Effective and flexible SMT-streamlined software model checking

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Sepideh Asadi

under the supervision of
Prof. Natasha Sharygina

April 2023

Dissertation Committee

Prof. Kenneth McMillan	University of Texas at Austin, USA
Prof. Roberto Sebastiani	University of Trento, Italy
Prof. Evanthia Papadopoulou	Università della Svizzera Italiana, Switzerland
Prof. Kai Hormann	Università della Svizzera Italiana, Switzerland

Dissertation accepted on 6 April 2023

Natasha Sharygina

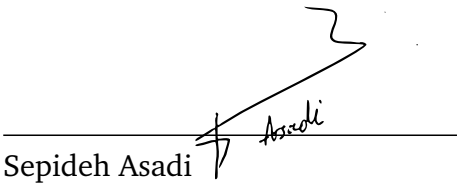
Research Advisor

Prof. Natasha Sharygina

PhD Program Director

Prof. Walter Binder and Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.



Sepideh Asadi
Lugano, 6 April 2023

To my best friend and beloved husband, Masoud.

I also dedicate this work to all brave women of IRAN who stand for freedom, equality, and peace.

"Women, Life, Freedom"

Abstract

Formal verification by model checking is an award-winning (Turing award, 2007) technology to verify systems exhaustively and automatically in order to increase the degree of confidence that a program would perform correctly. Symbolic verification techniques such as bounded model checking (BMC) supported by highly efficient Boolean satisfiability (SAT) solvers have advanced the state of the art in model checking substantially. Nevertheless, scalability issues persist in these techniques due to the state-space explosion problem.

The goal of this thesis is to address such scalability issues of model checking of software. We base the proposed solutions on Satisfiability Modulo Theories (SMT) as a logical representation of programs. Although SMT reasoning framework is one of the most successful approaches to verifying software in a scalable way, it poses new challenges to verification. The main issue of SMT is that it offers little direct support for adapting the constraint language to the task at hand. Another issue with SMT encoding of programs is that the light-weight theories are often imprecise. This means if a program is encoded in SMT, it may not be a ready-to-use solution for verification; it would require various (sometimes major) tuning to find a level of abstraction suitable for efficient symbolic model checking. This thesis addresses such challenges and trade-offs.

This thesis draws together a range of techniques and combines them in a novel way so that not only improves scalability but also enables flexibility by allowing different levels of SMT abstractions. A new verification technique is proposed based on SMT-based model checking that verifies programs incrementally, reusing the computational history of a program, namely function summaries. In addition to the provided reusability supported by the incremental approach, the technique allows adjusting the precision with different levels of SMT encodings. Overall, this thesis extends the state of the art by (i) designing a verification technique that interleaves the SMT reasoning with model checking, (ii) introducing an incremental verification approach where it reuses the computation history of related verification tasks, and (iii) exploring the trade-offs between program encoding *precision* and solving *efficiency* using a novel theory-aware ab-

straction refinement approach that models a program using the lightest theories and strengthens locally on demand if the precision is not sufficient for verification needs.

This thesis considers two application domains: (i) efficiently verifying a single program with a sequence of safety properties (i.e., no assertion failure in the program), and (ii) incrementally verifying program revisions with respect to the same safety property. The effectiveness of the proposed approaches has been validated by developing two new SMT-based model checking frameworks: HiFrog and UpProver. These frameworks have been integrated into the interpolating SMT solver OpenSMT. This thesis presents real-world model checking experiments using the proposed verification tools, demonstrating the viability and strengths of the techniques.

Acknowledgements

This thesis represents the culmination of a six-year-long memorable journey, where hard work and pure joy have been constantly intertwined. I believe that these were the years of my greatest personal and professional growth, and I feel a strong desire to express my gratitude to all those who have contributed to this journey in any way, whether scientifically or privately.

First and foremost, I would like to thank my PhD advisor, Natasha Sharygina, for her guidance, support, and supervision throughout my doctoral studies. She has always been full of insights and vision that have not only advised me in my scientific path but also in my life. A true role model, she believed in me even when I did not. I am honored to have had the opportunity to work in her research group at USI Formal Verification and Security Lab.

Throughout my PhD studies, I had the privilege of collaborating with and learning from many wonderful and brilliant people, for which I am sincerely grateful. I would like to extend a special thanks to Prof. Grigory Fedyukovich, whose insights were invaluable when I felt stuck in my work. I would also like to express my sincere gratitude to Dr. Antti Hyvärinen and Dr. Martin Blicha for their valuable contributions to my work and for our many insightful and enjoyable discussions. I truly enjoyed working with them and learning from them, and much of my work would not have been completed without their contributions.

Six years ago, studying computer science at Università della Svizzera italiana (USI Lugano) was just a dream for me. USI Lugano is a great place to be, and I enjoyed both the location and the people there, from various research groups. My colleagues were supportive, helpful, and good people to be around, especially Dr. Leonardo Alt, Dr. Matteo Marescotti, Rodrigo Otoni, Konstantin Britikov, and Masoud Asadzadeh. I would also like to thank my collaborators, Professor Hana Chockler and Dr. Karine Even-Mendoza, for our many insightful discussions and the opportunity to visit them at the University of King's College London.

I am deeply grateful to the reviewers of this thesis and my papers, who took the time to thoroughly analyze this work and provide valuable feedback and insightful questions. In particular, I would like to express my appreciation to

Professor Kenneth McMillan for his insightful feedback on theory refinement. It is fascinating to note how even the smallest observation, insight, recommendation, or criticism can easily turn into great guidance.

I am also grateful for the generous financial support provided by the Swiss National Science Foundation (projects 200021_185031. I also acknowledge my funding source from USI Lugano.

This entire journey would have been unimaginable without my husband, Masoud. I can never forget that he moved to Lugano because of me and believed in me every step of the way. The transition from undergraduate studies in Electrical Engineering to graduate studies in Computer Science could not have been made easy without his unwavering support. He has always been excited to hear about my work and has always been proud of me. Masoud is my source of energy, my complement, and my best friend.

I would like to conclude this by expressing a few words in Persian, to let my parents and sisters know how grateful I am for their lifelong unconditional love and support, for raising me with true values, and for giving me wings to fly freely. I am what I am today thanks to them!

مامان و بابا و خواهرای عزیزم سیمین و سارا، از صمیم قلبم از محبت و حمایتی که در زندگیم به من کردید ممنونم. از شما ممنونم که به من آموختید که ارزش های واقعی را بدانم و انسان خوبی باشم. ممنونم که هیچوقت جلوی من را نگرفتید، اما به من بال دادید و اجازه دادید آزادانه پرواز کنم. من به لطف شما هینی هستم که امروز هستم!

Sepideh Asadi, April 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preface

Nowadays with the prevalence, reliance, and ubiquity of software in our everyday lives, the importance of ensuring the correctness of software gets more acute. In practice, software quality usually does not refer to *total correctness* of a design, since ensuring the absence of all bugs is too expensive for most applications. In contrast, a guarantee of the absence of specific flaws is achievable and is a good metric of quality. *Formal verification* by model checking [Clarke and Emerson, 1981; Queille and Sifakis, 1982] exhaustively checks whether a given property (e.g., assertions in the program code) is satisfied by the program for all possible inputs (not just a limited set of inputs as in traditional testing).

Model Checking, an award-winning (Turing award, 2007) paradigm, is the collection of strategies that provides a formal and algorithmic way to verify the program in a fully automated manner. These approaches present a cost and a benefit: the system and its specification (properties) are modeled in restricted forms, but in exchange for this, proofs and counterexamples are produced automatically. The main issue for model checking is its high computational burden, which is known as the *state-space explosion* (SSE) problem.

Although formal verification is widely thought to be one of the grand challenges (in general it is undecidable) for computer science, it has benefited from a myriad of solutions to combat the SSE problem. This chapter gives an overview of the state-of-the-arts in verification by model checking which are proposed for mitigating SSE, and highlights their limitations that are the motivation for the research conducted. We then outline the contributions of the thesis as the solutions to some of the limitations.

1.1.1 Symbolic model checking

Symbolic model checking was first introduced by McMillan in his thesis [McMillan, 1993]. This breakthrough technique represented the state space *symbolically* by using binary decision diagrams (BDD) [Bryant, 1986] as a data structure to store and manipulate a set of reachable states implicitly, instead of doing it explicitly proposed by the original model checking approach. The introduction of symbolic representation permitted verifying properties of finite state systems that had more than 10^{20} states [Burch et al., 1990]. However, the use of BDDs does not prevent a state explosion in all cases. Even though BDD-based symbolic model checking was integrated into the quality assurance process of several major hardware design companies, the main bottleneck was its exponential growth and large memory usage that would hinder its wide usage to different applications, such as software verification.

An advanced variant of symbolic verification technique was inspired by improvements in the constraint-solving techniques with the ability to efficiently solve propositional satisfiability problems (or SAT, the "classic" NP-complete example) [Biere et al., 2009]. SAT-based *bounded model checking (BMC)* introduced in [Biere et al., 1999] allows us to represent a large set of program states in a more compact way than previous techniques. BMC is mainly used for error detection instead of an approach for full correctness proof. The key idea is to search for counterexamples of bounded length k. The problem is translated to a Boolean formula, such that the formula is satisfiable iff there exists a counterexample of length k. If no such counterexample is found, k is increased. This process continues until either a longer counterexample is found, a *predetermined upper bound* is reached, or the problem possibly becomes intractable due to hitting the time or memory limits. In practice there is often just a time limit and the tool reports the depth that it checked within that time.

With the improvements of *Satisfiability Modulo Theories (SMT)* solvers built over efficient SAT solvers, there has been a renewed effort in developing logic-based model checking tools that focus on satisfiability and scalability [Ramalho et al., 2013; Armando et al., 2009; Cordeiro et al., 2012; Ganai and Gupta, 2006; Gadelha et al., 2018; Xu, 2008]. SMT solvers can determine the satisfiability of first-order formulas with respect to various background theories, which enable them to reason about different data types and data structures. With SMT solvers it is possible to use more natural translations for systems, have fewer limitations on specifications, and often still have significant performance gains over SAT-based tools. In principle, SMT can be interpreted as an extension of propositional SAT to first-order logic. In order to remain decidable, first-order theories

are typically restricted to decidable fragments, e.g. quantifier-free fragments. Examples of the state-of-the-art SMT solvers are Z3 [de Moura and Bjørner, 2008], BOOLECTOR [Brummayer and Biere, 2009a], CVC5 [Barbosa et al., 2022], YICES [Dutertre, 2014], MATHSAT SMT SOLVER [Cimatti et al., 2013], SMTINTERPOL [Christ et al., 2012], OPENSMT [Bruttomesso et al., 2010]. The prosperity of research on SAT- and SMT solvers is also supported by a wide range of competition activities in the community [Bartocci et al., 2019].

Constrained Horn clauses (CHC) as a representation of software is another trend in the application of logic to symbolic verification. Originally [Grebenshchikov et al., 2012] proposed a unified format in the language of logical constraints to capture various verification tasks (e.g., safety, termination, and loop invariants computation) from different domains such as transition systems, functional programs, procedural and recursive programs, concurrent and distributed systems [Gurfinkel and Bjørner, 2019; McMillan and Rybalchenko, 2013; Gurfinkel et al., 2015; Kahsai et al., 2016; Dietsch et al., 2019]. In general, the satisfiability of CHC modulo theory of arithmetic is undecidable. Thus, solving them is a very complex task on its own.

Symbolic model checkers based on Property Directed Reachability (PDR/IC3) [Bradley, 2011; Eén et al., 2011] is known for finding safe inductive invariants. The key idea is to use a SAT/SMT solver to iteratively prove or disprove the reachability of proof obligations, i.e. states that are guaranteed to reach an error state. The iteration continues until the conjunction of all the learned IC3-lemmas is a formula strong enough to block any other proof obligation inductively, making it a safe inductive invariant [Beyer and Dangl, 2019; Marescotti et al., 2017]. The power of PDR algorithm is in how it is capable of constructing the invariant incrementally, instead of the monolithic approaches. However, for the class of unbounded systems, the inferred quantified invariant is not in the fragment of first-order logic, and the unbounded check may not succeed using current SMT solving techniques.

Since SAT-based approaches operate on the pure bit-level representation by converting formulas into Boolean circuits, it is not expressive enough to model most real-world problems, especially large-scale software. For this reason, the solving process at propositional level is often computationally time and resource-demanding, and sometimes not tractable at all. Compare to SAT, SMT offers a much more expressive modeling language. The first-order logic over different theories allows us to express the software with a natural and close formulation to programming language statements. These formulations can also be significantly more compact than propositional ones, since in bit-precise encoding all the arithmetic operations need to be hard-coded into logical circuits, resulting in

some cases in quadratic growth in the representation size. Therefore, encoding problem instances with first-order theories could be handled more efficiently by SMT solvers. SMT is a prospering research topic for many researchers around the world, both in industry and academia. For instance Z3 SMT solver is part of crypto blockchain verification utilities, and LLVM toolsets [Bjørner, 2018]. The companies Intel [*Decision Procedure Toolkit*, 2007], Meta [Dill et al., 2021], Microsoft [Ball et al., 2011], Ethereum Foundations [Grishchenko et al., 2018], and Amazon [Backes et al., 2018] routinely apply decision procedures for verifying their product. SMT solvers also play a central role in verifying compilers [Böhme et al., 2010], operating systems [Nelson et al., 2017], distributed systems [Hawblitzel et al., 2015], and protocol designs Padon et al. [2016].

1.1.2 Abstraction in model checking

The rise of various techniques for symbolic reasoning made it easier for the model checkers to establish over-approximations and under-approximations of programs. As a result, various solutions of abstraction have taken a big step in program verification [McMillan and Amla, 2003; Cook et al., 2005; Colón and Uribe, 1998; Graf and Saïdi, 1997]. The key insight behind over-approximations used in symbolic model checking is that they allow representing a large set of program states in a more compact way. Abstraction techniques reduce the state space of the system by mapping the set of states of the original system to an abstract one, and a smaller set of states in a way that keeps the necessary relevant information about the system required for proving the properties.

As the level of abstraction gets higher, we run the risk of missing some important details of the system's model. When model checking of the abstract model fails, the reported counterexamples might not correspond to any concrete counterexample of the actual system. This is called a *spurious error*. To recover from the spurious errors in verification, the abstraction is accompanied by a refinement process. In this regard, the paradigm of abstract-check-refine was first proposed by Kurshan [Kurshan, 1994]. The basic idea is to build an abstract model, then check the intended property, and if the check fails, refine the model and start over. In other words, the abstraction refinement techniques are to create a new abstract model which contains more detailed information in order to refute the spurious counterexample. This process is iterated until the property is either proved or disproved. It is known as the Counterexample-Guided Abstraction Refinement framework, or CEGAR for short [Clarke et al., 2000]. There are number of successful techniques based on abstraction refinement [Heizmann et al., 2009; Lahtinen et al., 2015]. However, in abstraction-based approaches finding the

right abstraction granularity remains an open *challenge*. The abstraction level should have enough detail to allow verification but remain at the manageable level.

1.1.3 Interpolation-based model checking

Craig (binary) interpolation [Craig, 1957] is widely acknowledged as one of the fundamental approaches in computing abstraction. For any two formulas whose conjunction is unsatisfiable, there exists an interpolant in the sense of the following definition: given a partition of a set of clauses into a pair of subsets (A, B), and a proof by resolution that the clauses are unsatisfiable, one can generate an interpolant in linear time[Pudlák, 1997]. The Craig Interpolant for the pair (A, B) is a formula I with the following properties: (i) A implies I, (ii) $I \wedge B$ is unsatisfiable, and (iii) the symbols in I occur in both of A and B.

Interpolation and bounded model checking can be synergistically combined to exploit their individual strengths. With interpolation, an over-approximation is generated such that it then can be used efficiently in symbolic model checking. This also provides the possibility for performing unbounded model checking. In [McMillan, 2003a], McMillan presents the use of interpolants in order to obtain a complete model checker based on a BMC-like reasoning engine. [McMillan, 2004] also showed how the interpolation algorithm can be tuned to work in SMT context. The algorithm computing partial interpolants for nodes in the resolution proof is extended to the nodes representing theory lemmas (logical tautologies with respect to the given background theory). For these nodes, a theory-specific interpolation algorithm is required and McMillan gave examples of these for the theory of linear arithmetic and the theory of equality and uninterpreted functions. More work followed introducing various algorithms for computing interpolants in SMT context, including [Kapur et al., 2006; Rybalchenko and Sofronie-Stokkermans, 2007]. However, they had various limitations such as dealing with a specific theory only (e.g. bit-vectors) or relying on algorithms that became obsolete with the progress in SMT solvers.

Interpolants are used in several different ways in symbolic model checking. One of the successful applications of interpolants is in *function summarization* [Henzinger et al., 2004; Sery et al., 2011; Albarghouthi et al., 2012b]. An over-approximating *function summary* enables the reuse of information among verification runs. Summaries are extracted using Craig interpolation after a successful verification run for one property and used as a light-weight replacement of the precise encoding of the corresponding functions while verifying other properties. Function summarization will be described in more detail in the next chapter

as it constitutes one of the basic abstraction mechanisms used for the development of new techniques.

The existing leading software model checking algorithms such as PDR-, CEGAR- or BDD-based model checkers which allow sound and complete model checking are not guaranteed to terminate for programs with infinite state space. In contrast, symbolic bounded model checking that under-approximates the set of reachable states has shown to be efficient for quick finding of counterexamples, as opposed to computing an unbounded proof [Cook et al., 2020; Mentel et al., 2021]. Even though plain BMC is incomplete (it can only show existence of counterexamples), it is a major building block of many complete model checking algorithms, such as interpolation-based model checking [McMillan, 2003b; Vizel and Grumberg, 2009; Vizel and Gurfinkel, 2014], and induction-based unbounded model checker, (e.g., [Sheeran et al., 2000; Donaldson et al., 2010]) and more. This approach to symbolic model checking, i.e., bounded model checking (BMC) is the focus of this dissertation. The next section describes the open challenges that this thesis aims to tackle and overviews the solutions of this research work.

1.2 Challenges and Contributions

Verifying software systems is a longstanding research goal. Throughout the past four decades, hardware symbolic verification has been advancing, whereas verification of large-scale software remains an open challenge. One of the main reason for the success of hardware verification is the physical constraints in hardware design that leads to better modularization of hardware designs. Software is more complicated: it contains a variety of scalar and non-scalar data types: integers, reals, arrays, collections, as well as user-defined data types and recursive functions. Furthermore, software may contain a complex heap data structure which destroys modularity by introducing a single global data structure that every line of code can access. One can only regain modularity at a high cost or at the price of losing automation, for example using separation logic in the specifications [O'Hearn et al., 2001; Lei et al., 2019]. Hence, the classical finite-state model checking approaches that are performing well in the hardware domain are not applicable to software.

