n.\psi_{>1}(\mathbf{t})$. Whenever a recursive call failed for an existential quantifier block, i.e., the remaining formula turned out to be false, we add the clause $\bigvee_{l \in \mathbf{b}} \bar{l}$ to the existential clausal abstraction. Analogously, when a recursive call failed for a universal quantifier block, i.e., the remaining formula turned out to be true, we add the clause $\bigvee_{l \notin \mathbf{b}} l$ to the universal clausal abstraction. The next iteration will either bring up a new assignment for the variables B or fail to do so, in which case the formula is violated in the existential case and satisfied in the universal case, respectively.

A. Reusing Clausal Abstractions and their Refinements

Reusing the state of SAT solvers is critical for performance. Instead of regenerating the clausal abstractions for inner quantifier for every instantiation of the variables of the outer quantifier blocks, we set up one SAT solver per quantifier block that we keep for the complete run of the algorithm. Reusing the SAT solvers for each quantifier level also enables us to efficiently reuse all previous refinements for the same level. The clauses by which we refined stay valid for all times.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

```
procedure $\textsc{solve}_\exists(\exists X.\,\Psi,\ \mathbf{t})$
    while true do
        $result, \mathbf{b}, failed \leftarrow \textsc{sat}(\varphi_X, \mathbf{t})$
        if $result = $ UNSAT then
            return UNSAT, _, $failed$
        else if $\Psi$ is propositional then
            return SAT, $\mathbf{t}$, _
        $\mathbf{t_b} \leftarrow \{t_i \mid b_i \notin \mathbf{b},\ 1 \leq i \leq k\}$
        $result, \mathbf{t}', failed' \leftarrow \textsc{solve}_\forall(\Psi, \mathbf{t} \cup \mathbf{t_b})$
        if $result = $ UNSAT then
            $\varphi_X \leftarrow \varphi_X \wedge (\bigvee_{t \in failed'} \neg b_t)$
        else
            return SAT, $\mathbf{t}'$, _
```

```
procedure $\textsc{solve}_\forall(\forall X.\,\Psi,\ \mathbf{t})$
    while true do
        $result, \mathbf{t}', failed \leftarrow \textsc{sat}(\varphi_X, \mathbf{t}^+)$
        if $result = $ UNSAT then
            return SAT, $failed$, _
        $result, \mathbf{t}'', failed' \leftarrow \textsc{solve}_\exists(\Psi, \mathbf{t}')$
        if $result = $ SAT then
            $\varphi_X \leftarrow \varphi_X \wedge (\bigvee_{t \in \mathbf{t}''} \neg t)$
        else
            return UNSAT, _, $failed'$
```

Given a QBF with matrix ψ we prepare a SAT solver for each existential quantifier block $\exists X$ with the clauses

$$\varphi_X \coloneqq \bigwedge_{C_i \in \psi} \Big(\big(\bigvee_{l \in C_i \wedge var(l) \in X} l \big) \vee t_i \vee b_i \Big) , \qquad (1)$$

and for each universal quantifier block $\forall X$, we prepare a SAT solver with the clauses

$$\varphi_X \coloneqq \bigwedge_{C_i \in \psi} \Big(\bigwedge_{l \in C_i \wedge var(l) \in X} (\bar{l} \vee t_i) \Big) . \qquad (2)$$

In the construction of the clausal abstraction for the final level, i.e., φ_{X_k}, we additionally require the bottom literals B to be false, as there is no quantifier level below the current level to which we could pass the proof obligations.

To obtain the clausal abstraction of a QBF $\Psi(\mathbf{t})$ we then simply *assume* the assignment $\mathbf{t}$. Solving formulas under assumptions is a common feature of modern SAT solvers.

Note that formula (2) does not contain B variables. For the universal clausal abstractions it is possible to join the two types of literals and ask for a satisfying assignment of φ_X that assigns a minimal number of T variables to true, under the assumption that at least those that were given in the function call are true. This is merely a measure to reduce the number of variables used in the formula.

B. Algorithm with Optimizations

The input to the algorithm $\textsc{solve}$ consists of a QBF $Q_i X_i.\,\Psi$. Depending on the type of the outermost quantifier, it calls $\textsc{solve}_\exists$ or $\textsc{solve}_\forall$ and fixes an initial assignment $\mathbf{t} = \emptyset$ of T that indicates that no clauses were satisfied so far.

```
procedure $\textsc{solve}(\Phi)$
    return $\Phi = \exists X.\,\Psi$ ? $\textsc{solve}_\exists(\Phi,\ \emptyset)$ : $\textsc{solve}_\forall(\Phi,\ \emptyset)$
```

The algorithm $\textsc{solve}_\exists$ makes use of a feature of modern SAT solvers by *assuming* a partial assignment for a particular call. This feature enables us to deactivate those clauses that are satisfied already. The notation $\textsc{sat}(\varphi_X, \mathbf{t})$ denotes a SAT call for which we additionally assume the assignment $\mathbf{t}$. SAT calls with assumptions either return a satisfying assignment, which is guaranteed to have the sub-assignment $\mathbf{t}$, or in case the formula is unsatisfiable under the assumptions they additionally return a set of *failed assumptions*, denoted by *failed*. The failed assumptions are a subset of the assumptions $\mathbf{t}$ and suffice to make the formula unsatisfiable. In line 11, $\neg b_t$ denotes the variable $b \in B$ that corresponds to the same clause as variable t. The algorithm refines the clausal abstraction only by those variables B that made the recursive call to $\textsc{solve}_\forall$ unsatisfiable. This significantly strengthens the refinement compared to the basic algorithm.

Similar to the algorithm $\textsc{solve}_\exists$, the algorithm $\textsc{solve}_\forall$ makes use of assumptions for SAT calls. Since, we joined the T and B variables, we only assume the positively assigned variables in $\mathbf{t}$, denoted by the call $\textsc{sat}(\varphi_X, \mathbf{t}^+)$. The SAT call then produces an assignment $\mathbf{t}'$ for which it possibly sets further variables T to true. In contrast to the existential case, we can use t' directly for the recursive call. The refinement step only refines by the variables T occurring in the assignment $\mathbf{t}''$ returned by the recursive call. The assignment $\mathbf{t}''$ is a subset of $\mathbf{t}'$ and therefore represents the information that the call to $\textsc{solve}_\exists$ satisfied more clauses than required by the assignment $\mathbf{t}'$.

Theorem 1. $\textsc{solve}(\Phi)$ *is correct and terminates.*

The proof is a simple induction over the quantifier quantifier hierarchy.

C. Stronger Refinements

The algorithm above always refines by a single clause. In certain cases, however, we can strengthen the refinement in $\textsc{solve}_\exists$ by excluding a conjunction of clauses $\mathcal{C}$, that are equivalent in the following sense: If some clause C corresponds to a failed assumption, then all other clauses $\mathcal{C} \setminus \{C\}$ would also lead to a failed assumption. Formally, we characterize this criterion by the subset relation between clauses restricted to the lower level literals. Let $\exists_i X_i$ be a quantifier block of a QBF with matrix ψ, let $failed'$ be the failed assumptions returned by the lower level (line 9), and let $t_k \in failed'$ be one of the failed assumptions. If C_k cannot be satisfied by a quantifier block $Q_j X_j$ with $j > i$, then any C_l with $C_l \cap \Big(\bigcup_{j>i} X_j \Big) \subseteq C_k \cap \Big(\bigcup_{j>i} X_j \Big)$, $C_l \subseteq_i C_k$ for short, cannot be satisfied either. Hence, given the failed assumptions $failed'$, we refine

$$\bigvee_{t_k \in failed'} \ \bigwedge_{\substack{C_l \in \psi, \\ C_l \subseteq_i C_k}} \neg b_l \, , \qquad (3)$$

```
p cap 3 3
d
d
6 -3
u SAT
d
4 5 3
u SAT
u SAT
1
u SAT
r SAT
```

$$\begin{tikzcd}
 & \arrow[u, phantom, "\uparrow u"] \langle \emptyset, \{x_1\}, \text{SAT}\rangle \arrow[d, dash, "\downarrow d"', "\uparrow u"] & \\
 & \langle \emptyset, \emptyset, \text{SAT}\rangle \arrow[dl, dash, "\swarrow d"', "\nearrow u"] \arrow[dr, dash, "\nwarrow u", "\searrow d"'] & \\
\langle \{t_3\}, \{\overline{x_3}\}, \text{SAT}\rangle & & \langle \{t_1, t_2\}, \{x_3\}, \text{SAT}\rangle
\end{tikzcd}$$

Fig. 1. A clausal abstraction proof in the CAP format (left) and its tree structure (right).

The refinement has to be transformed to CNF using the Tseitin transformation [16].

Additionally, after we have found an assignment for an existential quantifier block, we have a routine that checks whether the assignment satisfies clauses that are deactivated by the current T assignment. If so, we delete the corresponding literals from the T assignment.

D. Preprocessing Techniques

We use basic preprocessing techniques that can be easily integrated into our certification infrastructure: tautology clauses, pure literals, unit clauses, universal reduction, and miniscoping. For miniscoping, we apply the well known rule $\forall X. \exists Y_1, Y_2. \varphi(X, Y_1) \wedge \psi(X, Y_2) \equiv (\forall X. \exists Y_1. \varphi(X, Y_1)) \wedge (\forall X. \exists Y_2. \psi(X, Y_2))$. That is, we search for a partitioning of the matrix according to the existential variables of the current scope. By applying this rule bottom-up, we get a tree-shaped quantification header. Note that this tree only branches after an existential quantifier, hence, we modify the algorithm to split the current entry according to the partitioning and solve every child individually. For true QBF instances, this transformation can significantly reduce the size of the Skolem functions.

V. Certification

Similar to the QBFCert [15] framework, we propose a two step approach to certification that allows us to keep the certification infrastructure separate from the solver. First, our solver outputs a *clausal abstraction proof* (CAP), that is a sequence of assignments of T variables together with the corresponding assignments of X variables as well as navigation symbols to determine the quantification level. A clausal abstraction proof is essentially a *post*-order linearization of the recursion tree.

Clausal abstraction proofs contain the following elements:

- Header: `p cap` v c where v is the maximal variable number and c is the number of clauses.
- Result: `r` *res* where *res* $\in \{\text{SAT}, \text{UNSAT}\}$.
- Quantifier tree navigation: `d` for **d**own, `u` *res* for **up** with subtree result *res* $\in \{\text{SAT}, \text{UNSAT}\}$, and `n` for **n**ext sibling (in the case of miniscoping).
- T and variable assignments: $t_1\ t_2\ \ldots\ l_1\ l_2\ \ldots$, with $v < t_i \leq v + c$ for every i and $0 < |l_j| \leq v$ for every j.
- The strengthening instruction `s c` $t_1 \ldots t_n$ representing the cube of T variables used in the strong refinement optimization described by equation (3) in Section IV-C.

As an example, consider the following QBF:

$$\exists x_1 \forall x_2 \exists x_3 : \underbrace{(x_1 \vee x_2 \vee \overline{x_3})}_{t_1 \equiv x_4} \underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{t_2 \equiv x_5} \underbrace{(\overline{x_1} \vee \overline{x_2} \vee x_3)}_{t_3 \equiv x_6}$$

Figure 1 shows a clausal abstraction proof for this QBF. After the header, the proof descends to the lowest quantifier (lines `d` and `d`) where the decision $x_3 = 0$ is a satisfying assignment given that clause 3, corresponding to t_6, is satisfied by a higher level quantifier (line `6 -3`). The algorithm presented in Section IV guarantees that there is an assignment of the higher quantifier levels that satisfies the this set of clauses (i.e. clause 3). The proof format, however, delays printing the assignments of higher quantifier levels until it *successfully* ascends to the upper levels of the proof. Omitting assignments for failed proof branches saves a significant amount of space. Next we ascend to the universal quantifier level (line `u SAT`). The previous assignment of the universal quantified variable x_2 can be omitted, as it did not refute the current proof branch. The universal level chooses a new assignment for x_2 and the proof descends again (line `d`). This time clauses 1 and 2, corresponding to t_4 and t_5, are satisfied by a higher level clause and $x_3 = 1$ is a satisfying assignment for the remaining clauses (line `4 5 3`) and we ascend again (line `u SAT`). We exhausted the assignments of x_2 and the proof hence ascends from the universal quantifier level (line `u SAT`). Upon returning successfully from this proof branch, we print the assignment of $x_1 = 1$ (line `1`) that was chosen in the beginning. We return successfully from the outermost existential quantifier level (line `u SAT`) and the QBF is concluded to be true (`r SAT`).

We proceed with the general description of the certification approach. From the clausal abstraction proof, we build a circuit—encoded as an And-Inverter-Graph (AIG)—representing the Skolem or Herbrand function. First, we parse the clausal abstraction proof into a tree structure, where the levels of the tree correspond to the quantifier blocks of the QBF. Since the clausal abstraction proof is a post-order linearization, we can build the proof tree bottom-up. For a quantifier block $Q\,X.\,\Psi$, a node $\langle \mathbf{t}, \mathbf{x}, r\rangle$ in the tree is a tuple consisting of a T assignment $\mathbf{t}$, an X assignment $\mathbf{x}$, and the subtree result $r \in \{\text{SAT}, \text{UNSAT}\}$. Next, we prune the tree according to the result: If the QBF is true, we dismiss universal levels as well as nodes that are labeled as UNSAT. Analogously, if the QBF is false, we dismiss existential levels as well as nodes with $r = \text{SAT}$. All remaining nodes are relevant for the certificate.

Given a quantifier block $Q\,X.\,\Psi$, we first collect the list of nodes $\mathcal{N}_X$ corresponding to this level (according to the navigation commands in the proof trace). For every variable $x \in X$ we then build the Skolem/Herbrand function f_x with the algorithm ConstructFunction, which takes three arguments: the variable x, the list of nodes $\mathcal{N}_X$, and the type of quantifier $Q \in \{\exists, \forall\}$.

```
1: procedure ConstructFunction($x$, $\mathcal{N}$, $Q$)
2:     $f_x \leftarrow$ false
3:     $f_{pre} \leftarrow$ true
4:     for $\langle \mathbf{t}, \mathbf{x}, r \rangle \in \mathcal{N}$ do
5:         if $x \in \mathbf{x}$ then
6:             $f_x \leftarrow f_x \vee (f_{pre} \wedge \text{Precondition}(\mathbf{t}, Q))$
7:         $f_{pre} \leftarrow f_{pre} \wedge \neg\text{Precondition}(\mathbf{t}, Q)$
8:     return $f_x$
```

```
1: procedure Precondition($\mathbf{t}$, $Q$)
2:     if $Q = \exists$ then
3:         return $\bigwedge_{t \in \mathbf{t}} \left( \bigvee_{l \in C_t \wedge var(l) \in \bigcup_{j<i} X_i} l \right)$
4:     else
5:         return $\bigwedge_{t \in T \setminus \mathbf{t}} \left( \bigwedge_{l \in C_t \wedge var(l) \in \bigcup_{j<i} X_i} \bar{l} \right)$
```

In the algorithm above the formula f_x characterizes when the Skolem/Herbrand function will set x to true and f_{pre} characterizes the cases that are not yet covered by f_x. In each iteration over the list of nodes we extend f_{pre} by the case that is covered by the current node (line 7). If the variable x occurs positively in the assignment, we also extend the function f_x (line 6). Each case is described by the conjunction over the clauses described by the T assignment restricted to the variables of smaller quantifier levels (see algorithm Precondition). For a given $t \in T$, we denote with C_t the clause that corresponds to the t variable. The formulas f_x computed by the algorithm ConstructFunction then define the output signals of the AIG.

In the example of Fig. 1, the list of nodes for x_1 consists of the single node: $\langle \emptyset, \{x_1\}, \text{SAT} \rangle$, indicating that without precondition ($\emptyset$) x_1 is set to true. For x_3 there are two nodes: $\langle \{t_3\}, \{\overline{x_3}\}, \text{SAT} \rangle$ and $\langle \{t_1, t_2\}, \{x_3\}, \text{SAT} \rangle$, indicating that if clause 3 is satisfied, x_3 is set to false, and if clauses 1 and 2 are satisfied, x_3 is set to true. The Skolem function computed by the algorithm above is $f_{x_3} = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$, which simplifies to $f_{x_3} = x_2$.

For a true (false) QBF, the resulting AIG certificates can be checked by substituting the existential (universal) variables in matrix by applications of their Skolem (Herbrand) functions and then query a SAT solver to ask for an assignment of the variables such that some clause is falsified (all clauses are satisfied). The certificate is valid if, and only if, the SAT solver returns UNSAT and the Skolem/Herbrand functions depend only on variables in their dependency set.

VI. Experimental Evaluation

We implemented the algorithm and its optimizations in a tool named CAQE[1] (Clausal Abstraction for Quantifier Elimination). The tool is written in the programming language C and we use a generic SAT solver interface that can be instantiated with PicoSAT [17] (default) or MiniSat [18]. In this section, we evaluate the implementation on the instances from QBF Gallery 2014 [19] in several categories: the number of solved instances per benchmark family with and without preprocessing, the number of instances solved in certification mode, and the size of the generated certificates. We compare CAQE to the (available) best performing solvers of the QBF Gallery 2014. For our experiments, we used a machine with a 3.6 GHz quad-core Intel Xeon processor and 32 GB of memory. The timeout was set to 10 minutes.

[1]available at http://react.uni-saarland.de/tools/caqe/

A. *Solved Instances per Family*

Table I shows for each solver how many instances of the QBFGallery benchmark set are solved within 10 minutes. We removed the preprocessing track of QBFGallery and instead ran every solver with and without preprocessing using Bloqqer [20]. For three of the seven families, a configuration of CAQE solved the highest number of instances. Overall, CAQE using PicoSAT ranked second after RAReQS when using the number of solved instances as a measure.

Most notably, CAQE solved an exceptionally high number of problems in the *hardness* family, which consists of bounded model checking queries for incomplete designs [6]. One characteristic of this family is that number of quantifier alternations is relatively high; going up to 60 alternations. Figure 2 shows the performance of all solvers for the 188 instances of the full benchmark set that have a high number of quantifier alternations. The plot suggests that CAQE performs well on instances with a high number of quantifier alternations—in particular compared to other CEGAR-based solvers.

Unsurprisingly, the choice of the underlying SAT solver has a significant impact on the performance of CAQE. In this setting, the variant using PicoSAT performs better, however, more testing and optimization was done for this variant.

The effect of preprocessing is unusual. While preprocessing by Bloqqer improved the performance overall (in particular for CAQE using MiniSat), the analysis per family reveals that Bloqqer decreased the performance of CAQE using PicoSAT for three benchmark families.

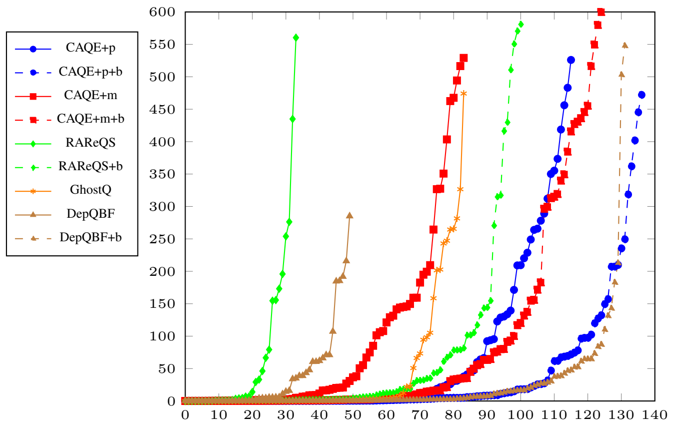


Fig. 2. Number of solved instances within 10 minutes among the 188 instances from QBFGallery 2014 with more than 6 quantifier alternations.

TABLE I
NUMBER OF INSTANCES OF THE QBFGALLERY 2014 BENCHMARK SET SOLVED WITHIN 10 MINUTES.

		CAQE				RAReQS		GhostQ	DepQBF	
Family	total	picosat	picosat+bloqqer	minisat	minisat+bloqqer	rareqs	rareqs+bloqqer	ghostq	depqbf	depqbf+bloqqer
eval2012r2	276	75	112	55	98	81	**129**	124	88	128
bomb	132	**91**	74	75	59	84	82	75	67	80
complexity	104	50	67	60	67	75	**91**	26	49	57
dungeon	107	46	31	22	**69**	57	62	45	44	66
hardness	114	78	**103**	58	94	15	68	57	8	81
planning	147	84	79	50	55	**146**	135	31	57	47
testing	131	54	77	25	84	36	92	**102**	57	76
all	1011	478	543	345	526	494	**659**	460	370	535

B. Certificates

Table II shows the number of instances of the QBF Gallery 2014 benchmark set that was solved within 10 minutes in certifying mode (#solved) and the number of certificates that was verified within 10 minutes (#verified). The evaluation also shows that CAQE can certify more results than DepQBF and provides certificates that are only half the size in average. The size of the certificates is measured in terms of AND-gates in the AIGER file that encodes the Skolem or Herbrand function after minimization with the DFRAIG algorithm of ABC [21].

Compared to the non-certifying run in Table I, the solvers solved between 84% (DepQBF) and 90% (CAQE) of the instances in certifying mode. This can be traced back to two factors. First, certain optimizations have to be disabled in certification mode, and second, there is a significant amount of time spent writing the proofs to disk. Furthermore, not all of the instances solved in certification mode could be verified within the given amount of verification time.

Lastly, we compare the size of the certificates of CAQE and DepQBF on the commonly solved instances in Fig. 3. The variance of the relative certificate sizes is high, but the number of instances where CAQE generated certificates of significantly smaller size than DepQBF is larger than the number of instances where DepQBF generated certificates of significantly smaller size than CAQE.

TABLE II
CERTIFYING RUNS OF DEPQBF AND CAQE.

Solver	# solved	# verified	# unique	avg. size
CAQE	428	340	146	3138
DepQBF	312	239	44	7447
virtual best	468	384	-	5357

VII. RELATED WORK

Various techniques have been proposed for QBF, including expansion [22], [23], BDDs [24], [25], (DPLL-like) search [14], [26], and CEGAR [9]. The CEGAR approach has been first explored in the context of model checking [27]. Janota and Silva successfully applied CEGAR to 2QBF [9].

RAReQS: Subsequently, Janota et al. extended the CEGAR approach to full QBF [8]. They implemented the approach in the tool RAReQS (and to a certain extend also in GhostQ), which lead to significant performance gains for several problem families. To evaluate the truth of a QBF $\varphi = Q_1 X_1 \ldots Q_n X_n . \varphi$ with n quantifier alternations, the algorithm picks an assignment $\mathbf{x}_1$ to X_1 and recursively determines the truth of $\varphi'(\mathbf{x}_1) = Q_2 X_2 \ldots Q_n X_n . \varphi(\mathbf{x}_1)$. If that call returns a counter-example, that is an assignment $\mathbf{x}_2$ to X_2, then φ is refined by the formula $\varphi'' = Q_1 X_1 . Q_3 X_3 \ldots Q_n X_n . \varphi(\mathbf{x}_2)$, which has two quantifier alternations fewer than φ. Before checking $\varphi'(\mathbf{x}_1')$ for other assignments $\mathbf{x}_1'$, it is first checked whether the assignment is already excluded by $\mathbf{x}_1'$. (That is, we first check $\varphi''(\mathbf{x}_1')$.) In this way, the assignment $\mathbf{x_1}$ cannot occur a second time as a counter-example. A potential problem with this approach is that the formula φ'' is itself a QBF, and may itself be refined with other QBFs in later iterations. We may therefore have to check each new assignment $\mathbf{x}_1'$ for a *tree* of counter-examples that each are QBFs, and the size of the tree of counter-examples may grow exponentially with the quantifier alternation depth. Our experiments suggest that this problem actually occurs in practice: while being very effective for low quantifier alternation depths, RAReQS solves only few instances with a higher number quantifier alternations.

In this work we propose an alternative CEGAR algorithm in which we refine only by single clauses. This notion of refinement coincides with the RAReQS refinement step in the case of 2QBF, but for two or more quantifier alternations it is *weaker*: Assignments that lead to a counter-example may reappear later. This explains why RAReQS outperforms CAQE on benchmarks with a low number of quantifier alternations. The weaker notion of refinement in CAQE, however, avoids the need for the tree of counter-examples and therefore scales well to instances with many quantifier alternations.

Clause selection: Very recently and independently from this work, Janota and Marques-Silva proposed *clause selection* and implemented the approach in the tool Qesto [28]. Similar to clausal abstractions, they reason about the satisfaction of sets of clauses and the algorithms have a similar structure. The encodings of the individual quantifier blocks, however, reveal interesting differences in the execution of this idea:

- Qesto uses equality constraints for the variables connecting the quantification levels while CAQE uses implications, requiring fewer clauses in the encoding.
- For universally quantified levels, CAQE needs to add only one variable per clause, while Qesto needs two.
- Qesto considers the clauses for existential and universal

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

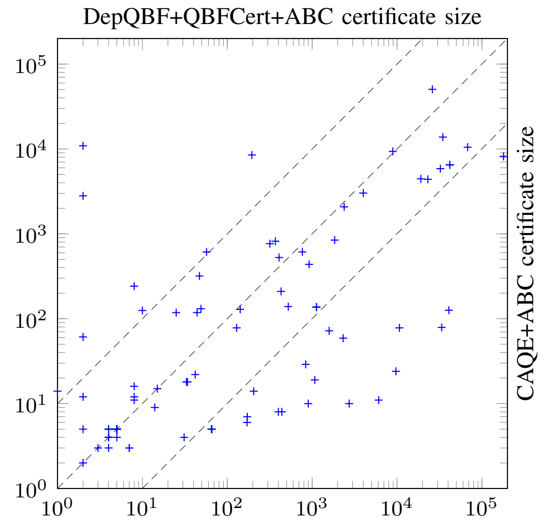


Fig. 3. The size of certificates computed by CAQE and DepQBF.

levels in a negated form, while CAQE does not negate existential levels.

It will be interesting to see whether CAQE and Qesto share the same runtime characteristics or whether these are particular to our encoding, and whether Qesto can be extended to certification as well.

Certifying QBF: Certifying QBF solvers enable a rich set of applications like encodings of bounded model checking [3]–[6] and synthesis that use the certificates as implementations [12]. Previous certifying QBF solvers were based on a DPLL-like search [14], [15] or expansion [29]. Our work shows how to enable certification for the CEGAR approach.

There has been work on certifying QBF preprocessing techniques based on QRAT proofs in Bloqqer [30], [31]. It may be possible to integrate CAQE in combination with Bloqqer in a similar setting as in [32].

VIII. Conclusions and Future work

We presented clausal abstractions, a decomposition of QBFs into sequences of propositional formulas, and a new CEGAR algorithm for QBF. The overall performance is competitive and our experiments suggest that the new algorithm is effective for instances with a high number of quantifier alternations. We showed how to certify the results of the algorithm and the evaluation shows that significantly more instances can be certified with our solver compared to the state-of-the-art.

In the future, we plan to consider joining the two notions of refinement used in RAReQS and CAQE and to integrate our algorithm in a certification framework like [32] to enable its use together with certified preprocessing.