Without detracting from the merits of the cutting-edge techniques of automated formal verification, there is a further demand for new methods to make software verification workflow more efficient. This research thesis addresses the following challenges that are hindrances to a broader application of symbolic model checking techniques in verifying software: (P1) expensive bit-precise rea-

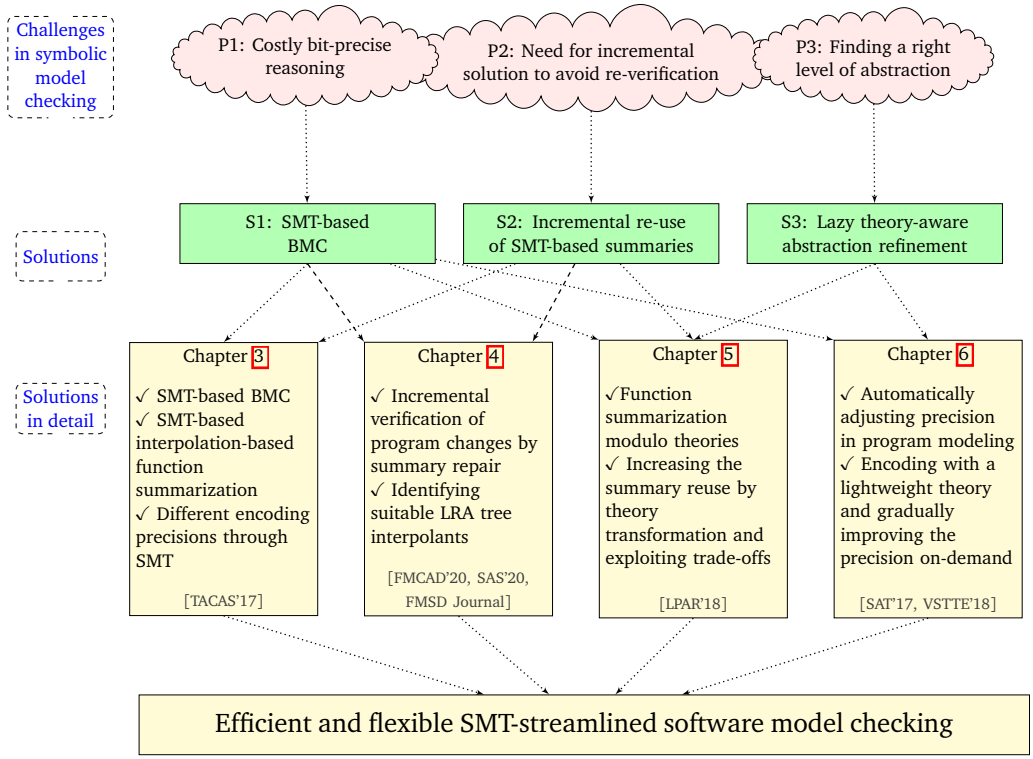


Figure 1.1. Main contributions of the dissertation addressing three unresolved problems in symbolic model checking. The solutions proposed in chapters 3, 5, and 6 target the verification of multi-properties in a single program. The solution proposed in chapter 3 targets verifying several program revisions w.r.t a single property.

soning, (P2) need for incremental solution to avoid re-verification for closely related programs, and (P3) finding a right level of abstraction suitable for efficient symbolic model checking. These challenges are naturally related to the complexity problems within the model checking paradigm. To address these challenges, this work enables the on-demand use of SMT technology and ultimately improves the efficiency of symbolic model checking.

The overall contribution of this dissertation is a collection of novel verification techniques that enable flexibility in symbolic verification and enhance the efficiency of software verification. Figure 1.1 depicts the classification of the individual contributions with respect to the open challenges in symbolic model checking. This dissertation proposes three corresponding solutions S1, S2, and S3 that address the three above-mentioned challenges P1, P2, and P3 respectively: (S1) SMT-based BMC, (S2) incremental re-use of SMT-based summaries

and (S3) lazy theory-aware abstraction refinement. The proposed techniques are highly interchangeable and have a degree of compatibility and interoperability that allows the combination of diverse techniques to obtain a superior SMT-based solution. Even though the solutions are presented in four chapters as individual and self-contained solutions, they are naturally related and one solution can benefit from the others and ultimately are integrated into a unique framework on-demand, as it was done, for example, in the model checkers developed as part of this work: HiFrog [Alt, Asadi, Chockler, Mendoza, Fedyukovich, Hyvärinen and Sharygina, 2017] and UpProver [Asadi et al., 2020b].

The solution S1, discussed in Chapter 3, proposes to prevent expensive bit-precise reasoning by building bridges between model checking and SMT solving. The challenge arises from the fact that SAT/SMT solvers are being used as a black box in model checkers, preventing the utilization of their full potential. The solution S1 suggests the seamless interaction between the model checker and the underlying SMT solver. This integration of the SMT solver into the BMC is manifested on the one hand in the artifacts required for computation of approximations, i.e., interpolation-based function summarization, and on the other hand – in the refinement of program models and checking satisfiability. The interleaving of SMT with BMC has been validated by implementing an integrated SMT-based verification framework HiFrog and experimented for efficient verification of a single program with a sequence of safety properties (i.e., no assertion failure in the program). Notably, the proposed solution opens new opportunities for enhancing scalability and forms the basis for the other solutions which will be described in Chapters 4, 5, and 6.

The solution S2 is built on top of the solution S1, in order to reuse computation history across different verification tasks that are closely related, namely program revisions. Chapter 4 investigates if it is beneficial to summarize a function precisely, or at least approximately, to help to avoid re-verification of the same verification tasks over and over again. This solution proposes to employ SMT-based function summaries across program versions to enhance the efficiency of model checking. The effectiveness of S2 has been validated by implementing a novel tool (on top of S1), UpProver, for the purpose of incremental verification of program revisions with respect to the same safety property and will be described in Chapter 4.

The solution S3 deals with the problem of rigid program modeling restricted to one level of precision. In other words, the existing SMT-based model checkers have no control whatsoever on the modeling precision to adjust the encoding if it is not suitable in the initial phase. The proposed approach focuses on finding a right level of abstraction to manage the scalability and precision. To explore

such trade-offs, this thesis proposes a lazy approach for theory-aware abstraction refinement which will be discussed in Chapters 5 and 6: the former focuses on adjusting the precision through transforming function summarization across theories, and the latter focuses on devising a solution that models a program using the lightest possible (i.e., less expensive) theories that suffice to verify the safety properties and strengthens locally on demand if the precision is not satisfying the verification needs. Both approaches have been validated by major implementation in HiFrog and will be discussed in Chapters 5 and 6.

The following sections discuss the challenges and the corresponding research solutions in more detail.

1.2.1 SMT-based BMC by means of function summarization

As described before, BMC is one of the most successful formal methods in academia and industry. As every solution creates new problems, the application of BMC to software also poses new challenges. The majority of the state-of-the-art techniques in software BMC are restricted to bit-precise reasoning. In effect, SAT-based BMC translates a program into a propositional Boolean formulas that are then fed directly to a SAT solver. The main issue of such approaches is that they often do not scale to bigger programs as the size of the Boolean formulas grows rapidly in the presence of large data-paths. Furthermore, when the verification conditions are turned into Boolean logic, the high-level program-specific information is lost, preventing potential optimizations to reduce the program state space. Overall, due to the these complexity of bit-precise encoding, SAT-based BMC techniques are not efficient especially in the context of large scale systems.

As a complementary approach, SMT offers a wide variety of light-weight theories based on first-order-logic that enhances expressiveness of program encoding considerably. However, there is a foundational issue in software model checking based on decision procedures. In particular, with a few notable exceptions (see, e.g., [Ramalho et al., 2013; Armando et al., 2009; Cordeiro et al., 2012; Ganai and Gupta, 2006; Gadelha et al., 2018; Xu, 2008]), the majority of research efforts concentrate on separate development of sophisticated model checkers on one hand and on impressively performing decision procedures for the logical models on the other hand. The existing SMT-based symbolic model checking solutions use SMT solvers as black-box and the research developments of SMT-solving and model checking are quite disconnected and are traditionally being developed by different scientific communities. This complexity of the interactions between model checker and SMT solver is related to the sheer magnitude of the task, as both fields are highly complex alone and require a significant

amount of in-depth knowledge in both theory and practice. Lacking fruitful interaction of model checker with SMT solver gives verification techniques little chance of further optimization and flexibility.

Another hindrance to the scalability of classic BMC is lack of modularity in the way the formula is constructed. The classical BMC approach encodes a program into a logic as a *monolithic* formula by full inlining and unwinding of the source code through a pre-determined bound. By inlining the entire function bodies, the resulting BMC formula becomes too large to handle. As a consequence, when a program is supplied with a sequence of pre-defined assertions, full re-verification of the formula from scratch for each property (properties could be represented as assertions in the program code) becomes prohibitively expensive. As this thesis suggests, searching for reusable specifications between verification runs is crucial for scalability. Unlike CEGAR- or PDR-based model checking, BMC is not typically driven by maintaining a safe inductive invariant. Hence, once the verification run is successful, a reusable specification should be generated. The works [Sery et al., 2011, 2012b] address this issue of finding a reusable computation history formalized for the SAT reasoning. They proposed a solution that exploits the fact that the bounded safety of the program is indicated by the unsatisfiability of the BMC formula. From the proof of unsatisfiability, their algorithm discovers over-approximating constructs, i.e., *SAT-based function summaries* that gather all important information of the function's behavior relevant to prove the bounded safety. Even though this idea of extracting and reusing SAT-based function summarization has been shown useful in BMC, the choice of bit-precise encoding with the direct use of a SAT solver, sometimes causes major problems due to the large sizes of the corresponding formulas. The large formulas often are expensive to solve making the overall verification procedure impractical in such cases.

This thesis proposes employing first-order solver to construct more efficient summaries, without sacrificing the precision of program encoding. Compared to the earlier propositional tool FUNFROG [Sery et al., 2011, 2012a], the SMT summaries in the proposed framework are smaller and more efficient in verification. They are also often significantly more human-readable, enabling their easier reuse.

Motivated by constructing function summaries of a better quality, we formulated two research questions that this dissertation addresses:

RQ1 How SMT can be used in summarization, one of the successful approaches making iterative model checking scale well?

RQ2 How to effectively reuse SMT-based function summaries for verification of

a program with multiple properties?

Research contribution. Chapter 3 of this thesis proposes a verification approach in which bounded model checking is interleaved with SMT reasoning for various computational tasks. In particular, to prevent expensive bit-precise reasoning the proposed verification framework leverages the natural and succinct encodings of SMT, SMT solving, and various SMT interpolating procedures for verification and program abstraction. The key distinguishing feature of the proposed verification framework is its capacity to re-use SMT-based summaries across properties in a single program. The proposed solution, SMT interpolation-based function summarization is used as a means of incremental verification based on the structure of the program. Based on the natural reliance of the summaries on the safety properties, the proposed framework performs function summarization iteratively, i.e., checking each safety property at a time and refining the imprecise summaries on demand. The proposed verification framework supports three encoding precisions through SMT: equality with uninterpreted functions (*EUF*), linear real arithmetic (*LRA*), and propositional logic. This work advocates the necessity to offer various encoding options to the user. Therefore, in addition to the provided SMT-level light-weight modeling and the corresponding SMT-level summarizations supported by the proposed incremental verifier, the framework would allow adjusting the precision and efficiency with different levels of encodings. We have developed HiFrog, a summarization-based bounded model checker that uses interpolants to represent function summaries. The extensive evaluations on the practical impact of different SMT precisions on model checking demonstrate that the use of SMT speeds up the calculation and allow scaling the model checking applications to verify large C programs.

The results of this work have been published in [Alt, Asadi, Chockler, Mendoza, Fedyukovich, Hyvärinen and Sharygina, 2017] and [Alt, Hyvärinen, Asadi and Sharygina, 2017], and are presented in Chapter 3.

1.2.2 SMT-based incremental verification of program revisions

As a continuation of studies described in Section 1.2.1, this thesis further studies the applicability of the proposed SMT-based function summarization approach to another application domain, i.e., verifying frequent changes that can occur during software evolution. In fact the programs are often developed in an iterative way, by successive improvement of the last version. However, most software verification techniques are not designed to support sequences of program versions, and they force each changed version to be verified from scratch which

makes re-verification computationally demanding or even impractical. Contrary to verifying the programs in isolation, incremental verification is an approach that aims to reuse the invested efforts between verification runs and consequently can achieve speedup in the subsequent analysis of the other versions. While this line of research is promising, and already included results on SAT-based incremental update checking [Sery et al., 2012b; Fedyukovich et al., 2013], bit-precise reasoning poses yet another scalability challenge. This work addresses this problem and produces an SMT-based solution for efficient analysis of a program after a change in the code.

Then further study concentrates on finding suitable interpolation algorithms to support the incremental verification of program revisions. There exist various Craig (binary) interpolation algorithms for the theory of LRA with flexibility in strength and size which are important for efficient over-approximation and convergence of program verification [McMillan, 2004; Alt, Hyvärinen and Sharygina, 2017; Blicha et al., 2019], and its practical success has motivated a line of research on tree interpolation [Rümmer et al., 2013; Blanc et al., 2013; Heizmann et al., 2010a; McMillan and Rybalchenko, 2013; Gupta et al., 2011]. However, applications requiring the *tree interpolation property* (e.g. incremental verification of program revisions) have limited choice for LRA interpolation algorithms. we are interested in *efficient* tree interpolation (i.e., without solving new SMT or SAT problems as in [Henzinger et al., 2004].)The goal of this work is to investigate Craig binary interpolation algorithms in LRA and propositional logic to guarantee tree interpolation property which is suitable for incremental verification of program revisions.

The study concentrates on the following two research questions:

RQ3 Is the use of SMT instead of SAT beneficial for incremental verification of real-world program revisions? If so, how the reuse SMT-based function summaries between program versions could be effectively enabled?

RQ4 Which interpolation algorithms can be used to support update checking? In other words, lifting function summaries from one program version to another version requires generalizations of Craig binary interpolants, namely tree interpolants. This means that the study of SMT-interpolation algorithms is required to understand how they can adequately guarantee the computation of tree interpolants.

Research contribution. Chapter 4 of this thesis proposes a model checking technique for incrementally verifying software while it is being gradually changed.

The goal is to make software analysis more efficient and scalable by reusing invested efforts between verification runs. This research suggests to reuse or adapt SMT-based function summaries across program versions and to exploit first-order theories available in SMT solvers, thus enabling flexibility. In addition to the provided SMT-level light-weight modeling and the corresponding SMT-level summarizations supported by the proposed incremental verifier, the proposed BMC framework allows adjusting the precision and efficiency with different levels of encodings. This approach not only allows the reuse of summaries obtained from SMT-based interpolation, but also provides an innovative capability of repairing them automatically and using them in the subsequent verification runs. The extensive experimental evaluations demonstrate that the proposed algorithm achieves an order of magnitude speedup compared to prior approaches. The further study investigates the known SMT interpolation algorithms in linear real arithmetic and classify them based on whether they guarantee the tree interpolation property. This work extends the state-of-the-art LRA interpolation algorithms by identifying the Craig binary LRA interpolation algorithms that can be used as a basis for tree interpolation and thus are suitable for update checking. The details of this contribution have been published in [Asadi et al., 2020b] and [Asadi et al., 2020a], and FMSD journal [Asadi et al., 2023], and are presented in Chapter 4.

1.2.3 Model checking by theory transformation

Another track of the thesis builds on the observation that different theories in SMT have interesting properties with respect to over-approximating program behavior. While the most precise encoding in case of model checking is to express the algorithm as a bit-precise propositional formula, this thesis suggests that modeling a verification problem with less expensive theories of SMT reduces the verification cost and improves the scalability.

The previous tracks concentrate on building the incremental SMT-based BMC. However, the use of SMT in BMC poses new challenges. SMT offers little direct support for adapting the constraint language to the task at hand. The main issue with SMT encoding of programs is that the light-weight theories are often imprecise. This means if a program is encoded in SMT, it may not be a ready-to-use solution for verification and might introduce a spurious result; it would require various (sometimes major) tuning to be reliable. Exploring such trade-offs between precision and a right level of abstraction is a challenge in program verification to which this track provides a solution.

This work addresses the problem of finding a suitable level of abstraction in

modeling the program that is sufficiently accurate to prove the correctness of programs and not to be expensive to reason on. Given that SMT offers light-weight theories, it is of utmost importance to allow flexibility in tuning the modeling precision to find a *balance* in theory encoding. This challenge marks a salient motivation for this research. Consequently, we formulated the following research questions:

RQ5 To manage the scalability how can one find a safe over-approximation of the program with SMT that is sufficiently precise but not too expensive to reason on? In other words, how can one manage trade-offs between modeling precision and performance of model checkers?

RQ6 How to deal with undesirable false alarms introduced through the use of SMT abstractions? How to extract useful information from unsuccessful verification runs to guide the consecutive runs?

Research contribution. The proposed approach in Chapter 5 alternates precision of the program modules on demand. The idea is to model a program using the lightest possible (i.e., less expensive) theories that suffice to verify the desired property. This work employs safe over-approximations for the program based on both function summaries and light-weight SMT theories. If during verification it turns out that the precision is too low, the proposed approach lazily strengthens all affected summaries or the theory through an iterative refinement procedure. The resulting summarization framework provides a natural and light-weight approach for carrying information between different theories. In order to make the full advantage of already generated abstract models we propose a technique for tuning the abstractions via transformation of one theory to another one. Designing the *theory interface* enables migrating information among formulas in different theories. An experimental evaluation with a bounded model checker for a subset of C program on a wide range of benchmarks demonstrates that the proposed technique scales well, often effortlessly solving instances where for example the state-of-the-art model checker CBMC runs out of time or memory.

The results of this work have been published in [Asadi et al., 2018] and are presented in Chapter 5.

1.2.4 Counterexample-guided theory-aware refinement

The last track of the thesis builds on the observation that different statements of the program may exhibit different degrees of precision. Thus mixing different

abstraction layers would provide more granularity and flexibility in the program modeling, as this work suggests, this would increase the scalability of model checking to a great extent. Motivated by finding a balance in program modeling, this work proposes theory-aware abstraction refinement which considers mixing two theories that are known as two opposite extremes of precision, namely *EUF* and bit-precise encoding.

This requires a fundamental study to understand how to automatically identify statements whose exact semantics can be ignored in model checking. This shift of viewpoint has several advantages: (i) the guidance from the source code allows the use of more powerful heuristics for choosing which statements should remain abstract; (ii) the approach can be used both to obtain speed-up in solving, and as a means for synthesis and finding fix-points for transition relations; and (iii) the refinement takes place on the level of the program, not at the level of the theory query, an approach potentially more natural from the point of view of the semantics of the program.

Research contribution. Chapter 6 presents a new approach for abstraction refinement in software verification. In the proposed approach a program is encoded using *less precise theories* and it is encoded to *precise theories* only when necessary. The building block of theory refinement has a CEGAR loop, and the main contribution is the process of gradually encoding a program using the most precise theory only for a critical subset of the program statements, while keeping lower precision for the rest of the program. The critical subset of program statements is identified automatically based on counterexamples, and theories of different precision are bound to each other through special identities. One important feature of the contributed algorithm is that it employs several heuristics for refinement to point the model checker at which exact statement strengthening is required. To demonstrate the advantages of the theory-aware abstraction refinement, the idea of theory refinement approach was implemented in HiFrog and compared against flattening approach (EAGAR) and CBMC model checker on a large set of C programs. The experiments show promising results both with respect to speed and the number of refined program statements. The experiments demonstrate that the approach has a potential of several orders of magnitude of improvement over the approach based solely on flattened bit-vectors, as implemented in the state-of-the-art tool for example CBMC and in HiFrog. The results of this work have been published in [Hyvärinen et al., 2017] and are presented in Chapter 6 in detail.

1.2.5 Outline

The dissertation is structured as follows. Chapter 2 introduces the required concepts and notation. Chapter 3 presents a technique for interleaving SMT-reasoning with BMC. Chapter 4 describes incremental verification of program revisions via summary repair and discusses the practical applicability of LRA interpolation algorithms in incremental verification. The proposed verification framework is extended in Chapter 5 and 6 to support finding a right level of abstraction suitable for efficient symbolic model checking. Related work, limitation of the proposed approach, and possible future work ideas are discussed at the end of each chapter. Finally, Chapter 7 states the conclusion of the work.

Chapter 2

Background

This chapter introduces background concepts and terminology that is used in the rest of the thesis.

2.1 Formal modeling and properties

To reason formally about a software system automatically, one should first build a formal model of it. Once a system model and a property are defined, the verification procedure can be performed.

Transition systems. Transition systems (TS) are the most common formal models used for the faithful representation of system behavior. Intuitive understanding of a transition system is quite simple: it is a directed graph with nodes that represent program states and edges that reflect transitions among states.

Program graph. Another program modeling structure – program graph – allows explicit reasoning about the program states to be avoided. Instead, it uses program locations as nodes and program commands as edges that connect locations. A program graph is often used as an intermediate modeling structure in program analysis. In particular, it is used to represent a control-flow graph of a program.

Properties specification. After the system behavior is formalized and prior to its analysis, we need to define an important element of the verification process — a property of interest. Once a system model and a property are defined, the verification itself can be performed. The specified property answers a question: what

does it mean for the system to be correct? System correctness (or equivalently incorrectness) is then defined with regard to properties, which hold (or may not) for the analyzed system. In this thesis the main focus is verifying the particular class of properties namely safety properties. Assertions are properties of the state of the program when the program reaches a particular program location. Assertions are often written by the programmer using the *assert* macro. *Reachability properties* are in the form "Is there a path through the program such that some property is violated?"; instances are buffer overflows, arithmetic overflow, pointer safety (check for NULL-pointer dereferences), memory leaks, division by zero, Not-a-Number, etc. Most of these properties relate to behaviors that are left undefined by the C language semantics. This thesis only concentrates on safety reachability properties that are already included in C program as user-specified assertions.

2.2 Abstraction

From the various of approached introduced for abstraction computation we elaborate on those that are relevant to the algorithms developed in this dissertation. The techniques are abstract interpretation, predicate abstraction and predicate abstraction-based counterexample-guided abstraction refinement.

2.2.1 Abstract interpretation

Abstract interpretation [Cousot and Cousot, 1977] is a theory of sound approximation of program models. It constructs an abstraction of a program with regards to values from an abstract domain by iteratively applying the instructions of a program to abstract values until the fixpoint is not reached.

2.2.2 Predicate abstraction

Predicate abstraction [Graf and Saïdi, 1997; Colón and Uribe, 1998] is one of the widely applied methods for abstracting data by only keeping track of certain predicates on the states. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state model that is an abstraction of the original system with respect to a set of predicates. Predicate abstraction techniques have been shown to be a powerful technique for verifying imperative programs, which can

solve the problem of state space explosion pretty well. However, despite its high efficiency, it fails to work out some kinds of the programs because the predicates produced are so bad to make the verification divergent.

2.2.3 Counterexample-guided abstraction refinement (CEGAR)

In a high-level view the paradigm of counterexample-guided abstraction refinement Kurshan [1994]; Balarin and Sangiovanni-Vincentelli [1993] is as following:

1 **Initial abstraction:** *Create an oversimplified model.*

2 **Verification:** *Verify the task. If the verification is successful, terminate with success. Else go to the next step.*

3 **Analyze the failure:** *Analyze the failure report from model checker and determine whether the failure is inherent in the original program or it is unfeasible due to the oversimplification. If the former, terminate with violation of the property. If the latter, go to the next.*

4 **Counterexample-driven refinement:** *Refine the abstract model in a way that the unfeasible reported error is eliminated. Then go to Verification step.*

As an analogy, the abstraction-refinement methodology proposed in this dissertation relates to the above technique in essentially the same way as it follows four main steps. However, a fundamental problem in the classical CEGAR approach is that sometimes the entire program needs to be presented in the most precise way, resulting in considerable overhead. Instead, the proposed approach in this thesis uses different theories of SMT to adjust the level of abstraction.

2.2.4 Abstraction using interpolation

The notion of interpolant goes back to Craig's interpolation theorem for first-order logic. Nowadays interpolation is widely acknowledged as one of the fundamental instruments in computing abstraction [McMillan, 2003a].

This thesis considers approaches where interpolants are constructed from proofs of unsatisfiability. For any two logical formulas whose conjunction is unsatisfiable, there exists always an interpolant.[1]

[1] This is the interpolation property, which holds for first-order logic, and for certain fragments and theories, but not others. Sometimes a theory can be extended to provide the interpolation property.

Definition 1 (Craig binary interpolation) *Given an unsatisfiable CNF formula ϕ partitioned into two disjoint formulas A and B, we denote a* binary interpolation instance *by* $(A|B)$. *An* interpolation algorithm *Itp is a procedure that maps an interpolation instance to a formula $I = Itp(A|B)$ such that (i) $A \implies I$, (ii) $I \implies \neg B$, and (iii) $symb(I) \subseteq symb(A) \cap symb(B)$.*

If I is an interpolant for $(A|B)$, then $\neg I$ is an interpolant for $(B|A)$. This interpolant is called *dual interpolant* of $(B|A)$. Intuitively interpolant can be seen as a formula that is weaker than A, so it over-approximates A while it is still conflicting with B. This type of problems shows up naturally in various verification approaches. For instance, assume that $A \wedge B$ is a symbolic encoding of the program P. If A describes a set of states and B encodes an example of error-free behavior, the interpolant I is an over-approximation of the part of program described in A but still sufficiently detailed to guarantee unsatisfiability with the problem description in B.

In the next section we describe one abstraction technique for over-approximation of program *functions*.

2.3 Interpolation-based function summarization

One of the successful applications of interpolation in model checking is computing a summary for the functions of the program being verified, called a *function summary* [Sery et al., 2011; Albarghouthi et al., 2012b]. Function summarization is a generic technique for abstracting programs which allows both propositional and first-order instantiation. The main idea of this approach is to construct abstractions for some parts of programs, namely functions. In a program P, let A-part consist of a description of a specific function f, and B-part consist of the rest of the program together with negation of a property being checked. The resulting interpolant I can be interpreted as an *over-approximation* of the function f satisfying the property.

Function summaries are computed as over-approximations to preserve the most relevant information of the function bodies to the property being checked. If such an over-approximation of a function exists, it can be used instead of the original encoding of function representation. Since summaries can be constructed to be considerably smaller, it significantly speeds up the verification process.

The approach for extracting and reusing interpolation-based function summaries in the context of BMC was proposed by [Sery et al., 2011]. This work focuses only on propositional logic and does not consider the rich field of first-

order-logic over different theories available in modern SMT solvers. Consequently, despite behaving in an incremental manner, in practice it is computationally expensive, and intractable in large-scale programs.

2.3.1 Function summaries

A function summary relates input and output arguments of a function. Therefore, a notion of arguments of a function is necessary. For this purpose, we expect to have a set of program variables $\mathbb{V}$ and a domain function $\mathbb{D}$ which assigns a domain (i.e., set of possible values) to every variable from $\mathbb{V}$. In the following we adopt the definitions from [Sery et al., 2011].

Definition 2 *For a function f, sequences of variables $args^f_{in} = \langle in_1, ..., in_m \rangle$ and $args^f_{out} = \langle out_1, \ldots, out_n \rangle$ denote the input and output arguments of f, where $in_i, out_j \in \mathbb{V}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. In addition, $args^f = \langle in_1, \ldots, in_m, out_1, \ldots, out_n \rangle$ denotes all the arguments of f. As a shortcut, this thesis uses $\mathbb{D}(f) = \mathbb{D}(in_1) \times \cdots \times \mathbb{D}(in_m) \times \mathbb{D}(out_1) \times \cdots \times \mathbb{D}(out_n)$.*

In the following, we assume that functions do not have arguments other than input and output parameters, which include also the return value. Note that an $in-out$ argument (e.g., a parameter passed by reference) is split into one input and one output argument. Similarly, a global variable accessed by a function is rewritten into the corresponding input or/and output argument, depending on the mode of access (i.e., read or/and write). Precise behavior of a function can be defined as a relation over values of input and output arguments of the function as follows.

Definition 3 (Relational Representation) *Let f be a function, then the relation $R^f \subseteq \mathbb{D}(f)$ is the relational representation of the function f, if R^f contains exactly all the tuples $\vec{v} = \langle v_1, \ldots, v_{|args^f|} \rangle$ such that the function f called with the input values $\langle v_1, \ldots, v_{|args^f_{in}|} \rangle$ can finish with the output values $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|} \rangle$.*

Note that Definition 3 admits multiple combinations of values of the output arguments for the same combination of values of the input arguments. This is useful to model nondeterministic behavior, and for abstraction of the precise behavior of a function. In this thesis, the summaries are applied in BMC. For this reason, the rest of the text in the section will be restricted to the following bounded version of Definition 3.

Definition 4 (Bounded Relational Representation) *Let f be a function and ν be a bound, then the relation $R^f_\nu \subseteq R^f$ is the bounded relational representation of the function f, if R^f_ν contains only the tuples representing computations with all loops and recursive calls unwound up to ν times.*

Then a summary of a function is an over-approximation of the set of precise behaviors of the given function under the given bound. In other words, each bounded function behavior is captured by a summary, but not necessarily each summary behavior belongs to the bounded function.

Definition 5 (Summary) *Let f be a function and ν be a bound, then a relation S such that $R^f_\nu \subseteq S \subseteq \mathbb{D}(f)$ is a summary of the function f.*

The relational view on a function behavior in Definition 5 is intuitive but impractical for implementation. Definition 6 makes a connection between these two views.

Definition 6 (Summary Formula) *Let f be a function, ν a bound, σ_f a formula with free variables only from $args^f$, and S a relation induced by σ_f as $S = \{\vec{v} \in \mathbb{D}(f) \mid \vec{v} \models \sigma_f\}$. If S is a summary of the function f and bound ν, then σ_f is a summary formula of the function f and bound ν.*

A summary formula of a function can directly substitute precise representation of the function. This way, the part of the program encoding corresponding to the called function does not have to be created and converted to a part of the BMC formula.

The important property of the resulting BMC formula with the use of summaries is that if the formula is unsatisfiable, then the formula without summaries is also unsatisfiable. Therefore, no errors are missed due to the use of summaries.

Lemma 1 *Let ϕ be a BMC formula of an unwound program P for a given bound ν, and let ϕ' be a BMC formula of P and ν, with some function calls substituted by the corresponding summary formulas bounded by ν', $\nu' \geq \nu$. If ϕ' is unsatisfiable then ϕ is unsatisfiable as well.*

Proof 1 *Suppose that there is only one summary formula σ_f substituted in ϕ' for a call to a function f. If multiple summary formulas are substituted, we can apply the following reasoning for all of them.*

For a contradiction, suppose that ϕ' is unsatisfiable and ϕ is satisfiable. From the satisfying assignment of ϕ, we get values $\langle v_1, \ldots, v_{|args^f|}\rangle$ of the arguments to

the call to the function f. Assuming correctness of construction of the BMC formula ϕ, the function f given the input arguments $\langle v_1, \ldots, v_{|args^f_{in}|}\rangle$ can finish with the output arguments $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|}\rangle$ and with all loops and recursive calls unwound at most ν times. Therefore, by definition of the summary formula, the values $\langle v_1, \ldots, v_{|args^f|}\rangle$ also satisfy σ_f. Since the rest of the formulas ϕ and ϕ' is the same, the satisfying assignment of ϕ is also a satisfying assignment of ϕ').

SMT offer a more intuitive and lightweight representation of logical expressions than solely using propositional logic [de Moura and Bjørner, 2009]. This makes it possible to generate more concise summaries in first-order logic as compared to summaries based on propositional logic.

2.4 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) [Barrett et al., 2009; Detlefs et al., 2005] plays a central role in various heterogeneous solutions for problem such as scheduling, optimization, testing, synthesis, and verification [de Moura and Bjørner, 2011].

SMT solvers build on the success of solvers for propositional satisfiability (SAT), which over the last two decades have become an invaluable tool for symbolic reasoning. They represent a first-order formula by propositional abstraction where the atoms are inequalities in a first-order theory such as the theory of linear arithmetic. The abstraction is handed to a SAT solver, which, once having found a solution for the Boolean part, will query the satisfiability of the solution from a solver specific to a theory. In the case of linear arithmetic, an altered version of the Simplex algorithm [Dutertre and de Moura, 2006] could be used to determine the satisfiability of a linear system.

2.4.1 SMT notational conventions

This dissertation relies heavily on concepts used in SMT solving. In the following, we define the essential notations of SMT.

A *signature* Σ is a union of $\mathcal{C}, \mathcal{F}, \mathcal{P}$, pairwise disjoint sets of constants, functions, and predicate symbols, respectively. Each function in $\mathcal{F}$ is associated with an arity $n \geq 1$.

A set of *terms* $\mathcal{R}_\Sigma$ is defined inductively comprising constants, variables, and more complex expressions by applying function symbols on terms. The rules are

captured by the following grammar:

$$\begin{aligned} term ::= & \; const \\ & \mid var \\ & \mid f(term, \ldots, term) \end{aligned}$$

where $const \in \mathcal{C}$ is a constant, var is a variable, and $f \in \mathcal{F}$ is a function symbol with arity equal to the number of terms in parentheses.

The set $\mathcal{P}$ a set of predicate symbols, always contains the symbols $\top$ and $\bot$ of arity 0, and $=$ of arity 2. An application of a predicate symbol p with arity n on n terms is called an *atom*. Given an atom At, we denote by $symb(At)$ the set of variables in At. A *literal* is either an atom At or its negation $\overline{At}$. A *clause* cl is a finite disjunction of literals, and a *formula* in *conjunctive normal form* (CNF) is a conjunction of clauses. We interchangeably interpret a clause as a set of literals and a CNF formula as a set of clauses. For a formula ϕ, we denote by $\neg\phi$ its negation. We extend the notation $symb$ to CNF formulas, writing $symb(\phi)$ for the set of atoms in ϕ.

Similar to the above grammar a set of *formulas* $\mathcal{S}_\Sigma$ is built inductively using the following grammar:

$$\begin{aligned} formula ::= & \; Bvar \\ & \mid p(term, \ldots, term) \\ & \mid term = term \mid \top \mid \bot \mid \neg formula \\ & \mid formula \wedge formula \mid formula \vee formula \end{aligned}$$

where $Bvar$ is a Boolean variable, $p \in \mathcal{P}$ is a predicate symbol with arity equaling to the number of terms in parentheses, and $\top$ and $\bot$ are Boolean constants denoting *true* and *false* respectively.

The context of the thesis is that of SMT on quantifier-free formulas. A quantifier-free first-order theory $\mathcal{T}_\Sigma \subseteq \mathcal{S}_\Sigma$ is a set of formulas defined over the signature Σ. When $\mathcal{T}$ is clear from the context, we call a formula from $\mathcal{S}_\Sigma$ an SMT instance.

A CNF formula ϕ is *satisfiable* if there exists an assignment to its variables so that each clause in ϕ contains a *true* literal. A *resolution refutation* (or *refutation*) of a CNF formula ϕ is a tree labeled with clauses. The root of the tree has the empty clause $\bot$, and the leaves have either *source clauses* appearing directly in ϕ, or *theory clauses* that are tautologies in the background theory learned through an unsatisfiable conjunctive query to the theory solver.The inner nodes are clauses

derived by the *resolution rule*

$$\frac{C_1 \vee p \qquad C_2 \vee \overline{p}}{C_1 \vee C_2}$$

where $C_1 \vee p$ and $C_2 \vee \overline{p}$ are the *antecedents*, $C_1 \vee C_2$ the *resolvent*, p is the *pivot* of the resolution step.

2.4.2 Theories of SMT

Boolean satisfiability or SAT is the problem of deciding whether a propositional logic formula can be satisfied, i.e., if there exist an appropriate values for the variables that will make the formula equivalent to true. For example, the formula $x^2 - 9 = 0$ is satisfiable and a solution is $x = 3$. But in general the satisfiability of formulas, e.g., $x^2 + 9 = 0$, depends on what values x is permitted to range over. This is where theories appear. A theory determines what values a variable can have and what the symbols in the formula mean. For instance in the theory of real arithmetic the equation $x^2 + 9 = 0$ is not satisfiable because x must be a real number.

The power of SMT comes from its ability to handle various kinds of theories. SMT solvers are able to reason on Boolean operators (such as AND, OR), arrays, arithmetic, strings, floating point numbers, and software data structures such as lists and trees. SMT framework can also be extended by adding theories to accommodate specific application domains. Another power of SMT is the ability to combine different theories, e.g., an array of strings. Finding efficient integration methods to combine various kind of theories and/or logics within the SAT solver has become a significant subject of research nowadays.