References

[1] M. Benedetti and H. Mangassarian, “QBF-based formal verification: Experience and perspectives,” *JSAT*, vol. 5, no. 1-4, pp. 133–191, 2008.
[2] W. Zhang, “QBF encoding of temporal properties and QBF-based verification.” in *IJCAR*, 2014, pp. 224–239.
[3] T. Jussila and A. Biere, “Compressing BMC encodings with QBF.” *Electr. Notes Theor. Comput. Sci.*, pp. 45–56, 2007.
[4] B. Finkbeiner and L. Tentrup, “Fast DQBF refutation,” in *Proceedings of SAT*, 2014, pp. 243–251.
[5] N. Dershowitz, Z. Hanna, and J. Katz, “Bounded model checking with QBF,” in *Proceedings of SAT*, 2005, pp. 408–414.
[6] C. Miller, C. Scholl, and B. Becker, “Proving QBF-hardness in bounded model checking for incomplete designs,” in *Proceedings of MTV*, 2013, pp. 23–28.
[7] M. N. Rabe, C. M. Wintersteiger, H. Kugler, B. Yordanov, and Y. Hamadi, “Symbolic approximation of the bounded reachability probability in large Markov chains,” in *Proceedings of QEST*. Springer, 2014, pp. 388–403.
[8] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, “Solving QBF with counterexample guided refinement.” in *Proceedings of SAT*, 2012, pp. 114–128.
[9] M. Janota and J. P. M. Silva, “Abstraction-based algorithm for 2QBF,” in *Proceedings of SAT*, 2011, pp. 230–244.
[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Proceedings of TACAS*, 1999, pp. 193–207.
[11] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, “Programming by sketching for bit-streaming programs,” in *Proceedings of PLDI*, 2005, pp. 281–294.
[12] R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing, “SAT-based methods for circuit synthesis,” in *Proceedings of FMCAD*, 2014, pp. 31–34.
[13] B. Finkbeiner and L. Tentrup, “Detecting unrealizable specifications of distributed systems,” in *Proceedings of TACAS*, 2014, pp. 78–92.
[14] F. Lonsing and A. Biere, “DepQBF: A dependency-aware QBF solver,” *JSAT*, vol. 7, no. 2-3, pp. 71–76, 2010.
[15] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere, “Resolution-based certificate extraction for QBF - (tool presentation),” in *Proceedings of SAT*, 2012, pp. 430–435.
[16] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” *Studies in constructive mathematics and mathematical logic*, vol. 2, no. 115-125, pp. 10–13, 1968.
[17] A. Biere, “PicoSAT essentials,” *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
[18] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Proceedings of SAT*, 2003, pp. 502–518.
[19] “QBF Gallery - Joint Evaluation of Quantified Boolean Formulas,” 2014, http://qbf.satisfiability.org/gallery/.
[20] A. Biere, F. Lonsing, and M. Seidl, “Blocked clause elimination for QBF,” in *Proceedings of CADE-23*, 2011, pp. 101–115.
[21] R. K. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *Proceedings of CAV*, 2010, pp. 24–40.
[22] A. Biere, “Resolve and expand,” in *Proceedings of SAT*, 2004.
[23] F. Lonsing and A. Biere, “Nenofex: Expanding NNF for QBF solving,” in *Proceedings of SAT*, 2008, pp. 196–210.
[24] G. Audemard and L. Sais, “A symbolic search based approach for quantified boolean formulas,” in *Proceedings of SAT*, 2005, pp. 16–30.
[25] O. Olivo and E. A. Emerson, “A more efficient BDD-based QBF solver,” in *Proceedings of CP*, 2011, pp. 675–690.
[26] E. Giunchiglia, M. Narizzano, and A. Tacchella, “QuBE: A system for deciding quantified boolean formulas satisfiability.” in *Proceedings of IJCAR*, 2001, pp. 364–369.
[27] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
[28] M. Janota and J. Marques-Silva, “Solving QBF by clause selection,” in *Proceedings of IJCAI*. AAAI Press, 2015, pp. 325–331.
[29] A. Goultiaeva, A. Van Gelder, and F. Bacchus, “A uniform approach for generating proofs and strategies for both true and false QBF formulas,” in *Proceedings of IJCAI*, 2011, pp. 546–553.
[30] M. Heule, M. Seidl, and A. Biere, “A unified proof system for QBF preprocessing,” in *Proceedings of IJCAR*, ser. LNCS, vol. 8562. Springer, 2014, pp. 91–106.
[31] ——, “Efficient extraction of skolem functions from QRAT proofs,” in *Proc. of FMCAD*, 2014, pp. 107–114.
[32] M. Janota, R. Grigore, and J. Marques-Silva, “On QBF proofs and preprocessing,” in *Proceedings of LPAR*, 2013, pp. 473–489.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Difference Constraints: An adequate Abstraction for Complexity Analysis of Imperative Programs

Moritz Sinn, Florian Zuleger, Helmut Veith
TU Wien, Austria

Abstract—**Difference constraints have been used for termination analysis in the literature, where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of x in the current state is at most the value of y in the previous state plus some constant $c \in \mathbb{Z}$. In this paper, we argue that the complexity of imperative programs typically arises from counter increments and resets, which can be modeled naturally by difference constraints. We present the first practical algorithm for the analysis of difference constraint programs and describe how C programs can be abstracted to difference constraint programs. Our approach contributes to the field of automated complexity and (resource) bound analysis by enabling automated amortized complexity analysis for a new class of programs and providing a conceptually simple program model that relates invariant- and bound analysis. We demonstrate the effectiveness of our approach through a thorough experimental comparison on real world C code: our tool Loopus computes the complexity for considerably more functions in less time than related tools from the literature.**

I. Introduction

Automated program analysis for inferring program complexity and (resource) bounds is a very active area of research. Amongst others, approaches have been developed for analyzing functional programs [14], C# [13], C [5], [21], [16], Java [4] and Integer Transition Systems [4], [7], [10].

Difference constraints (*DCs*) have been introduced by Ben-Amram for termination analysis in [6], where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of x in the current state is at most the value of y in the previous state plus some constant $c \in \mathbb{Z}$. We call a program whose transitions are given by a set of difference constraints a *difference constraint program* (*DCP*).

In this paper, we advocate the use of *DCs* for program complexity and (resource) bounds analysis. Our key insight is that *DCs* provide a *natural abstraction* of the standard manipulations of counters in imperative programs: counter *increments/decrements* $x := x + c$ resp. *resets* $x := y$, can be modeled by the *DCs* $x' \leq x + c$ resp. $x' \leq y$ (see Section IV on program abstraction). In contrast, previous approaches to bound analysis can model either only resets [13], [5], [21], [4], [7], [10] or increments [16]. For this reason, we are able to design a more powerful analysis: In Section II-A we discuss that our approach achieves *amortized analysis* for a new class of programs. In Section II-B we describe how our approach performs *invariant analysis* by means of bound analysis.

Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED and ICT12-059.

In this paper, we establish the practical usefulness of *DCs* for bound (and complexity) analysis of imperative programs: 1) We propose the first algorithm for bound analysis of *DCPs*. Our algorithm is based on the dichotomy between increments and resets. 2) We develop appropriate techniques for abstracting C programs to *DCPs*: we describe how to extract *norms* (integer-valued expressions on the program state) from C programs and how to use them as variables in *DCPs*. We are not aware of any previous implementation of *DCPs* for termination or bound analysis. 3) We demonstrate the effectiveness of our approach through a thorough experimental evaluation. We present the first comparison of bound analysis tools on source code from real software projects (see Section V). Our implementation performs significantly better in time and success rate.

II. Motivation and Related Work

A. Amortized Complexity Analysis

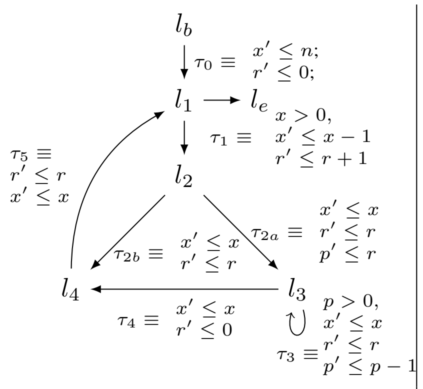
Example 1 stated in Figure 1 is representative for a class of loops that we found in parsing and string matching routines during our experiments. In these loops the inner loop iterates over disjoint partitions of an array or string, where the partition sizes are determined by the program logic of the outer loop. For an illustration of this iteration scheme, we refer the reader to Example 3 stated in the extended version [18], which contains a snippet of the source code after which we have modeled Example 1. Example 1 has the linear *complexity* $2n$, because the inner loop as well as the outer loop can be iterated at most n times (as argued in the next paragraph). However, previous approaches to bound analysis [13], [5], [21], [16], [4], [7], [10] are only able to deduce that the inner loop can be iterated at most a *quadratic* number of times (with loop bound n^2) by the following reasoning: (1) the outer loop can be iterated at most n times, (2) the inner loop can be iterated at most n times *within* one iteration of the outer loop (because the inner loop has a local loop bound p and $p \leq n$ is an invariant), (3) the loop bound n^2 is obtained from (1) and (2) by multiplication. We note that inferring the linear complexity $2n$ for Example 1, even though the inner loop can already be iterated n times *within* one iteration of the outer loop, is an instance of *amortized complexity analysis* [19].

In the following, we give an overview how our approach infers the linear complexity for Example 1:

1. Program Abstraction. We abstract the program to a *DCP* over $\mathbb{Z}$ as shown in Figure 1. We discuss our algorithm for abstracting imperative programs to *DCP*s based on symbolic

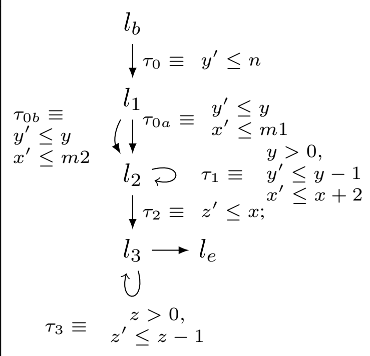
ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

```
     void foo(uint n) {
       int x = n;
       int r = 0;
l1     while(x > 0) {
         x = x - 1;
         r = r + 1;
l2       if(*) {
           int p = r;
l3         while(p > 0)
             p--;
           r = 0;
         }
l4     } }
```



l_b
$\tau_0 \equiv \begin{matrix} x' \le n; \\ r' \le 0; \end{matrix}$
l_1
l_e
$\tau_1 \equiv \begin{matrix} x > 0, \\ x' \le x - 1 \\ r' \le r + 1 \end{matrix}$
l_2
$\tau_5 \equiv \begin{matrix} r' \le r \\ x' \le x \end{matrix}$
$\tau_{2a} \equiv \begin{matrix} x' \le x \\ r' \le r \\ p' \le r \end{matrix}$
$\tau_{2b} \equiv \begin{matrix} x' \le x \\ r' \le r \end{matrix}$
l_4
l_3
$\tau_4 \equiv \begin{matrix} x' \le x \\ r' \le 0 \end{matrix}$
$\tau_3 \equiv \begin{matrix} p > 0, \\ x' \le x \\ r' \le r \\ p' \le p - 1 \end{matrix}$

```
     foo(uint n, uint m1,
         uint m2) {
       int y = n;
       int x;
l1     if(*)
         x = m1;
       else
         x = m2;
l2     while(y > 0) {
         y--;
         x = x + 2; }
       int z = x;
l3     while(z > 0)
         z--; }
```



l_b
$\tau_0 \equiv y' \le n$
l_1
$\tau_{0b} \equiv \begin{matrix} y' \le y \\ x' \le m2 \end{matrix}$
$\tau_{0a} \equiv \begin{matrix} y' \le y \\ x' \le m1 \end{matrix}$
l_2
$\tau_1 \equiv \begin{matrix} y > 0, \\ y' \le y - 1 \\ x' \le x + 2 \end{matrix}$
$\tau_2 \equiv z' \le x;$
l_3
l_e
$\tau_3 \equiv \begin{matrix} z > 0, \\ z' \le z - 1 \end{matrix}$

Complexity: $T\mathcal{B}(\tau_5) + T\mathcal{B}(\tau_3) = n + n = 2n$ | Complexity: $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_3) = \max(m_1, m_2) + 3n$

Example 1 | abstracted DCP of Example 1 | Example 2 | abstracted DCP of Example 2

Fig. 1. Running Examples, * denotes non-determinism (arising from conditions not modeled in the analysis)

execution in Section IV.

2. Finding Local Bounds. We identify p as a variable that limits the number of executions of transition τ_3: We have the guard $p > 0$ on τ_3 and p decreases on each execution of τ_3. We call p a *local bound* for τ_3. Accordingly we identify x as a *local bound* for transitions $\tau_1, \tau_{2a}, \tau_{2b}, \tau_4, \tau_5$.

3. Bound Analysis. Our algorithm (stated in Section III) computes *transition bounds*, i.e., (symbolic) upper bounds on the number of times program transitions can be executed, and *variable bounds*, i.e., (symbolic) upper bounds on variable values. For both types of bounds, the main idea of our algorithm is to reason *how much* and *how often* the value of the local bound resp. the variable value may increase during program run. Our algorithm is based on a mutual recursion between variable bound analysis ("how much", function $V\mathcal{B}(\mathtt{v})$) and transition bound analysis ("how often", function $T\mathcal{B}(\tau)$). Next, we give an intuition how our algorithm computes transition bounds: Our algorithm computes $T\mathcal{B}(\tau) = n$ for $\tau \in \{\tau_1, \tau_{2a}, \tau_{2b}, \tau_4, \tau_5\}$ because the local bound x is initially set to n and never increased or reset. Our algorithm computes $T\mathcal{B}(\tau_3)$ (τ_3 corresponds to the loop at l_3) as follows: τ_3 has local bound p; p is reset to r on τ_{2a}; our algorithm detects that before each execution of τ_{2a}, r is reset to 0 on either τ_0 or τ_4, which we call the *context* under which τ_{2a} is executed; our algorithm establishes that between being reset and flowing into p the value of r can be incremented up to $T\mathcal{B}(\tau_1)$ times by 1; our algorithm obtains $T\mathcal{B}(\tau_1) = n$ by a recursive call; finally, our algorithm calculates $T\mathcal{B}(\tau_3) = 0 + T\mathcal{B}(\tau_1) \times 1 = n$. We give an example for the mutual recursion between $T\mathcal{B}$ and $V\mathcal{B}$ in Section II-B.

We contrast our approach for computing the loop bound of l_3 of Example 1 with classical invariant analysis: Assume 'c' counting the number of inner loop iterations (i.e., c is initialized to 0 and incremented in the inner loop). For inferring $c <= n$ through invariant analysis the invariant $c + x + r <= n$ is needed for the outer loop, and the invariant $c + x + p <= n$ for the inner loop. Both relate 3 variables and cannot be expressed as (parametrized) octagons (e.g., [11]). Further, the expressions $c + x + r$ and $c + x + p$ do not appear in the program, which is challenging for template based approaches to invariant analysis.

B. Invariants and Bound Analysis

We explain on Example 2 in Figure 1 how our approach performs *invariant analysis* by means of bound analysis. We first motivate the importance of invariant analysis for bound analysis. It is easy to infer x as a bound for the possible number of iterations of the loop at l_3. However, in order to obtain a bound in the *function parameters* the difficulty lies in finding an invariant $x \le \mathtt{expr}(n, m_1, m_2)$. Here, the most precise invariant $x \le \max(m_1, m_2) + 2n$ cannot be computed by standard abstract domains such as *octagon* or *polyhedra*: these domains are *convex* and cannot express non-convex relations such as *maximum*. The most precise approximation of x in the polyhedra domain is $x \le m_1 + m_2 + 2n$. Unfortunately, it is well-known that the polyhedra abstract domain does not scale to larger programs and needs to rely on heuristics for termination. Next, we explain how our approach computes invariants using bound analysis and discuss how our reasoning is substantially different from invariant analysis by abstract interpretation.

Our algorithm computes a transition bound for the loop at l_3 by $T\mathcal{B}(\tau_3) = T\mathcal{B}(\tau_2) \times V\mathcal{B}(x) = 1 \times V\mathcal{B}(x) = V\mathcal{B}(x) = T\mathcal{B}(\tau_1) \times 2 + \max(m_1, m_2) = (n \times T\mathcal{B}(\tau_0)) \times 2 + \max(m_1, m_2) = (n \times 1) \times 2 + \max(m_1, m_2) = 2n + \max(m_1, m_2)$. We point out the mutual recursion between $T\mathcal{B}$ and $V\mathcal{B}$: $T\mathcal{B}(\tau_3)$ has called $V\mathcal{B}(x)$, which in turn called $T\mathcal{B}(\tau_1)$. We highlight that the variable bound $V\mathcal{B}(x)$ (corresponding to the invariant $x \le \max(m_1, m_2) + 2n$) has been established during the computation of $T\mathcal{B}(\tau_3)$.

Standard *abstract domains* such as *octagon* or *polyhedra* propagate information *forward* until a fixed point is reached, *greedily* computing all possible invariants expressible in the abstract domain at every location of the program. In contrast, $V\mathcal{B}(x)$ infers the invariant $x \le \max(m1, m2) + 2n$ by modular reasoning: *local information* about the program (i.e., increments/resets of variables, local bounds of transitions) is combined to a *global* program property. Moreover, our variable and transition bound analysis is *demand-driven*: our algorithm performs only those recursive calls that are indeed needed to derive the desired bound. We believe that our analysis complements existing techniques for invariant analysis and will find applications outside of bound analysis.

C. Related Work

In [6] it is shown that termination of $DCPs$ is undecidable in general but decidable for the natural syntactic subclass of *deterministic* $DCPs$ (see Definition 2), which is the class of $DCPs$ we use in this paper. It is an open question for future work whether there is a complete algorithm for bound analysis of deterministic $DCPs$.

In [16] a bound analysis based on constraints of the form $x' \leq x + c$ is proposed, where c is either an integer or a symbolic constant. The resulting abstract program model is strictly less powerful than $DCPs$. In [21] a bound analysis based on so-called *size-change constraints* $x' \lhd y$ is proposed, where $\lhd \in \{<, \leq\}$. Size-change constraints form a strict syntactic subclass of DCs. However, termination is decidable even for non-deterministic size-change programs and a complete algorithm for deciding the complexity of size-change programs has been developed [9]. Because the constraints in [21], [16] are less expressive than DCs, the resulting bound analyses cannot infer the linear complexity of Example 1 and need to rely on external techniques for invariant analysis.

In Section V we compare our implementation against the most recent approaches to automated complexity analysis [10], [7], [16]. [10] extends the COSTA approach by control flow refinement for cost equations and a better support for multi-dimensional ranking functions. The COSTA project (e.g. [4]) computes resource bounds by inferring an upper bound on the solutions of certain recurrence equations (so-called *cost equations*) relying on external techniques for invariant analysis (which are not explicitly discussed). The bound analysis in [7] uses approaches for computing polynomial ranking functions from the literature to derive bounds for SCCs in isolation and then expresses these bounds in terms of the function parameters using invariant analysis (see next paragraph).

The powerful idea of expressing locally computed loop bounds in terms of the function parameters by alternating between loop bound analysis and variable upper bound analysis has been explored in [7], [16] (as discussed in the extended version [17]) and [12]. We highlight some important differences to these earlier works. [7] computes upper bound invariants only for the *absolute* values of variables; this does, for example, not allow to distinguish between variable increments and decrements during the analysis. [17] and [12] do not give a general algorithm but deal with specific cases.

[20] discusses automatic parallelization of loop iterations; the approach builds on summarizing inner loops by multiplying the increment of a variable on a single iteration of a loop with the loop bound. The loop bounds in [20] are restricted to simple syntactic patterns.

The recent paper [8] discusses an interesting alternative for amortized complexity analysis of imperative programs: A system of linear inequalities is derived using Hoare-style proof-rules. Solutions to the system represent valid *linear* resource bounds. Interestingly, [8] is able to compute the linear bound for l_3 of Example 1 but fails to deduce the bound for the original source code (provided in the extended version [18]). Moreover, [8] is restricted to linear bounds, while our approach derives polynomial bounds (e.g., Example B in Figure 2) which may also involve the maximum operator. An experimental comparison was not possible as [8] was developed in parallel.

III. Program Model and Algorithm

In this section we present our algorithm for computing worst-case upper bounds on the number of executions of a given transition (transition bound) and on the value of a given variable (variable bound). We base our algorithm on the abstract program model of DCPs stated in Definition 2. In Section III-B we generalize DCPs and our algorithm to the non-well-founded domain $\mathbb{Z}$.

Definition 1 (Difference Constraints). *Let $\mathcal{V}$ be a finite set of variables and $\mathcal{C}$ be a finite set of symbolic constants. $\mathcal{A} = \mathcal{V} \cup \mathcal{C} \cup \mathbb{N}$ is the set of* atoms. *A* difference constraint *over $\mathcal{A}$ is an inequality of form $x' \leq y + c$ with $x \in \mathcal{V}$, $y \in \mathcal{A}$ and $c \in \mathbb{Z}$. $\mathcal{DC}(\mathcal{A})$ is the set of all difference constraints over $\mathcal{A}$.*

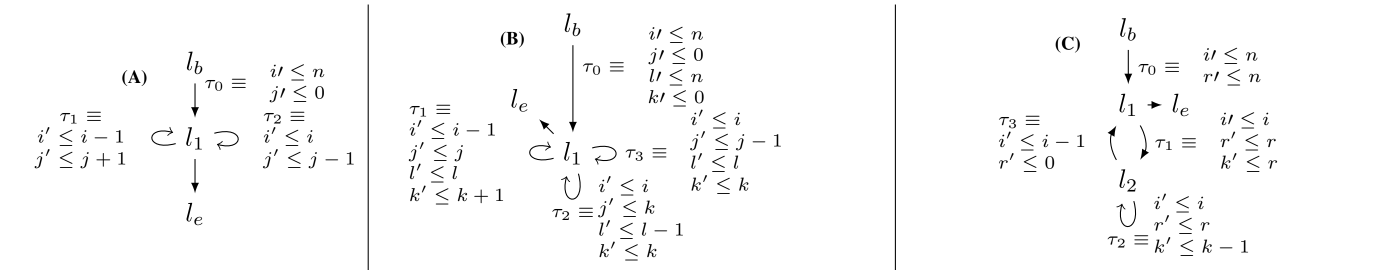
Definition 2 (Difference Constraint Program). *A* difference constraint program *(DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta\mathcal{P} = (L, T, l_b, l_e)$, where L is a finite set of* locations, *$l_b \in L$ is the entry location, $l_e \in L$ is the exit location and $T \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{u} l_2$ to denote a transition $(l_1, u, l_2) \in T$ labeled by a set of difference constraints $u \in 2^{\mathcal{DC}(\mathcal{A})}$. Given a transition $\tau = l_1 \xrightarrow{u} l_2 \in T$ of $\Delta\mathcal{P}$ we call l_1 the source location of τ and l_2 the target location of τ. A* path *of $\Delta\mathcal{P}$ is a sequence $l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \cdots$ with $l_i \xrightarrow{u_i} l_{i+1} \in T$ for all i. The set of* valuations *of $\mathcal{A}$ is the set $Val_\mathcal{A} = \mathcal{A} \to \mathbb{N}$ of mappings from $\mathcal{A}$ to the natural numbers with $\sigma(\mathtt{a}) = \mathtt{a}$ if $\mathtt{a} \in \mathbb{N}$. A* run *of $\Delta\mathcal{P}$ is a sequence $(l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ such that $l_b \xrightarrow{u_0} l_1 \xrightarrow{u_1} \cdots$ is a path of $\Delta\mathcal{P}$ and for all i it holds that (1) $\sigma_i \in Val_\mathcal{A}$, (2) $\sigma_{i+1}(x) \leq \sigma_i(y) + c$ for all $x' \leq y + c \in u_i$, (3) $\sigma_i(s) = \sigma_0(s)$ for all $s \in \mathcal{C}$. Given $\mathtt{v} \in \mathcal{V}$ and $l \in L$ we say that $\mathtt{v}$ is* defined *at l and write $\mathtt{v} \in \mathcal{D}(l)$ if $l \neq l_b$ and for all incoming transitions $l_1 \xrightarrow{u} l \in T$ of l it holds that there are $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u$.*

$\Delta\mathcal{P}$ is deterministic *(*fan-in-free *in the terminology of [6]), if for every transition $l_1 \xrightarrow{u} l_2 \in T$ and every $\mathtt{v} \in \mathcal{V}$ there is at most one $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u$.*

Our approach assumes the given DCP to be *deterministic*. We further assume that DCPs are *well-defined*: Let $\mathtt{v} \in \mathcal{V}$ and $l \in L$, if $\mathtt{v}$ is *live* at l then $\mathtt{v} \in \mathcal{D}(l)$. Our abstraction algorithm from Section IV generates only deterministic and well-defined DCPs.

In Definitions 3 to 10 we assume a DCP $\Delta\mathcal{P}(L, T, l_b, l_e)$ over $\mathcal{A}$ to be given.

Definition 3 (Transition Bound). *Let $\tau \in T$, τ is* bounded *iff τ appears a finite number of times on any run of $\Delta\mathcal{P}$. An expression $\mathtt{expr}$ over $\mathcal{C} \cup \mathbb{Z}$ is a* transition bound *for τ iff τ is* bounded *and for any* finite *run $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \ldots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ it holds that τ appears not more than $\sigma_0(\mathtt{expr})$ often on ρ. We say that a transition bound $\mathtt{expr}$ of τ is* precise *iff there is a run ρ of $\Delta\mathcal{P}$ s.t. τ appears $\sigma_0(\mathtt{expr})$ times on ρ.*


(A)
l_b
$\tau_0 \equiv$ $i' \le n$ $j' \le 0$
$\tau_1 \equiv$ $i' \le i - 1$ $j' \le j + 1$
l_1
$\tau_2 \equiv$ $i' \le i$ $j' \le j - 1$
l_e
(B)
l_b
$\tau_0 \equiv$ $i' \le n$ $j' \le 0$ $l' \le n$ $k' \le 0$
l_e
$\tau_1 \equiv$ $i' \le i - 1$ $j' \le j$ $l' \le l$ $k' \le k + 1$
l_1
$\tau_3 \equiv$ $i' \le i$ $j' \le j - 1$ $l' \le l$ $k' \le k$
$\tau_2 \equiv$ $i' \le i$ $j' \le k$ $l' \le l - 1$ $k' \le k$
(C)
l_b
$\tau_0 \equiv$ $i' \le n$ $r' \le n$
l_1 l_e
$\tau_3 \equiv$ $i' \le i - 1$ $r' \le 0$
$\tau_1 \equiv$ $i' \le i$ $r' \le r$ $k' \le r$
l_2
$\tau_2 \equiv$ $i' \le i$ $r' \le r$ $k' \le k - 1$

Complexity: $\mathcal{TB}(\tau_1) + \mathcal{TB}(\tau_2) = 2n$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_2 \mapsto j\}$
$\mathcal{TB}(\tau_1) = n$, $\mathcal{TB}(\tau_2) = n$

Complexity: $\mathcal{TB}(\tau_1) + \mathcal{TB}(\tau_2) + \mathcal{TB}(\tau_3) = 2n + n^2$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_2 \mapsto l, \tau_3 \mapsto j\}$
$\mathcal{TB}(\tau_1) = n$, $\mathcal{TB}(\tau_2) = n$, $\mathcal{TB}(\tau_3) = n^2$

Complexity: $\mathcal{TB}(\tau_2) + \mathcal{TB}(\tau_3) = 2n$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_3 \mapsto i, \tau_2 \mapsto k\}$
Def. 8: $\mathcal{TB}(\tau_1) = n$, $\mathcal{TB}(\tau_2) = n^2$, $\mathcal{TB}(\tau_3) = n$
Def. 10: $\mathcal{TB}(\tau_1) = n$, $\mathcal{TB}(\tau_2) = n$, $\mathcal{TB}(\tau_3) = n$

Fig. 2. Example DCP's (A), (B), (C)

We want to infer the complexity of the examples in Figure 2 (Examples A, B, C), i.e., we want to infer how often location l_1 can be visited during an execution of the program. We will do so by computing a bound on the number of times transitions τ_0, τ_1, τ_2 and τ_3 may be executed. In general, the complexity of a given program can be inferred by summing up the transition bounds for the back edges in the program.

Definition 4 (Counter Notation)**.** *Let $\tau \in T$ and $\mathtt{v} \in \mathcal{V}$. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots (l_e, \sigma_n)$ be a finite run of $\Delta\mathcal{P}$. By $\sharp(\tau, \rho)$ we denote the number of times that τ occurs on ρ. By $\downarrow(\mathtt{v}, \rho)$ we denote the number of times that the value of $\mathtt{v}$ decreases on ρ, i.e. $\downarrow(\mathtt{v}, \rho) = |\{i \mid \sigma_i(\mathtt{v}) > \sigma_{i+1}(\mathtt{v})\}|$.*

Definition 5 (Local Transition Bound)**.** *Let $\tau \in T$ and $\mathtt{v} \in \mathcal{V}$. $\mathtt{v}$ is a* local bound *for τ iff on all finite runs $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ it holds that $\sharp(\tau, \rho) \le \downarrow(\mathtt{v}, \rho)$. We call a* complete *mapping $\zeta : T \to \mathcal{V} \cup \{\mathbf{1}\}$ a* local bound mapping *for $\Delta\mathcal{P}$ if $\zeta(\tau)$ is a* local bound *of τ or $\zeta(\tau) = \mathbf{1}$ and τ can only appear at most once on any path of $\Delta\mathcal{P}$.*

Example A: i is a local bound for τ_1, j is a local bound for τ_2. *Example C:* i is a local bound for τ_1 and for τ_3.

A variable $\mathtt{v}$ is a *local transition bound* if on any run of $\Delta\mathcal{P}$ we can traverse τ not more often than the number of times the value of $\mathtt{v}$ decreases. I.e., a local bound $\mathtt{v}$ limits the potential number of executions of τ as long as the value of $\mathtt{v}$ does not increase. In our analysis, *local transition bounds* play the role of *potential functions* in classical *amortized complexity analysis* [19]. Our bound algorithm is based on a mapping which assigns each transition a local bound. We discuss how we find local bounds in Section III-C.

Definition 6 (Variable Bound)**.** *An expression* $\mathtt{expr}$ *over $\mathcal{C} \cup \mathbb{Z}$ is a* variable bound *for $\mathtt{v} \in \mathcal{V}$ iff for any finite run $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \ldots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ and all $1 \le i \le n$ with $\mathtt{v} \in \mathcal{D}(l_i)$ it holds that $\sigma_i(\mathtt{v}) \le \sigma_0(\mathtt{expr})$.*

Let $\mathtt{v} \in \mathcal{V}$. Our algorithm is based on a *syntactic* distinction between transitions which *increment* $\mathtt{v}$ or *reset* $\mathtt{v}$.