To make use of an SMT solver to solve a problem, the statement of the problem must first be encoded as a logical formula. The SMT-LIB [Barrett et al., 2021] library provides a standard input format for SMT solvers, and also maintains a wide variety of benchmarks for all supported theories, and their combinations as well. The prosperity of research on SMT-solving is also supported by a wide range of competition activities in the community as SMT solver competitions since 2005 [Bartocci et al., 2019].

2.4.3 Theories of interest

An interesting observation on proving the correctness of program is that in many cases when verifying safety properties of a program, often precise encoding does

not play any role in the correctness of the program and over-approximation is sufficient to prove the correctness of the program with respect to the property. Therefore this thesis exploits this case further whenever possible to allow the verification process to scale to large problems.

The theories of interest in this thesis are *Equality and Uninterpreted Functions* (*EUF*), *Linear Real Arithmetic* (*LRA*), and *Linear Integer Arithmetic* (*LIA*). Through these different theories and bit-precise propositional logic (*Prop*) one can explore different program encodings, resulting in different possibilities for abstraction of the original program.

The theory of EUF extends propositional logic by adding equality ($=$) and disequality ($\neq$) to the logical symbols, and allowing functions and predicates as non-logical symbols. The theory of EUF poses no restrictions on the interpretations of constants, functions, or predicates. Most EUF solvers rely on the congruence closure algorithm Nelson and Oppen [Nelson and Oppen, 1980; Nieuwenhuis and Oliveras, 2005] to decide the satisfiability of a set of equalities and disequalities. The key idea of the algorithm is to compute equivalence classes, that is, sets of terms that are equivalent.

In the theory of LRA, the universe consists of real numbers, the function symbols are $*$ and $+$ of arity two restricted to expressing only linear terms, and the predicate symbol $\leq$ and $<$; all them with their usual arithmetic interpretations.

2.5 Programs and Summaries

A *loop-free program* is a tuple $P = (F, main)$, such that F is a finite set of non-recursive functions, and $main \in F$ is an entry point. Let set $\hat{F}$ gather all function calls from F, where $\hat{f}$ is a call of function f. In $\hat{F}$ we distinguish different calls to the same function f by enumerating them as $\hat{f}_1, \ldots, \hat{f}_n$. A *summary* of a function f is a relation over the input and output variables of f that over-approximates the precise behavior of f. That is, if a formula $f_{precise}$ encodes the body of f, and f_{sum} encodes its summary, then $f_{precise} \implies f_{sum}$ must hold. The resulting interpolant f_{sum} can be used in place of $f_{precise}$ when creating the formula again because by construction f_{sum} over-approximates $f_{precise}$.

In this thesis, a program is encoded to a quantifier-free first-order formula in a given theory $\mathcal{T}$, which is then solved for satisfiability. Our intent is to substitute function calls in the considered programs by summaries whenever applicable. If the program encoding is inconsistent with the negation of safety property, then the program is safe.

In the context of this thesis, unsatisfiable formulas originate from bug-free

programs, and thus the summaries express that no trace allowed by the function body leads to a violation of the considered safety specification. In order to construct and use function summaries in the context of BMC, we assume that a BMC formula is a conjunction of encodings of individual function calls. Thus, the problem of determining whether the program is safe with respect to a safety assertion Q reduces to the problem of determining the satisfiability of the SMT formula

$$\bigwedge_{\hat{f} \in \hat{F}} \textsc{encode}(\hat{f}) \wedge \neg \textsc{encode}(Q) \implies \bot.$$

Chapter 3

SMT-based BMC by means of function summarization

BMC supported by highly-efficient Boolean satisfiability (SAT) solvers has advanced the state-of-the-art in model checking substantially. However, due to the high complexity of the model checking problem, verification of large software raises a challenge for SAT-based approaches. The major issue with the SAT-based model checking is that the underlying SAT solvers rely on bit-precise encoding and bit-blasting, which typically scales poorly on large programs.

This chapter proposes an approach to enable the on-demand use of SMT technology and ultimately improve the efficiency of symbolic model checking. To further improve the scalability, the proposed SMT-based BMC is capable of employing one of the most successful abstraction techniques, namely interpolation-based function summarization. This work presents the algorithm on how to compute function summaries using Craig interpolation, and then to *reuse* them for the subsequent analysis of the other properties. The novelty of the proposed approach is in the unique way it combines function summaries with the expressiveness of SMT.

An unintended consequence of using the proposed summarization-based model checking is that it can introduce spurious behaviors due to the use of coarse approximations in the summaries. To automatically rule out the spurious behaviors, the proposed solution employs the well-known paradigm of *counterexample guided refinement* of the function summaries. In order to demonstrate the efficacy of our proposed approach, we developed a novel framework for model checking using SMT and implemented it in our tool HiFrog. Through this implementation, we showcased how varying levels of SMT encoding precision can impact the efficiency of model checking in practice. The results of our extensive experi-

mentation confirm that using a combination of two abstraction techniques – SMT encoding and SMT-based function summarization – shows great promise for verifying large-scale programs. The results reported in this chapter were published in the following papers: [Alt, Asadi, Chockler, Mendoza, Fedyukovich, Hyvärinen and Sharygina, 2017] and [Alt, Hyvärinen, Asadi and Sharygina, 2017].

3.1 Program modeling

Programs are modeled as loop-free through preprocessing of the programs. The goal of preprocessing is to compile source code down to a simplified format that is more amenable to conversion into logical formulae. As the first step, the source code is parsed and transformed into control flow graph (CFG) as an intermediate representation *goto-program*. This transformation is done using the GOTO-CC symbolic compiler developed by CPROVER team[1]. Then in the created CFG the loops are unwound based on the pre-determined number of iterations. The proposed solution, identifies the set of assertions from the source code, reads the user-defined function summaries (if any) in the SMTLIB2-format, and makes them available for the subsequent analysis.

We encode loop-free programs as tuples $P = (F, f_{main})$ where F represents the finite set of (unique) function calls, i.e., function invocation with a unique combination of a program location, a call stack, and a target function. $f_{main} \in F$ denotes the call of the entry point of the program. Interchangeably F also corresponds to the set of functions in the call tree of the unrolled program. We use f for function, and when it is clear from the context it can refer to the *function call* as well. We use relations $child \subseteq F \times F$ and $subtree \subseteq F \times F$, where *child* relates each function f to all the functions invoked by f, and *subtree* is a reflexive transitive closure of $child$. Since each node has at most one parent, we write $parent(n_2)$ to refer to n_1 if $child(n_1, n_2)$ holds.

Next we first describe the classic BMC approach where a program is encoded into a monolithic BMC formula by inlining all the function calls. and demonstrate that it does not allow modular summary-based verification. Then Section 3.1.2 presents a partitioned-BMC technique in which the functions are encoded into separate partitions. This allows modular verification and would be suitable for function summarization.

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

3.1.1 Monolithic program encoding into SMT formula

Assigning logical meaning to programs was pioneered in the sixties by Floyd [W., 1967] and Hoare [Hoare, 1969]. Over the years many forms and enhancements have been developed for connecting between logic and programs; among which the current state-of-the-art includes symbolic model checking tools that are characterized by building symbolic representations of the reachable program states. These symbolic representations can be formulas that represent an over-approximation of reachable program states. As a result, in this case there is a direct mapping from programs to logical formula that retains the input-output relations of the program.

Classical BMC encodes an unwound program to a BMC formula [Clarke, Kroening and Lerda, 2004]. We discuss some of the key ideas of encoding program to first-order-logic by taking a toy example written in an imperative language of C. The main obstacle to encode imperative languages in first-order-formula is inclusion of programs loops and recursion. This obstacle can be resolved by restricting the program bounded loops and recursion. Bounded means posing a fixed number of iterations.

The encoding of a variable assignment follows the Single Static Assignment (SSA) where each assignment to a program variable introduces a new SMT variable that is assigned to only once. When a program variable is changed inside different branches of execution, a new variable after the branch is created to merge the different values after the branches. A so-called *ite*-function (cf. the ϕ function in SSA) is used to re-combine values from different branches of *if-then-else*. We use $ite(c, x1, x2)$ where c is the branch condition and $x1$ and $x2$ are the two SSA variables corresponding to x at the ends of the branches. Functions are expanded in the call site as if being inlined. Once the program is in the SSA form, we can extract a logical formula by treating each assignment as equality.

Example 1 *Figure 3.1 illustrates the above-mentioned process for generating SMT formula as performed in the classic BMC. An unwound program is encoded into a formula by inlining all the function calls. Figure (a) depicts a program in C, Figure (b) shows its SSA form, and Figure (c) shows the corresponding (simplified) encoding of the program into an SMT formula. The formula consists of three parts: a conjunct representing function* `main`*, one equivalent (modulo renaming) conjuncts representing calls of* `func`*, and one conjuncts representing the negated assertion. As customary in BMC, each program variable has its indexed copies (induced by the SSA form).*

The resulting monolithic BMC formula in which all the function calls are in-

```
1  void main() {
2
3   int x = nondet();
4   int y = nondet();
5
6
7   if (x > 3)
8   y = func(x);
9   else
10  y = 2;
11
12  assert(y >= 0);
13  assert(y >= 1);
14
15 }
16
17 int func(int z) {
18  if (z <= 6)
19  return z;
20  else
21  return z - 6;
22 }
```

(a) Program in C

```
// main
x0 = nondet();
y0 = nondet();
if (x0 > 3) {
 z0 = x0;
 // func
 if (z0 <= 6)
 ret0 = z0;
 else
 ret1 = z0 - 6;
 ret2 = phi(z0 <= 6,
    ret0, ret1);
 // end func
 y1 = ret2;
}
else
y2 = 2;

y3 = phi(x0 > 3, y1, y2)
    ;
assert(y3 >= 0);
assert(y3 >= 1);
```

(b) SSA form

```
x0 = nondet0 ∧
y0 = nondet0 ∧
z0 = x0 ∧

ret0 = z0 ∧
ret1 = z0 - 6 ∧

( x0 > 3 ∧ z0 <= 6 ⇒
ret2 = ret0) ∧

( x0 > 3 ∧ z0 > 6  ⇒
ret2 = ret1) ∧


y1 = ret2 ∧

(x0 > 3 ⇒ y3 = y1) ∧
(x0 ≤ 3 ⇒ y3 = y2) ∧

y3 < 0

y3 < 1
```

(c) SMT formula

Figure 3.1. Program encoding in classical BMC

lined is passed to an SMT solver. In case the formula is unsatisfiable, it implies the assertion holds. Otherwise, a satisfying assignment manifests an error trace in the program.

Note that the variable $x0$, highlighted in red, which is local in the function main, appears in the encoding of the body of the function `func`. This is problematic for generating function summary in a modular manner. Next section presents a partitioned-BMC technique in which the functions are encoded into separate partitions. This would allow modular summary-based verification.

3.1.2 PBMC formula construction for summarization

This section uses the approach of the partitioned bounded model checking formula (PBMC formula) construction [Sery et al., 2011] to create formulas in a particular form. The PBMC formula construction approach partitions a program into smaller segments namely functions, such that each function is encoded us-

ing only its own interface symbols and maintains the partition boundaries with the rest of the functions. The interface symbols include input and output parameters, accessed global variables, and helper symbols. In particular, for each function call f, there is a helper boolean variable, that evaluates to true when an error (assertion violation) is reachable in that function given the valuation of its input parameters.

Let β_f be the BMC encoding of the body of a function f, i.e., the logical formula obtained from the SSA form of the body of the function f. Note that β_f does not include inlining of called functions. A PBMC formula is constructed recursively as

$$f_{precise} \equiv \texttt{PrecisePBMCformula}(f) \triangleq \beta_f \wedge \bigwedge_{h \in F : child(f,h)} \texttt{PrecisePBMCformula}(h) \tag{3.1}$$

For each $f \in F$, the formula is built by conjoining the partition β_f and a separate partition for all nested calls. The PBMC formula $\texttt{PrecisePBMCformula}(f_{main})$ is conjoined with the negation of a safety property $error_{f_{main}}$ and it is called a *safety query*. Typically $error_{f_{main}}$ represents disjunction of the negations of each of the assertion in the program. A program is *safe* if the safety query is unsatisfiable.

Once the PBMC formula is unsatisfiable, it is straightforward to partition it for the interpolant generation for each function call. The idea is that one part (call it A) corresponds to the function implementation (including its callees and nested functions) and the other part (call it B) corresponds to the calling context (rest of the formula). After partitioning, Craig interpolation can be applied for the part A and part B. The computed interpolants are then over-approximations of the functions input/output behavior and expressed only over the interface variables of the functions.

3.2 Obtaining function summaries

This section presents how interpolation can be used to construct over-approximation of the function behaviors after a successful verification run. By exploiting the proof of unsatisfiability for the safety query one can construct *function summaries* for each function call f.

Algorithm 1 outlines interpolation-based SMT solving which is the core part of function summarization in SMT-based BMC. The idea is if the BMC formula is unsatisfiable, i.e., the program is safe, the algorithm proceeds with interpolation. Algorithm 1 describes the method for constructing function summaries in BMC using interpolating SMT solver. At line 1 once the safety query in a certain theory

Input: Program $P = (F, f_{main})$ with function calls F and main function f_{main}; theory $\mathcal{T}$.
Output: Verification result: {*SAFE, UNSAFE*}, summary mappings $\sigma_{\mathcal{T}}$ in a particular theory $\mathcal{T}$
Data: PBMC formula $\phi_{f_{main}}$, resolution refutation: π

1 $\phi_{f_{main},\mathcal{T}} \leftarrow$ `PrecisePBMCformula`$(f_{main}) \wedge \neg error_{f_{main}}$ `//create inlined formula in theory` $\mathcal{T}$
2 $\langle result, \pi, \epsilon \rangle \leftarrow$ `CheckSAT`$(\phi_{f_{main}})$; `// run SMT-solver`
3 **if** $result =$ SAT **then**
4 | **return** *UNSAFE*, `ExtractCex`(ϵ) ;
5 **foreach** $f \in F$ **do**
6 | $\sigma_{\mathcal{T}}(f) \leftarrow$ `GetInterpolant`(f, π); `// extract summaries`
7 **return** *SAFE*, $\sigma_{\mathcal{T}}$;

Algorithm 1: SMT-based BMC with function summarization (Bootstrapping)

$\mathcal{T}$ is created, it is sent to an SMT solver. If the result is unsatisfiable, i.e., the program is safe, the method `GetInterpolant` (line 6) computes an interpolant for each $f \in F$ from the proof of unsatisfiability.

Function summaries are constructed as interpolants from proof of unsatisfiability of the BMC formula. In particular, for a function f, the PBMC formula Φ is divided into two parts $\Phi_f^{subtree} \wedge \Phi_f^{rest}$. The first, $\Phi_f^{subtree}$ corresponds to the partitions representing the function call f and its nested function calls:

$$\Phi_f^{subtree} \triangleq \bigwedge_{h \in F : subtree(f,h)} \beta_h \tag{3.2}$$

The second, Φ_f^{rest} corresponds to the rest of the program including the negation of safety properties:

$$\Phi_f^{rest} \triangleq \neg error_{f_{main}} \wedge \bigwedge_{h \in F : \neg subtree(f,h)} \beta_h \tag{3.3}$$

Then for each f, formula (3.2) is considered as A-part and formula (3.3) as B-part for the interpolant construction. The `GetInterpolant` method generates an interpolant I_f for the interpolation instance $(A|B)$, which acts as a summary for the function f. We map functions to their summaries encoded in the theory $\mathcal{T}$ with $\sigma_{\mathcal{T}}$: $F \rightarrow S$ such that $\sigma_{\mathcal{T}}(f) = I_f$.

We call the generated interpolant of function f (line 6) $\sigma_{\mathcal{T}}(f)$ which can be

used in place of $f_{precise}$ when creating the formula again. This replacement is sound because by construction $f_{precise} \implies \sigma_{\mathcal{T}}(f)$.

3.2.1 Interpolation algorithms used in the proposed framework

The proposed framework relies on different interpolation algorithms for the different theories it supports.

The solutions in this thesis exploits an EUF-Interpolation framework proposed in [Alt, Hyvärinen, Asadi and Sharygina, 2017], for generating interpolants for the quantifier-free theory of *EUF*. In this interpolation system two EUF interpolation algorithms are available: Strong and Weak. These algorithms generate interpolants with the smallest number of equalities, guaranteeing, respectively, the strongest and the weakest interpolants in the entire framework. This framework computes interpolants that are much more compact and expressive compared to SAT-based interpolants and it is able to generate a multitude of interpolants of different strength. The strength of interpolants can be controlled by special labeling functions so that the strongest and weakest labeling functions of the EUF interpolation framework generate the interpolants with the smallest number of equalities among all the possible labeling functions.

The solutions in this thesis also exploits interpolation system for LRA implemented following the technique presented in [McMillan, 2005]. The dual version of this algorithm is also available, and can be sorted by the strenght of the created interpolants. Thus, we call here *Strong* the interpolation algorithm given in [McMillan, 2005], and *Weak* its dual. As a result the generation of *LRA* interpolants can be controlled with respect to strength and size by the user-given labeling. The LRA-interpolation system [Alt, Hyvärinen and Sharygina, 2017] allows the generation of an infinite amount of interpolants from the same simplex explanation by using a strength factor.

Another feature related to interpolants which is exploited in this thesis is Proof Compression for propositional proofs. For propositional logic the *Labeled Interpolation Systems* [D'Silva et al., 2010] is used including the *Proof-Sensitive* (PS) interpolation algorithms [Rollini et al., 2013]. This framework supports a range of techniques to reduce the size of the generated interpolants through removing redundancies in propositional proofs. Reducing the proof has the consequence of also reducing the size of the generated function summaries.

3.3 SMT-based incremental verification for verifying multiple properties

This chapter presents a technique to verify incrementally multiple safety properties.

To verify a single program against a list of safety properties $Q_1, \ldots, Q_n$, there are two approaches. The first approach is to check each property individually, which can be time-consuming and not very effective because it requires building and solving formulas from scratch at each iteration. The second approach is to check the properties incrementally, one by one, and reuse computation history after each successful verification run. However, this approach has drawbacks. If even just one property fails to hold, the failure has to be examined and confirmed. Despite this, the overall cost of verification is often much less than verifying the program from scratch for each individual property, even after several iterations. In the worst case, the computational history while checking one property may not be reusable for the next, forcing the verification procedure to run without reuse.

Our solution to the verifying multiple properties in a single program is to construct function summaries as a fundamental building block for storing the computation history and to reuse them in subsequent verification runs. The proposed solution by incremental verification avoids repeating the same tasks again and over again. Consequently, the application of this approach would lead to significant speed up in verification, and more importantly would scale to large programs.

3.3.1 Applying function summaries during formula construction

This section proposes an algorithm to reuse function summaries between verification tasks when checking different assertions in the same program. While constructing an SMT formula for checking a new assertion, if a summary of a function call already exists, it can substitute the precise encoding the function body. In the proposed solution a summary formula of a function can be applied during construction of the BMC formula to represent a function call. This way, the part of the SSA form corresponding to the called function does not have to be created and converted to a part of the BMC formula. Consequently, the summary formula tends to be smaller and removes the need of encoding the formula from scratch.

Substitution of summaries might fit well and the verification might succeed

and lead to performance speedup, or due to an over-approximating nature of summaries, spurious bugs might which have to be identified immediately. The important property of the resulting BMC formula is that if it is unsatisfiable then also the original representation of formula (function `func` in Figure 3.1) is unsatisfiable as well. Therefore, no errors are missed due to the use of the summaries.

Example 2 *In the example in Figure 3.1 a possible function summary for* `func` *obtained after verifying the assertion is* $(z_0 > 0) \implies (func\_ret \geq 1)$. *This summary can successfully replace the call to* `func` *while verifying the second assertion.*

Algorithm 1 considers the case of checking a given program with respect to a single assertion. Suppose, the program contains multiple properties which are required to be checked: $\{error\}_1^n$. Instead of verifying the program assertions in isolation and repeating the full re-verification over and over again, the thesis proposes to reuse function summaries to reduce the amount of proofs required during verification. The generated function summaries after a successful verification of assertion $error_i$ are reused for checking assertion $error_{i+1}$.

Recall that construction of the formulas by inlining as in `precisePBMCformula` (Eq. 3.1) which results in a monolithic formula where the whole function bodies are included. Contrary to such inlined construction of formulas, this chapter proposes the construction of PBMC formula in the presence of function summaries that can be substantially smaller compared to the inlining of the entire function bodies since functions summaries tend to be more compact.

Consider function f as a root of a subtree of a program. Suppose in the subtree of f there is a function h and its summary was already computed. Then the summary of h can be substituted for body of h while building the PBMC formula of f. This way, the PBMC formula ϕ_f corresponding to the encoding of subtree of f can be considerably more succinct compared to the inlining of the entire function bodies in subtree of f.

Algorithm 2 creates a PBMC formula for a subtree rooted at f, i.e., ϕ_f. The algorithm initially accepts a *substitution scheme* for the function representations and uses it while constructing formula ϕ_f.

Definition 7 *A substitution scheme for program function calls is a mapping function* $Sb : F \rightarrow \{precise, sum, nondet\}$ *determines how each function call should be handled.*

The level of approximation for each function $f \in F$ is determined as one of the following three cases: (i) *precise* when the entire f is required to be processed,

Input: Program P with function calls F; Summary mapping $\sigma_{\mathcal{T}} : F \to S$; A theory of SMT $\mathcal{T}$; the function f for which a formula is created for it and its subtree.
Output: ϕ_f: Encoding of subtree of f (PBMC formula)
Data: $WL \subseteq F$, mapping of substitution scheme for function calls $Sb : F \to \{precise, sum, nondet\}$

```
1  WL ← {f} , φ_f ← true;
2  while WL ≠ ∅ do
3      pick g ∈ WL, and WL ← WL \ {g};
4      φ_g ← true;
5      foreach h s.t. child(g,h)              // process children of g
6       do
7          switch Sb(h) do
8              case sum: do φ_g ← φ_g ∧ σ_T(h);        // apply summary
9              case precise: do WL ← WL ∪ {h};   // defer process of h
10             case nondet: do skip; // treat h nondeterministically
11     φ_g ← β_g ∧ φ_g;                          // create SMT formula
12     φ_f ← φ_f ∧ φ_g;
13 return φ_f
```

Algorithm 2: `CreateFormula`$(P, \sigma_{\mathcal{T}}, \mathcal{T}, f, Sb)$

(ii) *sum* when a pre-computed summary substitutes f, and (iii) *nondet* when f is treated as a nondeterministic function. Since nondet abstracts away the function, it is equivalent to using a summary formula *true*.

We define three substitution schemes: $Sb_{precise}$: $F \to \{precise\}$ inlines all the function bodies. Sb_{eager}: $F \to \{sum, precise\}$ inlines functions without summaries and otherwise employs summaries:

$$Sb_{eager}(g) = \begin{cases} sum, & \text{if g has summaries} \\ precise, & \text{otherwise} \end{cases} \tag{3.4}$$

Note that summary mapping $\sigma_{\mathcal{T}}$ is total and all functions initially have summaries. Finally, Sb_{lazy}: $F \to \{sum, nondet\}$ treats functions without summaries as nondeterministic calls and the rest as *sum*, as follows:

$$Sb_{lazy}(g) = \begin{cases} sum, & \text{if g has summaries} \\ nondet, & \text{otherwise} \end{cases} \tag{3.5}$$

This results in a smaller initial PBMC formula and leaves the identification of the critical function calls to the refinement loop.

3.3.2 Summary refinement

Although function summaries are lightweight replacements for precise encodings of corresponding functions, their loss of precision can sometimes lead to spurious counterexamples as a result of over-approximation. In such cases, the summaries must be refined; in simple cases, they may be discarded and replaced with the direct encoding. To address false alarms that arise due to abstraction, we propose a refinement procedure. Specifically, we employ the well-known technique of Counterexample-guided refinement (CEGR) to handle coarse function summaries.

The summary refinement approach involves analyzing the error trace, which is determined by a satisfying assignment from the solver. If the error trace includes any function summaries, they must be removed from the PBMC formula and replaced with the precise representation of the full body of the corresponding functions. In our proposed solution, we have implemented this approach by adding the precise representation of a function f to the PBMC formula in case the error trace contains a summary of f. Then the satisfiability check is repeated to ensure accuracy. If no function summary appeared along the error trace, the error is real.

3.4 HiFrog: SMT-based incremental verification via function summaries

This section presents the architecture of the proposed interpolating SMT-based BMC for verifying different assertions of a program in an incremental way.

The proposed framework has been implemented in the tool HIFROG that is based on function summarization and invokes the OpenSMT2 solver [Hyvärinen et al., 2016] with support to the propositional logic, *LRA*, *LIA*, and *EUF* theories. HIFROG is unique in its combination of function summarization and interleaving with SMT, which provides numerous benefits both in efficiency of verification and in readability of the proofs. In particular, the SMT-based representation of function summaries preserves the structural information of the summaries. This makes them reusable specifications, provides user-readable feedback, and offers numerous verification features. The broad spectrum of features supported

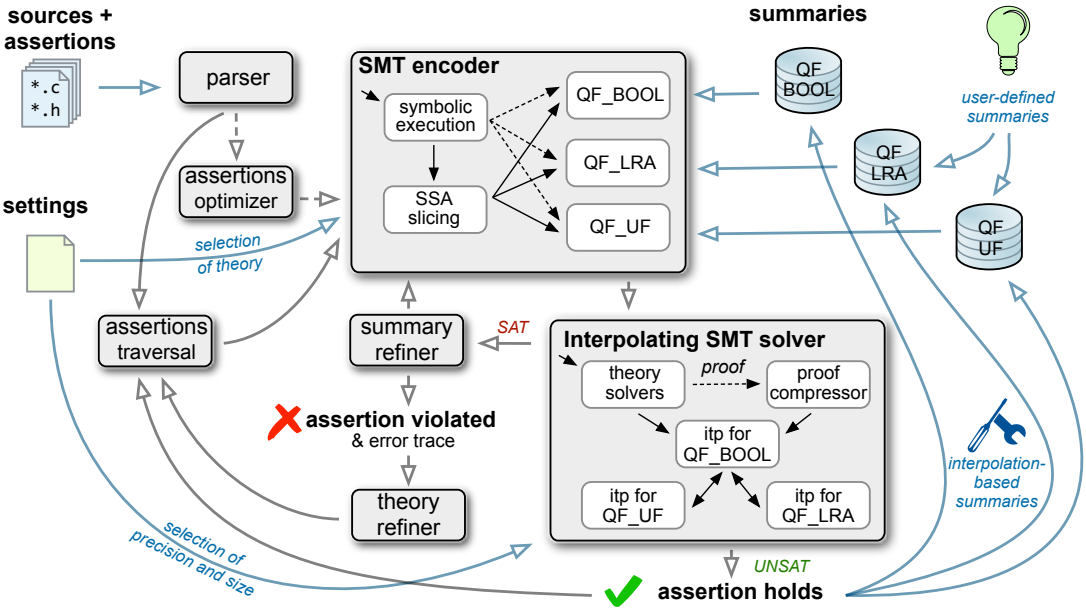


Figure 3.2. HiFrog overview. Grey and black arrows connect different modules of the tool (dashed - optional). Blue arrows represent the flow of the input/output data.

by HiFrog exploits BMC verification and includes verification of recursive programs. It also supports the injection of user-defined summaries, allowing the user to add known mathematical theorems to the theory (such as a program computing trigonometric functions).

The tool provides an additional flexibility of the verification process, allowing to construct interpolants of different strength. It also supports several optimizations, which we describe in more detail later. In particular, it supports optimization of assertions by removing the redundant assertions from the verification process. If the verification fails, the tool presents a counterexample in an easy-to-understand way, with the direct mapping to the source code.

HiFrog consists of two main components: the *SMT encoder* and the *interpolating SMT solver*; as well as the function *summaries* (see Figure 6.5). The components are initially configured with the theory and the interpolation algorithms. The tool then processes assertions sequentially using function summaries when possible. The results of a successful assertion verification are stored as interpolated function summaries, and failed verifications trigger a refinement phase or the printing of a violation witness. This section details the tool features.

SMT encoding and Function Summarization. For a given assertion, the goto-program is *symbolically executed* function-per-function resulting in the "modular" Static Single Assignment (SSA) form of the unwound program, i.e., a form where each variable is assigned at most once, and each function has its own isolated SSA-representation. To reduce the size of the SSA form, HIFROG performs *slicing* that keeps only the variables in the SSA form that are syntactically dependent on the variables in the assertion.

When the SSA form is pruned, HIFROG creates the SMT formula in the pre-determined logic (e.g., *Prop*, *EUF* or *LRA*). The modularity of the SSA form comes in handy when the function summaries of the chosen logic (either user-defined, interpolation-based, or treated non-deterministically) are available. If this is the case, the call to a function with the available summary is replaced by the summary. The final SMT formula is pushed to an SMT solver to decide its satisfiability.

Due to over-approximating nature of function summaries, the program encoded with the summaries may contain spurious errors. The *summary refiner* identifies and marks summaries directly involved in the detected error, and HIFROG returns to the encoding stage to replace the marked summaries by the precise (up to the pre-determined logic) function representations. Note that due to refinement, HIFROG reveals nested function calls (including recursive ones) which are again replaced by available summaries. When faced with an unsatisfiable SMT formula, HIFROG employs interpolation to create function summaries. These summaries are then serialized and stored in persistent storage, making them accessible for future runs of HIFROG.

Theories of SMT in HiFrog. HIFROG supports three different quantifier-free theories in which the program can be modeled: bit-precise *Prop*, *EUF* and *LRA*. The use of theories beyond *Prop* allows the system to scale to large problems since encoding in particular the arithmetic operations using bit-precision can be very expensive. As the precise arithmetic often do not play a role in the correctness of the program, substituting them with linear arithmetic, uninterpreted functions, or even nondeterministic behavior might result in a significant reduction in model checking time. If a property is proved using one of the light-weight theories *EUF* and *LRA*, the proof holds also for the exact BMC encoding of the program. However, the loss of precision can sometimes produce spurious counterexamples due to the over-approximating encoding. The light-weight theories therefore need to be refined (i.e., using *theory refiner*) to *Prop* if the provided counter-example does not correspond to a concrete counterexample.

Assertion Optimizer. In addition to incremental verification of a set of assertions, HIFROG supports the basic functionality of classical model checkers to verify all assertions at once. For the cases when the set of assertions is too large, it can be optimized by constructing an *assertion implication relation* and exploiting it to remove redundant assertions [Fedyukovich et al., 2015]. In a nutshell, the assertion optimizer considers pairs of spatially close assertions a_i and a_j and uses the SMT solver to check if a_i conjoined with the code between a_i and a_j implies a_j (if there is any other assertion between a_i and a_j then it is treated as assumption). If the check succeeds then a_j is proven redundant and its verification can be safely skipped. Indeed, if a_i holds then a_j holds as well; and if there is a counter-example to a_j then it is a counter-example to a_i as well.

Type constraints. To limit the domain to a specific range of values, HIFROG provides an option `-type-constraints` to control the level of expressiveness of type constrains. By default, it limits range of the variable values to the range of their data type in C and the check is applied only to numerical data types (e.g., int, long, double, unsigned). The type constraints check can be extended to include overflow/underflow of numerical data types .

Verification outputs. In the end of each verification run, HIFROG either reports `VERIFICATION SUCCESSFUL` or `VERIFICATION FAILED` accompanied by a violation witness. An error trace presents a sequence of steps with a direct reference to the code and the values of variables in these steps. In most cases when *EUF* and *LRA* introduce a spurious error, HIFROG outputs a warning, and thus the user is advised to use HIFROG with a more precise theory. HIFROG also reports the statistics on the running time and the number of the summary-refinements performed.

Implementation. The major implementation effort has been devoted to developing the HIFROG framework, which serves as the platform for validating the algorithms proposed in this thesis. The tool is available as an open-source software. As a front-end of HIFROG, we use the infrastructure from CPROVER v5.11 to transform C program to obtain a basic unrolled BMC representation that we use as a basis for producing the final logical formula. Apart from the distinguishing feature of HIFROG compared to its earlier version FUNFROG, note the difference in the use of *Prop* summaries between FUNFROG and HIFROG. The former uses an old version of CPROVER and is no longer maintained, which leads to many parsing issues. In contrast, each component of HIFROG, from parsing to modeling to

Table 3.1. Number of solved instances in HiFrog using different theories.

C Benchmarks	total #assertion	#EUF correct solved	#LRA correct solved	#Prop solved / holding assertion
token.c	54	34	34	34
s3.c	131	18	21	26
mem.c	149	96	96	96
disk.c	79	6	6	23
ddv.c	152	47	47	142
café.c	115	15	20	30
tcas_asrt.c	162	16	29	29
p2p.c	244	8	20	94
floppy1.c	18	15	16	18
floppy2.c	21	15	16	21
floppy4.c	22	11	13	22
floppy3.c	19	13	14	19
diskperf1.c	14	9	10	14
diskperf2.c	4	2	2	4
kbfilter1.c	10	10	10	10
kbfilter2.c	13	13	13	13
kbfilter3.c	14	11	11	14
Total	1221	339	378	609
Percentage of success		55%	62%	100%

solving procedures, has been significantly optimized compared to FUNFROG.

3.5 Evaluation

This section provides an experimental evidence of the performance of interpolating SMT-based BMC. We evaluated the implemented algorithm on HIFROG on a large set of C programs for verifying different assertions one after the other.

Benchmarks. Function summarizations allows HIFROG to efficiently verify C code with many assertions. To demonstrate the tool's capability, we use an independent suite taken from several resources: our crafted benchmarks, part of benchmarks from Competition on Software Verification SV-COMP[2], and modified SV-COMP benchmarks with additional assert statements which were generated with *Daikon* [Ernst et al., 2001]. All benchmarks contained multiple assertions

[2]Software Verification Competition, `http://sv-comp.sosy-lab.org/`

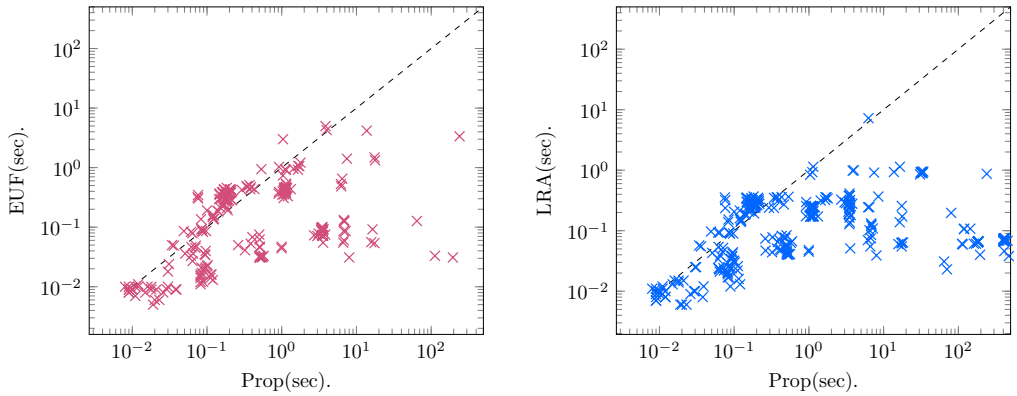


Figure 3.3. Running time of HiFrog by propositional (*Prop*) encoding against *EUF* and *LRA* encoding.

to be checked.[3]

To demonstrate the advantages of the SMT-based summarization, Table 3.1 reports the number of solved instances (safe assertions) in HiFrog using different theories.[4] The table provide data for analysis of benchmarks containing 1221 assertions from which 609 were proven safe using *Prop* (meaning that those properties satisfy the system specifications). Among 609 shortlisted safe assertions (verification tasks) solved by *Prop*, 339 assertions (55%) validated by *EUF* and 378 assertions (62%) validated by *LRA*. Even with the over-approximating nature of *EUF* and *LRA* theories, the experiments observed a large number of properties that were also confirmed to be correct by employing the lightweight theories of the HiFrog. The remaining cases were SAT indicating spurious results. In other words, the results of HiFrog with *EUF* and *LRA* are only reliable for holding assertions (i.e., the counter-example may be spurious due to abstract encoding). Therefore, we report on the success rate of the verification results to ensure consistency.

Furthermore, the experiments revealed that model checking using the *EUF* and *LRA*-based summarization was extremely efficient. Figure 3.3 presents two logarithmic plots comparing SAT encoding against SMT-encoding. The *left* and *right* plot depicts the comparison of running times[5] of HiFrog with *Prop* to respectively *EUF* and *LRA*. Each point represents a pair of verification runs of a

[3]The benchmarks set is available at `http://verify.inf.usi.ch/hifrog/bench`.