Definition 7 (Resets and Increments)**.** *Let $\mathtt{v} \in \mathcal{V}$. We define the* resets $\mathcal{R}(\mathtt{v})$ *and* increments $\mathcal{I}(\mathtt{v})$ *of* $\mathtt{v}$ *as follows:*

$$\begin{aligned}\mathcal{R}(\mathtt{v}) &= \{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in T \times \mathcal{A} \times \mathbb{Z} \mid \\ &\qquad \mathtt{v}' \le \mathtt{a} + \mathtt{c} \in u, \mathtt{a} \not\equiv \mathtt{v}\} \\ \mathcal{I}(\mathtt{v}) &= \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in T \times \mathbb{Z} \mid \mathtt{v}' \le \mathtt{v} + \mathtt{c} \in u, \mathtt{c} > 0\}\end{aligned}$$

Given a path π of $\Delta\mathcal{P}$ we say that $\mathtt{v}$ is reset *on π if there is a transition τ on π such that $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ for some $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$.*

Example B: $\mathcal{I}(k) = \{(\tau_1, 1)\}$ and $\mathcal{R}(k) = \{(\tau_0, n, 0)\}$.
I.e., we have $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ if variable $\mathtt{v}$ is reset to a value $\le \mathtt{a} + \mathtt{c}$ when executing the transition τ. Accordingly we have $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ if variable $\mathtt{v}$ is incremented by a value $\le \mathtt{c}$ when executing the transition τ.
Our algorithm in Definition 8 is build on a *mutual recursion* between the two functions $\mathcal{VB}(\mathtt{v})$ and $\mathcal{TB}(\tau)$, where $\mathcal{VB}(\mathtt{v})$ infers a *variable bound* for $\mathtt{v}$ and $\mathcal{TB}(\tau)$ infers a *transition bound* for the transition τ.

Definition 8 (Bound Algorithm)**.** *Let $\zeta : T \to \mathcal{V} \cup \{\mathbf{1}\}$ be a* local bound mapping *for $\Delta\mathcal{P}$. We define $\mathcal{VB} : \mathcal{A} \mapsto Expr(\mathcal{A})$ and $\mathcal{TB} : T \mapsto Expr(\mathcal{A})$ as:*
$\mathcal{VB}(\mathtt{a}) = \mathtt{a}$, if $\mathtt{a} \in \mathcal{A} \setminus \mathcal{V}$, else

$$\mathcal{VB}(\mathtt{v}) = \mathtt{Incr}(\mathtt{v}) + \max_{(\_, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} (\mathcal{VB}(\mathtt{a}) + \mathtt{c})$$

$\mathcal{TB}(\tau) = \mathbf{1}$, if $\zeta(\tau) = \mathbf{1}$, else

$$\begin{aligned}\mathcal{TB}(\tau) = \; & \mathtt{Incr}(\zeta(\tau)) \\ & + \sum_{(\mathtt{t}, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} \mathcal{TB}(\mathtt{t}) \times \max(\mathcal{VB}(\mathtt{a}) + \mathtt{c}, 0)\end{aligned}$$

where

$$\mathtt{Incr}(\mathtt{v}) = \sum_{(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})} \mathcal{TB}(\tau) \times \mathtt{c} \;\; (\mathtt{Incr}(\mathtt{v}) = 0 \textit{ for } \mathcal{I}(\mathtt{v}) = \emptyset)$$

Discussion: We first explain the subroutine $\mathtt{Incr}(\mathtt{v})$: With $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ we have that a single execution of τ *increments* the value of $\mathtt{v}$ by not more than $\mathtt{c}$. $\mathtt{Incr}(\mathtt{v})$ multiplies the transition bound of τ with the increment $\mathtt{c}$ for summarizing the total amount by which $\mathtt{v}$ may be incremented over all executions of τ. $\mathtt{Incr}(\mathtt{v})$ thus computes a bound on the total amount by which the value of $\mathtt{v}$ may be *incremented* during a program run.
The function $\mathcal{VB}(\mathtt{v})$ computes a variable bound for $\mathtt{v}$: After executing a reset transition $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$, the value of $\mathtt{v}$ is bounded by $\mathcal{VB}(\mathtt{a}) + \mathtt{c}$. As long as $\mathtt{v}$ is not *reset*, its value cannot increase by more than $\mathtt{Incr}(\mathtt{v})$.
The function $\mathcal{TB}(\tau)$ computes a transition bound for τ based on the following reasoning: (1) The total amount by which

the local bound $\zeta(\tau)$ of transition τ can be *incremented* is bounded by $\texttt{Incr}(\zeta(\tau))$. (2) We consider a reset $(\texttt{t}, \texttt{a}, \texttt{c}) \in \mathcal{R}(\zeta(\tau))$; in the worst case, a single execution of $\texttt{t}$ resets the local bound $\zeta(\texttt{t})$ to $V\mathcal{B}(\texttt{a}) + \texttt{c}$, adding $\max(V\mathcal{B}(\texttt{a}) + \texttt{c}, 0)$ to the potential number of executions of $\texttt{t}$; in total all $T\mathcal{B}(\texttt{t})$ possible executions of $\texttt{t}$ add up to $T\mathcal{B}(\texttt{t}) \times \max(V\mathcal{B}(\texttt{a}) + \texttt{c}, 0)$ to the potential number of executions of $\texttt{t}$.

Example A, ζ as defined in Figure 2: j is *reset* to 0 on τ_0 and incremented by 1 on τ_1. i is reset to n on τ_0. Our algorithm computes $T\mathcal{B}(\tau_2) = T\mathcal{B}(\tau_1) \times 1 + T\mathcal{B}(\tau_0) \times 0 = T\mathcal{B}(\tau_1) = T\mathcal{B}(\tau_0) \times n = n$. Thus the overall complexity of Example A is inferred by $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_2) = 2n$.

Example B, ζ as defined in Figure 2: i and l are *reset* to n on τ_0. Our algorithm computes $T\mathcal{B}(\tau_1) = T\mathcal{B}(\tau_0) \times n = n$ and $T\mathcal{B}(\tau_2) = T\mathcal{B}(\tau_0) \times n = n$. j is *reset* to 0 on τ_0 and *reset* to k on τ_2. Our algorithm computes $T\mathcal{B}(\tau_3) = T\mathcal{B}(\tau_0) \times 0 + T\mathcal{B}(\tau_2) \times V\mathcal{B}(k)$. Since k is *reset* to 0 on τ_0 and incremented by 1 on τ_1, our algorithm computes $V\mathcal{B}(k) = T\mathcal{B}(\tau_1) \times 1 = n \times 1 = n$. Thus $T\mathcal{B}(\tau_3) = T\mathcal{B}(\tau_2) \times V\mathcal{B}(k) = n \times n = n^2$. Thus the overall complexity of Example B is inferred by $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_2) + T\mathcal{B}(\tau_3) = n + n + n^2 = 2n + n^2$.

Example 2 (Figure 1): $\zeta = \{\tau_0, \tau_{0_a}, \tau_{0_b}, \tau_2 \mapsto 1, \tau_1 \mapsto y, \tau_3 \mapsto z\}$, $\mathcal{R}(z) = \{(\tau_2, x, 0)\}$, $\mathcal{I}(x) = \{(\tau_1, 2)\}$, $\mathcal{R}(x) = \{(\tau_{0a}, m1, 0), (\tau_{0b}, m2, 0)\}$, $\mathcal{R}(y) = \{(\tau_0, n, 0)\}$. We have stated the computation of $T\mathcal{B}(\tau_3)$ in Section II-B.

Termination: Our algorithm does not terminate if recursive calls cycle, i.e., if a call to $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\texttt{v})$ (indirectly) leads to a recursive call to $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\texttt{v})$. This can be easily detected, we return the value $\perp$ (undefined).

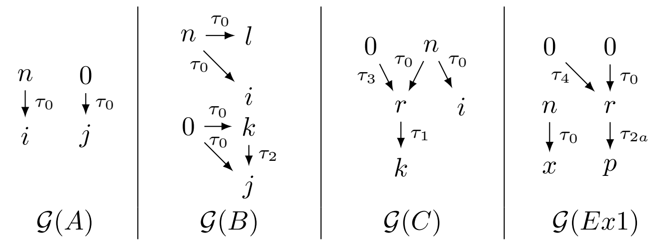
Theorem 1 (Soundness). *Let $\Delta\mathcal{P}(L, T, l_b, l_e)$ be a well-defined and deterministic DCP over atoms $\mathcal{A}$, $\zeta : T \mapsto \mathcal{V} \cup \{1\}$ be a* local bound mapping *for $\Delta\mathcal{P}$, $\texttt{v} \in \mathcal{V}$ and $\tau \in T$. Either $T\mathcal{B}(\tau) = \perp$ or $T\mathcal{B}(\tau)$ is a* transition bound *for τ. Either $V\mathcal{B}(\texttt{v}) = \perp$ or $V\mathcal{B}(\texttt{v})$ is a* variable bound *for* $\texttt{v}$.

A. Context-Sensitive Bound Analysis

So far our algorithm reasons about resets occurring on single transitions. In this section we increase the precision of our analysis by exploiting the context under which resets are executed through a refined notion of resets and increments.

Definition 9 (Reset Graph). *The* Reset Graph *for $\Delta\mathcal{P}$ is the graph $\mathcal{G}(\mathcal{A}, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{A} \times T \times \mathbb{Z} \times \mathcal{V}$ s.t. $\mathcal{E} = \{(x, \tau, \texttt{c}, y) \mid (\tau, y, \texttt{c}) \in \mathcal{R}(x)\}$. We call a* finite *path* $\kappa = \texttt{a}_n \xrightarrow{\tau_n, \texttt{c}_n} \texttt{a}_{n-1} \xrightarrow{\tau_{n-1}, \texttt{c}_{n-1}} \ldots \texttt{a}_0$ *in $\mathcal{G}$ with $n > 0$ a* reset path *of $\Delta\mathcal{P}$. We define $in(\kappa) = \texttt{a}_n$, $c(\kappa) = \sum_{i=1}^{n} \texttt{c}_i$, $trn(\kappa) = \{\tau_n, \tau_{n-1} \ldots, \tau_1\}$, and $atm(\kappa) = \{a_n, a_{n-1} \ldots, a_0\}$. κ is* sound *if for all $1 \leq i < n$ it holds that $\texttt{a}_i$ is* reset *on all paths from the target location of τ_1 to the source location of τ_i in $\Delta\mathcal{P}$. κ is* optimal *if κ is sound and there is no sound reset path $\hat{\kappa}$ s.t. κ is a suffix of $\hat{\kappa}$, i.e., $\hat{\kappa} = \texttt{a}_{n+k} \xrightarrow{\tau_{n+k}, \texttt{c}_{n+k}} \texttt{a}_{n+k-1} \xrightarrow{\tau_{n+k-1}, \texttt{c}_{n+k-1}} \ldots \texttt{a}_n \xrightarrow{\tau_n, \texttt{c}_n} \texttt{a}_{n-1} \xrightarrow{\tau_{n-1}, \texttt{c}_{n-1}} \ldots \texttt{a}_0$ with $k \geq 1$. Let $\texttt{v} \in \mathcal{V}$, by $\mathfrak{R}(\texttt{v})$ we denote the set of optimal reset paths ending in* $\texttt{v}$.

We explain the notions *sound* and *optimal* in the course of the following discussion. Figure 3 shows the reset graphs of Examples A, B, C and Example 1 from Figure 1. For a given reset $(\tau, \texttt{a}, \texttt{c}) \in \mathcal{R}(\texttt{v})$, the reset graph determines which atom flows into variable $\texttt{v}$ under which context. For example, consider $\mathcal{G}(C)$: When executing the reset $(\tau_1, r, 0) \in \mathcal{R}(k)$ under the context τ_3, k is set to 0, if the same reset is executed under the context τ_0, k is set to n. Note that the reset graph does not represent *increments* of variables. We discuss how we handle increments below.



$$\begin{tikzcd} n \arrow[d, "\tau_0"] & 0 \arrow[d, "\tau_0"] \\ i & j \end{tikzcd}$$

$\mathcal{G}(A)$

$$\begin{tikzcd} n \arrow[r, "\tau_0"] \arrow[dr, "\tau_0"'] & l \\ & i \\ 0 \arrow[r, "\tau_0"] \arrow[dr, "\tau_0"'] & k \arrow[d, "\tau_2"] \\ & j \end{tikzcd}$$

$\mathcal{G}(B)$

$$\begin{tikzcd} 0 \arrow[dr, "\tau_3"'] & & n \arrow[dl, "\tau_0"'] \arrow[dr, "\tau_0"] & \\ & r \arrow[d, "\tau_1"] & & i \\ & k & & \end{tikzcd}$$

$\mathcal{G}(C)$

$$\begin{tikzcd} 0 \arrow[dr, "\tau_4"'] & 0 \arrow[d, "\tau_0"] \\ n \arrow[d, "\tau_0"] & r \arrow[d, "\tau_{2a}"] \\ x & p \end{tikzcd}$$

$\mathcal{G}(Ex1)$

Fig. 3. Reset Graphs, increments by 0 are not depicted

We assume that the reset graph is a DAG. We can always force the reset graph to be a DAG by abstracting the *DCP*: we remove all program variables which have cycles in the reset graph and all variables whose values depend on these variables. Note that if the reset graph is a DAG, the set $\mathfrak{R}(\texttt{v})$ is finite for all $\texttt{v} \in \mathcal{V}$.

Let $\texttt{v} \in \mathcal{V}$. Given a reset path κ of length k that ends in $\texttt{v}$, we say that $(trn(\kappa), in(\kappa), c(\kappa))$ is a reset of $\texttt{v}$ with context of length $k - 1$. I.e., $\mathcal{R}(\texttt{v})$ from Definition 7 is the set of *context-free* resets of $\texttt{v}$ (context of length 0), because $(trn(\kappa), in(\kappa), c(\kappa)) \in \mathcal{R}(\texttt{v})$ iff κ ends in $\texttt{v}$ and has length 1. Our algorithm from Definition 8 reasons *context free* since it uses only *context-free* resets.

Consider Example C. The precise bound for τ_2 is n because we can iterate τ_2 only in the first iteration of the loop at l_1 since r is reset to 0 on τ_3. But when reasoning context-free, our algorithm infers a *quadratic* bound for τ_2: We assume ζ to be given as stated in Figure 2. In $\mathcal{G}(C)$ $\kappa = r \xrightarrow{\tau_1, 0} k$ is the only reset path of length 1 ending in k. Thus $\mathcal{R}(k) = \{(\tau_1, r, 0)\}$. Our algorithm from Definition 8 computes: $T\mathcal{B}(\tau_1) = T\mathcal{B}(\tau_0) \times n = n$, $V\mathcal{B}(r) = T\mathcal{B}(\tau_0) \times n + T\mathcal{B}(\tau_3) \times 0 = n$, $T\mathcal{B}(\tau_2) = T\mathcal{B}(\tau_1) \times V\mathcal{B}(r) = n \times n = n^2$.

We show how our algorithm infers the *linear* bound for τ_2 when using *resets with context*: If we consider κ with contexts, we get $\kappa_1 = 0 \xrightarrow{\tau_3, 0} r \xrightarrow{\tau_1, 0} k$ and $\kappa_2 = n \xrightarrow{\tau_0, 0} r \xrightarrow{\tau_1, 0} k$. Note that κ_1 and κ_2 are *sound* by Definition 9 because r is reset on all paths from the target location l_2 of τ_1 to the source location l_1 of τ_1 in Example C (namely on τ_3). Thus $\mathfrak{R}(k) = \{(\{\tau_3, \tau_1\}, 0, 0), (\{\tau_0, \tau_1\}, n, 0)\}$. We can compute a bound on the number of times that a sequence $\tau_1, \tau_2, \ldots \tau_n$ of transitions may occur on a run by computing $\min\limits_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$. Thus, basing our analysis on $\mathfrak{R}(k)$ rather than $\mathcal{R}(k)$ we compute: $T\mathcal{B}(\tau_2) = \min(T\mathcal{B}(\tau_3), T\mathcal{B}(\tau_1)) \times 0 + \min(T\mathcal{B}(\tau_0), T\mathcal{B}(\tau_1)) \times n = \min(n, 1) \times n = n$.

We have demonstrated that our analysis gains precision when adding context to our notion of resets. It is, however, not sound to base the analysis on maximal reset paths (i.e., resets with maximal context) only: Consider Example B with ζ as stated in Figure 2. There are 2 maximal reset paths ending in j (see

$\mathcal{G}(B)$): $\kappa_1 = 0 \xrightarrow{\tau_0,0} j$ and $\kappa_2 = 0 \xrightarrow{\tau_0,0} k \xrightarrow{\tau_2,0} j$. Thus $\mathfrak{R}(j)' = \{(\{\tau_0,\tau_2\},0,0),(\{\tau_0\},0,0)\}$ is the set of resets of j with *maximal* context. Using $\mathfrak{R}(j)'$ rather than $\mathcal{R}(j)$ our algorithm computes: $\mathcal{TB}(\tau_3) = \min(\mathcal{TB}(\tau_0), \mathcal{TB}(\tau_2)) \times 0 + \mathcal{TB}(\tau_0) \times 0 + \mathcal{TB}(\tau_1) \times 1 = \mathcal{TB}(\tau_1) \times 1 = n$, but n is not a transition bound for τ_3. The reasoning is unsound because κ_2 is *unsound* by Definition 9: k is *not* reset on all paths from the target location l_1 of τ_2 to the source location l_1 of τ_2 in Example B: e.g., the path $\tau_2 = l_1 \xrightarrow{u_2} l_1$ of Example B does not reset k.

We base our *context sensitive* algorithm on the set $\mathfrak{R}(\mathtt{v})$ of *optimal* reset paths. The optimal reset paths are those that are maximal within the *sound* reset paths (Definition 9).

Definition 10 (Bound Algorithm with Context)**.** *Let $\zeta : T \to \mathcal{V} \cup \{\mathbf{1}\}$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $\mathcal{VB} : \mathcal{A} \mapsto Expr(\mathcal{A})$ be as defined in Definition 8. We override the definition of $\mathcal{TB} : T \mapsto Expr(\mathcal{A})$ in Definition 8 by stating:*

$$\mathcal{TB}(\tau) = \mathbf{1} \textit{ if } \zeta(\tau) = \mathbf{1} \textit{ else}$$
$$\mathcal{TB}(\tau) = \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} \mathcal{TB}(trn(\kappa)) \times \max(\mathcal{VB}(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$$

where

$$\mathcal{TB}(\{\tau_1,\tau_2,\dots,\tau_n\}) = \min_{1 \le i \le n} \mathcal{TB}(\tau_i)$$

Discussion and Example: The main difference to the definition of $\mathcal{TB}(\tau)$ in Definition 8 is that the term $\mathtt{Incr}(\zeta(\tau))$ is replaced by the term $\sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$. Consider the abstracted *DCP* of Example 1 in Figure 1. We have discussed in Section II-A that r may be incremented on τ_1 between the reset of r to 0 on τ_0 resp. τ_4 and the reset of p to r on τ_{2a}. The term $\sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$ takes care of such increments which may increase the value that finally flows into $\zeta(\tau)$ (in the example p) when the last transition on κ (in the example τ_{2a}) is executed: We use the local bound mapping $\zeta = \{\tau_0 \mapsto 1, \tau_1 \mapsto x, \tau_{2a} \mapsto x, \tau_{2b} \mapsto x, \tau_4 \mapsto x, \tau_5 \mapsto x, \tau_3 \mapsto p\}$ for Example 1. The reset graph of Example 1 is shown in Figure 3. We have $\mathfrak{R}(p) = \{0 \xrightarrow{\tau_0} r \xrightarrow{\tau_{2a}} p, 0 \xrightarrow{\tau_4} r \xrightarrow{\tau_{2a}} p\}$. Thus our algorithm computes $\mathcal{TB}(\tau_3) = \sum_{\kappa \in \mathfrak{R}(p)} \mathcal{TB}(trn(\kappa)) \times \max(\mathcal{VB}(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a}) = \mathcal{TB}(\{\tau_0,\tau_{2a}\}) \times \max(\mathcal{VB}(0),0) + \mathtt{Incr}(r) + \mathcal{TB}(\{\tau_4,\tau_{2a}\}) \times \max(\mathcal{VB}(0),0) + \mathtt{Incr}(r) = 2 \times \mathtt{Incr}(r) = 2 \times \mathcal{TB}(\tau_1) \times 1 = 2 \times n$ (with $\mathcal{TB}(\tau_1) = n$).

Complexity: In theory there can be exponentially many resets in $\mathfrak{R}(\mathtt{v})$. In our experiments this never occurred, enumeration of (optimal) reset paths did not affect performance.

Further Optimization: We have shown in Section II that transitions τ_3 of Example 1 has a *linear* bound, precisely n. The Bound $2n$ that is computed by our bound algorithm from Definition 10 is *linear* but not precise. We compute $2n$ because r appears on both reset paths of p and therefore $\mathtt{Incr}(r) = n$ is added twice. However, there is only one transition (τ_{2a}) on which p is reset to r and between any two executions of τ_{2a} r will be reset to 0. For this reason each increment of r can only contribute once to the increase of the local bound p of τ_3, and not twice. We thus suggest to further optimize our algorithm from Definition 10 by distinguishing if there is more than one way how $\mathtt{a} \in atm(\kappa)$ may flow into the target variable of κ or not. We divide $atm(\kappa)$ into two disjoint sets $atm_2(\kappa) = \{\mathtt{a} \in atm(\kappa) \mid$ more than 1 path from $\mathtt{a}$ to target variable of κ in $\mathcal{G}(\Delta\mathcal{P})\}$, $atm_1(\kappa) = atm(\kappa) \setminus atm_2(\kappa)$. We define

$$\mathcal{TB}(\tau) = \Big(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)} \mathtt{Incr}(\mathtt{a})\Big) + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} \mathcal{TB}(trn(\kappa)) \times \max(\mathcal{VB}(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a} \in atm_2(\kappa)} \mathtt{Incr}(\mathtt{a})$$

for $\zeta(\tau) \neq 1$. Note that for Example 1 $atm_1(\kappa) = \{r\}$ and $atm_2(\kappa) = \emptyset$ for both $\kappa \in \mathfrak{R}(p)$. Therefore $\mathcal{TB}(\tau_3) = \mathcal{I}(r) = n$ with the optimization.

Theorem 2 (Soundness of Bound Algorithm with Context)**.** *Let $\Delta\mathcal{P}(L,T,l_b,l_e)$ be a well-defined and deterministic DCP over atoms $\mathcal{A}$, $\zeta : T \mapsto \mathcal{V} \cup \{1\}$ be a* local bound mapping *for $\Delta\mathcal{P}$, $\mathtt{v} \in \mathcal{V}$ and $\tau \in T$. Let $\mathcal{TB}(\tau)$ and $\mathcal{VB}(\mathtt{a})$ be defined as in Definition 10. Either $\mathcal{TB}(\tau) = \bot$ or $\mathcal{TB}(\tau)$ is a* transition bound *for τ. Either $\mathcal{VB}(\mathtt{v}) = \bot$ or $\mathcal{VB}(\mathtt{v})$ is a* variable bound *for* $\mathtt{v}$.

B. *DCPs over non-well-founded domains*

In real world code, many data types are not well-founded. The abstraction of a concrete program is much simpler and more information is kept if the abstract program model is not limited to a well-founded domain. Below we extend our program model from Definition 2 to the non-well-founded domain $\mathbb{Z}$ by adding guards to the transitions in the program. Interestingly our bound algorithm from Definition 8 resp. Definition 10 remains sound for the extended program model, if we adjust our notion of a *local transition bound* (Definition 11).

We extend the range of the *valuations* $Val_{\mathcal{A}}$ of $\mathcal{A}$ from $\mathbb{N}$ to $\mathbb{Z}$ and allow constants to be integers, i.e., we define $\mathcal{A} = \mathcal{V} \cup \mathcal{C} \cup \mathbb{Z}$. We extend Definition 2 as follows: The transitions T of a *guarded DCP* $\Delta\mathcal{P}(L,T,l_b,l_e)$ are a subset of $L \times 2^{\mathcal{V}} \times 2^{\mathcal{DC}(\mathcal{A})} \times L$. A sequence $(l_b,\sigma_0) \xrightarrow{g_0,u_0} (l_1,\sigma_1) \xrightarrow{g_1,u_1} \cdots$ is a *run* of $\Delta\mathcal{P}$ if it meets the conditions required in Definition 2 and additionally $\sigma_i(x) > 0$ holds for all $x \in g_i$. For examples see Figure 1.

Definition 11 (Local Transition Bound for *DCP*s with guards)**.** *Let $\Delta\mathcal{P}(L,T,l_b,l_e)$ be a DCP with guards over $\mathcal{A}$. Let $\tau \in T$ and $\mathtt{v} \in \mathcal{V}$. $\mathtt{v}$ is a* local bound *for τ if for all finite runs $\rho = (l_b,\sigma_0) \xrightarrow{\tau_0} (l_1,\sigma_1) \xrightarrow{\tau_1} \cdots (l_e,\sigma_n)$ of $\Delta\mathcal{P}$ it holds that $\sharp(\tau,\rho) \le \downarrow(\max(\mathtt{v},0),\rho)$.*

The algorithms in Sections III-C and IV are based on the extended program model over $\mathbb{Z}$, it is straightforward to adjust them for *DCP*s without guards.

C. *Determining Local Bounds*

We call a path of a *DCP* $\Delta\mathcal{P}(L,T,l_b,l_e)$ *simple and cyclic* if it has the same start- and end-location and does not visit a location twice except for the start- and end-location. Given a transition $\tau \in T$ we assign it $\mathsf{v} \in \mathcal{V}$ as local bound if for all simple and cyclic paths $\pi = l_1 \xrightarrow{g_1,u_1} l_2 \xrightarrow{g_2,u_2} ...l_n$ $(l_n = l_1)$ of $\Delta\mathcal{P}$ that traverse τ it holds that (1) $\exists 0 < i < n$ s.t. $\mathsf{v} \in g_i$ and (2) $\exists 0 < i < n$ s.t. $\mathsf{v}' \leq \mathsf{v} + \mathsf{c} \in u_i$ for some $\mathsf{c} < 0$. Our implementation avoids an explicit enumeration of the simple and cyclic paths of $\Delta\mathcal{P}$ by a simple data flow analysis.

IV. Program Abstraction

In this section we present our concrete program model and discuss how we abstract a given program to a *DCP*.

Definition 12 (Program). *Let* Σ *be a set of* states. *The set of* transition relations $\Gamma = 2^{\Sigma \times \Sigma}$ *is the set of relations over* Σ. *A* program *is a directed labeled graph* $\mathcal{P} = (L, E, l_b, l_e)$, *where* L *is a finite set of* locations, $l_b \in L$ *is the entry location,* $l_e \in L$ *is the exit location and* $E \subseteq L \times \Gamma \times L$ *is a finite set of* transitions. *We write* $l_1 \xrightarrow{\rho} l_2$ *to denote a transition* (l_1, ρ, l_2). *A* norm $e \in \Sigma \rightarrow \mathbb{Z}$ *is a function that maps the states to the integers.*

Programs are labeled transition systems over some set of states, where each transition is labeled by a transition relation that describes how the state changes along the transition. Note, that a *DCP* (Definition 2) is a program by Definition 12.

Definition 13 (Transition Invariants). *Let* $e_1, e_2, e_3 \in \Sigma \rightarrow \mathbb{Z}$ *be norms, and let* $c \in \mathbb{Z}$ *be some integer. We say* $e_1' \leq e_2 + e_3$ *is* invariant for $l_1 \xrightarrow{\rho} l_2$, *if* $e_1(s_2) \leq e_2(s_1) + e_3(s_1)$ *holds for all* $(s_1, s_2) \in \rho$. *We say* $e_1 > 0$ *is* invariant for $l_1 \xrightarrow{\rho} l_2$, *if* $e_1(s_1) > 0$ *holds for all* $(s_1, s_2) \in \rho$.

Definition 14 (Abstraction of a Program). *Let* $\mathcal{P} = (L, E, l_b, l_e)$ *be a program and let* N *be a finite set of norms. A DCP* $\Delta\mathcal{P} = (L, E', l_b, l_e)$ *with atoms* N *is an abstraction of the program* $\mathcal{P}$ *iff for each transition* $l_1 \xrightarrow{\rho} l_2 \in E$ *there is a transition* $l_1 \xrightarrow{u,g} l_2 \in E'$ *s.t. every* $e_1' \leq e_2 + c \in u$ *is invariant for* $l_1 \xrightarrow{\rho} l_2$ *and for every* $e_1 \in g$ *it holds that* $e_1 > 0$ *is invariant for* $l_1 \xrightarrow{\rho} l_2$.

We propose to abstract a program $\mathcal{P} = (L, E, l_b, l_e)$ to a *DCP* $\Delta\mathcal{P} = (L, E', l_b, l_e)$ as follows: Let N be some initial set of norms.
1) For each transition $l_1 \xrightarrow{\rho} l_2 \in E$ we generate a set of difference constraints $\alpha(\rho)$: Initially we set $\alpha(\rho) = \emptyset$ for all transitions $l_1 \xrightarrow{\rho} l_2$. We then repeat the following construction until the set of norms N becomes stable: For each $e_1 \in N$ and $l_1 \xrightarrow{\rho} l_2 \in E$ we check whether there is a difference constraint of form $e_1' \leq e_2 + \mathsf{c}$ for e_1 in $\alpha(\rho)$. If not, we try to find a norm e_2 (possibly not yet in N) and a constant $\mathsf{c} \in \mathbb{Z}$ s.t. $e_1' \leq e_2 + \mathsf{c}$ is invariant for ρ. If we find appropriate e_2 and c, we add $e_1' \leq e_2 + \mathsf{c}$ to $\alpha(\rho)$ and e_2 to N. I.e., our transition abstraction algorithm performs a fixed point computation which might not terminate if new terms keep being added (see discussion in next section).
2) For each transition $l_1 \xrightarrow{\rho} l_2$ we generate a set of guards $G(\rho)$: Initially we set $G(\rho) = \emptyset$ for all transitions $l_1 \xrightarrow{\rho} l_2$. For each $e \in N$ and each transition $l_1 \xrightarrow{\rho} l_2$ we check if $e > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$. If so, we add e to $G(\rho)$.
3) We set $E' = \{l_1 \xrightarrow{G(\rho),\alpha(\rho)} l_2 \mid l_1 \xrightarrow{\rho} l_2 \in E\}$.

In the following we discuss how we implement the above sketched abstraction algorithm.

A. Implementation

0. Guessing the initial set of Norms.: We aim at creating a suitable abstract program for bound analysis. In our non-recursive setting, complexity evolves from iterating loops. Therefore we search for expressions which limit the number of loop iterations. For this purpose we consider conditions of form $a > b$ resp. $a \geq b$ found in loop headers or on loop-paths if they involve loop counter variables, i.e., variables which are incremented and/or decremented inside the loop. Such conditions are likely to limit the consecutive execution of single or multiple loop-paths. From each such condition we form the integer expression $b - a$ and add it to our initial set of norms. Note that on those transitions on which $a > b$ holds, $b - a > 0$ must hold.
1. Abstracting Transitions.: For a given norm $e \in N$ and a transition $l_1 \xrightarrow{\rho} l_2$ we derive a transition predicate $e' \leq e_2 + \mathsf{c} \in \alpha(\rho)$ as follows: We symbolically execute ρ for deriving e' from e. In order to keep the number of norms low, we first try
i) to find a norm $e_2 \in N$ s.t. $e' \leq e_2 + e_3$ is invariant for ρ where e_3 is some integer valued expression. If $e_3 = \mathsf{c}$ for some integer $\mathsf{c} \in \mathbb{Z}$ we derive the transition predicate $e' \leq e_2 + \mathsf{c}$. Else we use our bound algorithm (Section III) for over-approximating e_3 by a constant expression $k \geq e_3$ and infer the transition predicate $e' \leq e_2 + k$ where we consider k to be a symbolic constant.
ii) If i) fails, we form a norm e_4 s.t. $e' \leq e_4 + \mathsf{c}$ by separating constant parts in the expression e' using associativity and commutativity of the addition operator. E.g., given $e' = \mathsf{v} + 5$ we set $e_4 = \mathsf{v}$ and $\mathsf{c} = 5$. We add e_4 to N and derive the predicate $e' \leq e_4 + \mathsf{c}$.
Since case ii) triggers a recursive abstraction for the newly added norm we have to ensure the termination of our abstraction procedure: Note that we can always stop the abstraction process at any point, getting a sound abstraction of the original program. We therefore enforce termination of the abstraction algorithm by limiting the chain of recursive abstraction steps triggered by entering case ii) above: In case this limit is exceeded we remove all norms from the abstract program which form part of the limit exceeding chain of recursive abstraction steps. This also ensures well-definedness of the resulting abstract program.
Further note that the *DCP*s generated by our algorithm are always *deterministic*: For each transition, we get at most one predicate $e' \leq e_2 + \mathsf{c}$ for each $e \in N$.
2. Inferring Guards: Given a transition $l_1 \xrightarrow{\rho} l_2$ and a norm e, we use an SMT solver to check whether $e > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$. If so, we add e to $G(\rho)$.
Non-linear Iterations.: We handle counter updates such as $x' = 2x$ or $x' = x/2$ as discussed in [16].

V. Experiments

Implementation: We have implemented the presented algorithm into our tool Loopus [1]. Loopus reads in the LLVM [15]

	Succ.	1	n	n^2	n^3	$n^{>3}$	2^n	Time	TO
Loopus'15	806	205	489	97	13	2	0	15m	6
Loopus'14	431	200	188	43	0	0	0	40m	20
KoAT	430	253	138	35	2	0	2	5,6h	161
CoFloCo	386	200	148	38	0	0	0	4.7h	217

Fig. 4. Tool Results on analyzing the complexity of 1659 functions in the cBench benchmark, none of the tools infers *log* bounds.

intermediate representation and performs an intra-procedural analysis. It is capable of computing bounds for loops as well as analyzing the complexity of non-recursive functions.

Experimental Setup: For our experimental comparison we used the program and compiler optimization benchmark *Collective Benchmark* [2] (cBench), which contains a total of 1027 different C files (after removing code duplicates) with 211.892 lines of code. In contrast to our earlier work we did not perform a loop bound analysis but a complexity analysis on function level. We set up the first comparison of complexity analysis tools on real world code. For comparing our new tool (Loopus'15) we chose the 3 most promising tools from recent publications: the tool KoAT implementing the approach of [7], the tool CoFloCo implementing [10] and our own earlier implementation (Loopus'14) [16]. Note that we compared against the most recent versions of KoAT and CoFloCo (download 01/23/15).[1] The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor and 16 GB memory. We used the following experimental set up:

1) We compiled all 1027 C files in the benchmark into the llvm intermediate representation using clang.

2) We extracted all 1751 functions which contain at least one loop using the tool llvm-extract (comes with the llvm tool suite). Extracting the functions to single files guarantees an intra-procedural setting for all tools.

3) We used the tool llvm2kittel [3] to translate the 1751 llvm modules into 1751 text files in the Integer Transition System (ITS) format read in by KoAT.

4) We used the transformation described in [10] to translate the ITS format of KoAT into the ITS format of CoFloCo. This last step is necessary because there exists no direct way of translating C or the llvm intermediate representation into the CoFloCo input format.

5) We decided to exclude the 91 recursive functions in the set because we were not able to run CoFloCo on these examples (the transformation tool does not support recursion), KoAT was not successful on any of them and Loopus does not support recursion.

In total our example set thus comprises 1659 functions.

Evaluation: Table 4 shows the results of the 4 tools on our benchmark using a time out of 60 seconds. The first column shows the number of functions which were successfully bounded by the respective tool, the last column shows the number of time outs, on the remaining examples (not shown in the table) the respective tool did not time out but was also not able compute a bound. The column *Time* shows the total time used by the tool to process the benchmark. Loopus'15 computes the complexity for about twice as many functions as KoAT, CoFloCo and Loopus'14 while needing an order of magnitude less time than KoAT and CoFloCo and significantly less time than Loopus'14. We conclude that our approach is both scalable and more successful than existing approaches.

Pointer and Shape Analysis: Even Loopus'15, computed bounds for only about half of the functions in the benchmark. Studying the benchmark code we concluded that for many functions pointer alias and/or shape analysis is needed for inferring functional complexity. In our experimental comparison such information was not available to the tools. Using optimistic (but unsound) assumptions on pointer aliasing and heap layout, our tool Loopus'15 was able to compute the complexity for in total 1185 out of the 1659 functions in the benchmark (using 28 minutes total time).

Amortized Complexity: During our experiments, we found 15 examples with an amortized complexity that could only be inferred by the approach presented in this paper. These examples and further experimental results can be found on [1] where our new tool is offered for download.

[1]https://github.com/s-falke/kittel-koat, https://github.com/aeflores/CoFloCo

References

[1] http://forsyte.at/software/loopus/.
[2] http://ctuning.org/wiki/index.php/CTools:CBench.
[3] https://github.com/s-falke/llvm2kittel.
[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
[5] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133, 2010.
[6] A. M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.
[7] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, page to appear, 2014.
[8] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. PLDI, 2015.
[9] T. Colcombet, L. Daviaud, and F. Zuleger. Size-change abstraction and max-plus automata. In *MFCS*, pages 208–219, 2014.
[10] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS*, pages 275–295, 2014.
[11] T. M. Gawlitza, M. D. Schwarz, and H. Seidl. Parametric strategy iteration. *arXiv preprint arXiv:1406.5457*, 2014.
[12] S. Gulwani and S. Juvekar. Bound analysis using backward symbolic execution. Technical Report MSR-TR-2004-95, Microsoft Research, 2009.
[13] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.
[14] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
[15] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
[16] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, pages 745–761. Springer, 2014.
[17] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. *CoRR*, abs/1401.5842, 2014.
[18] M. Sinn, F. Zuleger, and H. Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. *CoRR*, abs/1508.04958, 2015.
[19] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, Apr. 1985.
[20] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Languages and Compilers for Parallel Computing*, pages 427–441. Springer, 2003.
[21] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Simulation Graphs for Reverse Engineering

Mathias Soeken[1,2,4] Baruch Sterin[3] Rolf Drechsler[2,4] Robert Brayton[3]
[1] Faculty of Engineering, University of Freiburg, Freiburg, Germany
[2] Faculty of Mathematics and Computer Science, University of Bremen, Germany
[3] Electrical Engineering and Computer Sciences, UC Berkeley, CA, USA
[4] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{msoeken,drechsle}@cs.uni-bremen.de {sterin,brayton}@berkeley.edu

Abstract—**Reverse engineering is the extraction of word level information from a gate-level netlist. It has applications in formal verification, hardware trust, information recovery, and general technology mapping. A preprocessing step finds blocks in a circuit in which word level components are expected. A second step searches for word level components in these blocks. For this second step, we propose two variants of equivalence checking that consider subfunction containment. We propose algorithms to solve these variants by using subgraph isomorphism. A *simulation graph* (SG) is constructed for the block and for each library component, using a set of permutation-invariant simulation vectors for that component. If a library component SG is a subgraph of the block SG, we have a candidate match, which is then checked by standard equivalence checking. We extend a state-of-the-art subgraph isomorphism algorithm, LAD, to handle simulation graphs efficiently and also propose a SAT-based formulation. Experimental evaluations show that our algorithms can efficiently find 32-bit arithmetic components in blocks with over 300 primary inputs.**

I. Introduction

The problem of reverse engineering (RE) is, given a gate-level netlist, find a word level netlist description which has the same behavior. It can be seen as a generalized technology mapping problem in the following sense. An instance would be to start with a gate-level circuit and a list of word level components, called a library, which contains components such as adders, multipliers, shifters, and other word level components of various bit widths. The goal is to find occurrences of these components in the circuit. While there are many objective functions to be optimized in regular technology mapping, e.g., area, delay, power, clause count in a conjunctive normal form (CNF), and wire count, in RE the single objective is to find all the word level components in the circuit.

RE is of interest for a number of reasons:

1) A register transfer level description (RTL) may not be available; the design may be a legacy without an initial RTL; it may be that an AIG is being passed between various tools with no accompanying RTL, it may have come from bit blasting an RTL and synthesizing it at the bit level.
2) To create a word level description to enable and improve formal verification or synthesis.
3) To understand the structure of an unknown chip.
4) To analyze a chip to help isolate Trojan hardware.

In general, RE is very difficult, but there are many circumstances where the problem is made simpler. The goal of our research is to create set of algorithms that are as efficient as possible and to extend the range of situations where RE is feasible.

RE methodologies typically consist of two main parts: (i) a structural method that decomposes the gate-level circuit into blocks and (ii) a functional method to match sub-circuits of a block with components from a library, assuming that a block's inputs and outputs contain the inputs and outputs of any component to be found.

In this work, we propose an algorithm that targets the second part. Given a block in a circuit and a library, our algorithm finds sub-circuits, called candidates, in the block that have the same simulation behavior, modulo a selected set of simulation vectors, as the component. For this purpose, *simulation graphs* (SGs) of the blocks are created and subgraph isomorphism is applied to match a SG of a component in the library. Once such a candidate has been found it is checked if it is indeed a component using standard combinational equivalence checking. Thus our entire matching algorithm is functional and not structural.

The contributions of this paper are as follows:

1) The problem of finding components in a block of logic is reduced to the subgraph isomorphism problem.
2) An efficient LAD-based subgraph isomorphism algorithm is developed to find matches of component candidates to subgraphs of the block. The requirement that the component outputs be exactly on the block output boundary is relaxed.
3) A symbolic SAT-based subgraph isomorphism algorithm is discussed that is capable of detecting candidates in the presence of inverters at the inputs and outputs in the block.
4) An open source framework is provided with implementations of all algorithms to reproduce the experimental evaluations.

The paper is organized as follows. Relevant definitions and related literature are provided in Sects. II and III, respectively. Sect. IV provides two problem formulations that generalize equivalence checking and discusses how they arise in practice. Sect. V discusses solving such problems by a reduction to subgraph isomorphism of simulation graphs and Sects. VI and VII present implementations based on LAD and SAT, respectively. Sect. VIII discusses an extension that relaxes the assumption that the component's outputs must be contained in the block's outputs. Sect. IX presents experimental results. Finally Sect. X concludes, outlines many future research directions.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

II. Preliminaries

General notation and graphs: The notation $[n]$ is a shorthand for the set $\{1,\ldots,n\}$. A *digraph* $G=(V,A)$ consists of a set of vertices V and arcs $A \subseteq V \times V$. Each vertex $v \in V$ has an *in-degree* $d^{-}(v) = \#\{w \in V \mid (w,v) \in A\}$ and *out-degree* $d^{+}(v) = \#\{w \in V \mid (v,w) \in A\}$. A digraph is called *k-partite* if V can be partitioned into disjoint vertex sets $V_1,\ldots,V_k$ such that there exists no arc $(v,w) \in A$ and no $i \in [k]$ such that $v,w \in V_i$. A k-partite digraph is called *k-layered* if $A \subseteq \bigcup_{i=1}^{k-1} V_i \times V_{i+1}$, i.e., arcs only connect adjacent vertex sets.

A *vertex-labeled* graph also has a set of labels L and a labeling function $l : V \to L$. A *labeled subgraph isomorphism* from $G' = (V', A')$ with labeling function l' to $G=(V,A)$ with labeling function l is an injective function $\mu : V' \hookrightarrow V$ such that $(u,v) \in A' \Leftrightarrow (\mu(u),\mu(v)) \in A$ and $l(u) = l'(\mu(u))$ for all $u \in V'$.

Functions: An n-input m-output Boolean function is an m-tuple of Boolean functions over the variables $x_1,\ldots,x_n$. A Boolean combinational circuit is associated with the function it computes. We will refer to a circuit and the function it computes interchangeably.

Definition 1 (Embedding): An n'-input m'-output Boolean function f' is *embedded* in an n-input m-output Boolean function f if (*1*) there is an injective input-matching function $\pi : [n'] \hookrightarrow [n]$, (*2*) there is an injective output-matching function $\sigma : [m'] \hookrightarrow [m]$ such that $f_{\sigma(j)}(x_1,\ldots,x_n) = f'_j(x_{\pi(1)},\ldots,x_{\pi(n')})$, for all $j \in [m']$ and $x_1 \ldots, x_n$.

Definition 2 (Simulation vector): A *simulation vector* for a circuit with n inputs is a bitstring of size n. If $(s_1,\ldots,s_n)$ is a simulation vector, then $(f_1(s_1,\ldots,s_n),\ldots,f_m(s_1,\ldots,s_n))$ is the *output of a simulation vector*. The simulation vector is called *k-hot* encoded if exactly k bits are set to one and it is called *k-cold* if exactly k bits are set to zero. Let $S^{\text{hot}}_{n,k}$ and $S^{\text{cold}}_{n,k}$ be the sets of all k-hot and all k-cold simulation vectors, respectively. Note that $\#S^{\text{hot}}_{n,k} = \#S^{\text{cold}}_{n,k} = \binom{n}{k}$ and $S^{\text{hot}}_{n,k} = S^{\text{cold}}_{n,n-k}$.

We also extend the input-matching function from Definition 1 to simulation vectors by padding with zeros where π does not map any inputs of f', i.e., mapping k-hot vectors of length n' to k-hot vectors of length n.

Definition 3: Let $\pi : [n'] \hookrightarrow [n]$ be an input-matching function as above, and let s' be a simulation vector of f'. We define $\pi(s')$ by

$$\pi(s')_j := \begin{cases} s'_i & j = \pi(i), \quad \text{for some } i \\ 0 & \text{(otherwise).} \end{cases}$$

III. Related Work

According to [1] the two main steps to solve a reverse engineering problem are: (1) block identification and (2) matching blocks against components in a library. A component is assumed to be inside a block with its inputs and outputs being contained in the primary inputs and primary outputs of the block. Blocks may contain multiple components and additional logic.

First preliminary approaches have been presented for the first step in [2], [3], however, this step is still considered an open problem with no satisfactory solution proposed so far. One way to circumvent this problem is to have a variety of block matching algorithms for the second step, thereby relaxing the constraints on the blocks. Different approaches for Step 2 make different assumptions on the blocks that are identified by Step 1. Since inputs and outputs of the component are assumed to be primary inputs and primary outputs of the block, all approaches are generalizations of equivalence checking.

In [4], [5], [6], [7], it is assumed that the block neither contains additional logic nor additional inputs and outputs, however, the order of inputs and outputs is unknown. More formally, given two n-input m-output Boolean functions f' and f, the *permutation-independent equivalence checking* (PIEC) problem asks whether f' is embedded (as in Definition 1) in f.

In [8] primary inputs of the block are partitioned into control bits $c_1,\ldots,c_k$, and data bits $x_1,\ldots,x_n$. The order of inputs and outputs is unknown, as well as the values of the control bits that will cause the same functional behavior as that of the component. More formally, given two functions $f(c_1,\ldots,c_k,x_1,\ldots,x_n) = (f_1,\ldots,f_m)$ and $f'(x_1,\ldots,x_n) = (f'_1,\ldots,f'_m)$, the *permutation-independent conditional equivalence checking* (PICEC) problem asks whether there exist two permutations $\pi \in S_n$ and $\sigma \in S_m$ and a propositional function $\psi : [k] \to \mathbb{B}$ such that for all $x_1,\ldots,x_n$ and all $j \in [m]$

$$f_j(\psi(1),..,\psi(k),x_1,\ldots,x_n) = f'_{\sigma(j)}(x_{\pi(1)},..,x_{\pi(n)}) \quad (1)$$

IV. Problem Formulation and Motivation

1) Problem formulation: We address two problems in this work, called the *subset permutation-independent equivalence checking* (SPIEC) and the *subset negation-and-permutation-independent equivalence checking* (SNPIEC). SPIEC asks whether a smaller function is embedded into a larger one when no input and output correspondence is known. SNPIEC extends the problem and allows inputs and outputs of the larger function to be negated.

The input to the SPIEC problem is an n'-input m'-output Boolean function f' and an n-input m-output Boolean function f. The problem asks whether f' is embedded in f.

The input to the SNPIEC problem is an n'-input m'-output Boolean function f' and an n-input m-output Boolean function f. The problem asks whether there exist two propositional functions

$$p : [n] \to \mathbb{B} \quad \text{and} \quad q : [m] \to \mathbb{B}$$

such that f' is embedded in the function h defined by

$$h_j(x_1,\ldots,x_n) = q(j) \oplus f(p(1) \oplus x_1, \ldots, p(n) \oplus x_n),$$

i.e., the function h is f whose i-th input is inverted if $p(i) = 1$ and j-th output is inverted if $q(j) = 1$. The SNPIEC problem detects a subfunction in a block in the presence of misplaced inverters at the primary inputs and outputs of the block.

2) Motivation: We exploit in our algorithms the fact that many components of interest can have a uniquely characteristic input/output behavior even for a small set of simulation vectors.

As an example, let $f : \mathbb{B}^{2n} \to \mathbb{B}^{n+1}$ be the function of an n-bit adder for which unknown permutations have been

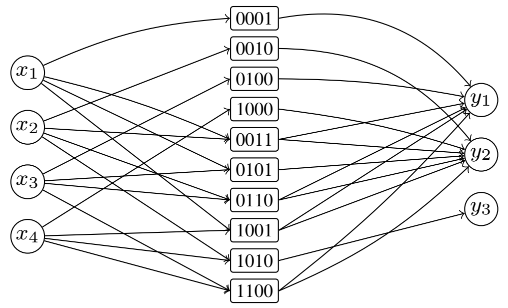


Fig. 1. A Simulation graph of a 2-bit adder. Simulation nodes are annotated with the simulation vector associated with it for convenience.

applied to the inputs and outputs. All one-hot and all two-hot simulation vectors are sufficient to find these permutations using the following arguments. From addition with 0, we know there are one-hot simulation vectors s and s', $s \neq s'$, with $s_i = 1$ and $s'_j = 1$, such that $f(s) = f(s') = y$ is also one-hot with $y_k = 1$. Therefore, i, j, and k refer to the same bit position p in the operands and the output of the adder. To determine the next position $p+1$, the two-hot simulation vector $s \,|\, s'$ ('$|$' refers to bitwise OR) yields a one-hot output value $y' = f(s \,|\, s')$ with $y'_l = 1$. Therefore, $l = p + 1$.

Note that since binary addition is bit-wise commutative, i.e., a_i and b_i for any i can be swapped without changing the result $a + b$; we cannot uniquely determine the partition of input bits into a left-hand operand and a right-hand operand. For other components, such as a multiplier, a different set of simulation vectors is needed. Additionally each component would have its own sequence of deductions for determining the permutations. Therefore, we seek a method that does not depend on the type of component to be found.

V. Reduction to Subgraph Isomorphism

This section describes how to find components in blocks by reducing it to a labeled subgraph isomorphism problem. The key to our approach is the concept of a simulation graph (SG), which captures the behavior of a circuit on a set of simulation vectors.

Definition 4 (Simulation Graph): Let f be an n-input m-output Boolean function and let $S = \{s^{(1)}, \dots, s^{(t)}\}$ be a set of simulation vectors for f. The *simulation graph* G_f^S is a 3-layered vertex-labeled directed graph $G = (X \cup S \cup Y, A_1 \cup A_2)$ where *(1)* $X = \{x_1, \dots, x_n\}$, *(2)* $Y = \{f_1, \dots, f_m\}$, *(3)* $(x_i, s^{(j)}) \in A_1 \Leftrightarrow s_i^{(j)} = 1$, and *(4)* $(s^{(j)}, f_r) \in A_2 \Leftrightarrow f_r(s^{(j)}) = 1$. Labeling is defined by *(1)* $L = \{\mathbf{PI}, \mathbf{PO}\} \cup \mathbb{N}$, *(2)* $l(x_i) = \mathbf{PI}$, *(3)* $l(f_r) = \mathbf{PO}$, and *(4)* $l(s^{(j)}) =$ number of 1s in $s^{(j)}$.

In other words, in an SG, a simulation vector is connected to the inputs that are 1 on it, and to the outputs where it produces a 1. The labels denote the types of node in the graph, inputs, outputs, and the number of ones in a simulation vector.

Example 1: Fig. 1 shows an SG for a 2-bit adder. It has four vertices for the inputs and three vertices for the outputs. Simulation vectors used are all the one-hot and two-hot vectors, resulting in 10 vertices for the simulation vectors. Nodes in the figure are annotated for readability and are not the labels used in the algorithm.

For the remainder of this section, f is an n-input m-output Boolean function, f' is an n'-input and m'-output Boolean function, $K \subseteq \{0, \dots, n'\}$, $S = \bigcup_{k \in K} S^{\mathrm{h}}_{n,k}$ and $S' = \bigcup_{k \in K} S^{\mathrm{h}}_{n',k}$. Let $G_{f'}^{S'}$ denote the SG for f' using S' and let G_f^S denote the SG for f using S. We chose these sets of simulation vectors because of the following property.

Proposition 1: For all $K \subseteq \{0, \dots, n\}$ the set $\bigcup_{k \in K} S^{\mathrm{h}}_{n,k}$ is closed under permutations of the order of the inputs.

Theorem 1: If f' is embedded in f, then there exists a labeled subgraph isomorphism from $G_{f'}^{S'}$ to G_f^S.

Proof: We can use the input and output matching functions of the embedding π and σ to construct a mapping μ from $G_{f'}^{S'}$ to G_f^S:

1) $\mu(x'_i) := x_{\pi(i)}$,
2) $\mu(f'_r) := f_{\sigma(r)}$,
3) $\mu(s'^{(j)}) := \pi(s'^{(j)})$.

Note that $s'^{(j)}$ is a k-hot vector if and only if $\pi(s'^{(j)})$ is k-hot. Since S is closed under permutation, $\pi(s'^{(j)})$ is a simulation vector node in G_f^S. This mapping preserves the labeling because it maps inputs to inputs, outputs to outputs, and k-hot simulation vectors to k-hot simulation vectors. To show that μ is a subgraph isomorphism observe that

1) $(x'_i, s'^{(j)}) \in A'_1 \Leftrightarrow (s'^{(j)})_i = 1 \Leftrightarrow \mu(s'^{(j)})_{\pi(i)} = 1 \Leftrightarrow (\mu(x'_i), \mu(s'^{(j)})) \in A_1$, and
2) $(s'^{(j)}, f'_r) \in A'_2 \Leftrightarrow f'_r(s'^{(j)}) = 1 \Leftrightarrow f_{\sigma(r)}(\mu(s'^{(j)})) = 1 \Leftrightarrow (\mu(s'^{(j)}), \mu(f'_r)) \in A_2$ ■

Because a simulation graph is constructed using only a small subset of all possible simulation vectors, the converse does not hold. Instead we can just state the obvious result that if there is a labeled subgraph isomorphism from $G_{f'}^{S'}$ to G_f^S, then f has a subset of inputs and outputs, that behaves like f' on the set of vectors used to construct the simulation graphs.

The existence of a subgraph isomorphism as stated in Theorem 1 depends on the set of simulation vectors used being closed under permutation. Consequently, a single simulation vector can only be supported by the proposed approach if *all* its permutations are considered.

Finding candidate components: Theorem 1 supports an algorithm for finding candidate components in the block. Using the same types of k-hot simulation vectors, we construct a simulation graph for the block, called the *target graph*, and a simulation graph for the component, called the *pattern graph*. If there is a labeled subgraph isomorphism from the pattern graph to the target graph, the mapping of inputs and outputs of the pattern graph can be used to extract a subcircuit of the target graph for use as a candidate for equivalence checking (CEC). If no labeled subgraph isomorphism is found, we conclude from Theorem 1 that the component is not embedded in the block. Note that the subgraph isomorphism problem is NP-complete [9].

Candidate quality: The quality of the candidate, or the likelihood that a candidate is definitely the component, depends on the set of simulation vectors used to construct the simulation graphs.

For example, using just the 0-hot vector, almost guarantees a false positive, and on the adder example, using just the 1-hot vectors does not differentiate between the MSB and LSB.

For each component we can use a simple criteria to judge the quality of candidate sub-blocks generated when using a specific set of simulation vectors; if component f' has its inputs and outputs permuted to create f, does any candidate isomorphism between the corresponding SGs derived from the simulation set, provide a correct matching of the inputs, up to symmetries in f'.

Note that only the matching of the inputs is mentioned. Once the inputs are matched correctly, it becomes easy to match the outputs using random simulation or formal techniques.

k-cold Simulation vectors: The quality of a candidates is likely to be higher if more simulation vectors are used. However, because of the sheer number of k-hot vectors, it becomes impractical to construct the graph beyond 2-hot simulation vectors.

It is tempting to also use n-hot, $(n'-1)$-hot or $(n'-2)$-hot simulation vectors, but if $n > n'$, the size of the target graph may still be very large.

For example, if $n' = 10$ and $n = 100$, the number of $(n'-1) = 9$-hot simulation vectors is just $\binom{10}{9} = 9$, but at the block it is $\binom{100}{9}$ which is impractically huge.

A simple alternative is to use k-cold vectors. We can generalize the definition of a simulation graph to allow k-cold vectors. An input vertex will be connected to a k-cold vector if the input is 0 on that vector. A *k-cold* label on the vectors is used to distinguish them from k-hot vectors.

For simplicity, in this paper we only formally defined and proved results for the k-hot vectors.