[4]The tool FunFrog is not used in the comparison, and instead, HiFrog with *Prop* is used to ensure that the comparison is not affected by unrelated implementation differences.

[5]The timing results were obtained on an Ubuntu 14.04.1 LTS server running two Intel(*R*) Xeon(*R*) *E*5620 *CPUs*@2.40*GHz* and 16GB RAM.

holding assertion with the two corresponding theories using the interpolation-based summaries. Note that for most of the assertions, the verification with *EUF* and *LRA* is an order of magnitude faster than the verification with *Prop*.

This experimentation reveals that while the use of the theories speeds up the model checking significantly, it comes with the obvious downside that sometimes verification conditions that in reality hold for the program are identified as errors due to the over-approximating encoding. The light-weight theories *EUF* and *LRA* therefore need to be refined to propositional logic if the provided counterexample does not correspond to a concrete counterexample. However, if a property is proved with one of these theories, the proof holds also for the exact BMC encoding of the program.

3.6 Related work

This section provides a brief overview of the application of interpolation in formal verification, different approaches to construct function summaries, and different verification approaches based on SMT.

3.6.1 Related work on interpolation in verification

The first application of interpolation to formal verification was employed in the complete technique for finite-state model checking by [McMillan, 2003a], where interpolants were used to over-approximate the sets of reachable states. This work constructs fixed points of transition functions for programs encoded as a satisfiability problem. In the later work (the IMPACT algorithm) [McMillan, 2006], this approach was adapted into the abstraction-refinement style [Clarke et al., 2003], allowing for the verification of infinite-state systems, such as programs with loops. In the IMPACT algorithm, interpolation is applied on demand in lazy fashion and only to individual program paths. Therefore, no unrolling of the entire program is needed, and some paths could remain abstracted away as long as it does not prevent proving safety. Generalizations of the IMPACT algorithm to programs with functions are given in [Heizmann et al., 2010b] and formulated using nested word automata and [Albarghouthi et al., 2012b].

Since McMillan's first application of interpolants in formal verification, interpolation has been applied in algorithms with various extensions in model checking [Cabodi et al., 2006; Vizel and Grumberg, 2009; Heizmann et al., 2010b; Alberti et al., 2012; Rümmer et al., 2013; Albarghouthi and McMillan, 2013;

McMillan, 2014; Cabodi et al., 2014; Vizel et al., 2015; Fedyukovich, Sery and Sharygina, 2017; Fedyukovich and Bodík, 2018; Iosif and Xu, 2018; Beyer and Podelski, 2022; Vick and McMillan, 2023]. There are several techniques for interpolant generation [Cimatti et al., 2008; Griggio et al., 2011] which help the model checkers to leverage interpolants in some form to improve the performance: CPACHECKER [Beyer and Keremoglu, 2011], SEAHORN [Gurfinkel et al., 2015], ULTIMATE AUTOMIZER [Heizmann et al., 2018] and many others.

3.6.2 Related work on function summaries

Function summaries date back to Hoare logic [Hoare, 1971]. Since then a wide variety of techniques for computing function summaries have been proposed to achieve scalable inter-procedural analysis [Henzinger et al., 2004; Engler and Ashcraft, 2003; Gurfinkel et al., 2011]. Each function is processed only once, its summary is created and applied for other calls of the function. [Reps et al., 1995] proposes an algorithm for explicitly computing function summaries using the data-flow analysis. The approach was further exploited in the tool BEBOP [Ball and Rajamani, 2000] (a part of the framework SLAM [Ball et al., 2011]) that uses BDDs to represent the program states. Later the work by [Basler et al., 2007] which is another domain of function summaries in model checking of pushdown systems (PDS) replaced BDDs by SAT- and QBF-based techniques. However, as stated in [Basler et al., 2007], QBF queries still poses a major bottleneck. In contrast, our work proposes a procedure to efficiently reuse SMT-based summaries across multiple properties.

There is another work presented in [McMillan, 2010] which supports handling function calls and uses function summaries. It employs symbolic execution to remember a reason for infeasibility of an execution path, i.e., a blocking annotation. Blocking annotations are used to reject other execution paths as early as possible. Compared to the proposed technique of this thesis, lazy annotation uses interpolation to derive and propagate the blocking annotations backwards for every program instruction. If the annotation is to be propagated across a function call, a function summary merging blocking annotations from all paths through the function is generated and stored for a later use. The proposed technique of this thesis uses interpolation on the whole BMC formula and creates one function summary from one interpolant and requires a single proof of unsatisfiability of the whole program and generates all the summaries at once.

The research for computing over-approximation of function behaviors instead of explicit enumeration of them has been fortified by the advent of SAT-based model checking techniques. In particular, bounded model checkers SATURN [Xie

and Aiken, 2005b,a] and CALYSTO [Babic and Hu, 2008] compute summaries for each function by an iterative discovery of modification of variable values in the function behaviors. Such computation requires several calls to a SAT solver, but can end up with more general summaries compared to [Reps et al., 1995]. This is conceptually different technique to the proposed Algorithm 1 that requires a single proof of unsatisfiability of the whole program and generates all the summaries at once. In spite of the difference in the methods for computing summaries, they all advocate on the way of reusing them on demand.

The work [McMillan and Rybalchenko, 2013] proposed a technique to compute function summaries using tree interpolants and use them as abstractions of functions possibly with refinements. Their proposed approach was implemented in the DUALITY model checker. In DUALITY, the computed summaries are from bounded unfoldings, and summaries are used to replace function calls and refined on demand. While DUALITY approach re-uses the summaries across different call sites for proving the same property, HIFROG re-uses SMT-based summaries across multiple properties in a single program. Duality also has a BMC mode and it still could terminate before unfolding to the BMC bound if an inductive invariant was found. Nevertheless DUALITY makes large interpolation queries as more relations are unfolded and there remain significant challenges in handling mutual recursion and in scalability. For improving scalability, we ensure that the SMT queries in our proposed method are always bounded in size to speed up bounded model checking.

Regarding inter-procedural software model checking with interpolants, in the context of predicate abstraction, the work [Jhala and McMillan, 2007] discusses how well-scoped invariants can be inferred in the presence of function calls. Another related approach are property-directed reachability (PDR)-based approaches SPACER [Komuravelli et al., 2016] that effectively compute summaries and use interpolants as abstractions. However they compute summaries in a different way and they do not perform BMC unfolding. There is another approach that synthesizes function summaries within the SEAHORN verification framework. SEAHORN generates summaries by encoding and solving a system of non-linear Horn clauses [Bjørner et al., 2012] which requires developing an appropriate SMT solver that supports quantifier elimination [Loos and Weispfenning, 1993] for each first-order theory. In contrast, the method of this thesis is able to use off-the-shelf SMT-based techniques without the necessity to support quantifier elimination.

While function summary is a non-inductive over-approximation of function behavior (unless an upper bound is known), synthesizing an *inductive safe invariant* is a key concept in automated proof-based verification [Komuravelli et al.,

2013; Albarghouthi et al., 2012a]. There are a number of successful approaches for finding invariants for programs, e.g., CEGIS [Solar-Lezama et al., 2006], k-induction [Sheeran et al., 2000], and PDR/IC3 algorithm. Nevertheless, synthesizing invariants for programs is one of the most challenging problems in verification today.

As another related work to this area is the approaches that are based on guessing summaries such as HoIce [Champion et al., 2018], FreqHorn [Fedyukovich, Kaufman and Bodík, 2017], and LinearArbitrary [Zhu et al., 2018]. All of these approaches have trade-offs between scalability of the search space and expressivity of guessed summaries.

Much less activity was observed for using the idea of function summaries in concolic execution [Godefroid, 2007] and explicit-state model checking [Qadeer et al., 2004]. For instance, the model checker ZING records explicit summaries as a set of tuples of explicit input and output values that were observed during state space traversal. The summaries employed in Zing also include lock-related information required for synchronization of concurrent programs. In contrast, in HIFROG algorithm each summary symbolically defines an over-approximation of all explicit execution traces through a function, but currently without concurrency related information.

FUNFROGthe model checking algorithm HIFROG built on top of, used an approach for constructing and reusing interpolation-based function summaries in the context of SAT-based bounded model checking. Unlike HIFROG, the work in FUNFROG performs only on propositional logic and does not consider the rich field of first-order theories available in modern SMT solvers. Hence, despite behaving incrementally, FUNFROG was computationally expensive in many cases in practice.

There are several tools that support reuse or exchange of verification results, similar to our function summaries. In the last decade there has been progress on standardized formats [Beyer et al., 2016] of exchange between analysis tools. For instance CPACHECKER tool is able to migrate predicates across program versions [Beyer et al., 2013]. Deductive verification tools such as VIPER and DAFNY offer modular verification [Müller et al., 2016] and caching the intermediate verification results [Leino and Wüstholz, 2015] respectively. CBMC [Kroening and Tautschnig, 2014] is a symbolic bounded model checker for C that to a limited extent exploits incremental capabilities of a SAT solver, but does not use or output any reusable information like function summaries. Similar to HIFROG, ESBMC also shares the CPROVER infrastructure and is based on an SMT solver. To the best of our knowledge, it does not support incremental verification [Cordeiro and de Lima Filho, 2016].

3.6.3 Related work on SMT-based verification

There is a large body of work on SMT-based verification of various domains such as [Alt and Reitwießner, 2018], GPU kernel functions [Li and Gopalakrishnan, 2010], hybrid systems [Cimatti et al., 2012], concurrent programs (including the verification of mutual exclusion, deadlocks) [Güdemann, 2022], Petri nets [Pólrola et al., 2014], distributed network control planes [Raghunathan et al., 2022], security protocols [Szymoniak et al., 2021], cyber-physical system [Nigam and Talcott, 2022], finding adversarial inputs in neural network [Huang et al., 2016] and many more. SMT-based verification can be combined with other verification techniques, such as theorem proving and abstract interpretation, to achieve better results.

In the context of SMT-based symbolic model checking of C programs, the closest body of work can be found in CBMC[Kroening et al., 2023] and ESBMC [Gadelha et al., 2018], which both to a limited extent exploit the capabilities of an SMT solver as a black box. These tools are unable to tune the SMT solver to efficiently decide formulas for symbolic model checking, nor are they able to adapt the program encoding to find a suitable level of abstraction. Instead, they delegate a big monolithic formula to the SMT solver without providing any modularity or the possibility of adjusting the level of precision of the analysis. On the other hand, HiFrog provides support for adapting the constraint language based on the program under verification, allowing for greater flexibility and control over the encoding process.

Kind (K-induction for Invariant Discovery) [Kahsai et al., 2012; Champion et al., 2016] is an SMT-based model checker that allows the simultaneous verification of multiple properties incrementally by saving and reusing the invariants that were used by JKind [Gacek et al., 2018] to prove the properties of a model in parallel. Unlike Kind, HiFrog has not been supporting simultaneous verification of properties in parallel; however, we find this work interesting and relevant to the work in this thesis, especially since OpenSMT2 had recently applied algorithms for parallel and distributed solving [Asadzade et al., 2021; Marescotti et al., 2018].

While the proposed approach differs from other SMT-based model checkers, there are widely-used software verification tools that employ SMT solvers for specific tasks. Here we provide some examples of such model checkers which are orthogonal to the proposed approach. Ultimate Automizer [Heizmann et al., 2013] is a unified framework for unbounded verification and analysis of software based on automaton and SMT. CPAchecker Beyer and Keremoglu [2011] in an SMT-based software verification tool that is based on the concept of con-

figurable program analysis. LLBMC [Falke et al., 2013] is based on the LLVM compiler infrastructure which analyzes programs at the intermediate representation level. SEAHORN is a verification framework for LLVM-based languages that works by iteratively constructing a sequence of inductive invariants, each of which is used to prove the correctness of the next state in the system. It uses over- and under-approximations to improve performance and detect convergence to a proof. UFO [Albarghouthi et al., 2012a] is also built on top of the LLVM compiler infrastructure and allows definition of different abstract post operators, refinement strategies and exploration strategies, and exploits the power of SMT solvers for enumerating paths. In particular it uses MATHSAT4 for SMTchecking and interpolation, and Z3 for quantifier elimination. The DUALITY model checker constructed tree-like SMT problems for unfoldings of recursive programs, and computed functions summaries from tree interpolants. There are conceptually different techniques to the proposed approaches in this thesis.

To wrap up the brief overview of related techniques, we emphasize the distinguishing features of our SMT-streamlined BMC which allows re-using SMT-based summaries across various properties and exploits different SMT encoding for symbolic abstraction.

3.7 Synopsis

This chapter addresses scalability issues in symbolic model checking, contributing to the verification of software. Solutions for enhancing scalability are provided by circumventing costly bit-precise reasoning (P1 in Figure 1.1) while verifying multiple properties in a program. Additionally, the thesis proposes an incremental approach that interleaves with SMT reasoning for various computational tasks. This chapter presents a framework for generating a reusable computation modules of the safe program, namely SMT-based function summaries (a solution to problem P2 in Figure 1.1).

The proposed algorithm is a fully-featured function-summarization-based model checker that uses SMT as the modeling and summarization language for verifying a sequence of properties. By exploiting the interpolating SMT solver, it is possible to construct function summaries after the initial successful verification run. These summaries can then be used for more efficient subsequent analysis of properties.

The effectiveness of the proposed approach has been validated by implementing the SMT-based incremental BMC in a new tool HIFROG. To the best of our knowledge, this is the first incremental SMT-based software model checker in

which a sequence of safety properties is verified. To examine the practical impact of the different SMT precisions on model checking, we ran HIFROG on three levels of encoding: *EUF*, *LRA*, and *Prop*. The results indicate that incorporating SMT into the model checking enables the efficient verification of large C programs.

The results of this chapter were published in [Alt, Asadi, Chockler, Mendoza, Fedyukovich, Hyvärinen and Sharygina, 2017] and [Alt, Hyvärinen, Asadi and Sharygina, 2017].

3.8 Limitation and future work

A general note about soundness of the proposed SMT-based BMC is that it is achieved by incrementing the bound until a witness is found, but completeness can only be achieved when the number of steps to reach all states is finite.

Although BMC has found many industrial acceptance and has shown to be efficient for bug catching, i.e., quick finding of counterexamples, BMC tools, including HIFROG, have some inherent limitations that impede verifying the general class of software (unbounded): *(i)* BMC is not capable of finding inductive invariants which are necessary for unbounded proof. The functions summaries computed by HIFROG is not inductive unless an upper bound is known. To shed light on this issue, consider a program with a loop that scans all elements of an array of size *n* where n is arbitrary. BMC tool requires a concrete bound to unwind the arbitrary-size loop. For instance, by 10 unrolling steps, BMC can generate summary formulas that are only valid for arrays with at most size 10. This is certainly restrictive and we are interested in more general over-approximative summaries that are valid for an arbitrary length of the array. *(ii)* the performance of HIFROG is negatively affected by the increase in the number of unrolling steps. It suffers from an encoding that produces an exponential number of formulas in the depth of the specification.

One possible future direction for addressing these problems is to base the proposed approach of this chapter on Constrained Horn Clauses, which appears to be a promising solution. As a high-level intermediate language, CHCs enable concise expression of the verification problem. This would help one to focus on the fundamental issues, abstracting away from low-level details and specifics of practical programming languages. In fact, if a safe inductive invariant is identified, it would remain unaffected by the number of loop iterations.

Scalability continues to be a challenge in this thesis. The proposed tool, HIFROG, accepts programs written in a subset of the C11 standards [C. , 2011]

with the purpose of checking the correctness of safety assertions. This limitation is due to HiFrog's reliance on the CProver framework for parsing, symbolic executions, and front-end modeling of programs, which involves translating C programs into GOTO programs. For instance, in cases where programs rely on arrays for correctness, they are simply bit-blasted in our modeling language. Another limitation is that HiFrog only uses OpenSMT SMT solver as a solving back-end which supports a limited set of theories for program representation. This limitation also may prevent HiFrog from proving substantial properties of complex programs or achieving significant scalability in software model checking. Therefore, exploring the possibility of supporting modeling via additional theories in other interpolating SMT solvers, such as MathSAT or SMTInterpol, with the theory of arrays, bit-vectors, floating points, and so on, and accordingly supporting summarization in those theories, would be an interesting area for future work.

Supporting non-trivial programs with dynamic memory allocation or heap data structures remains as future work. Proving safety of heap-manipulating programs and solving such combined constraints requires non-trivial interaction between SMT solver and model checker. As the heap encoding is often one of the most complex components of an SMT or CHC-based verification tool, verifying heap data structure is known to be a difficult task. This is mainly due to the fact that when designing verification tools for different programming languages, and migrating a tool to a different style of heap encoding, it requires a massive implementation effort. Current verification tools based on CHC tend to handle heap either using the theory of arrays (e.g., as performed by SeaHorn which encodes heap as a set of non-overlapping arrays, or employ bespoke encodings of heap data using refinement types [Freeman and Pfenning, 1991], by using invariants that summarize the possible states of a reference at a program location (JayHorn [Kahsai et al., 2016]), prophecies (RustHorn [Matsushita et al., 2021]), or theory of heaps in TRICERA [Esen and Rümmer, 2021]. In HiFrog, if the property being verified depends on such complex data structures, the algorithm would work with propositional encoding, similar to CBMC. Alternatively, HiFrog can encode the problem into *EUF*. If the property being verified does not rely on such complex constructs, the result will not be spurious. Otherwise, the tool suggests choosing a more precise encoding, such as propositional encoding.

Another future direction is the use of machine learning techniques in SMT modeling. In this thesis a user has to select an SMT theory in advance for modeling the input program. It would be interesting to use data driven patterns to inform the priorities that we pursue in SMT to find a suitable theories of SMT to guide program modeling technique. We firmly believe that summarization-

based approach could be even more successful if it could make use of domain knowledge, e.g., knowing before starting the verification process which program variables are likely to be needed for reasoning over a given verification task.

Chapter 4

Incremental verification of program changes

Modern software is developed in an iterative way, by successive improvement of the last version. As a result, the software undergoes frequent minor changes, e.g., bug fixes, introduction of new features, optimizations, refactoring, and so on. The problem of updating software is that this might break existing features—bugs might get introduced. The confidence of correctness can be increased by rigorous verification before a new revision of a piece of software is checked in. Model checking techniques verify program fully automatically and exhaustively. However, most software verification approaches are not designed to support sequences of program versions, and they force each changed program to be verified from scratch which often makes re-verification computationally impractical.