Our experimental results demonstrate that 1-hot, 2-hot, and 0-cold simulation vectors were enough to detect many common components, with the exception of multipliers for which 2-hot, 0-cold, and 1-cold simulation vectors were needed.

VI. LAD-based Approach

This section describes an algorithm to solve SPIEC that checks for subgraph isomorphism in SGs using a state-of-the-art algorithm LAD [10], [11]. LAD works on general graphs and does not take the special structure of SGs into account. It is implemented in terms of a constraint satisfaction framework which starts by assigning each vertex $u \in V_P$ a *domain* $D_u \subseteq V_T$ that contains possible candidates for node matching. These domains are refined in the search process using several filtering techniques until either inconsistencies are found, indicated by an empty domain, or no further refinement is possible.

To speed up the search process, it is important to keep the sizes of the domains small at each step. There are two possibilities to reduce the sizes: (i) when initializing the problem and (ii) by using implications during the search process. The latter is more difficult to implement. LAD offers two methods to decrease the domains initially. First, the in-degree $d^-(u)$ and out-degree $d^+(u)$ of a vertex u in the pattern graph cannot exceed the in-degree and out-degree of vertices in the target graph. LAD also supports labeled subgraph isomorphism using a labeling function l. Based on these observations the initial domain for a vertex $u \in V_P$ is

$$\begin{aligned} D_u = \{v \in V_T \mid d^-(v) \geq d^-(u) \wedge d^+(v) \geq d^+(u)\} \\ \cap \{v \in V_T \mid l(v) = l(u)\}. \end{aligned} \tag{2}$$

Extending LAD-based subgraph isomorphism: We restrict the domains further by extracting information from the underlying circuits of the SGs. Given a circuit and a primary output u we define $\operatorname{supp}_s(u)$ to be the *structural support* of u, i.e., the set of primary inputs that are reachable from the outputs in a backwards traversal of the circuit starting at u. The *functional support* of u, denoted $\operatorname{supp}(u)$, is the set of primary inputs on which the function represented by u depends. Clearly $\operatorname{supp}(u) \subseteq \operatorname{supp}_s(u)$, i.e., the structural support over-approximates the functional support. Matching output vertices must have the same functional support size and therefore the following constraint is added to Eq. (2) for all $u \in Y_P$:

$$D_u = \ldots \cap \{v \in Y_T \mid \#\operatorname{supp}(v) = \#\operatorname{supp}(u)\} \tag{3}$$

If computing the functional support for the target circuit is too inefficient, one can also use the structural support for a weaker constraint, i.e.,

$$D_u = \ldots \cap \{v \in Y_T \mid \#\operatorname{supp}_s(v) \geq \#\operatorname{supp}(u)\}. \tag{4}$$

These constraints only take the size of the support into account but not the actual inputs in the support. If the functional support is computed, one can add additional so-called *support arcs* (x_i, u_r) to the simulation graph, if and only if $x_i \in \operatorname{supp}(u_r)$. The experimental evaluation will show that these arcs can lead to an improvement of the run-time.

Further improvement can be achieved by making use of *simulation signatures*. There are $\binom{n}{k}$ k-hot and k-cold simulation vectors for circuits with n primary inputs. While simulation can be performed efficiently for small k, their explicit representation as vertices in the SG causes a significant degradation in the run-time for subgraph isomorphism. It is therefore of interest to only include the most effective simulation vectors for SGs. But simulation results of other simulation vectors can still be used in other ways.

We compute for each output u a *simulation signature*, which in our experiments, is a tuple containing the number of 0,1,2-hot, and 0,1,2-cold simulation vectors that drive u to 1. More formally, the simulation signature of an output u in a circuit with n primary inputs is a tuple of values

$$\operatorname{sig}_{n,k}(u) = \#\{x \in S_{n,k}^{\text{hot}} \mid u(x) = 1\}$$

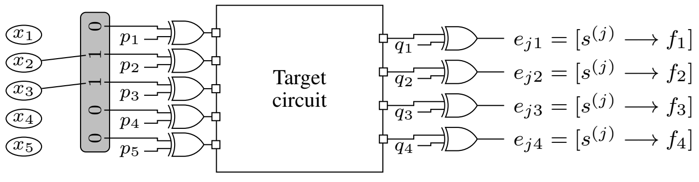
The definitions are analogous for k-cold simulation vectors.

This requires a notion of signature compatibility. Two simulation signatures of a target output $u \in Y_T$ and $u' \in Y_P$ are compatible, denoted $\operatorname{sigcomp}(u, u')$, if and only if for all values in the tuple

$$\operatorname{sig}_{n,k}(u) = \sum_{i=0}^{k} \binom{n-n'}{i} \cdot \operatorname{sig}_{n',k-i}(u') \tag{5}$$

holds. As one instance of this equation, we have

$$\operatorname{sig}_{n,1}(u) = \operatorname{sig}_{n',1}(u') + (n-n') \cdot \operatorname{sig}_{n',0}(u),$$



Target circuit
$e_{j1} = [s^{(j)} \longrightarrow f_1]$
$e_{j2} = [s^{(j)} \longrightarrow f_2]$
$e_{j3} = [s^{(j)} \longrightarrow f_3]$
$e_{j4} = [s^{(j)} \longrightarrow f_4]$

Fig. 2. Handling inverters in SAT-based subgraph isomorphism (circuit construction for simulation vector 00110)

i.e., the size of the on-set for one-hot encoded simulation vectors in the target graph, is the sum of $\mathrm{sig}_{n',1}(u')$ (considering the '1' is assigned to one of the n' matching inputs) and $(n-n')\,\mathrm{sig}_{n',0}(u)$ (considering the '1' is assigned to one of the $n-n'$ non-matching inputs).

Simulation signatures further restrict the domains:

$$D_u = \ldots \cap \{v \in Y_T \mid \mathrm{sigcomp}(u,v)\} \qquad (6)$$

VII. SAT-based Approach

We introduce a SAT-based subgraph isomorphism based on the formulation given in [12] mainly for two reasons. First, to show the dominance of the LAD-based approach in terms of scalability (see experimental results in Sect. IX). Second, a symbolic representation of SGs makes it easier to solve the SNPIEC problem (where possible inverters are at the inputs and outputs). The presence of possible inverters affects the target graph such that it cannot be represented explicitly, which precludes application of the LAD-based approach.

The formulation as a SAT instance is inspired by the formulations described in [12], [13], [14]. The SAT formulation includes the same optimization techniques (blocking using labels, structural, functional support, support arcs, and simulation signatures) and additionally exploits symmetry breaking based on input symmetries. Our LAD algorithm does not consider the latter because it is more complicated and requires several nontrivial changes throughout the whole algorithm. Due to space limitations, we only focus on the extension of the formulation to solve SNPIEC problems.

SAT based formulation of SNPIEC: If inverters are possibly present at inputs and outputs (SNPIEC problem) in the target circuit, simulation results are changed and therefore also the target graph. This seems to prohibit the application of subgraph isomorphism algorithms to find candidate components. However, a SAT based formulation can be made for SNPIEC.

The presence of inverters at inputs and outputs changes the function values of the target circuits and causes arcs between simulation vectors in S_T and outputs in Y_T in the target graph, which cannot be determined explicitly. To formulate SNPIEC the presence of arcs must be determined symbolically based on additional polarity variables $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ with $n = \#X_T$ and $m = \#Y_T$. One way to determine the polarity variables is to construct a circuit for each simulation vector as illustrated in Fig. 2 that contains variables e_{jr} which represent an arc between simulation vertex $s^{(j)}$ and output f_r in the target graph. In this case there are 5 inputs and 4 outputs and the simulation vector is 00110. As can be seen, the input vertices and simulation vector vertices can be connected explicitly. For each input and output of the target circuit, an XOR gate is added that is controlled by a polarity variable. Besides the polarity variables, inputs to the XOR gates are the simulation bits and the original outputs. The new outputs of the circuit are the symbolic values for e_{jr}. This circuit is copied for *each* simulation vector, which results in a very large circuit with $m+n$ primary inputs (the polarity variables) and $m \cdot \#S_T$ primary outputs (the number of e_{jr} variables). This large circuit is transformed into a CNF formula and added to the SAT formulation. In order to get a smaller formula, we optimized the circuit in our experiments using ABC [15] before translating it into a CNF. However, this approach is not tractable (experiments showed reasonable runtimes only for small instances) and further research is needed if motivated.

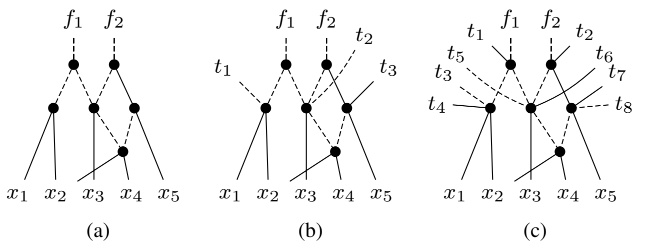


Fig. 3. Output feathering: (a) original circuit, (b) feathering with respecting edge polarities, and (c) feathering all polarities. Dashed edges are inverterd.

VIII. Relaxing the Constraints of Block Identification

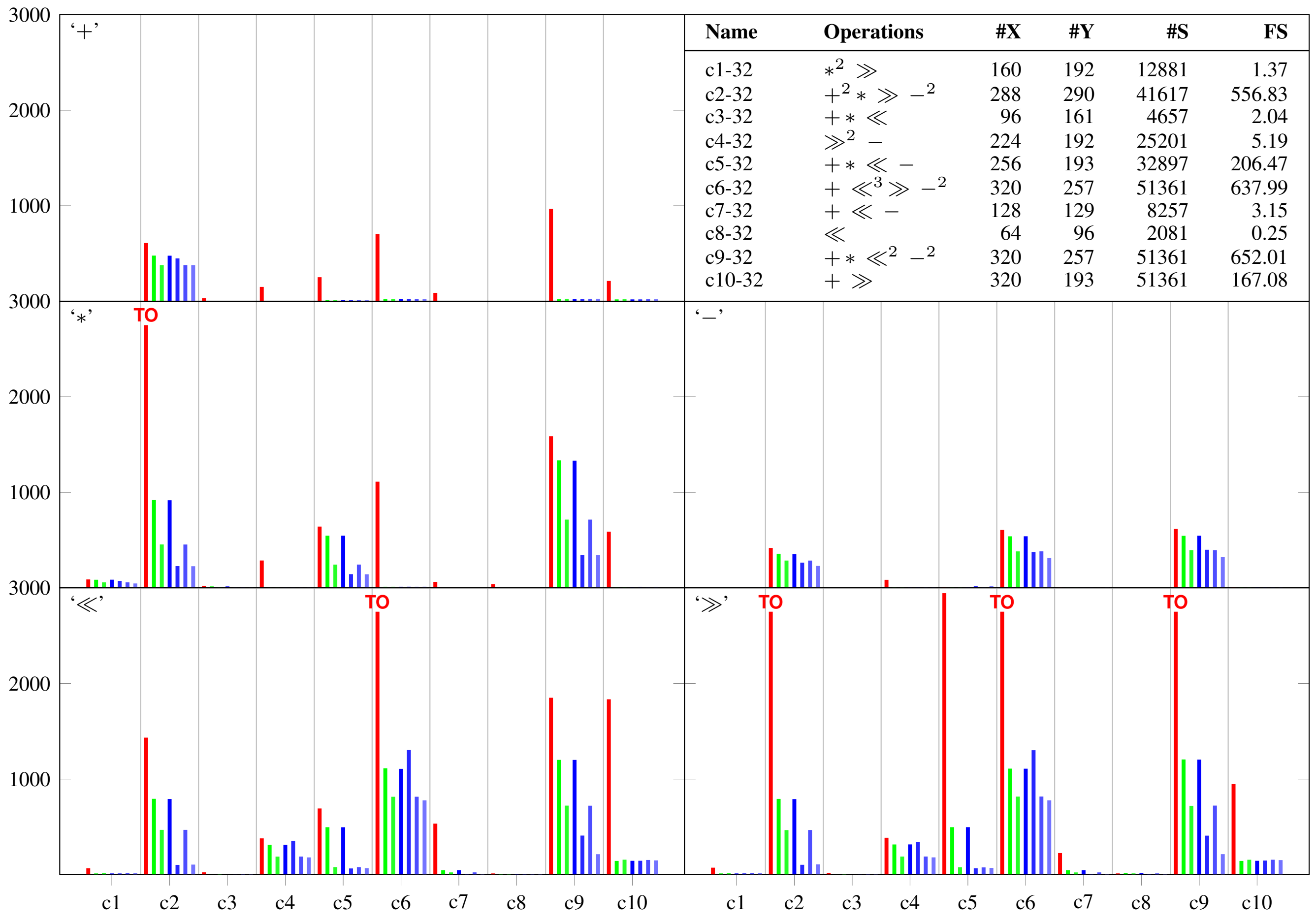
All discussed generalized equivalence checking problems assume that the primary inputs and primary outputs of the component f' are also primary inputs and primary outputs of the block f. Relaxing this assumption can help in the block identification problem. We propose a technique called *output feathering* as a preprocessing step to a SPIEC solver for this. Note that output feathering exploits the subset relation in the problem definition of SPIEC and is therefore not applicable to the other block matching algorithms that were described in Sect. III.

Output feathering first levelizes the circuit—in our case an AIG—and then creates outputs for each node in the k topmost levels. We allow two modes of output feathering. The first creates outputs according to the polarities of outgoing edges whereas the second mode creates an output and its negation for each node. Fig. 3 illustrates output feathering and both modes. This is practical because our SG based method is quite insensitive to the number of outputs of the block.

IX. Experimental Evaluation

We implemented the approaches, discussed in this paper, in C++ and present our evaluations in this section.[1] We implemented a tool (part of the above mentioned source code) that generates gate-level circuits meeting the assumptions on the blocks our algorithm expects from block identification. The tool randomly chooses from multiple arithmetic components in a

[1]The implementation is called '`find_subcircuit`.' The source code and all benchmarks are available at `http://www.informatik.uni-bremen.de/~msoeken/revenge-1.0.tar.gz`.



Name	Operations	#X	#Y	#S	FS
c1-32	$*^2\ \gg$	160	192	12881	1.37
c2-32	$+^2 * \gg -^2$	288	290	41617	556.83
c3-32	$+ * \ll$	96	161	4657	2.04
c4-32	$\gg^2\ -$	224	192	25201	5.19
c5-32	$+ * \ll -$	256	193	32897	206.47
c6-32	$+ \ll^3 \gg -^2$	320	257	51361	637.99
c7-32	$+ \ll -$	128	129	8257	3.15
c8-32	$\ll$	64	96	2081	0.25
c9-32	$+ * \ll^2 -^2$	320	257	51361	652.01
c10-32	$+ \gg$	320	193	51361	167.08

Fig. 4. Experimental evaluation for the LAD-based approach. For each circuit and for each component, the seven bars (shown from left to right) represent the LAD algorithm with different sets of enhancements: (i) default LAD, (ii) structural support, (iii) structural support + simulation signatures, (iv) functional support, (v) functional support + simulation signatures, (vi) functional support + support arcs, and (vii) functional support + support arcs + simulation signatures

library as well as additional components not in the library (e.g., wide AND and OR gates) to create "noise" in the circuit. Then, inputs are randomly added and connected to the components; components may share common inputs; there may be several instances of the same component. The library used in the experiments consisted of an adder ($+$), multiplier ($*$), left-shifter ($\ll$), right-shifter ($\gg$), and a subtracter ($-$). These were mapped to gate level circuits for different bit widths. The experiments were carried out on an Intel Xeon processor with 2.4 GHz, 64 GB RAM, running Linux 3.17. Run-times are given in seconds with a timeout (TO) of 3600 seconds. The sets of experiments done, (i) show that the LAD-based approach is efficient for solving SPIEC problems, (ii) compare the LAD-based approach to the SAT-based approach, (iii) show how the SAT-based approach scales for SNPIEC problems, and (iv) use our approach to solve PICEC problems and compare with the state-of-the-art.

LAD-based approach: The results are listed in Fig. 4. The table in the top right lists properties of the 10 circuits used in the experiments. The first column lists the identifier of each circuit (c1 – c10). Each circuit in the library has 32-bit primary inputs. The second column lists which arithmetic operations are contained in the circuit and superscripts denote the number of instances of the operator. If no superscript is specified, it occurs only once. The columns $\#X$, $\#Y$, and $\#S$ give the numbers of inputs, outputs, and simulation vectors of the target graph. The sum of these three numbers is the number of nodes in the target graph. For the experiments, we used all-hot, one-hot, and two-hot simulation vectors for all components except the multiplier, for which we used one-cold instead of one-hot simulation vectors, since multiplication by 0 creates no arcs between simulation vertices and output vertices. We used the ABC [15] command 'print_supp' which uses a simple method to compute the functional support. The run-time to compute the functional support is given in column *FS*. These numbers can be improved considerably, however, note that the functional support needs to be computed only once for each circuit.

The rest of the figure consists of blocks of plots for each component in the library. The component being matched is illustrated by its operator symbol in the top left corner. Each block of plots is separated into 10 compartments, one for each circuit, and each compartment has seven bars that show the run-times respectively for the following configurations of the

LAD-based approach, from left to right:

(i) A modification of the original LAD approach for subgraph isomorphism from [10]. This implementation considers domain constraints for vertex degrees and vertex labels as described in Eq. (2), referred to as LAD.[2]

(ii) LAD with structural support for domain constraints (see Eq. (4)), referred to as LAD_s.

(iii) LAD_s + simulation signatures.

(iv) LAD with functional support for domain constraints (see Eq. (3)), referred to as LAD_f.

(v) LAD_f + simulation signatures.

(vi) LAD_f + support arcs.

(vii) LAD_f + support arcs + simulation signatures.

Note that the approaches only find a candidate mapping but do *not* perform the final CEC equivalence checks. We performed these checks separately using the ABC command '`iprove`' and did not add the run-times to the values in the plots. In fact, for all operations except the multiplier, the equivalence checks could be performed in less than a second. As equivalence checking multipliers is known to be a hard problem, we manually validated the correctness of the computed mapping based on the port names. All matchings determined by the algorithms were correct, but during the initial evaluations, we experienced several wrong matchings for the multiplier, which were resolved once we added the one-cold simulation vectors. This demonstrates that an appropriate set of types of simulation vectors must be chosen individually per component.

The main observations on the experimental results are:

1) When incorporating the support, the run-time is significantly better; in some cases LAD_s and LAD_f can find a matching within a few seconds while LAD does not find a solution within one hour (see, e.g., 'c6-32' and 'c9-32').
2) In the SPIEC problem, a left-shifter is equivalent to a right-shifter because $a \ll b = (a^R \gg b)^R$ where a^R is the reverse of a. This symmetry is evident in the run-times of LAD_s and LAD_f since run-times are not affected by which component is sought. However, in the LAD approach the run-times diverge significantly in some cases (see, e.g., 'c5-32' and 'c7-32').
3) The best performance is achieved for functional support with support arcs and simulation signatures. Often, the support arcs don't make a difference. Also, if neither support arcs nor simulation signatures are used, the difference between structural and function support based domain restriction is marginal.
4) Generally run-times were significantly better when the component was not present (see, e.g., the 32-bit circuits for adders, multipliers, and subtracters).
5) Subtracters and adders are only hard to find if they occur more than once in the block.

SAT-based approach for SPIEC: We scaled down the circuits of the previous experiment to 8 bits and ran the SAT-based approach mentioned in Sect. VII using MiniSAT [16] as the back-end solver, referred to as SAT in the following. Table I lists the results of comparing SAT with LAD_s. Since the instances become extremely large (the number of variables to formalize μ_S is of order $O(|X_P|^4)$), a long run-time is already spent on creating the instance; the solving time makes up approximately 25%. Note that the run-times of the LAD_s approach can be several orders of magnitude faster, e.g., 'c6' and 'c9'. Also the run time of SAT seems to be almost independent of whether the component is contained in the block or not, but is highly correlated with the number of nodes in the SG. Incremental SAT techniques (e.g., with activation literals) may improve the run-times since many considered instances are similar.

SAT-based approach for SNPIEC: We tried to evaluate the SAT-based approach to solve SNPIEC, denoted $\text{SAT}_\mathbf{n}$, for all circuits of bit width 8; however, no results were obtained within the timeout. The approach is not yet tractable and further research is needed if it turns out that SNPIEC problems occur frequently in RE. One possible direction for future research is the exploitation of exists-forall SAT solvers (see, e.g., [17]) as they are used to solve the PICEC problem in [8].

Comparison to PICEC: In [8], the PICEC problem was solved using a SAT-based method. The input of the problem is partitioned into control and data bits and the aim is to find an assignment of the control bits and a permutation of the data bits. When the number of control bits is small, one can solve such a PICEC problem as a sequence of SPIEC problems by enumerating all control bit assignments, propagating them through the circuit, and stopping once a match has been found. We performed this experiment based on LAD_s on satisfying instances reported in [8] and compared the results with the approach described in [8] with preprocessing, signatures, and Yices [18] as the back-end solver, called PICEC in the following.

TABLE I
EXPERIMENTAL EVALUATION FOR THE SAT-BASED APPROACH TO SOLVE SPIEC

Name	**Operations**	$+$		$*$		$\ll$		$\gg$		$-$	
		SAT	LAD_s	SAT	LAD_s	SAT	LAD_s	SAT	LAD_s	SAT	LAD_s
c1-8	$*^2 \gg$	7.2	0.0	8.6	0.2	8.6	0.0	7.9	0.0	7.6	0.0
c2-8	$+^2 * \gg -^2$	83.6	0.3	89.8	1.2	89.8	0.7	75.3	0.8	87.4	0.3
c3-8	$+ * \ll$	1.8	0.0	1.9	0.1	1.9	0.0	1.6	0.0	1.7	0.0
c4-8	$\gg^2 -$	22.7	0.0	22.2	0.0	22.2	0.3	29.5	0.3	30.3	0.0
c5-8	$+ * \ll -$	48.3	0.0	52.3	0.7	52.3	0.4	46.6	0.4	50.3	0.0
c6-8	$+ \ll^3 \gg -^2$	123.5	0.0	134.9	0.0	134.9	0.8	131.7	0.8	134.6	0.5
c7-8	$+ \ll -$	4.0	0.0	4.2	0.0	4.2	0.1	3.8	0.1	4.3	0.0
c8-8	$\ll$	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0
c9-8	$+ * \ll^2 -^2$	123.3	0.0	155.2	1.3	155.2	0.9	131.8	0.9	134.5	0.4
c10-8	$+ \gg$	120.8	0.0	93.5	0.0	93.5	0.1	117.6	0.1	99.0	0.0

TABLE II
SOLVING PICEC WITH SPIEC

Name	**#PI**	**#PO**	**#C**	$+$		$*$		$-$		$\equiv$		$<$	
				LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC
mul8	16	8	0			1.7	1.9						
mul16	32	16	0			TO	N/A						
simple_alu	64	32	2	6.9	361.3			1.4	182.2				
full_alu	64	32	4	55.4	N/A								
fake2670	133	1	5							0.7	0.1	0.1	0.5
c3540	16	22	34	TO	35.5								

[2]The run-time of our LAD implementation is significantly better compared to the original since we replaced an expensive recursive procedure that stores data on a stack by one that stores data on a heap. Further, by using our own implementation, we avoid writing the SGs to temporary files and can work directly on the data structure. Since the improved algorithm is based on this implementation, the comparison of LAD to LAD_s and LAD_f, is more fair.

The results are given in Table II, which lists the name of each circuit, its number of primary inputs and primary outputs, and the number of control bits (not counted in primary inputs). In the original experiments in [8], individual operations were specified for each circuit. These are listed in the last column together with the required run-times. The run-times for LAD_S include the time spent on equivalence checking since this step is included in PICEC. However, note that the partition of input bits is known in PICEC but not in LAD_S. For some benchmarks, no results were available (N/A). For benchmark 'mul16' the equivalence check did not terminate within one hour, however, the correct candidate was determined in 0.1 seconds. This shows an advantage of the SPIEC approach being decoupled from the equivalence checker. A different equivalence checker implementation might be able to determine equivalence faster. Benchmark 'c3540' created a too large search space with 34 control inputs, and for such cases PICEC is clearly superior.

X. Conclusions

We presented algorithms for new variants of combinational equivalence checking that integrates into other approaches required for tackling RE problems. They find a component in a larger circuit and a mapping of primary inputs and outputs of the component to the larger circuit. We showed that the problem can be solved efficiently using algorithms for subgraph isomorphism on their SGs, exploiting additional functional information of the circuits to reduce the search space. It was demonstrated that our approach is viable for solving PICEC problems. To solve equivalence checking problem in which the polarity of inputs and outputs may be inverted, we discussed an alternative SAT formulation.

There are many directions for future research. For the experiments in this paper we only considered a fixed set of simulation vector types for each type of component. However, for some components this may be an overkill and the result could have been found by using a smaller set. This seems to suggest that each component should come with its own set of simulation vectors that are known to be sufficient, reducing the sizes of the target and component SGs. However, we should experiment with this to see if it improves run-times. On the other hand, iterative methods can be used to dynamically extend the SGs with k-hot or k-cold vectors for larger k by using learned information to reduce the space of simulation vectors.

Although all matchings were correct in our experiments, it has not been discussed how to proceed if there remain ambiguities in the domains after LAD has terminated. There are several possible scenarios. The LAD-based approach could be extended to yield all possible matchings instead of only the first one. Iterative methods driven by counter examples are another promising solution, however, since all simulation-vectors must be invariant to permutation, counter examples cannot be exploited in a straight-forward manner.

The algorithms presented in this paper do not perform the equivalence check; rather they find a possible matching that can be given to a standard CEC checker. We noted that the run-time required by the equivalence check was negligible compared to the run-time required to determine the mapping, except for multipliers. We propose to investigate the use of structural equivalence checkers for such cases with a set of common multiplier implementations as additional components.

In general, we want to enlarge the large class of library components with its set of simulation types for which an SG-based method can correctly identify sub-isomorphisms. Operators might include square, square root, log, and compositions of various operators like multiply-add.

Acknowledgments: This research is supported by the German Academic Exchange Service (DAAD) in the PPP 57134066 and by the German Federal Ministry of Education and Research (BMBF) in the project SPECifIC under grant 01IW13001. We want to thank NSA for support through contract 'Equivalence checking in crypto-analytic applications' and NSF under contract 1219154.

References

[1] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *Int'l Symp. on Hardware-Oriented Security and Trust*, 2012, pp. 83–88.

[2] W. Li, A. Gascón, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "WordRev: finding word-level structures in a sea of bit-level gates," in *Int'l Symp. on Hardware-Oriented Security and Trust*, 2013, pp. 67–74.

[3] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 1, pp. 63–80, 2014.

[4] J. Mohnke, P. Molitor, and S. Malik, "Establishing latch correspondence for sequential circuits using distinguishing signatures," *Integration*, vol. 27, no. 1, pp. 33–46, 1999.

[5] Y. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *Int'l Conf. on Computer Design*, 1992, pp. 452–458.

[6] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 137–148, 1997.

[7] C. Lai, J. R. Jiang, and K. Wang, "Boolean matching of function vectors with strengthened learning," in *Int'l Conf. on Computer-Aided Design*, 2010, pp. 596–601.

[8] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," in *Formal Methods in Computer-Aided Design*, 2014, pp. 83–90, benchmarks and tools are available at `https://bitbucket.org/spramod/fmcad14-experiments`.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] C. Solnon, "AllDifferent-based filtering for subgraph isomorphism," *Artif. Intell.*, vol. 174, no. 12-13, pp. 850–864, 2010.

[11] S. Zampelli, Y. Deville, and C. Solnon, "Solving subgraph isomorphism problems with constraint programming," *Constraints*, vol. 15, no. 3, pp. 327–353, 2010.

[12] C. Anton and L. Olson, "Generating satisfiable SAT instances using random subgraph isomorphism," in *Canadian Conference on Artificial Intelligence*, 2009, pp. 16–26.

[13] V. Arvind, P. P. Kurur, and T. C. Vijayaraghavan, "Bounded color multiplicity graph isomorphism is in the #L hierarchy," in *Conf. on Computational Complexity*, 2005, pp. 13–27.

[14] J. Torán, "On the resolution complexity of graph non-isomorphism," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2013, pp. 52–66.

[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[16] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[17] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "Efficiently solving quantified bit-vector formulas," *Formal Methods in System Design*, vol. 42, no. 1, pp. 3–23, 2013.

[18] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, 2014, pp. 737–744.

Template-based Synthesis of Instruction-Level Abstractions for SoC Verification

Pramod Subramanyan, Yakir Vizel, Sayak Ray and Sharad Malik
Department of Electrical Engineering, Princeton University.

***Abstract*—Contemporary integrated circuits are complex system-on-chip (SoC) designs consisting of programmable cores along with accelerators and peripherals controlled by firmware running on the cores. The functionality of the SoC is implemented by a combination of firmware and hardware components. As a result, verifying these two components separately can miss bugs while attempting to formally verify the full SoC design considering both firmware and hardware is not scalable.**

An abstraction that can be used instead of the cycle-accurate and bit-precise hardware implementation can be helpful in scalably verifying system-level properties of SoCs. However, constructing such an abstraction to capture all the required details and interactions is error-prone, tedious and time-consuming. Another challenge is ensuring correctness of the abstraction so that properties proven using it are valid.

In this paper, we introduce a methodology for SoC verification. We *synthesize* an *instruction-level abstraction* (ILA) that precisely captures updates to all firmware-accessible states spanning the cores, accelerators and peripherals. The synthesis algorithm uses a blackbox simulator to synthesize the ILA from a *template* specification. A "golden-model" generated from the ILA is used to verify whether the hardware implementation matches the ILA. We demonstrate the methodology using a small SoC design consisting of the 8051 microcontroller and two cryptographic accelerators. The methodology uncovered 14 bugs.

I. Introduction

Today's integrated circuits are complex system-on-chip (SoC) designs consisting of one or more programmable cores, several accelerators and peripheral devices [17]. The overall functionality of the SoC is determined by *firmware* that runs on the cores and orchestrates the operation of the accelerators and peripheral devices. Attempting to formally verify the complete SoC with all its hardware components and firmware is not scalable for even very small designs.

Firmware sits "below" the operating system and interacts closely with the hardware. Both firmware and hardware make many assumptions about the behavior of the other component. As a result, verifying the two components separately requires explicitly enumerating these assumptions and verifying that the other component satisfies these assumptions. An example from a commercial SoC highlighting the importance of capturing these interactions is provided in [21]. A series of I/O write operations could be executed by malicious firmware leaving a cryptographic accelerator in a "confused" state after which sensitive cryptographic keys could be exfiltrated. The bug was due to certain implicit assumptions made by hardware about the timing of firmware I/O writes. These were violated by the malicious code sequence.

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

A. Abstractions for SoC Verification

A general technique for making SoC verification tractable is to use an abstraction that accurately models all updates to firmware-accessible hardware states [9, 16, 25, 26]. When verifying properties involving firmware, the abstraction is used instead of the bit-precise cycle-accurate hardware model.

Although the idea of constructing abstractions for firmware verification is attractive, there are several challenges in applying the technique in practice. Firmware interacts with hardware components in a myriad of ways. For the abstraction to be useful, it needs to model all these interactions and capture all updates to firmware-accessible states.

- Firmware usually controls accelerators in the SoC by writing to memory-mapped registers within the accelerators. These registers may set the mode of operation of the accelerator, the location of the data to be processed, or return the current state of the accelerator's operation. The abstraction needs to model these "special" reads and writes to the memory-mapped I/O space correctly.
- Once operation is initiated, the accelerators step through a high-level state machine that implements the data processing functionality. Transitions of this state machine may depend on responses from other SoC components, the acquisition of semaphores, external inputs, etc. These state machines have to be modeled to ensure there are no bugs involving race conditions or malicious external input that cause unexpected transitions or deadlocks.
- Another concern is preventing compromised/malicious firmware from accessing sensitive data. To prove that such requirements are satisfied, the abstraction needs to capture issues such as a sensitive value being copied into a firmware-accessible temporary register.

We argue that manually constructing an abstraction which captures these details, as proposed for example in [25, 26], is not practical because it is error-prone, as well as tedious and very time-consuming. Abstractions that focus on specific types of properties, like the control flow graph from [16], can ease certain verification concerns, but this does not capture all the requirements mentioned above. A third alternative is to verify the firmware using a software model of the hardware [9]. This too misses bugs present in the hardware implementation but not the software model.

The problem with these approaches is correctness of the abstraction. If the hardware implementation is not consistent with the abstraction, properties proven using it are not valid.

B. Synthesizing Instruction-Level Abstractions

In this paper, we propose a general methodology for constructing correct abstractions for SoC verification. The abstraction captures all updates to firmware-accessible states which includes the architectural state of the cores, memory-mapped and I/O addressable registers in the accelerators and peripheral devices as well as high-level state machines that model the operation of the cores and other hardware components.

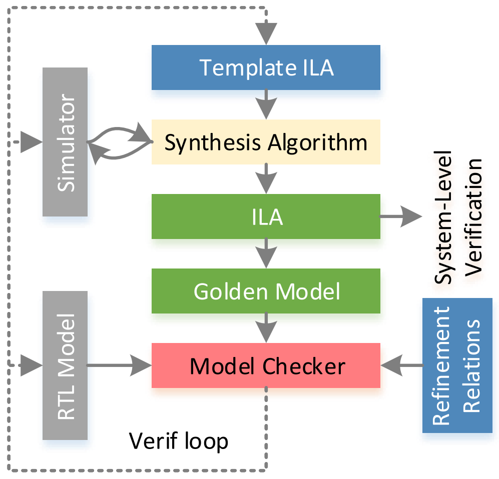


Fig. 1: Block Diagram of Template-Based Synthesis of Instruction-Level Abstractions

We call this an *instruction-level abstraction* or *ILA*. The insight is that firmware can only view changes in system state at the granularity of instructions. So it is sufficient to model hardware components of the SoC at this granularity. Then, the ILA of an SoC is a product of deterministic finite state transition systems that are abstractions of each of the SoCs hardware components constructed at the granularity of instructions. For example, Figure 2 shows an ILA that is a product of three finite state transition systems: a processor, accelerator and an I/O peripheral. The ILA for the processor is analogous to an instruction-granularity control flow graph, while for the accelerator it is an instruction-granularity high-level state machine. If we construct an ILA and prove it is an overapproximation of the hardware components, system-level properties proven using the ILA will be valid.

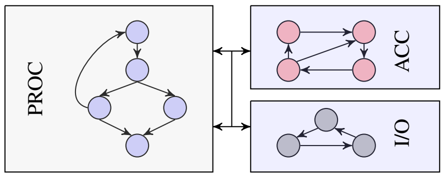


Fig. 2: Instruction-Level Abstraction

To enable the abstraction to be easily constructed in a semi-automated manner, we build on recent progress in syntax-guided synthesis [1, 11, 19]. We propose *synthesizing the ILA* from a *template*. Instead of manually constructing the complete abstraction, the verification engineer now has the much easier task of writing a *template* that partially defines the operation of the hardware components. The synthesis framework infers the complete abstraction and fills in the missing details by using a *blackbox simulator* of the hardware components.

The term *blackbox* simulator means the simulator can be used to find the next state and outputs of the system given its current state and input values, but it is not possible to "look inside" the simulator and get a full-definition of the system's behavior.[1] Simulators are often constructed during SoC design for validation purposes, e.g., simulation-based testing of firmware. In principle, it may be possible to extract an abstraction of the SoC through automated analysis of the simulator, however, in practice, due to the scale and complexity of the codebase it is not possible to do so. Our work constructs an abstraction of the system in this scenario.[2]

To validate the abstraction and ensure that the hardware implementation conforms to the abstraction, we automatically generate a "golden model" from the abstraction. A set of temporal refinement relations are model checked to ensure that the behavior of the implementation matches the behavior of the golden model. If the refinement relations are proven, we have a guarantee that the abstraction is a correct over-approximation of the hardware components and any properties proven using the abstraction are in fact valid. If the proof fails, we get counterexamples that can be used to "fix" either the implementation or the template.

Figure 1 is an overview of the methodology. The blue-boxes show the components that are provided by the verification engineer. We assume that the register-transfer level (RTL) model and a simulator are already available; these are gray. Automatically generated artifacts are green and off-the-shelf tools are red. The synthesis algorithm is in yellow.

C. Contributions

We introduce a general methodology for template-based synthesis of instruction-level abstractions for SoC verification. The methodology has three advantages. It helps verification engineers *easily* construct *correct* abstractions that are *useful* in verifying system-level properties of SoCs.

We introduce a parameterized synthesis framework that allows scalable synthesis of the complex functionality in modern SoCs and a language for template-based synthesis that is tailored to modeling hardware components. We show how correctness of the synthesized abstraction can be verified.

Finally, we present a case study applying the methodology to the verification of a simple SoC design consisting of the 8051 microcontroller and two cryptographic accelerator cores. We discuss construction of the instruction-level abstraction and describe the bugs found during verification. The synthesis framework and experimental artifacts are available online [5].

[1]The *blackbox* simulator is akin the I/O oracle in [11].

[2]The hardware (RTL) implementation can also be used for simulation if a dedicated simulator is not available.

II. Definitions and Formal Model

We model the hardware implementation as a deterministic finite state transition system. Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain. The state space is defined by the union of two sets of Boolean variables encoding the states: $X = X_F \cup X_M$. $X_F = \{x_1, x_2, \ldots, x_m\}$ represent the firmware-accessible states. $X_M = \{u_1, u_2, \ldots, u_n\}$ is the microarchitectural state, and is not visible to the firmware. For example, in a microprocessor core, X_F will contain the architectural registers and program counter while X_M may contain the pipeline registers and reorder buffer. The transition system is then defined as the tuple $M = (X, I, \mathit{Init}, T)$. I is the set of external inputs to the transition system. Init is a predicate over X and defines the initial states of the transition system. $T(I, X, Y)$ defines the transition relation where Y is the set of next-state variables.

The instruction-level abstraction is also modeled as a deterministic finite state transition system. The state space of the abstraction is defined over the set of variables X_A. All firmware-accessible states are included in X_A so $X_F \subseteq X_A$. The transition system is then defined by the tuple $M_A = (X_A, I_A, \mathit{Init}_A, T_A)$ where I_A, Init_A and $T_A(I_A, X_A, Y_A)$ are analogously defined. We also define the blackbox simulation function $\mathit{eval} : I \times X_A \mapsto Y_A$. $\mathit{eval}(\mathtt{I}, \mathtt{X_A}) = \mathtt{Y_A}$ iff $T_A(\mathtt{I}, \mathtt{X_A}, \mathtt{Y_A})$ is true. Here $\mathtt{I}$, $\mathtt{X_A}$ and $\mathtt{Y_A}$ are specific values of I, X_A and Y_A respectively.

III. Synthesizing Instruction-Level Abstractions

The synthesis problem is to construct the finite state transition system $M_A = (X_A, I, \mathit{Init}_A, T_A)$ using the blackbox simulation function eval. One potential solution to this problem is to use results on learning finite state automata [2, 18]. Unfortunately the running time of these algorithms grows as a polynomial function of the number of states in the system. Since a typical hardware component has 2^m states with m state variables and m is in the range of hundreds, thousands or even more, these algorithms are not practical.

We tackle the problem using two insights. The first is that it is reasonable to expect the verification engineer to identify the state variables of the ILA: X_A. We then build on recent progress in syntax-guided synthesis [1] to synthesize the transition relation T_A from a template. The challenge here is that the transition relation for a hardware component is likely too complex to synthesize directly. For instance, consider the transition relation for the ILA of a microprocessor. The inputs to the relation will be all state the processor can access: all data registers, all memory, all external I/O ports, the program counter, flag register, etc. The relation captures the functionality of each opcode by performing a "case-split" for each opcode, which can take hundreds of different values. Synthesis algorithms are currently limited to templates with a few tens to hundreds of synthesis elements [1, 8, 11]. Since the opcode can take hundreds of different values, synthesizing the complete transition relation appears to be out of reach.

Our solution is to simplify the synthesis problem by eliminating the "case-split" structure. The synthesis framework starts with a template, *i.e.,* a specification containing "holes" [1, 19] and a *synthesis parameter*. Synthesis is done for each value of the parameter and the complete transition relation combines the individually synthesized elements.

A. Synthesis Problem Formulation

To synthesize the ILA, the verification engineer constructs a *template* transition relation $\mathcal{T}_A(S, I, X_A, Y_A)$. S is the set of *synthesis variables* which have to be assigned appropriately to make the template $\mathcal{T}_A$ equivalent to T_A. The synthesis problem is parameterized over the parameter p_i where $i = 1, 2, \ldots N$. p_i is a family of predicates defined on X_A such that $p_1 \vee p_2 \vee \cdots \vee p_N = 1$ and $(i \neq j) \implies \neg(p_i \wedge p_j)$. For each i, the synthesis algorithm uses the function $\mathit{eval} : I \times X_A \mapsto Y_A$ and attempts to find an assignment $\mathtt{S_i}$ to S such that $\forall I, X_A :$ $p_i \implies \big(\mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A) \iff T_A(I, X_A, Y_A)\big)$. The conjunction of these relations yields the ILA T_A.

A formal definition of the parameterized synthesis problem is as follows. Find $\mathtt{S_1} \ldots \mathtt{S_N}$ such that for all I, X_A, Y_A:

$$\Big(\bigwedge_{i=1}^{N} \big(p_i \implies \mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A)\big)\Big) \iff T_A(I, X_A, Y_A)$$

Consider the example of synthesizing an ILA for a microprocessor. The template transition relation expresses the different ways in which architectural state can be updated by each instruction. The synthesis parameter is the currently executing opcode, therefore, the predicate p_i would be defined over the ROM values pointed to by the current program counter. The predicate p_0 would be true when the opcode 0 is being executed, predicate p_1 would be true for opcode 1 and so on. Synthesis is done for each opcode and the conjunction $(p_0 \implies \mathcal{T}_A(\mathtt{S_0}, I, X_A, Y_A) \wedge \cdots \wedge (p_N \implies \mathcal{T}_A(\mathtt{S_N}, I, X_A, Y_A))$ defines the operation of the microprocessor under every possible opcode. This is the complete ILA.

B. Template Language Definition

The transition relation is defined using the template language shown in Figure 3. To easily model hardware behavior, a number of primitives to manipulate Boolean and bitvector values are included in the language. It also models memory-like structures (RAM/ROM/register files) and uninterpreted functions from bitvectors to bitvectors.

The template consists of a list of statements. Each statement is either an assignment or an **output** statement. An **output** statement indicates this identifier is one of the outputs of the transition relation. The constructs **bool**, **bv**, **mem** and **func** create Boolean, bitvector, memory and function variables of the appropriate sizes, respectively. A memory variable is specified with two parameters: the bitvector size of the address and the bitvector size of the data cells. A function variable is a map from a bitvector of size `in_width` to a bitvector of size `out_width`. ***bvop*** and ***boolop*** represent all usual bitvector and boolean operators: and, or, not, addition, subtraction etc.

Synthesis is supported using three constructs. The **choice** construct takes a list of expressions as its argument and specifies that the synthesized ILA must replace the choice construct with one of the argument expressions. For example,

$\langle$*template*$\rangle$ ::= $\langle$*stmt*$\rangle$; $\langle$*template*$\rangle$
  | $\langle$*empty*$\rangle$

$\langle$*stmt*$\rangle$ ::= $\langle$*id*$\rangle$ $\leftarrow$ $\langle$*exp*$\rangle$
  | **output** $\langle$*id*$\rangle$

$\langle$*exp*$\rangle$ ::= $\langle$*bv-exp*$\rangle$ | $\langle$*bool-exp*$\rangle$ | $\langle$*mem-exp*$\rangle$

$\langle$*bv-exp*$\rangle$ ::= $\langle$*id*$\rangle$ | **bv** `width` | **bvcnst** `value width`
  | ***bvop*** $\langle$*bv-exp*$\rangle$ ...
  | **if** $\langle$*bool-exp*$\rangle$ **then** $\langle$*bv-exp*$\rangle$ **else** $\langle$*bv-exp*$\rangle$
  | **readmem** $\langle$*mem-exp*$\rangle$ $\langle$*bv-exp*$\rangle$
  | **apply** $\langle$*func-exp*$\rangle$ $\langle$*bv-exp*$\rangle$
  | **choice** $\langle$*id*$\rangle$ [$\langle$*bv-exp*$\rangle$ $\langle$*bv-exp*$\rangle$ ...]
  | **read-slice-choice** $\langle$*id*$\rangle$ $\langle$*bv-exp*$\rangle$ `length`
  | **bv-in-range** $\langle$*bv-exp*$\rangle$ $\langle$*bv-exp*$\rangle$

$\langle$*bool-exp*$\rangle$ ::= $\langle$*id*$\rangle$ | **bool** | **true** | **false**
  | ***boolop*** $\langle$*bool-exp*$\rangle$ ...
  | $\langle$*bv-exp*$\rangle$ == $\langle$*bv-exp*$\rangle$ | $\langle$*bv-exp*$\rangle$ $\neq$ $\langle$*bv-exp*$\rangle$
  | **if** $\langle$*bool-exp*$\rangle$ **then** $\langle$*bool-exp*$\rangle$ **else** $\langle$*bool-exp*$\rangle$
  | **choice** $\langle$*id*$\rangle$ [$\langle$*bool-exp*$\rangle$ $\langle$*bool-exp*$\rangle$ ...]

$\langle$*mem-exp*$\rangle$ ::= $\langle$*id*$\rangle$
  | **mem** `addr_width data_width`
  | **write-mem** $\langle$*mem-exp*$\rangle$ $\langle$*bv-exp*$\rangle$ $\langle$*bv-exp*$\rangle$
  | **if** $\langle$*bool-exp*$\rangle$ **then** $\langle$*mem-exp*$\rangle$ **else** $\langle$*mem-exp*$\rangle$
  | **choice** $\langle$*id*$\rangle$ [$\langle$*mem-exp*$\rangle$ $\langle$*mem-exp*$\rangle$ ...]

$\langle$*func-exp*$\rangle$ ::= $\langle$*id*$\rangle$
  | **func** $\langle$*id*$\rangle$ `in_width out_width`

Fig. 3: Template Language Grammar

suppose we want to model an 8-bit ALU that performs addition, subtraction and the increment operations. This is written as:

$\mathtt{ALU_{INC}} \leftarrow \mathtt{SRC_1}$ + **bvcnst** 1 8
$\mathtt{ALU_{ADD}} \leftarrow \mathtt{SRC_1} + \mathtt{SRC_2}$
$\mathtt{ALU_{SUB}} \leftarrow \mathtt{SRC_1} - \mathtt{SRC_2}$
$\mathtt{ALU_{RESULT}} \leftarrow$ **choice** $\mathtt{ALU_{OP}}$ [$\mathtt{ALU_{INC}}$ $\mathtt{ALU_{ADD}}$ $\mathtt{ALU_{SUB}}$]

The **read-slice-choice** has bitvector b and width k as arguments and synthesizes an expression that extracts bits i to $i + k - 1$ of b for some index i. In other words, it provides a convenient way to operate on slices of bitvectors without specifying the indices of the slice. For example, if one of the bits in the PSW register is the carry flag, we can write this as follows: $\mathtt{CY} \leftarrow$ **read-slice-choice** $\mathtt{CY_F}$ `PSW` 1.

The final synthesis operator is **bv-in-range** which synthesizes a bitvector value within the specified range. Somewhat surprisingly, we found this minimal set of synthesis operators to be sufficiently expressive for our case study.[3] It is very easy to add more synthesis primitives.

C. Synthesis Algorithm

The synthesis procedure first "compiles" the template into a satisfiability modulo theory (SMT) formula which uses the theories of bitvectors, arrays and uninterpreted functions with equality. Most elements in the template language can be mapped to SMT using a straightforward recursive algorithm. The synthesis primitives: **choice**, **read-slice-choice** and **bv-in-range** require special treatment. These primitives introduce new synthesis variables, whose values have to be inferred by the synthesis procedure. As an example consider the **choice** primitive. The statement **choice** `id` [$\mathtt{c_1}$ $\mathtt{c_2}$ ... $\mathtt{c_k}$] is converted to the SMT formula $\mathrm{ITE}(\mathtt{id_1}, \mathtt{c_1}, \mathrm{ITE}(\mathtt{id_2}, \mathtt{c_2}, \mathrm{ITE}(\mathtt{id_3}, \mathtt{c_3}, \ \ldots\ , \mathtt{c_k})))$ where $\mathtt{id_1}$ ... $\mathtt{id_{k-1}}$ are Boolean synthesis variables.

Algorithm 1 Synthesis Algorithm

Function: *synthesize*.
Inputs: $\mathcal{T}_A(S, I, X_A, Y_A)$, p_i and $eval$.
Output: $\mathtt{S_i}$

```
1: $j \leftarrow 1$
2: $\mathcal{T}_1 = \mathcal{T}_A(S_1, I, X_A, Y_{A1})$
3: $\mathcal{T}_2 = \mathcal{T}_A(S_2, I, X_A, Y_{A2})$
4: $F^1 = p_i \wedge \mathcal{T}_1 \wedge \mathcal{T}_2$
5: while $sat[F^j \wedge (Y_{A1} \neq Y_{A2})]$ do
6:     $(\mathtt{I^j}, \mathtt{X^j}) \leftarrow \textsc{SatAssignment}_{I,X_A}(F^j)$
7:     $\mathtt{Y^j} \leftarrow eval(\mathtt{I^j}, \mathtt{X^j})$
8:     $\mathtt{T}_1^j \leftarrow \textsc{Substitute}(\mathcal{T}_1, I = \mathtt{I^j}, X_A = \mathtt{X^j}, Y_{A1} = \mathtt{Y^j})$
9:     $\mathtt{T}_2^j \leftarrow \textsc{Substitute}(\mathcal{T}_2, I = \mathtt{I^j}, X_A = \mathtt{X^j}, Y_{A2} = \mathtt{Y^j})$
10:    $F^{j+1} \leftarrow F^j \wedge \mathtt{T}_1^j \wedge \mathtt{T}_2^j$
11:    $j \leftarrow j + 1$
12: end while
13: $\mathtt{S_i} \leftarrow \textsc{SatAssignment}_{S_1}(F^j)$
```

The translation procedure yields the SMT formula $\mathcal{T}_A(S, I, X_A, Y_A)$ which is then synthesized using Algorithm 1. The key idea is to repeatedly find *distinguishing inputs* [11] while ensuring the simulation input/output values observed thus far are satisfied. A distinguishing input for $\mathtt{S_1}$ and $\mathtt{S_2}$ is an assignment to I and X_A such that the $\mathcal{T}_A$ transitions to a different states with $\mathtt{S_1}$ and $\mathtt{S_2}$. The distinguishing input is found in line 6. Next we use $eval$ to find the correct next-state $\mathtt{Y^i}$ and assert that the next distinguishing input must satisfy this transition (line 10). When no more distinguishing inputs can be found, then all assignments to S define the same transition relation and we pick one of these assignments in line 13.

1) Template Bugs: We say the template $\mathcal{T}_A$ and parameters p_i *can express* the relation T_A if for all i: $p_i \implies T_A(I, X_A, Y_A)$ is a member of the set of relations defined by $\{s \in \mathbb{B}^{|S|} \mid \mathcal{T}_A(s, I, X_A, Y_A)\}$. If $p_i \implies T_A(I, X_A, Y_A)$ does not belong to this family of relations, we say that the $\mathcal{T}_A$ and p_i *cannot express* T_A.

We refer to the scenario when $\mathcal{T}_A$ and p_i *cannot express* T_A as a template bug. A template bug may result in the call to the SMT solver in line 13 to be unsatisfiable. When this happens, our synthesis framework prints out the unsat core of F^j. In our experience, examining the simulation inputs and outputs present in the unsat core is sufficient to identify the bug. The algorithm may also return an incorrect transition relation and this will be discovered either when verifying the ILA (see §IV) or when verifying system-level properties using the ILA.

[3]This finding is consistent with SKETCH where the supported "holes" are of only three types: index expressions, lookup tables and bitmasks.

2) Simulator Bugs: Since $eval$ models a simulator and real-world simulators may contain bugs, it is possible that $eval$ is not equivalent to the idealized transition relation T_A, *i.e.*, $eval(\mathtt{I},\mathtt{X_A}) = \mathtt{Y_A} \iff T_A(\mathtt{I},\mathtt{X_A},\mathtt{Y_A})$ does not hold. This will also either cause an unsatisfiable result or an incorrect transition relation. The former can be debugged using the unsat core of F^j while the latter will be detected during verification.

3) Correctness of Algorithm 1: In the absence of template bugs and if $eval$ is equivalent to T_A, we have the following result about Algorithm 1.

(Theorem) If $\mathcal{T}_A$ and p_i can express T_A and $eval(\mathtt{I},\mathtt{X_A}) = \mathtt{Y_A} \iff T_A(\mathtt{I},\mathtt{X_A},\mathtt{Y_A})$ then $\left(\wedge_{i=1}^{N}(p_i \implies \mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A))\right) \iff T_A(I, X_A, Y_A)$.

IV. Verifying the Instruction-Level Abstraction

Once we have ILA, the next step is to verify that it correctly abstracts the hardware implementation. Our first attempt at this might be to verify properties of the form $\mathbf{G}(x_a = x_f)$ where $x_a \in X_A$ and $x_f \in X_F$ are elements of the firmware-accessible states present in both the abstraction and the implementation. Unfortunately, this property is likely to be false for most internal state in hardware designs. Consider an accelerator design. The ILA may only model a high-level state machine of the accelerator which "executes" an entire operation in one transition but the implementation will step through many intermediate states to accomplish the equivalent. The property is likely invalid during these intermediate states.

A. Verifying Abstraction Correctness

When considering the internal state of the hardware components, we verify the ILA by defining *refinement relations* as proposed by McMillan [14]: $\mathbf{G}(cond \implies x_a = x_f)$. The predicate $cond$ specifies when the equivalence between state in the ILA and the corresponding state in the implementation holds. For example, in a pipelined microprocessor, we might expect that when an instruction commits, the architectural state of the implementation matches the ILA. Defining the refinement relations as above allows compositional verification [12]. Consider the property $\neg(\phi\ \mathbf{U}\ (cond \wedge (x_a \neq x_f)))$ where ϕ states that all refinement relations hold until time $t-1$. This is equivalent to the above property, but we can abstract away irrelevant parts of ϕ when proving equivalence of x_a and x_f.

For state variables that are outputs of hardware components being modeled, we expect that ILA outputs always match the implementation. In this case, the property is $\mathbf{G}(x_a = x_f)$.

1) Discussion of Verification Issues: One part of our case study is a pipelined microcontroller with limited speculative execution. Our refinement relations are of the form $\mathbf{G}(inst\_finished \implies (x_a = x_f))$, *i.e.*, the state of the ILA and implementation must match when each instruction commits. The other part of the case study involves the verification of two cryptographic accelerators. Here the refinement relations are: $\mathbf{G}(hlsm\_state\_changed \implies x_a = x_f)$.

If we had to verify a superscalar processor, the ILA would execute multiple instructions in each transition. The exact number of instructions to be executed with each transition is an output of the implementation and an input to the abstraction. The property would state that after these many instructions are executed, the states of the ILA and implementation match.

B. Verification Correctness

If we prove the refinement relations for all outputs of the ILA and implementation: $\mathbf{G}(x_a = x_f)$, then we know that the ILA and implementation have identical externally-visible behavior. Hence any properties proven about the behavior of the external inputs and outputs of the ILA are also valid for the implementation.

In practice, proving the property $\mathbf{G}(x_a = x_f)$ for all external outputs may not be scalable, so we will have to adopt McMillan's compositional approach. We prove refinement relations of the form $\neg(\phi\ \mathbf{U}\ (cond \wedge x_a^i \neq x_f^i))$ for internal state and use these to prove the equivalence of the outputs.

If these compositional refinement relations are proven for all firmware-visible state in the ILA and implementation, then we know that all firmware-visible state updates are equivalent between the ILA and the implementation. Further, we know that transitions of the high-level of state machines in the ILA are equivalent to those in the implementation. These properties guarantee that firmware/hardware interactions in the ILA are equivalent to the implementation, capturing the requirements mentioned in Section I-A.

V. Evaluation

This section describes the evaluation methodology, the example SoC used as a case study, and then presents the synthesis and verification results.

A. Evaluation Methodology

We implemented the template-based synthesis framework as a Python library using the Z3 SMT solver [4]. Besides synthesis of the ILA, the library also provides a set of functions for generating behavioral Verilog corresponding to the "golden model". This is used to verify that the ILA matches the implementation. We have made the synthesis framework, template abstractions, synthesized ILA, Verilog netlists and other experimental artifacts available online [5].

We used a slightly-modified version of the open source Yosys [24] tool to synthesize netlists from behavioral Verilog. We used ABC [22] for property verification. Experiments were run on Intel(R) Xeon(R) E5645 and E3-1230 CPUs. The E5645 has 12 cores and 128 GB of RAM, while the E3-1230 has 6 cores and 32 GB of RAM. All experiments were run on Ubuntu Linux v12.04.

1) Example SoC Structure: A block diagram of the example SoC is shown in Figure 4. It consists of the 8051 microcontroller and two cryptographic accelerators. The register-transfer level (RTL) Verilog implementation of the 8051 was taken from OpenCores.org [23]. We used *i8051sim* from UC Riverside as a blackbox instruction-level simulator of the 8051 [13]. One accelerator implements encryption/decryption using the Advanced Encryption Standard (AES) [7]. This is from OpenCores.org [10]. The second accelerator [20] implements the SHA-1 cryptographic hash function [6]. We wrote interface modules that "expose" the AES and SHA-1 accelerators to the 8051 using a memory-mapped I/O interface.