As a continuation of studies described in Chapter 3, this chapter further studies the applicability of the proposed SMT-based function summarization approach to another application domain. In particular, this work addresses the problem of efficient analysis of a program after it undergoes changes. As a viable solution to this problem, incremental verification is a promising approach that aims to *reuse* the invested efforts between verification runs. The previous work [Fedyukovich et al., 2013] has shown that the idea of constructing and reusing *function summaries* across program changes is useful in BMC when using the so called bit-blasting approach, with the direct use of a SAT solver. However, due to the known complexity of bit-precise encoding it suffers from scalability issues. Using SAT-based function summaries also creates summaries that can be significantly larger than the original formulas, and that are not human-readable, impeding their reuse and maintainability.

This chapter presents an innovative approach for verification by model check-

ing of programs that undergo continuous changes. To tackle the problem of repeating the entire model checking for each new version of the program, the proposed approach verifies programs incrementally. It reuses computational history of the previous program version, namely function summaries. In particular, the summaries are over-approximations of the bounded program behaviors. The proposed solution tackles scalability issues that arise due to verification of industrial-size program versions by (i) modeling the program with fragments of quantifier-free first-order logic, in particular in *LRA* and *EUF*) which allows to leverage success of nowadays SMT solvers, and (ii) reusing and maintaining pre-computed function summaries in SMT to localize the checks of changed functions. Basing the proposed solution on SMT allows lightweight program modeling and at the same time to obtain concise function summarization. Whenever reusing of summaries is not possible straight away, the proposed algorithm repairs the summaries to maximize the chance of reusability of them for subsequent runs.

We implemented our SMT-based incremental verification algorithm with the new concept of summary repair in the UPPROVER tool. We advocate the necessity to offer various encoding options to the user. Therefore, in addition to the provided SMT-level light-weight modeling and the corresponding SMT-level summarizations supported by our incremental verifier, the tool allows adjusting the precision and efficiency with different levels of encodings. For this purpose UPPROVER enables the *LRA* and *EUF* theories (and in the future, more). This was not possible in the previous-generation tools based on bit blasting (e.g., in its predecessor EVOLCHECK [Fedyukovich et al., 2013]), and this distinguishes UPPROVER from them. Furthermore, our approach not only allows the reuse of summaries obtained from SMT-based interpolation, but also provides an innovative capability of repairing them automatically and using them in the subsequent verification runs. The extensive experimentation on the benchmark suite of primarily Linux device drivers versions demonstrates that the proposed algorithm achieves an order of magnitude speedup compared to prior approaches. The results reported in this chapter were published in the following conference papers: [Asadi et al., 2020b] and [Asadi et al., 2020a], and the journal paper [Asadi et al., 2023].

4.1 Motivating example

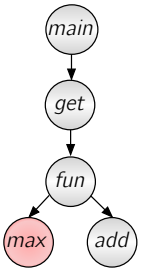
The idea of adapting summaries is motivated by the high computational burden to model check the second program from scratch. In this work, we adapt the already generated function summaries to become the function summaries of

```
1  int D; //global
2
3  int nondet(){
4   int r;
5   assume(r>0 && r<5);
6   return r;
7  }
8
9  int add(int x){
10  return x+10;
11 }
12
13 int max(int a,int b){
14  if(a >= b) return 2*a;
15  return 2*b;
16 }
17
18 int fun(int a,int b){
19  int out = max(a, b);
20  D = add(a);
21  assert(D - a >= 1);
22  return out + D;
23 }
24
25 int get(int a,int b){
26  int c = fun(a,b);
27  if(a>=c && b>=c)
28   return 2 * a;
29  return 2 * c;
30 }
31
32 void main() {
33  int a = nondet();
34  int b = nondet();
35  D = nondet();
36  int w = get(a,b);
37  assert(w >= a+b);
38 }
```

(a) Base program

```
int D;

int nondet() {
 int r;
 assume(r>0 && r<5);
 return r;
}

int add(int x){
 return x+10;
}

int max(int a,int b){
 // if(a >= b) return 2*a;
 return 2*b;
}

int fun(int a,int b){
 int out = max(a, b);
 D = add(a);
 assert(D - a >= 1);
 return out + D;
}

int get(int a,int b){
 int c = fun(a,b);
 if(a>=c && b>=c)
  return 2 * a;
 return 2 * c;
}

void main(){
 int a = nondet();
 int b = nondet();
 D = nondet();
 int w = get(a,b);
 assert(w >= a+b);
}
```

(b) Changed program



(c) Call tree

	Summary formula
$\sigma_{\mathcal{T}}(\texttt{max})$	$(ret \geq 2b) \wedge (ret \geq 2a)$
$\sigma_{\mathcal{T}}(\texttt{add})$	$ret \geq x + 10$
$\sigma_{\mathcal{T}}(\texttt{fun})$	$(ret \geq 10 + a + 2b) \wedge (ret \geq 10 + 3a)$
$\sigma_{\mathcal{T}}(\texttt{get})$	$(a > -5) \Rightarrow ((ret \geq 2a) \wedge (ret \geq 2a + 4b + 20))$

(d) Function summaries of base program

Figure 4.1. Two versions of a C program with call tree and function summaries.

the second program. This section demonstrates summary reuse and summary weakening in the proposed incremental verification approach.

Consider two programs in Figure 4.1 as the base and changed program. We call them P_1 (Figure 4.1a) and P_2 (Figure 4.1b). Both versions consist of several functions out of which one function differs, namely `max`, highlighted with red. The function `nondet` represents a non-deterministic choice (e.g. user input) which is assumed to be in a certain range. The two assert statements capture the property of the program that should always hold after an execution of the program. Program P_1 can be encoded as a *LRA* formula together with the negation of assertions.

The proposed approach first performs a bootstrapping verification for P_1 prefer-

ably encoding and solving with a light theory of SMT like *LRA*. After successful verification of P_1 using Algorithm 1, summaries of functions are created with respect to all properties, as shown in Figure 4.1d. These function summaries represent the relation between the inputs and outputs of each function, and are expressed using a formula that includes the return value of the function, denoted by the variable *ret* in Figure 4.1d. Note that the proposed tool of this chapter implements Algorithm 1 for computing summaries automatically. Summaries are stored for future usage. When it comes to verifying P_2, in order to have an efficient verification procedure, instead of performing full-verification again it is desirable to reuse the summaries of P_1. We process the changed functions of P_2 and investigate if the previous summaries are good enough to over-approximate the changes.

Summary weakening. Let us denote $\sigma_{\mathcal{T}}(\texttt{max})$ as summary of `max` and $\phi'_{\texttt{max}}$ as *LRA* encoding of function `max` in P_2. Here the summary check $\phi'_{\texttt{max}} \Rightarrow \sigma_{\mathcal{T}}(\texttt{max})$ does not succeed. Since $\sigma_{\mathcal{T}}(\texttt{max})$ is conjunctive, we can weaken the summary by dropping some conjuncts to increase the chance of being valid for the changed `max`. As shown in Figure 4.1d $\sigma_{\mathcal{T}}(\texttt{max})$ is in conjunctive form:

$$\sigma_{\mathcal{T}}(\texttt{max}) := (ret \geq 2b) \wedge (ret \geq 2a). \tag{4.1}$$

A possible weakened formula is $\sigma^w_{\mathcal{T}}(\texttt{max}) := (ret \geq 2b)$. By dropping a conjunct, the resulting formula is still a summary, but it is coarser than the previous one. Since the implication $\phi'_{\texttt{max}} \Rightarrow \sigma^w_{\mathcal{T}}(\texttt{max})$ is valid, it indicates that the weakened summary is coarse enough to capture the changed function $\phi'_{\texttt{max}}$. However, the validation check has to propagate towards the caller, i.e., `fun` to make sure if the coarse summary is a valid over-approximation in the subtree rooted at `fun`.

Summary reuse. During the validation of `fun`'s summary, i.e., $\sigma_{\mathcal{T}}(\texttt{fun})$, if there are summaries available in its subtree they are used. Since function `add` is not changed, its summary $\sigma_{\mathcal{T}}(\texttt{add})$ is reused straightforwardly as well as $\sigma^w_{\mathcal{T}}(\texttt{max})$. However, the summary check for function `fun` fails because of the change in function `max`. The implication check $\sigma_{\mathcal{T}}(\texttt{add}) \wedge \sigma^w_{\mathcal{T}}(\texttt{max}) \wedge \phi_{\texttt{fun}} \Rightarrow \sigma_{\mathcal{T}}(\texttt{fun})$ by instantiating the summary template does not succeed:

$$(D \geq a+10) \wedge (out \geq 2b) \wedge (ret = out + D) \Rightarrow (ret \geq 10+a+2b) \wedge (ret \geq 10+3a)$$

Thus the summary of function `fun` should be weakened to $(ret \geq 10 + a + 2b)$, and the summary check should be performed for function `get`, where the check succeeds. Since there are no further changed functions unprocessed, the check does not continue further to the root of the program, and we can conclude that P_2

is safe too. The weakened summaries ($\sigma^w_{\mathcal{T}}$(max) and $\sigma^w_{\mathcal{T}}$(fun)) are stored instead of the original summaries and will be reused when a new version arrives.

4.2 Incremental verification of program changes

This section presents a solution to the problem of efficient verification of a program after a change. The problem of incremental verification is stated as follows: given the original program P_1 with the safety properties included in the code, a summary mapping (certificate of correctness) σ_1 of P_1, and the changed program P_2, it is necessary to adapt σ_1 to become a summary mapping of σ_2 for P_2, or to show that P_2 does not admit a correctness certificate (i.e., has a counterexample).

Note that this thesis presents the incremental verification algorithm instantiated in the context of BMC and SMT. However, the algorithm is more general and can be applied in other approaches relying on over-approximative function summaries.

Next section presents a first version of the incremental verification algorithm and then an optimized version of the algorithm.

4.2.1 Basic algorithm of summary validation

In the proposed incremental verification algorithm, the problem of determining whether a newly changed program still meets a safety property reduces to the problem of validating the family of summaries for the new program. As a result, in practice the verification is localized to the changed parts of the system, resulting in significant run time improvements.

We customize the definitions of substitution schemes of the refinement process defined in 3.3.1. The new definitions includes set N where it stores the set of functions with invalid summaries once they were identified as invalid. The substitution scheme $Sb^N_{precise}$: $F \rightarrow \{inline\}$ inlines all the function bodies.

The substitution scheme Sb^N_{eager}: $F \rightarrow \{sum, inline\}$ inlines functions with invalid summaries accumulated in set N and otherwise employs summaries:

$$Sb^N_{eager}(g) = \begin{cases} inline, & \text{if } g \in N \\ sum, & \text{otherwise} \end{cases} \tag{4.2}$$

Note that summary mapping $\sigma_{\mathcal{T}}$ is total and all functions initially have summaries. Finally, Sb^N_{lazy}: $F \rightarrow \{sum, nondet\}$ treats functions with invalid sum-

Input: function summaries of the first version, `tree`: call-tree of the second version, Δ : set of changed functions in the second version;
Result: second version is *Safe* or *Unsafe*;

```
1  while all Δ are not processed do
2      choose the first f in the reverse postorder of tree such that f ∈ Δ;
3      if f has a summary then
4          if the summary is invalid then
5              RemoveSum(f);     // Remove the summary of function f
6              if f has a parent (f is not root) then
7                  Add the parent to Δ to be processed;
8              else
9                  return Unsafe, error trace;
10         else
11             Re-compute summaries from subtree of f by interpolation;
12 return Safe, set of valid summaries;
```

Algorithm 3: Basic summary validation in incremental verification

maries as nondeterministic calls and the rest as *sum*, as follows:

$$Sb^{N}_{lazy}(g) = \begin{cases} nondet, & \text{if } g \in N \\ sum, & \text{otherwise} \end{cases} \tag{4.3}$$

Summary validation shown in Algorithm 3, consists of a series of local validation checks for all changed function calls and their possibly affected callers, beginning at the deepest node. If a local validation succeeded, but for some function call in the subtree a summary was invalidated, the algorithm call the method `RemoveSum` to remove the summary of f straight away (line 5). Note that this local validation continues until there are no more functions to be processed, and if it succeeds, the second version is reported as **Safe**, potentially along with a set of valid summaries that are made available for checking the next version.

It is worth noting that when the validation check propagates to the call tree root, i.e., *main* function, it corresponds to the classical BMC check where all functions are inlined. Thus in the worst case, since the programs that we check are bounded (a decidable problem), the algorithm falls backs to classical BMC.

Input: Program $P = (F, f_{main})$ with function calls F; the set of changed functions $\Delta \subseteq F$; total summary mapping $\sigma_{\mathcal{T}} : F \to S$ that maps functions to summaries; theory $\mathcal{T}$;
Output: $\langle$**Safe**, $\sigma_{\mathcal{T}}\rangle$ or $\langle$**Unsafe**, $CE\rangle$
Data: $N \subseteq F$ set of functions with invalidated summaries

```
1  WL ← Δ;
2  N ← ∅;
3  while WL ≠ ∅ do
4      pick f ∈ WL, s.t. ∀h ∈ WL : ¬subtree(f, h);
5      WL ← WL \ {f};
6      if f ∉ N then                                  // if has summary?
7          φ_f ← CreateFormula(P, σ_T, T, f, Sb^N_precise);   // create formula
8          ⟨result, π, ϵ⟩ ← CheckSAT(φ_f ∧ ¬σ_T(f));          // run SMT solver
9          if result = SAT then
10             if f ≠ f_main then
11                 WL ← WL ∪ {parent(f)};
12             else
13                 return ⟨Unsafe, ExtractCex(ϵ)⟩;
14             ⟨σ_T(f), N⟩ ← WeakenSummary(P, σ_T, f, N);
15         else // res = UNSAT
16             for all h ∈ F s.t. subtree(f, h) do
17                 σ_T(h) ← σ_T(h) ∧ GetInterpolant(π, h);
18                 N ← N \ {h};
19 return ⟨Safe, σ_T⟩; // second version safe, return updated
       summary
```

Algorithm 4: UPPROVER: Summary validation and repair in SMT

4.2.2 Algorithm with summary repair

This section presents the complete algorithm for incremental verification of program changes. First it describes the main points of Algorithm 4. Then it describes the important subroutines of the algorithm; (i) an improvement of the algorithm with the summary weakening in Sec. 4.2.3, and (ii) summary refinement in Sec. 4.2.4.

The goal of Algorithm 4 is performing incremental verification with the ability to *repair* over-approximating summaries of the updated program. The key idea behind the *summary repair* approach is to circumvent the deletion of invalidated

function summaries and instead attempt to adapt them to the changed functions. The proposed solution repairs an over-approximating summary of the program function which is coarse enough to enable rapid check but at the same time strong enough to cover more changes in an incremental checking scenario. The repair is done via two strategies: the first *weakens* the invalid summary formulas by removing the broken parts, and the second *strengthens* the weakened summary by recomputing the corresponding interpolant and adding missing parts. The conjunction of newly repaired summaries is kept for subsequent uses. The refinement procedure accompanies our incremental summary validation algorithm for dealing with spurious behaviors that might be introduced due to imprecise over-approximative summaries.

Overall, the proposed solution proceeds as follows. First, a program together with safety properties is modeled in SMT, and if properties hold, function summaries are computed. Then once a change arrives, it first determines whether the old summaries for functions are still valid after the change. This validation phase is local to the change and tends to be computationally inexpensive since it considers only the changed function bodies, their old summaries, and possibly the summaries of the predecessors of the changed functions. If this local validation phase succeeds, the new program version is also safe. If local validation fails, the approach attempts to widen the scope of the search while still maintaining some locality, by propagating the validation check to the callers of the modified functions. After each successful validation, any invalidated summaries become a candidate to be repaired and are made available for checking the next program version.

Algorithm 4 (similar to Algorithm 3) considers two versions of the program, P_1 and P_2, and the function summaries of P_1. If P_1 or its function summaries are not available (e.g., at the initial stage), a bootstrapping run (Algorithm 1) is required to verify the whole program P_2 to generate the summaries, which are then maintained during the subsequent verification runs.

As input, the pseudocode takes the new program (P_2), the set of functions that have been identified as changed Δ, the theory $\mathcal{T}$, and the summaries as a total mapping $\sigma_{\mathcal{T}}$ from functions F to the set of all summaries S. Initially in case no summary exists for f (e.g., newly introduced in P_2) its summary is initialized as *false*. By initializing the summary of f as *false*, we are being explicit about the fact that we do not yet have any summary. As output, it reports either *Unsafe* with a concrete counterexample *CE*, or *Safe* with a possibly updated total mapping representing the summaries.

The algorithm maintains a worklist *WL* of function calls that need to be checked against the pre-computed summaries. Initially, *WL* is populated by a

set of functions with code changes, namely Δ (line 1). Then the algorithm repeatedly chooses f from *WL* so that no function in the subtree of f exists in *WL* (line 4). Then it removes a function f from *WL* and attempts to check the validity of the corresponding summary in the second version. Note that this bottom-up traversal of the call tree ensures that summaries in the subtree of f have been already checked (shown either valid or invalid). The algorithm also maintains the set N to store the set of functions with invalid summaries and aims at repairing all of the summaries that were identified as invalid.

The if-condition at line 6 checks whether the summary of f is not invalid (i.e., has a summary). If so, `CreateFormula` constructs the formula ϕ_f that encodes the subtree of f. Note that here the substitution scenario *precise* is used where all function calls in the subtree of f are naively inlined. Later Section 4.2.4 introduces a more efficient way for constructing the formula.

The validation check of pre-computed summaries occurs in line 8. The validity of implication $\phi_f \Rightarrow \sigma_{\mathcal{T}}(f)$ is equivalent to the unsatisfiability of the negation of the formula, $\phi_f \wedge \neg\sigma_{\mathcal{T}}(f)$. This local formula is sent to an SMT solver for deciding its satisfiability. Performing the local check determines whether the summary is still a valid over-approximation of the new function's behavior. If the *result* is UNSAT, the validation is successful and the summary covers the changed function. Here, the algorithm obtains a proof of unsatisfiability π which is used to compute new summaries to update the invalid or missing summaries (line 17). This is called *strengthening* procedure in our approach. If *result* is SAT, the validation of the current summary fails for the changed function (line 9). In this case, either the check is propagated to the function caller towards the root of the call tree (line 11) or a real error is identified (line 13). In the latter case, since the validation fails for the root f_{main} of the call tree, the algorithm extracts and reports a concrete counterexample from the result of the SMT query (line 13).

Note that if a function f is introduced in P_2, the caller of f is marked as changed by our difference-checker. In such a scenario, since summary of f is trivially *false*, the validation check immediately fails, f gets added to N, and the algorithm continues to check the caller. A successful validation of some ancestor of f with inlined f generates a summary for f (line 17).