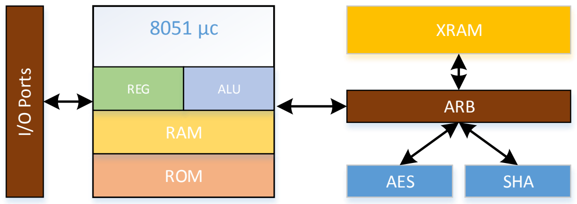


Fig. 4: Example SoC Block Diagram

2) Firmware Programming Interface: The firmware running on the 8051 initiates operation of the accelerators by writing the addresses of the data to be encrypted/decrypted/hashed to memory-mapped registers within the accelerators. Operation is started by writing to the start register which is also memory-mapped. Once the operation is started, the accelerators use direct memory access (DMA) to fetch the data from the external memory (XRAM), perform the operation and write the result back to XRAM. The processor determines completion by polling a memory-mapped status register.

3) Verification Objectives: In this work we focus on producing a verified ILA of the SoCs hardware components. The objectives here are to verify that each instruction in the 8051 is executed according to the ILA, firmware programming the cryptographic accelerators by reading/writing to appropriate memory-mapped registers produces the expected results and that the implementation of the cryptographic accelerators matches the high-level state machines in the ILA. We do not verify correctness of encryption or hashing itself.

B. *Verifying the Example SoC*

We performed the verification in a modular manner by constructing two ILAs: one for the 8051 microcontroller and another for the arbiter, XRAM, AES and SHA modules. The insight here is that the 8051 communicates with the accelerators and XRAM by reading/writing to XRAM addresses. So from the perspective of the 8051, it is sufficient to show that all instructions that modify the internal state of the 8051 are executed correctly and instructions which read/write XRAM produce the correct results at the external memory interface. What happens after these instructions "leave" the external memory interface - whether they modify the XRAM or start AES encryption, or return the current state of the SHA accelerator - need not be considered in this model. For the accelerators and the XRAM, we construct a separate ILA and the only instructions we need to consider here are reads and writes to XRAM addresses. In this ILA, we verify that these operations produce the expected results.

1) Synthesizing the 8051 ILA: We constructed a template ILA of the 8051 which is parameterized over the opcode and models all 256 opcodes of the microcontroller and other elements of architectural state including the internal RAM which contains the register banks, the accumulator and other registers. We used *i8051sim* as the blackbox simulator.

Note this is equivalent to synthesizing the instruction set architecture (ISA) of the 8051. Our methodology ensures that the constructed ILA specification is precisely-defined and correct; this is a significant challenge in practice. For example, Godefroid *et al.* [8] report that ISA documents only partially define some instructions and leave some state undefined. They report instances where implementation behavior contradicts the ISA document and cases where implementation behavior changes between different generations of the same processor-family. Our methodology avoids all of these pitfalls.

Model	**LoC**	**Size**
Template ILA	≈ 650	30 KB
C++ instruction-level simulator	≈ 3000	106 KB
Behavioral Verilog implementation	≈ 9600	360 KB

TABLE I: Lines of code (LoC) and size in bytes of each model.

As an indication of the effort involved in building the model, Table I compares the size of the template ILA with the simulator and the RTL implementation. The template ILA is significantly smaller than both the simulator and the RTL. Table II shows the execution time for synthesis of each element of architectural state. We report the average and maximum values over all 256 opcodes. Except for the internal RAM, all other elements are synthesized with a few seconds.

2) Verifying the 8051 ILA: We first attempted to verify the 8051 by generating a large monolithic golden model that implemented the entire functionality of the processor in a single cycle. The IRAM in this model was abstracted from a size of 256 bytes to 16 bytes. This abstracted golden model was generated automatically using the synthesis library. We manually implemented the abstraction reducing the size of the IRAM in the RTL implementation.

We used this golden model to verify properties of the form $\mathbf{G}(\mathit{inst\_finished} \implies x_a = x_f)$. For the external outputs of the processor, e.g., the external ram address and data outputs, the properties were of the form $\mathbf{G}(\mathit{output\_valid} \implies x_a = x_f)$. Verification was done using bounded model checking (BMC) with ABC using the `bmc3` command. After fixing some bugs and disabling the remaining (17) buggy instructions, we were able to reach a bound of 17 cycles after 5 hours of execution.

State	AVG/MAX Time (s)	State	AVG/MAX Time (s)
ACC	4.3/8.5	B	3.6/5.1
DPH	2.7/5.0	DPL	2.6/4.4
IRAM	1245.7/14043.6	P0	1.8/2.7
P1	2.4/3.8	P2	2.2/3.5
P3	2.7/4.6	PC	6.3/141.2
PSW	7.3/15.9	SP	2.8/5.0
XRAM/addr	0.4/0.4	XRAM/dataout	0.3/0.4

TABLE II: Synthesis execution time for 8051 ILA.

To improve scalability, we generated a set of "per-instruction" golden models which only implement the state updates for one of the 256 opcodes, the implementation of the other 255 opcodes is abstracted away. We then verified a set of properties of the form: $\neg(\phi\ \mathbf{U}(\mathit{inst\_finished} \wedge \mathit{opcode} = o_i \wedge x_a \neq x_f))$. Here ϕ states that all architectural state matches until time $t-1$. We then attempted to verify five important properties stating that: (i) PC, (ii) accumulator, (iii) the IRAM, (iv) XRAM data output and (iv) XRAM address must be equal for the golden model and the implementation.

Property	BMC bounds					Proofs
	CEX	≤ 20	≤ 25	≤ 30	≤ 35	
PC	0	0	25	10	204	96
ACC	1	0	8	39	191	56
IRAM	0	0	10	36	193	1
XRAM/dataout	0	0	0	0	239	238
XRAM/addr	0	0	0	0	239	239

TABLE III: Results with per-instruction golden model.

Results for these verification experiments are shown in Table III. Each row of the table corresponds to a particular property. Columns 2-6 show the bounds reached by BMC within 2000 seconds. For example, the first row shows that for 25 instructions, the BMC was able to reach a bound between 21 to 25 cycles without a counterexample; for 10 instructions, it achieved a bound between 26 to 30 cycles and for the remaining 204 instructions, the BMC reached a bound between 31 and 35 cycles. The last column shows the number of instructions for which we could prove the property. These proofs were done using the `pdr` command which implements the IC3 algorithm [3] with a time limit of 1950 seconds. Before running `pdr`, we preprocessed the netlists using the gate-level abstraction [15] technique with a time limit of 450 seconds.

We believe all instructions and all architectural states can be proven to match the ILA with some verification effort. We will have to apply the appropriate abstractions and possibly specify a few intermediate lemmas. Due to limited time we were unable to perform these proofs for all cases, so we report the partial results shown above. Yet, the current results do substantiate our claim that the ILA can be proven correct.

3) Bugs Found During 8051 Verification: In the simulator, we found 5 bugs in total. Bugs in `CJNE`, `DA` and `DIV` instructions were due to signed integers being used where unsigned values were expected. Another was a typo in `AJMP` and the last was a mismatch between RTL and the simulator when dividing by zero. These bugs were found during synthesis.

An interesting bug in the template was for the `POP` instruction. The `POP <operand>` instruction updates two items of state: (1) `<operand> = RAM[SP]` and (2) `SP = SP - 1`. But what if operand is `SP`? The RTL set `SP` using (1) while the ILA used (2). This was discovered during model checking and the ILA was changed to match the RTL. This shows one of the benefits of our methodology: all state updates are precisely-defined and consistent between the ILA and RTL.

In the RTL model, we found a total of 7+1 bugs. One of these is an entire class of bugs related to the forwarding of special function register (SFR) values from an in-flight instruction to its successor. This affects 17 different instructions and all bit-addressable architectural state. We partially fixed this. A complete fix appears to require significant effort.

Another interesting issue was due to reads from reserved/undefined SFR addresses. The RTL returned the previous value stored in a temporary buffer. This is an example of the methodology detecting and preventing unintended leakage of information through undefined state.

4) Synthesizing XRAM+AES+SHA ILA: The template ILA for the cryptographic accelerators models the high-level state machines (HLSM) for each accelerator. The synthesis parameter is the current state of the HLSM of the two accelerators. The template also models reads/write operations from the processor which read/write the external RAM or internal registers in the accelerators. The AES and SHA functions were modeled using uninterpreted functions.

Model	**LoC**	**Size**
Template ILA	≈ 500	26 KB
Python HLSM simulator	≈ 400	14 KB
Behavioral Verilog implementation	≈ 2800	87 KB

TABLE IV: Lines of code and size of each model.

The sizes of the model are shown in Table IV. Table V shows the time to synthesize each element of the abstraction's state space. Synthesis can be completed in about an hour.

5) Verifying the XRAM+AES+SHA ILA: As before, we generated a Verilog golden model for the XRAM+AES+SHA ILA. We reduced the size of the XRAM in the ILA and the implementation to just one byte because we were not looking to prove correctness of reads and writes to the XRAM. We then attempted to prove a set of properties of the form $\mathbf{G}(\mathit{hlsm\_state\_change} \implies (x_a = x_f))$. We were able to prove that the *AES:State*, *AES:Addr*, and *AES:Len* in the implementation matched the ILA using the `pdr` command. For other firmware-visible state, BMC found no property violation up to 199 cycles with a time limit of one hour.

VI. Related Work

Syntax-Guided Synthesis: Our work builds on recent progress in syntax-guided synthesis which is surveyed in [1]. The synthesis primitives we introduce are similar to the idea of "holes" and the `??` operator proposed in SKETCH [19].

State	AVG/MAX Time (s)	State	AVG/MAX Time (s)
AES:Addr	0.5/1.0	AES:BytesProcessed	0.6/1.5
AES:Ctr	0.6/1.6	AES:EncData	0.4/0.4
AES:Key0	0.7/1.7	AES:Key1	0.6/1.5
AES:Len	0.4/0.9	AES:ReadData	0.4/0.5
AES:State	0.8/2.0	Dataout	91.9/345.2
SHA:BytesProcessed	0.3/0.5	SHA:Digest	0.3/0.3
SHA:Len	0.4/0.4	SHA:RDAddr	0.4/0.4
SHA:Readdata	81.9/588.3	SHA:State	0.3/0.4
SHA:WRAddr	0.4/0.5	XRAM	22.4/58.1

TABLE V: Synthesis execution time for XRAM+AES+SHA ILA.

The synthesis algorithm is based on oracle-guided synthesis from [11]. Our contribution is in the application of synthesis to constructing abstractions for verification and the parameterized formulation which makes synthesizing the ILA tractable.

Synthesizing Abstractions: Godefroid *et al.* [8] synthesize a symbolic model for a subset of the ALU instructions in an x86-core using input/output samples. They cannot verify the correctness of the synthesized model, so it may or may not correspond to the implementation. As such, it is insufficient for our scenario where we wish to use the model for system-level verification with strong guarantees of correctness. Furthermore, our synthesis framework can be used to model general hardware components while they focus on a specific part of the microprocessor: the ALU result and flag outputs.

Verifying Abstraction Correctness: The *refinement relations* we use in proving that the abstraction and the implementation match are from [12, 14]. In [12], Jhala and McMillan show how refinement relations can be defined to prove the correctness of an out-of-order superscalar processor. While these verification techniques are very important, these are not the focus of our paper. We focus on *synthesizing* abstractions. To verify their correctness, we can leverage the rich body of work in hardware verification.

SoC Verification: One approach to compositional SoC verification is by Xie *et al.* [25, 26]. They suggest manually constructing a "bridge" specification that along with a set of hardware properties can be used to verify software components that rely on these properties. Our methodology makes it easy to construct the equivalent of the bridge specifications. It has the added benefit of ensuring the abstraction is correct.

Horn *et al.* [9] suggests symbolic execution on a software model that contains both firmware and software models of hardware components. This approach is complementary to ours because it can used for early-design stage verification, when an RTL model may not be available. However, once the RTL model is constructed, there is no easy way of ensuring that the software model and the RTL are in agreement. This is the critical challenge addressed by our work.

VII. Conclusion

Modern SoCs consist of a number of programmable cores and many accelerators and peripheral devices which are controlled by firmware running on the cores. The functionality of the SoC is derived by this combination of firmware and hardware. Verifying such SoCs is challenging because formally verifying the complete SoC with firmware and hardware is not scalable, while verifying the two separately may miss bugs.

In this paper, we introduced a methodology for SoC verification that *synthesizes* an *instruction-level* abstraction (ILA) of the SoC. The ILA captures updates to all firmware-accessible states in the SoC and can be used instead of the bit-precise cycle-accurate hardware model while proving system-level properties involving firmware and hardware. One advantage of our methodology is that the ILA is verifiably correct. A set of refinement relations are defined to prove that the behavior of the ILA matches the implementation. The other advantage is that instead of specifying the complete ILA, the verification has the much easier task of writing a template ILA which partially defines the operation of the hardware components, and the synthesis algorithm is able to synthesize the missing details. We demonstrated the applicability of our methodology by using it to verify a small SoC consisting of the 8051 microcontroller and two cryptographic accelerators. The verification process uncovered several bugs substantiating our claim that the methodology is effective.

References

[1] 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 *Formal Methods in Computer-Aided Design*, 2013.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computing*, November 1987.

[3] A. R. Bradley. SAT-based Model Checking Without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, 2011.

[4] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[5] Experimental artifacts and synthesis framework source code. https://bitbucket.org/spramod/fmcad-15-soc-ila, 2015.

[6] NIST FIPS. 180-2: Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technology, 2001.

[7] NIST FIPS. 197: Announcing the Advanced Encryption Standard (AES). Technical report, 2001.

[8] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Programming Language Design and Implementation*, 2012.

[9] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. Formal co-validation of low-level hardware/software interfaces. In *Formal Methods in Computer-Aided Design*, Oct 2013.

[10] H. Hsing. http://opencores.org/project,tiny_aes, 2014.

[11] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, 2010.

[12] R. Jhala and K. L. Mcmillan. Microarchitecture verification by compositional model checking. In *Computer-Aided Verification*, 2001.

[13] R. Lysecky, T. Givargis, G. Stitt, A. Gordon-Ross, and K. Miller. http://www.cs.ucr.edu/~dalton/i8051/i8051sim/, 2001.

[14] K. L McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods*. Springer, 2001.

[15] Alan Mishchenko, Niklas Een, Robert Brayton, Jason Baumgartner, Hari Mony, and Pradeep Nalla. GLA: Gate-level Abstraction Revisited. In *Design, Automation and Test in Europe*, 2013.

[16] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. Formal Hardware/Software Co-verification by Interval Property Checking with Abstraction. In *Design Automation Conference*, 2011.

[17] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6), 2006.

[18] R. E. Schapire. *The Design and Analysis of Efficient Learning Algorithms*. MIT Press, 1992.

[19] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

[20] J. Strömbergson. https://github.com/secworks/sha1, 2014.

[21] P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation and Test in Europe*, 2014.

[22] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2014.

[23] S. Teran and J. Simsic. http://opencores.org/project,8051, 2013.

[24] Clifford Wolf. http://www.clifford.at/yosys/, 2015.

[25] F. Xie, X. Song, H. Chung, and Ranajoy N. Translation-based Co-verification. In *Formal Methods and Models for Co-Design*, 2005.

[26] F. Xie, G. Yang, and X. Song. Component-based Hardware/Software Co-verification for Building Trustworthy Embedded Systems. volume 80, May 2007.

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

Transaction Flows and Executable Models: Formalization and Analysis of Message-passing Protocols

Muralidhar Talupur
murali.talupur@gmail.com

Sandip Ray
Strategic CAD Labs
Intel Corporation
sandip.ray@intel.com

John Erickson
MIC Methods, Tools, and Verification,
Intel Corporation
john.erickson@intel.com

***Abstract*—The lack of appropriate models is often the biggest hurdle in applying formal methods in the industry. Creating executable models of industrial designs is a challenging task, one that we believe has not been sufficiently addressed by existing research. We address this problem for distributed message passing protocols by showing how to synthesize executable models of such protocols from transaction message flows, which are readily available in architecture descriptions. We present industrial case studies showing that this approach to creating formal models is effective in practice. We also show that going the other way, *i.e.*, extracting flows from executable models, is at least as hard as the model-checking problem. These results indicate that transaction flows may provide a superior approach to capture design intent than executable models.**

I. Introduction

Over the last decade, significant advance has been made in the formal verification of industrial-scale designs. Formal tools scale to hardware designs with millions of gates and software systems with millions of lines of code [1], [2], [3]. However routine applications of formal methods have been confined to certain niche areas, *e.g.*, floating-point units, device drivers, etc. A key factor constraining the use of formal methods is the unavailability of appropriate models [4]. Constructing formally analyzable models of industrial designs is a complex enterprise, requiring significant expertise both in the artifact being modeled and in the formalism used. The model must be small and abstract to be tractable, while preserving the behaviors of interest from the original design for the analysis to be meaningful. Furthermore, there is a significant cost to maintaining models to keep up with the design evolution. Not surprisingly, most successful adoptions of formal methods have been in areas where the target was the implementation itself (*e.g.*, Register-Transfer Level (RTL) designs in hardware, or C/C++ implementations of software). Unfortunately, this means that verification occurs late in the design life-cycle, after these artifacts have been implemented. Moreover, for such low-level implementations, formal tools do not scale to complete designs [5]. The situation is exacerbated with shrinking time-to-market schedules, that make it infeasible to fix late-discovered deep errors which warrant significant design change. The result has been complex patches, point-fixes, and systems shipped with errors and vulnerabilities. To address these issues it is critical to facilitate easy creation of maintainable, high-level models early in the design life-cycle; the models can then serve as (1) targets for early analysis for catching architecture-level bugs, and (2) specifications driving later phases of development.

This work was done when the first author was at Intel Corporation.

In this paper, we address the problem of efficiently developing high-level executable models for asynchronous message-passing protocols. Such protocols include cache coherence, resource allocation, bus locking, etc., and form the bedrock of modern multicore and multiprocessor systems as well as SoC designs. Errors in these protocols tend to be particularly difficult to detect since they involve unanticipated, subtle interleavings of concurrent communications that are difficult to exercise through simulation. Furthermore, protocol errors discovered late are difficult to fix, since they typically involve design changes in a number of participating components.

Our approach is based on automatic generation of executable models of the protocols from artifacts already created by architects during the system design process. These artifacts typically take the form of diagrams specifying different message transactions. Fig. 1 shows two such diagrams for a toy cache coherence protocol. Observe that they provide a "transaction-centric view" of the protocol: executions are broken into transaction scenarios, and a diagram specifies the message communications for each transaction. A traditional distributed system model, on the other hand, provides an "agent-centric view": for each participating agent *agt*, it specifies the behavior of *agt* under all possible system scenarios. We will refer to these models as *executable models*. They have a closer correspondence to downstream implementations (*e.g.*, RTL or software) which are also developed on agent-by-agent basis. Traditionally executable models are manually created by formal methods experts after studying architecture documents.

The main insight for our work is that, with a little additional information, the seemingly informal transaction descriptions created by architects can be used to synthesize executable models. The paper makes four contributions. First, we develop a formal foundation for specifying protocol transactions. Two key ingredients of this foundation are (i) *transaction message flows* (or simply, *flows*); and (ii) a definition of *compliance* that formalizes correspondence between transaction-centric and an agent-centric views. Second, we develop a framework for synthesizing executable models from flows. This makes it feasible for architects, having little familiarity with formal

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.

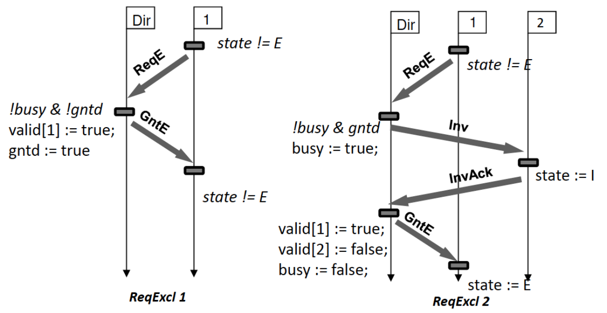


Fig. 1: Two architecture diagrams showing ReqExcl (request for exclusive line access) transaction for a toy cache coherence protocol. Agents are the clients (numbered as 1 and 2) and a memory controller or *Directory*. (a) The scenario in which no other agent has a copy of the cache line. So the line is granted immediately. (b) The scenario where another agent has a copy of the cache line. This copy must be invalidated (and its data written back) before the same line can be granted to 1.

methods, to develop, and analyze executable models of complex industrial protocols, *e.g.*, our tool has been used by architects at Intel to synthesize highly complex protocols in Intel's many-integrated-core (MIC) processors. Third, we report industrial case studies showing that protocol specifications via flows is effective in practice: model generation could be accomplished at *a fraction of the time* required by a formal methods expert to manually build a model. Finally, we show that while generation of models from flows is relatively simple, identifying flows from models is at least as hard as model-checking. This indicates that flows provide a strictly richer semantic information than an executable model.

The rest of the paper is organized as follows. Section II defines flows and formalizes the notion of compliance between flows and models. In Sections III and IV we discuss our approach to synthesize models from flows. In Section V we discuss the converse problem, *viz.*, extracting flows from models, and show that extracting (an appropriate notion of) compliant flows from a model $\mathcal{M}$ is as hard as model-checking $\mathcal{M}$. In Section VI we discuss application of automated synthesis of flows to models in practical case studies. We discuss related work in Section VII and conclude in Section VIII.

II. Systems, Transactions, and Executable Models

A. Executable Model

Our formalization of an executable model is based on guarded transitions. Such formalisms are well-known in concurrency literature [6], [7], and form the basis of system modeling in model-checking tools like Murphi [8] and SPIN [9].

A distributed system involves coordinated computation by a collection of *agents* with indices or *ids*. In our formalism ids are numbers $\{1, \ldots, n\}$. The system state is given by the local state of each agent and the states of communication channels. The local state of agent i is specified by the value of a finite collection *vars*(i) of *state variables*. For each pair of agents i, j, where $i \neq j$ *chans*(i, j) is a finite set of *channels* from i to j. To send a message m to j, agent i places it in some $c \in$ *chans*(i, j). All variables (both state variables and channels) are assumed to take values from a fixed, bounded, finite set $\mathcal{S}$. We assume that $\mathcal{S}$ includes a special "empty" value $\perp$ to represent an empty channel. For each agent i, denote the set *vars*$(i)\cup(\bigcup_{j\neq i}$ *chans*$(i,j))$ by Π_i. An *assignment* of a variable $v \in \Pi_i$ is given by $v := exp$ where *exp* is an expression over $\Pi_i \cup \mathcal{S}$. Unless otherwise noted we keep the syntax of operations involved in expression exp unspecified, but assume that any expression in this paper can be evaluated over $\mathcal{S}$. A guard g_i for agent i is a Boolean expression over $\Pi_i \cup \mathcal{S}$.

Definition 1 (Rule): A *rule* for agent i is a construct of the form $r : g_i \rightarrow a_i$ where r is a symbol called *rule name*, (1) $g_i$is a guard for i and (2) a_i is a collection of assignments of some variables $v \in \Pi_i$.

Definition 2 (Executable Model): An *executable system model* (or simply, *model*) M of n agents is a pair $M \triangleq \langle R, I\rangle$ where R is a set of rules with unique rule names and I is an initial set of assignments to all variables in Π_i, for $i \in \{1, \ldots, n\}$ to constants in $\mathcal{S}$.

We assume that each $c \in$ *chans*(i, j) is assigned to $\perp$ in I. A *system state* s is an assignment to all the variables in Π_i, for all $i \in \{1, \ldots, n\}$. Given a system state s and a rule $r : g_i \rightarrow a_i$ for agent i, we say that r is *enabled* at s if and only if g evaluates to true in state s.

Definition 3 (Rule Firing): Given a rule $r : g_i \rightarrow a_i$ for agent i, we say that s' *is the result of firing of* r *from* s if the following two conditions hold

- r is enabled in s; and
- s' is derived from s as follows: If there is no assignment to v in r then v is assigned the same value in s' as in s. Otherwise, if there is an assignment $v := exp$ in r then v is assigned the value obtained by evaluating exp in s.

Definition 4 (Execution Trace): A sequence of rules $\tau \triangleq [r_1, \ldots, r_k]$ is called an *execution trace* of model $M \triangleq \langle R, I\rangle$ (where $r_i \in Rfori = 1 \ldots k$) if there exists a sequence $[s_0, \ldots, s_k]$ of system states with $I = s_0$ such that following two conditions hold.

- r_i is enabled in s_{i-1}; and
- s_i is the result of firing r_i in s_{i-1}.

B. Flows and Flow Model

The notion of message flows is similar to Message Sequence Charts (MSCs) [10] used in the specification of multi-agent transaction systems. In particular, a flow is a partially ordered set of *events*, specifying the transactions of a protocol. An *event* is a 5-tuple $\langle \text{AGT}, \text{GD}, \text{RECV}, \text{SEND}, \text{UP}\rangle$ as described below. Fig. 2 shows the formalization of the *ReqExcl1* flow in Figure 1.

- AGT $\in \{1, \ldots, n\}$ specifies the index (or *id*) of the agent executing the event.
- GD is guard for $\Pi_{\text{AGT}}\cup\mathcal{S}$ and UP is set of assignments to variables in Π_{AGT}.

1)	$\langle 1, \texttt{true}, -, [\langle \text{ReqE}, \mathsf{Dir}\rangle], -\rangle$
2)	$\langle \mathsf{Dir}, \neg\,\text{busy} \wedge \neg\,\text{gntd}, [\langle \text{ReqE}, 1\rangle], [\langle \text{GntE}, 1\rangle], [\text{valid}[1] := \texttt{true}; \text{gntd} := \texttt{true}]\rangle$
3)	$\langle 1, \texttt{true}, [\langle \text{GntE}, \mathsf{Dir}\rangle], -, [\text{state} := \texttt{E}]\rangle$

Fig. 2: Formalization of three events in *ReqExcl1* flow of Fig. 1. Although process ids are restricted to be numbers in the formalization, we use the symbol Dir for the directory process for pedagogical reasons.

- SEND and RECV are lists of messages. Each message is a tuple $\langle \text{MSG}, \text{ID}\rangle$ where MSG is the actual message and ID is the index of the receiving agent (in case of SEND) or the sending agent (in case of RECV).

We use e.GD, e.AGT, etc. to denote the individual components of event e. As with rules, meaning is assigned to events via guarded commands: e is enabled whenever e.GD holds and the list of messages specified in e.RECV are available in the incoming channels of e.AGT; the execution causes the updates to the local state of e.AGT as specified by e.UP and the list of messages in e.SEND to be sent through the outgoing channels of e.AGT.

Based on the above semantics, we impose following syntactic requirements and restrictions on events: (1) e.GD is a Boolean expression over the state variables of e.AGT; (2) e.SEND provides a list of assignments to the outgoing channels of e.AGT where the right hand side of each assignment is a tuple $\langle \text{MSG}, \text{ID}\rangle$ specifying the message and receiver id; (3) e.RECV is similarly a list of tuples $\langle \text{MSG}, \text{ID}\rangle$ containing the message and sender id.

Definition 5 (Transaction Flows and Flow Model): A *transaction Message Flow* (or simply, *Flow*) is a pair $\langle f, \prec\rangle$, where f is a set of *events* and $\prec$ is a partial order relation over f.

Informally, $\prec$ specifies the temporal ordering on the events in a flow. For Fig. 1, we can view each diagram as a linearly ordered sequence of events; the two diagrams represent two flows describing the two different ways in which a request for exclusive access can be handled. Note that for this and many common cases, the events in a flow f form a sequence, *i.e.*, the partial order $\prec$ is in fact a total order. Nevertheless, there are situations where the generality of partial order is necessary, *e.g.*in the "diamond transaction" shown in Fig. 3. For the rest of the paper, we use f instead of $\langle f, \prec\rangle$ to refer to a flow when the relation $\prec$ is clear from context.

1) Mapping between rules and events: There is an direct connection between rules and events. For the purpose of formalization, we assume fixed syntactic mappings $rl2ev$ and $ev2rl$ that translate a rule into an event and vice versa. The mapping $ev2rl$ maps an event e to a rule of the following form. Here the right hand side of the rule, specified as a list of items enclosed by $\langle\rangle$ represents a sequence of assignments to local and channel variables.

$$\begin{aligned}&e.\text{GD} \wedge (\textit{chans}[(e.\text{RECV}).\text{ID}, e.\text{AGT}] = (e.\text{RECV}).\text{MSG})\\ &\rightarrow \langle e.\text{UP}; (\textit{chans}[e.\text{AGT}, (e.\text{SEND}).\text{ID}] := (e.\text{SEND}).\text{MSG});\\ &\quad \textit{chans}[e.\text{AGT}, (e.\text{RECV}).\text{ID}] := \bot\rangle\end{aligned}$$

Correspondingly, the mapping $rl2ev$ takes a rule rl and extracts the five fields above to get an event ev. The only possible source of ambiguity in a rule rl is about the identity of the sender of messages in case rl involves multiple agents. In our model, messages are communicated using fixed unidirectional channels, and it is simple to identify the sender.[1] This agent can be viewed as the executing agent of the event.

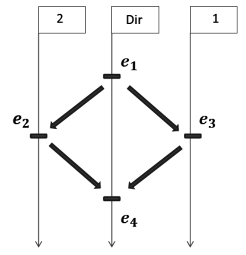


Fig. 3: A transaction requiring $\prec$ to be a partial order. The execution of e_1 enables both e_2 and e_3, and both these events must be executed before e_4 can be enabled.

2) Flows as templates: Given the correspondence between events and rules, a flow provides a template or a pattern for system execution, grouping together related rules with a temporal ordering on their firing. A flow can be invoked or *instantiated* several times, even concurrently, during a run of the system. To make precise the relation of an execution trace with flows, we need to disambiguate between these instances. The notion of tagging accomplishes that by augmenting a flow with a "tag". Here we assume that we have an unbounded set T of *tags* (which is different from all the previously defined sets, *viz.*, variables, values, events, rules, etc.).

Definition 6 (Tagged Events and Flows): A *tagged event* is a pair $[e, t]$ where e is an event and t is a tag. If $\langle f, \prec\rangle$ is a flow, then a *tagged flow* $\langle [f, t], \prec\rangle$ is obtained by replacing each event $e \in f$ with the corresponding tagged event $[e, t]$.

Definition 7 (Legal Tagging): Given a set of flows F and a set T of tags, a set $[F, T] \subseteq \{[f, t] : f \in F, t \in T\}$ is a *legal tagging* if and only if for $f, f' \in F$ such that $f \neq f'$, if $[f, t]$ and $[f', t']$ are members of $[F, T]$ then $t \neq t'$.

Informally, we want a *unique* tag to be associated with each instance of a flow in an execution trace of the system. The definition of compliance makes this notion explicit.

Definition 8 (Precedence-Preserving Mapping): Let $\tau \triangleq [r_1, \ldots, r_k]$ be an execution trace of model M, F be a set of flows, and $[F, T]$ be a legal tagging of F. Let $rl2ev_{\#}$ be a one-to-one mapping from rules in τ to tagged events in $[F, T]$. We say $rl2ev_{\#}$ is *precedence preserving* if for each r_i, $i = 1 \ldots k$

[1]In practice, the message names usually gives away the sender identity.

there exists a tagged flow $[f,t] \in [F,T]$ and a tagged event $[e,t] \in [f,t]$ such that the following conditions hold:

- $rl2ev_{\#}(r_i) = [e,t]$
- $rl2ev(r_i) = e$
- for each $p \in f$ such that $p \prec e$, there exists $k < i$ such that $rl2ev_{\#}(r_k) = [p,t]$.

Definition 9 (Compliance): Let $\tau \triangleq [r_1,\ldots,r_k]$ be an execution trace, and let F be a set of flows. We say that τ is compliant with F if there exists a precedence preserving mapping $rl2ev_{\#}$ from members of τ to events in a legal tagging $[F,T]$. A model M is compliant with flows F if every trace of M is compliant with with F.

Flows provide a generalization of control flow graph to a distributed setting. The definition of compliance essentially stipulates that the trace τ can be viewed as a composition of a collection of flow instances. The requirement of precedence preserving mapping guarantees that each such instance can be uniquely identified with a tag and respects the precedence constraints imposed by flows.

We end this section by briefly comparing the notion of flows introduced here with a related notion in previous work [11], [12] which was also called "flows". Flows in previous papers did not consider state annotations and updates. The more refined notion used here is necessary for synthesizing protocols, while previous work only used flows to infer invariants. Nevertheless, we use the same name in this paper for two reasons. First, the definition here strictly supersedes the previous notion, *viz.*, the previous usages can be accomplished with the current notion. Second, the notions are similar, both in structure and in "spirit", *e.g.*, both aim to exploit the transaction-centric view of protocols.

III. Synthesizing Models from Flows

Fig. 4 shows the high-level steps for applying our framework for synthesizing executable models. The user starts from an initial (possibly incomplete) set of flows F that captures the algorithmic aspects of protocol being designed, and progressively refines this set in response to feedback from a model-checker. In more detail, we automatically synthesize a (compliant) model M from F, and model-check M against a set of user-provided assertions I and a collection of sanity conditions. If model-checking detects deadlock or invariant violation, or a sanity condition fails, a counterexample is provided together with diagnostic information (cf. Section VI). The user modifies F, possibly by adding more flows or adjusting some existing one, so that the erroneous behavior is ruled out. The process is iterated until model-checking succeeds. Note that all the steps are automatic other than the obviously creative step of "fixing" F to rule out the counterexample.

The approach requires a set of assertions. The assertions we use are simple and straightforward, *e.g.*, for cache coherence protocols the obvious assertion is the coherence property itself. Sanity checks are also included to ensure that the generated model M (and, by implication, F) exhibit certain desired behaviors. We discuss some generic sanity checks in the next section. In addition, domain-specific sanity conditions can be added by the user. For instance, a simple sanity check for cache coherence protocols is that for every agent a there is a trace where a can get into *shared* and *exclusive* states.

```
while !(done) do
    Synthesize model $M$ *compliant* with set of flows $F$.
    if $M$ *does not deadlock,* $M \models I$ *and passes sanity checks* then
        $M$ is the required executable formal model.
        done := true.
    else
        Modify $F$. In case of deadlock or invariant violation use the counterexample trace as a guide.
    end
end
```

Fig. 4: Synthesis-Refinement Loop for Interacting with Flows

A. Synthesizing M

Consider synthesizing a model from a set of flows F. A first naïve approach may be simply mapping each event e in each flow $f \in F$ to $ev2rl(e)$. To see why this does not work, note that $ev2rl$ is a function of e alone and not the preceding events of e in f. Hence we are not guaranteed that rule $ev2rl(e)$ is enabled respecting the ordering relation "$\prec$". The key task in synthesizing a model M is to create rule guards in M such that for any execution trace τ each rule $r \in \tau$ respects "$\prec$". To achieve this, we augment the state variables for each agent a by the following additional components:

- A "local tag list" T_a for each agent a, that disambiguates concurrently executing flows (including different instances of the same flow) involving a.
- A mapping $\eta_a : T_a \to Id \times \cup_{b \neq a} T_b$ that associates every local tag of a with a set (possibly empty) of local tags of other agents b that a communicates with.
- A set of "history variables" $H_a[f,t] \subseteq E_f$, where E_f is the set of events in flow f, for each agent a, flow f, and each possible tag value t. The history variable $H_a[f,t]$ records the firing of events in the instance of flow f with tag t. For example, if event $e \in H_a[f,t]$ then it means event e occurring in f has been fired by agent a with associated local tag t.

The local tags are different from the global tag introduced in Section II. Since each agent has only local visibility, it is impossible to ensure that the tags chosen for concurrently executing flows are globally unique. The crux of the synthesis consists of "book-keeping" to disambiguate between different active transactions based on local history and tags.

Suppose that f is a flow which includes an event e. Then we generate a rule $ev2rl^*(e,f)$ by extending the rule $ev2rl(e)$ with (1) additional assignments updating the relevant tag and history variables and (2) additional conjuncts on the guard. Let a be the agent executing e, that is, $e.\text{AGT} = a$.

Updates to History and Tag. Fig. 5 shows how these variables are updated. Note that we must update history variables of a each time a rule $ev2rl(e)$ fires.

```
if $e$ is the first event for agent $a$ in flow $f$ then
    Pick a new tag $t$ and add it to $T_a$.
    Add $e$ to history variable $H_a(f,t)$.
    Let $m = e.\text{SEND}$. Replace $m$ with a new message
    $m'$ by augmenting it with the identifier $(a,t,f)$.
    if there is an incoming message $msg$ for $a$ in $e$ then
        Let $(b,t',f)$ be the identifier associated with
        $msg$. Add the mapping $t \to (b,t')$ to $\eta_a$. This
        indicates that agent $b$ uses local tag $t'$ for that
        particular instance of flow $f$.
    end
else
    Then there exists $e' \prec_f e$ such that $e'.\text{AGT} = a$. The
    local tag $t$ already picked for $e'$ is used as local tag
    for $e$ as well. Update $H_a(f,t)$, $\eta_a$ and message
    $m = e.\text{SEND}$ exactly as above.
end
```

Fig. 5: Updating History and η Variables when agent a fires event e

Additional Guard. The additional guard conjunct is given by the expression $g_a[e,f] \triangleq \exists t : \forall e' : e' \prec e \Rightarrow e' \in H_a[f,t]$. That is, $g_a[e,f]$ is true only when all events preceding e' in flow f have been executed. Note that although we wrote the guard as a quantified first-order expression, for any system involving a finite set of flows and active tags, it can be written as a Boolean expression by enumeration.

Fig. 5 defines an algorithm for each agent to keep track of the local tag it uses to communicate with other agents in specific flow instances. This information is sufficient to create a global tag as required by compliance definition (cf. Theorem 1). Indeed, the elaborate tagging and history updates mirror typical RTL implementation of protocols with multiple, concurrently executing flow instances. Although essential for correctness, this part of protocol design is typically boring to humans while still being tricky to get right. Indeed, many errors in protocols arise by incorrect handling of such disambiguation procedures. By synthesizing it automatically, we free the human to focus on the algorithmically interesting parts.

The history and tag variables can make the synthesized model unbounded, *e.g.*, we need a history variable $H_a[f,t]$ for each possible tag value t. In practice, we impose finiteness by setting an upper bound to the set of possible tag values. This is reasonable in our case since our protocols are implemented in hardware with finite resources; consequently, most protocol definitions already include an upper bound on the possible tag values to impose implementability. Nevertheless, the model is more restrictive than flow descriptions. In particular, the set of possible tags constrains the number of possible concurrent instantiations of a flow. We can avoid this constraint to some extent by reusing tags of completed flows.

The correctness of the procedure is given by Theorem 1. Here $ev2rl^*$ and $rl2ev^*$ are augmentations of $ev2rl$ and $rl2ev$ respectively so that the domain of $ev2rl^*$ (and range of $rl2ev^*$) include additional guards and updates to the history variables.

Lemma 1: Let $\tau \triangleq [r_1, \ldots, r_i]$ be any execution trace of the model M obtained by applying the above procedure to F. Then there exists $f \in F$ such that $rl2ev^*(r_i)$ is an event in f and for each $e \prec_f rl2ev^*(r_i)$ there exists $k < i$ such that $rl2ev^*(r_k) = e$.

Proof sketch: The proof follows from induction on the length of τ. In the induction step, we note that using the guard specified for *Additional Guard* and rules for history variable update, for each event e such that $e \prec rl2ev^*(r_i)$, $ev2rl^*(e,f)$ must be executed for r_i to be enabled. ■

Theorem 1: If M is an executable model synthesized from a set of flows F, then M is compliant with F.

Proof sketch: Let $\tau \triangleq [r_1, \ldots, r_k]$ be any execution trace of M. It is sufficient to show that there is a legal tagging $[F,T]$ and precedence preserving mapping $rl2ev_{\#}$ from rule firings in τ to $[F,T]$. Below we provide a construction of T. The result then follows from Lemma 1.

We construct $[F,T]$ inductively. Recall that each rule firing r in τ is associated with a unique $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ triple. We have to map every tuple $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ seen in the execution τ to a global tag so that all local tags used for the same instance of a flow f by different agents get mapped to the same global tag.

For the base step, the unique tuple of the first rule r_1 is mapped to global tag of 1. In the inductive step, suppose the tuples for rules $[r_1, ..., r_{i-1}]$ have been mapped to global tags $T_{i-1} \triangleq \{1...g\}$ such that T_{i-1} is a legal tagging. We then consider the following cases for r_i.

Case 1: There exists $e \prec rl2ev^*(r_i)$ such that both e and $rl2ev^*(r_i)$ are events in f and $e.\text{AGT} = (rl2ev^*(r_i)).\text{AGT}$. Then by Lemma 1 there exists a $k < i$ such that $ev2rl^*(e,f) = r_k$. By induction hypothesis, $ev2rl^*(e)$ is tagged by T_{i-1} and this global tag can be used for r_i.

Case 2: If there is no e satisfying *Case 1* then either $rl2ev^*(r_i)$ is an initial event of f or all preceding events of $rl2ev^*(r_i)$ are executed by an agent different from $(rl2ev^*(r_i)).\text{AGT}$. In the former case, we can augment T_{i-1} with any unused tag and map that to r_i. In the latter case, $(rl2ev^*(r_i)).\text{AGT}$ must have received a non-empty set P of tuples $\langle \text{AGT}, \text{TAG}, \text{FLOW} \rangle$ from preceding rule firings. If all these are mapped to the same global tag g we use that for the current rule as well. Otherwise, we pick an unused global tag g and map the tuple of the current rule r_i and all the tuples in P to g. ■

IV. Additional Checks to Ensure Realizability

The notion of compliance only ensures that each trace in the model M can be viewed as an interleaving of flow instances, but not that all flows in F can be exercised by some trace. Consider the trivial model $\mathcal{M}_{tr}$ in which the guard for each rule is the logical false. Since the only trace of $\mathcal{M}_{tr}$ is the empty trace, $\mathcal{M}_{tr}$ is compliant with F. To ensure that every flow is "necessary", we introduce the following additional sanity check.

Non-Triviality Check. We say that a flow $f \in F$ is *exercised* by M if there is a trace τ in M such that τ is *not complaint* with the set of flows $F \setminus \{f\}$. M is *non-trivial* with respect to F if every flow $f \in F$ is exercised by M.

The non-triviality check ensures that without f, there is no way to decompose τ into interleaving instances of flows from F. In general, it can require exhaustive exploration of M. However, it is efficient in practice since model-checking quickly discovers traces exercising all flows (cf. Section VI-A).

In addition to non-triviality, we perform two other checks which catch frequently observed mistakes in flows. These checks are done on the flows F directly, not on the synthesized model M. First check is that for every message m sent by some event e in flow f, there is some other event e' in f, $e \prec e'$ such that $e.\textsc{recv}$ includes m. The second check, called *prefix consistency* is more subtle. Here we assume that for flow $\langle f, \prec \rangle$, the relation $\prec$ is a total order over the events $\{e^1_{(f,a)}, \ldots, e^2_{(f,a)}\}$ executed by agent a (so that they form a sequence). This is a general restriction and follows well-known tradition of distributed system definitions [7].

Prefix Consistency Check. Let f_1 and f_2 be two flows such that the event sequences executed by agent a are $[e^1_{(f_1,a)}, \ldots, e^k_{(f_1,a)}]$ and $[e^1_{(f_2,a)}, \ldots, e^l_{(f_2,a)}]$ respectively. Then f_1 and f_2 are *prefix consistent* if the following holds for $n = 2, \ldots, \min(k, l)$: If the first $(n-1)$ events of both sequences is identical, and the GD and RECV fields of $e^n_{(f_1,a)}$ and $e^n_{(f_2,a)}$ are identical, then so must be the UP and SEND fields.

The motivation for prefix consistency comes from the fact that when two or more flows share a prefix there may be "misunderstanding" among agents regarding which flow is being executed (usually leading to a deadlock). For instance, suppose $e^n_{(f_1,a)}$ and $e^n_{(f_2,a)}$ cause the same message m^b_a to be sent from agent a to b, but local updates on a and the expected response from b are different. Suppose a executes $e^n_{(f_1,a)}$. Receipt of message m^b_a can enable b's response to $e^n_{(f_2,a)}$ in addition to $e^n_{(f_1,a)}$. That is, from the perspective of b, the flow its participating in is f_2 whereas from the perspective of a it is f_1. Consequently, the response sent by b is discarded by a which continues to wait for a response to $e^n_{(f_1,a)}$, leading to a deadlock. Indeed, we introduced this check after observing that this phenomenon is quite common for many industrial protocols and leads to subtle errors.

Finally, note that the checks described in this section only ensure that flows pass a minimum quality screening; they cannot ensure that only correct models are synthesized. Indeed, the quality of synthesized model is only as good as the flows. In practice, particularly in the initial iterations of the refinement loop of Fig. 4, flows do contain errors that have to be fixed by the user through analysis of model-checking counterexamples. The main advantage of using flows over directly writing executable models from scratch is that it provides an easier and more intuitive way of creating models.

V. Extracting Flows from Models

In this section we consider the inverse problem of the preceding section, *i.e.*, extracting a set of compliant flows from a given executable model. In addition to being of theoretical interest, *e.g.*, for comparing the semantic richness of flows vis-a-vis models, an efficient flow extraction algorithm has practical utilities. For instance, flows can yield powerful invariants facilitating formal verification [11], [12]. Moreover, there are legacy models of industrial protocols which are large and hard to understand; extracting flows may facilitate understanding by exposing the underlying transaction structures.

Flow Extraction Problem. Let M be any executable model. Construct a finite set F of flows such that (1) M is compliant with F, and (2) each flow $f \in F$ is exercised by M.

Condition 2 ensures that for each f in F, an instantiation of f occurs in some trace τ of M. This rules out trivial cases, *e.g.*, defining each rule in M to be a flow with a single event and calling the rule set to be the set of "extracted" flows.

Unfortunately, the flow extraction problem as stated is as hard as model-checking M. To see that, let I be an arbitrary predicate over M, and consider the model M^+ obtained by extending M with the single rule $r : I \to \texttt{NOP}$, where NOP does not involve any updates. Let F be a set of flows satisfying Conditions 1 and 2 above. Then $\neg I$ is an invariant of M if and only if no $f \in F$ includes the event $rl2ev(r)$. This follows by noting that r occurs in some trace if and only if I is true of some reachable state of M.

In spite of the result above, it is often possible to extract approximate flows from a sufficient set of bounded executions of M by heuristically inferring event precedence. Indeed, in many cases, we empirically found such an approach to produce flows similar to those created by architects. Thus, for legacy protocols with no available flows, such approximations can be used as substitutes to mine invariants. However, the result shows that extracting "perfect" flows is an intractable reverse-engineering problem.

VI. Applications

A. Flows2HLM Synthesizer

We developed a tool, *Flows2HLM*, to synthesize models from flows. The tool implements the framework in Section III, augmented with the following facets to facilitate adoption.

Parameterized Flows. The definition of flows required each event to include the id of the executing agent. But in practice, ids only specify the type of the executing agent (*e.g.*, *Directory* agent vs. *Cache* agent). Two or more agents of the same type occurring in the same flow are disambiguated by using numbers along with type. Our tool supports such parametric flows. The key change required in the algorithm is to add agent ids to the local tags to disambiguate same events from parametric flows executed by different concrete agents. Dealing with parametric flows keeps the synthesized model small and manageable.

Murphi Types. The model generated by Flows2HLM is a Murphi [8] model. Apart from adding tagging information and event precedence, it also adds a Murphi header file declaring the types of variables. Type inference in this case is simple, since all variables are finite enumerated types.

Flows2HLM follows the refinement loop of Section III, using Murphi as the model-checker. In addition to generic assertions (*e.g.*, cache coherence) and the sanity checks discussed in Section IV, the user can write project-specific sanity checks, constraints, etc. The generated model and any assertion violations are reported to the user, together with diagnostic information culled from the counterexample. The crucial diagnostic information is the interaction of flows leading to the failure. Typos and "shallow" errors are typically identified (and easily repaired) in initial synthesis iterations. For example, an error in message type manifests in a deadlock, as follows. Suppose agent a is inadvertently declared to send b a message of type E instead of correct type C; then b, which expects C, waits indefinitely, leading to a deadlock. Finally, while there is no guarantee, the initial iterations coupled with random simulations are enough to see that all flows are being exercised; thus, the non-triviality check of Section IV is easily satisfied.

B. Synthesizing Industrial Protocols

Flows2HLM has been used to synthesize several cache coherence protocols for Intel's next-generation many-integrated-core (MIC) processors; recently, it has been applied to bus lock and credit management protocols as well (cf. Table I). For pedagogical reasons, we also synthesized two academic cache protocols, German [13] and Flash [14], and their flows are publicly available [15]. We invite the reader to explore them, to appreciate the intuitive nature of flow specifications.

We elaborate a bit on our experience with Intel 2, since it is the most complex protocol synthesized by Flows2HLM so far (and perhaps the most complex cache coherence protocol formally analyzed). Interestingly, the synthesis was done fully by an architect with no prior formal modeling experience. The initial definition took two days and constituted 40 flows. Several sanity checks were added by the architect, *e.g.*, that the cache for each agent can get to an exclusive state. This description contained many shallow errors which were detected by Flows2HLM, including the deadlock scenarios discussed above. Subsequently subtle bugs were exposed, including an unexpected response to a local snoop message which required adding a new flow to fix. Overall *six major bugs were found*, which required significant modification of flows; any *one* of them, if leaked into RTL, would have led to a costly RTL churn. The effort took a month, including modification to Flows2HLM; in contrast, an earlier effort developing a hand-written Murphi model of similar complexity by a formal verification expert had required 6 months.

In addition to facilitating use of formal models by architects, a major gain of our approach is easy maintenance and modification of protocols. For Intel 2, the architect subsequently made major changes to flows to create a derivative protocol in matter of days, something that would be very hard with hand-written executable models. Furthermore, the architect used counterexamples returned by model-checker during the synthesis effort to estimate downstream RTL validation complexity. If model-checking counterexamples involve complex interleaving of several flow instances with a number of agents, one may expect bugs in further elaborated RTL implementations to also exhibit similar characteristics and hence require significant validation effort. Based on this insight, the architect simplified the initial protocol to keep the RTL validation complexity manageable.

Finally, since annotations are written manually as part of flow description, our approach provides reduction in specification detail over hand-written executable models only if they are small. This has been the case for most message-passing protocols we have seen, including academic cache coherence protocols as well as SoC and MIC protocols in Intel. Annotations account for less than 10% of model size in all our examples, and less than 5% for the larger protocols. This is not due to bloating from auto-generated code; hand-written models of protocols of comparable complexity were typically of a similar size. Note that for *microarchitectural* protocols, which involve lower level details like message buffering and arbitration, annotations may form a larger part of the description. Investigating application of flows for such protocols is an interesting future work.

VII. Related Work

Several protocol description techniques have been created in recent years, *e.g.*, table-based methods, message sequence charts (MSC), etc. [16], [4], [17], [10]. However, they have not been widely adopted in industrial practice. We speculate that a key bottleneck is the need to think about protocols in terms not natural to the architects. For instance, table-based methods require manually projecting system-level transactions to each agent and carefully tracking the set of transactions that the agent can participate in concurrently. Furthermore, a local change in one transaction requires modification of multiple tables. The comparison with MSCs [10] is more interesting. MSCs capture a diverse range of distributed computing artifacts, including interface protocols and real-time systems; consequently, they include several bells and whistles. While graphical formats provide a more intuitive visualization than text for small protocols, they become unwieldy and inflexible for protocols with 40 flows. Rather than identifying a subset of MSCs for specifying protocols, we found it easier to develop a simple language analogous to MSCs directly based on artifacts we observed the architects to use in informal specifications.

There has been recent related work on synthesizing distributed protocols. Udupa *et al.* [18] synthesize models from concolic execution snippets via user-guided iterative refinement using counterexamples from a model checker. Concolic snippets are analogous to tabular specification, with each snippet corresponding to a row. Synthesis based on snippets cannot account for the context of an event execution, *i.e.*, preceding events in a flow. We found flows to provide a more effective and natural starting point. Alur *et al.* [19], use *scenarios* and temporal properties to synthesize protocols; they address the problem of automatically synthesizing a model even when the number of scenarios is inadequate. There is superficial correspondence between scenarios and flows. However, while flows are self-contained, redundancy-free descriptions of protocol transactions, scenarios are sample executions. The problem addressed by Alur *et al.* is to automatically synthesize a model when the number of scenarios is inadequate. In contrast, we take the set of flows F itself to be the description of system transactions; thus, assertion violation in our refinement loop identifies inconsistency between the flows in F but no automatic repair mechanism. The analogous repair problem in our setting, *e.g.*, repairing protocol model by supplying a completely missing flow, would be unsolvable: if a flow is missing it cannot in general be inferred from other flows. Finally, using temporal assertions to capture behaviors of complex distributed protocols is hard and requires significant expertise in formal logic, making it unusable for architects. In our experience, flow-based design capture is more closely aligned to industrial design development.

Protocol	Type	No. of Flows	Murphi Model LOC	State Annotation LOC
German	Cache Coherence	4	600	30
Flash	Cache Coherence	10	1400	100
Intel 1	Cache Coherence	36	5000	200
Intel 2	Cache Coherence	43	6500	200
Intel 3	Bus Lock	3	1600	120
Intel 4	Credit Management	3	200	20

TABLE I: Some protocols synthesized with our tool

VIII. Conclusion

We formalized the notion of transaction message flows, and provided a method to synthesize executable models. We also showed that flows contain strictly richer semantic information than models: while generating models from flows is easy, extracting flows from models is intractable. To our knowledge, ours is the only technique for automated protocol modeling that has found consistent use in industry. The cache coherence protocols analyzed via Flows2HLM are some of the most complex ones to ever undergo formal verification. Note from Table I that the industrial cache coherence protocols have about an order of magnitude more flows than German, Flash, etc. that constitute representative benchmarks for state-of-the-art automatic formal verification techniques.

The problem of synthesizing models arose out of the need to analyze these highly complex industrial protocols early in the design life-cycle. Previous work [11], [12] addressed this problem by mining invariants from transaction-level descriptions to facilitate formal verification of protocol models at this scale. Nevertheless, a limiting factor for its practical adoption was the complexity of creating these formal models. This bottleneck, together with the observation that flow diagrams in architecture documents often represent "authoritative" source of protocol descriptions for designers and validation engineers, motivated the approach presented here. The results of this paper, together with the previous results [11], [12], suggest that flows provide a powerful and efficient method for modeling, analysis, and understanding of protocols.

In future work, we plan to exploit flows further in protocol modeling and analysis. One application is generating test harness for RTL simulation, *e.g.*, flows can be used to encode environment behaviors when exercising the RTL implementation of an agent. Another future work is to use repair techniques to fix missing annotations in user-provided flows, perhaps through a user-guided refinement loop. Finally, tabular specifications, although difficult for architects, are useful for downstream RTL designers; it will be interesting to explore synthesis of tabular specifications from flows.

References

[1] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation," in *21st International Conference on Computer-Aided Verification*, 2009.

[2] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, 2001.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Communications of the ACM*, vol. 5, no. 2, 2010.

[4] D. James, T. Leonard, J. O'Leary, M. Talupur, and M. Tuttle, "Extracting Models from Design Documents with Mapster," in *Proceedings of the 27th Annual ACL Symposium on Principles of Distributed Computing (PODC 2008)*, R. A. Bazzi and B. Patt-Shamir, Eds., 2008, p. 456.

[5] S. German, "Tutorial on Verification of Distributed Cache Memory Protocol," in *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, A. J. Hu and A. K. Martin, Eds., 2004, http://www.cs.utah.edu/~ganesh/presentations/fmcad04_tutorial2/german/steven-tutorial.pdf.

[6] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, 1976.

[7] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. 3, no. 2, pp. 125–143, 1977.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," in *IEEE International Conference on Computer Design*, 1992.

[9] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, 1997.

[10] ITU-TS Recommendation Z.120, "Message Sequence Chart (MSC) Annex B: Algebraic Semantics of Message Sequence Charts, ITU-TS, Geneva," 1995.

[11] M. Talupur and M. R. Tuttle, "Going with the Flow: Parameterized Protocol Verification using Message Flows," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*, A. Cimatti and R. B. Jones, Eds., 2008.

[12] J. W. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol Verification using Flows: An Industrial Experience," in *Proceedings of the 9th Internation Conference on Formal Methods and Computer Aided Design (FMCAD 2009)*, A. Biere and C. Pixley, Eds., 2009, pp. 172–179.

[13] A. Pnueli, S. Ruah, and L. Zuck, "Automatic Deductive Verification with Invisible Invariants," in *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, T. Margaria and W. Yi, Eds., 2001.

[14] J. Kustin and et. al, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ICSA 1994)*, 1994.

[15] M. Talupur, S. Ray, and J. Erickson, "Flows and Murphi Models for German and Flash Protocols," 2014, See URL http://www.cs.cmu.edu/~tmurali/flow_examples.

[16] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. D. Jeffords, "Tools for constructing requirements specifications: the SCR Toolset at the age of nine," *Computer Systems: Science & Engineering*, vol. 20, no. 1, 2005.

[17] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[18] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: Specifying Protocols with Concolic Snippets," in *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, 2013.

[19] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing Finite-state Protocols from Scenarios and Requirements," 2014.