Whenever a summary is identified as invalid (line 9), the sub-routine `WeakenSummary` (line 14) is called to make the summary coarser which is explained in the next section.

Input: Program $P = (F, f_{main})$ with function calls F; summary mapping $\sigma_{\mathcal{T}} : F \to S$; Theory $\mathcal{T}$; function f whose summary is being weakened, set of functions N with invalid summary
Output: Updated summary formula of f $\sigma_{\mathcal{T}}(f)$, updated invalid set N

1 For a formula $\sigma_{\mathcal{T}}(f) = \mathbb{S}_1 \wedge ... \wedge \mathbb{S}_n$, consider a set $Cands = \{\mathbb{S}_1, ..., \mathbb{S}_n\}$;
2 $\phi_f \leftarrow$ `CreateFormula`$(P, \sigma_{\mathcal{T}}, \mathcal{T}, f, Sb^N_{eager})$;
`// Create formula with substitution scheme (3.4)`
3 **while** $Cands \neq \emptyset$ **do**
4 $\langle result, \text{-}, \epsilon \rangle \leftarrow$ `CheckSAT`$(\phi_f \wedge \neg \bigwedge_{cnj \in Cands} cnj)$; `// validity`
5 **if** $result =$ UNSAT **then break**;
6 **for** $cand \in Cands$ **do**
7 **if** $\epsilon \models \neg cand$ **then**
8 $Cands \leftarrow Cands \setminus \{cand\}$;
9 $\sigma_{\mathcal{T}}(f) \leftarrow \bigwedge_{cnj \in Cands} cnj$; `// store valid summary conjuncts`
10 **if** $\sigma_{\mathcal{T}}(f) = true$ **then** $N \leftarrow N \cup \{f\}$;
11 **return** $\sigma_{\mathcal{T}}(f), N$;

Algorithm 5: `WeakenSummary`$(P, \sigma_{\mathcal{T}}, \mathcal{T}, f, N)$

4.2.3 Summary weakening

The previous algorithms of incremental verification in [Fedyukovich et al., 2013; Asadi et al., 2020b] check summaries one-by-one and whenever the validation fails, the invalidated summaries are removed straight away, thus the chance to reuse summaries becomes low. In this section instead of removing the invalidated summaries right away, our proposed algorithm attempts to compute coarser over-approximation (i.e. weaker summaries) by dropping some conjuncts, thus maximizing their usability. The key insight behind the algorithm is identifying which parts of the summary break the validity of implication, removing them from the set, and repeating the validation check.

The number of top-level conjuncts in a summary formula is a measure of generalizability of the interpolant. In some applications (see, e.g., [Komuravelli et al., 2013; Fedyukovich and Bodík, 2018]) it is useful to further abstract an over-approximation. The idea was inspired by HOUDINI algorithm [Flanagan and Leino, 2001]. Weakening a summary formula is performed by dropping the conjuncts that break the validity of the summary. Note that HOUDINI is only meaningful for the summaries with top-level structure in conjunctive form.

In Algorithm 4 once the summary turned out to be invalid (line 9) the sub-

routine `WeakenSummary` (i.e., Algorithm 5) is called to weaken the summaries. Algorithm 5 shows a simple implementation of an iterative check-and-refute cycle that iterates until the validation check of the subset of summary conjuncts succeeds. Initially, summary conjuncts are stored in the set *Cands* and as the algorithm proceeds, the conjuncts are removed if proven to be invalid.

In line 2 the PBMC formula ϕ_f for the function f is constructed. In this phase Sb^N_{eager} is used as the substitution scheme, i.e., whenever a function summary is available in the subtree of f, the summary is used as a substitute for the function body. Then in line 4 the containment of the resulting formula ϕ_f in the summary candidate is checked by the SMT solver. Once the solver generates a counterexample ϵ, it is used to prune the summary conjuncts that break the containment check. This iterates until the solver returns UNSAT. In the end, the remaining subset of summary candidates would form a new valid summary for f. In case no conjuncts are left, the function summary is assigned to the weakest possible summary, namely *true* (line 10) and it is added to the set N that contains functions whose summaries were not valid anymore in P_2.

Once the weakened summary is obtained from Algorithm 5, in Algorithm 4 the check always propagates to the caller (as the parent is already marked there) to make sure the new weakened summary is suitable in the subtree rooted at the caller. It can be the case that the weakened summary does not capture the whole relevant functionality of the changed function and needs more conjuncts (strengthening) that can be obtained during the validation check of the caller and interpolation in line 17 of Algorithm 4. Note that in this case even though the check propagates to the caller, it would be still beneficial in the sense that the actual encoding of the function body is substituted with the weakened summary, thus the overall check will be less expensive and still maintains some locality.

4.2.4 Summary refinement

In Algorithm 4 line 7 creates the PBMC formula ϕ_f in a way that all the nested functions in the subtree rooted at f are inlined. In other words, the initial substitution scheme was set to $Sb^N_{precise}$. To further speed up the incremental check while constructing the formula ϕ_f, pre-computed summaries can be used to abstract away the function calls in its subtree.

This section presents an algorithm for creating PBMC formulas in a more efficient way, then presents the proposed solution for refining the abstract summaries on demand. The pseudocode in this section can substitute lines 7 and 8 in Algorithm 4.

Algorithm 6 consists of two key points: (i) while constructing the PBMC for-

Input: Program P with function calls F, function f for which a formula is created, Summary mapping $\sigma_{\mathcal{T}} : F \to S$, theory $\mathcal{T}$, set of functions with invalid summary N;
Output: Solving result:{SAT, UNSAT}, proof of unsatisfiability π, updated $\sigma_{\mathcal{T}}$, updated N
Data: PBMC formula ϕ_f, tentative set of function calls to be refined $WL \subseteq F$, counterexample CE, mapping of substitution scheme for function calls $Sb^N : F \to \{precise, sum, nondet\}$

```
1  while true do
2      φ_f ← CreateFormula(P, σ_T, T, f, Sb^N_lazy);   // Create formula with
         substitution scheme in Eq. (3.5)
3      ⟨result, π, ϵ⟩ ← CheckSAT(φ_f ∧ ¬σ_T(f));        // run SMT solver
4      if result = SAT then
5          CE ← ExtractCex(ϵ);                          // extract error trace
6          RefCandid ← getCalls(CE);      // get refinement candidates
7          if {g ∈ RefCandid | Sb^N(g) ≠ precise} = ∅ then
8              break;                                   // nothing to refine
9          else
10             foreach g ∈ RefCandid do
11                 Sb^N(g) ← precise;     // set substitution scheme to precise
12                 σ_T(g) ← ⊤;
13                 N ← N ∪ {g};
14     else
15         return UNSAT, π, σ_T, N;
16 return SAT, CE, σ_T, N;
```

Algorithm 6: `SolveRefine`$(P, f, \sigma_{\mathcal{T}}, N)$

mula ϕ_f, whenever summaries are available in the subtree of f, they substitute the actual body of the function calls in the subtree, (ii) while checking the validity of summaries, the infeasible behaviors that are detected during analysis of abstract summaries are refined by an iterative refinement procedure.

The initial over-approximation in Algorithm 6 is set to Sb^N_{lazy} where it sets the precision for function calls in the *lazy* style. Then the PBMC formula ϕ_f is created based on Algorithm 2 with Sb^N_{lazy}. The resulting ϕ_f is expected to be substantially smaller compared to the encoding whose substitution scheme was initially set to $Sb^N_{precise}$.

Then the resulting ϕ_f is checked for the validity whether its pre-computed

summary contains the formula, i.e., $\phi_f \Rightarrow \sigma_{\mathcal{T}}(f)$ (line 3). If the resulting formula is satisfiable, it can be either a real or a spurious violation since over-approximative function summaries were used to substitute some of the nested function calls. This can be discovered by analyzing the presence of summaries along an error trace, determined by a satisfying assignment ϵ returned by a solver and by dependency analysis.

Based on the satisfying assignment the algorithm identifies the set of the summaries used along the counterexample and stores them in *RefCandid* (line 6). The algorithm applies dependency analysis that restricts *RefCandid* set to those possibly affecting the validity. Then every over-approximations (summary or nondet) in the *RefCandid* set is marked as *precise* in the next iteration (line 11). If the set is empty, the check fails and the summary is shown invalid. This refinement loop repeats until the validity of the summary is determined.

4.3 Correctness of the algorithm

This section defines *tree interpolation property* by which the correctness of Algorithm 4 can be established. Then the correctness theorem is defined assuming the tree interpolation property already holds in the interpolation algorithm. Lastly, we instantiate the proposed generic SMT-based incremental verification approach to certain theories of SMT and discuss which interpolation algorithms can guarantee tree interpolation property.

4.3.1 Tree interpolation property

Binary interpolation can be generalized so that the partitions of an unsatisfiable formula form a tree structure. We in particular concentrate on *tree interpolants*, generalizations of binary interpolants, obtained from a single proof that guarantees the tree interpolation property as defined in the following:

Definition 8 (tree interpolation property[1]) *Let $X_1 \wedge \ldots \wedge X_n \wedge Y \wedge Z$ be an unsatisfiable formula in first-order logic. Let $I_{X_1}, \ldots, I_{X_n}$ and $I_{X_1 \ldots X_n Y}$ be interpolants for interpolation instances $(X_1 \mid X_2 \wedge \ldots \wedge X_n \wedge Y \wedge Z)$, …, $(X_n \mid X_1 \wedge \ldots \wedge X_{n-1} \wedge Y \wedge Z)$, and $(X_1 \wedge \ldots \wedge X_n \wedge Y \mid Z)$, respectively. The tuple $(I_{X_1}, \ldots, I_{X_n}, Y, I_{X_1 \ldots X_n Y})$ has the* tree interpolation property *iff $I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge Y \Rightarrow I_{X_1 \ldots X_n Y}$.*

4.3.2 Correctness of the algorithm

This section discusses the correctness of the SMT-based incremental verification algorithm, i.e., given k unrolling steps, the algorithm always terminates with the correct answer with respect to k. Notice that in this thesis, program safety is considered with respect to the pre-determined unwinding bound k. In the remainder of this section, we assume the same k for both old and new programs. In case the user increases the bound for a specific loop, the corresponding function needs to be validated as if changed.

The correctness of Algorithm 4 is stated in the following theorem:

Theorem 1 *Assume the interpolation algorithm for $\mathcal{T}$ guarantees tree interpolation property. When the Algorithm 4 returns safe, then the entire program is safe, i.e., $error_{f_{main}} \wedge \Phi^{subtree}_{f_{main}} \Rightarrow \bot$.*

Proof 2 *Let f_{main} be the entry function, $\sigma_{\mathcal{T}}(f)$ be the summaries, f range over the function calls satisfying subtree(f_{main}, f), and $c_1, \ldots, c_n$ be the function calls in the (possibly empty) set of functions called by f. We first show that the properties*

$$error_{f_{main}} \wedge \sigma_{\mathcal{T}}(f_{main}) \Rightarrow \bot, \text{ and} \tag{4.4}$$

$$\sigma_{\mathcal{T}}(c_1) \wedge \ldots \wedge \sigma_{\mathcal{T}}(c_n) \wedge \beta_f \Rightarrow \sigma_{\mathcal{T}}(f) \tag{4.5}$$

are strong enough to prove that the entire program is safe, and then show that they hold in the algorithm both after successful bootstrapping and after a successful incremental verification run on a set of changes Δ.

Safety from properties (4.4) and (4.5). *After rewriting property (4.4) into $\sigma_{\mathcal{T}}(f_{main}) \Rightarrow (error_{f_{main}} \Rightarrow \bot)$, logical transitivity and iterative application of property (4.5) to substitute all interpolants on the right hand side of property (4.5) yields the inlined formula in the claim $error_{f_{main}} \wedge \Phi^{subtree}_{f_{main}} \Rightarrow \bot$.*

Bootstrapping phase. *We show that property (4.4) holds over the program call tree annotated by computed interpolants whenever bootstrapping verification in Algorithm 1 terminates. Recall that the summaries are generated only when the program is safe with respect to the property, i.e., $error_{f_{main}} \wedge \Phi^{subtree}_{f_{main}} \Rightarrow \bot$. Therefore, by definition of interpolation in Definition 1 (property (ii)), $error_{f_{main}} \wedge I_{f_{main}}$ is unsatisfiable, i.e., property (4.4) holds. Property (4.5) follows from our assumption that the interpolation algorithm guarantees the tree interpolation property. This can be seen by choosing the following partitions in Definition 8: $X_i \equiv \Phi^{subtree}_{c_i}$ for $i \in 1 \ldots n$, $Y \equiv \beta_f$, and $Z \equiv \Phi^{rest}_f$.*

Incremental phase. *Assume that properties (4.4) and (4.5) hold before the changes in Δ are introduced. We show that if Algorithm 4 running on a set of*

changes Δ successfully returns **Safe**, *both properties are maintained, from which the claim in the theorem follows. If Algorithm 4 successfully terminates, then each function call c obtains an updated summary $\sigma_{\mathcal{T}}(c)$ (line 17) when some of its predecessor f passed the summary validity check (line 15). Otherwise, the check propagates towards the root of the call tree and eventually may lead to an* UNSAFE *result. Thus, it suffices to show that the recomputed or repaired interpolants satisfy property (4.5). For this purpose, we again rely on the assumption that the interpolation algorithm guarantees the tree interpolation property. When constructing the formula of a function, Algorithm 2 uses all valid summaries in the subtree of the function. This is sound as we know from property (i) of Definition 1 that $\sigma_{\mathcal{T}}(c_i) \Rightarrow I_{X_i}$ where I_{X_i} is an interpolant obtained from the proof of unsatisfiability corresponding to the successful validity check of the predecessor of c_i.*

In case some change in Δ is not contained in pre-computed summary $\sigma_{\mathcal{T}}(f)$ but the algorithm introduced a weakened summary $\sigma^{w}_{\mathcal{T}}(f)$ that contains the change, the algorithm still propagates to the caller function (line 11) to determine whether property (4.5) holds. In case $\sigma^{w}_{\mathcal{T}}(f)$ is not precise enough in the subtree of the caller, the algorithm proceeds with the refinement of the weak summaries. Once the refined check succeeds, the proof of unsatisfiability is used (through interpolation) to strengthen $\sigma^{w}_{\mathcal{T}}(f)$ (line 17). Technically, the weakened summary and the recomputed summary are conjoined to form a new summary $\sigma^{\text{itp}}_{\mathcal{T}}(f) \wedge \sigma^{w}_{\mathcal{T}}(f)$. Again relying on the assumption that the interpolation algorithm guarantees the tree interpolation property, the recomputed and repaired interpolants satisfy property (4.5). □

4.3.3 Interpolation algorithms in a concrete theory

This section instantiates the proposed generic SMT-based incremental verification approach to certain theories of SMT. Interpolation algorithms that are the low-level primitives in the proposed approach are discussed based on whether they guarantee the tree interpolation property or not. The theories of our interest are *LRA* and *EUF*. Having different theories and interpolation algorithms is of great practical interest since the choice of a good interpolation algorithm may well determine whether an application terminates quickly or diverges.

In the theory of linear arithmetic over the reals, *LRA*, there are several efficient proof-based interpolation algorithms proposed in the literature so that the resulting interpolants can differ in ways that have practical importance in their use in incremental verification. For the theory of *LRA*, many SMT solvers produce interpolants using application of the Farkas lemma [Schrijver, 1999]. The most widely used approach computes weighted sum defined by Farkas coefficients of

all inequalities appearing in A part of $(A|B)$ [McMillan, 2005]. The interpolant computed in this way is always a *single* inequality. We call this approach Farkas interpolation procedure and denote it as Itp^F.

Recently [Blicha et al., 2019] introduced a new algorithm called decomposing Farkas interpolation procedure which is able to compute interpolants in linear arithmetic in the form of a *conjunction of inequalities*. The algorithm is an extension of Itp^F; it uses techniques from linear algebra to identify and separate independent components from the interpolant structure. We denote the decomposing interpolation procedure as Itp^D. Intuitively, Itp^Dworks as follows: Instead of using the whole weighted sum of A, it tries to *decompose* the vector of weights (Farkas coefficients) into several vectors. This effectively decomposes the single sum into several sub-sums. If each of the sub-sum still eliminates all A-local variables, the resulting inequalities can be *conjoined* together to yield a valid interpolant.

Since Itp^D is able to produce interpolants in the form of a conjunction of inequalities, it provides the opportunity to make more effective use of summaries in our incremental verification algorithm. Hence, Algorithm 5 can benefit from computing coarser over-approximation (i.e. weaker interpolants) by dropping conjuncts. In the later sections, we will experimentally verify the usefulness of the decomposition scheme by comparing two *LRA* interpolation algorithms.

As for the theory of *EUF*, we use the *EUF* Interpolation algorithm in [Alt, Hyvärinen, Asadi and Sharygina, 2017] that relies on a congruence [McMillan, 2005] graph data structure constructed while solving an *EUF* problem. *EUF* interpolation algorithm combines propositional and *EUF* interpolation which is useful for a model checking setting and some of the conjuncts in *EUF* interpolants are coming from the propositional structure.

For the propositional part, we use the well-known Pudlák's interpolation algorithm [Pudlák, 1997] which we treat as an instance of D'Silva et al.'s labeling interpolation system [D'Silva et al., 2010]. This is more suitable for function summaries than McMillan's, since it constructs weaker interpolants and can capture more changes in incremental verification. For scalability reason, combining a propositional interpolation algorithm with LRA interpolation algorithm is shown beneficial as an over-approximation of bit-vectors in software model checking. The approach first constructs the refutation using standard SMT methods. The interpolation works then by labeling each clause with an interpolant starting from the leaf clauses towards the empty clause. The leaf theory clauses are labeled using an LRA interpolation after which the propositional labeling can be applied in a standard way. For more details we refer the reader to [Asadi et al., 2020a].

Tree interpolants are used in our incremental algorithm for determining the satisfiability of a first-order logic formula that reuses summaries generated after the unsatisfiability of a slightly different formula is determined. Among the widely used family of interpolation algorithms for *LRA*, *EUF*, and *Prop* we rely on the ones that can guarantee the tree interpolation property, and thus are suitable for the application in incremental verification of program versions.

The following two lemmas state which *LRA* interpolation algorithms can guarantee tree interpolation property. The proofs can be found in [Asadi et al., 2020a] where we discussed under which conditions *LRA* interpolation procedures guarantee tree interpolation property.

Lemma 2 *LRA interpolants computed by Farkas interpolation algorithm have tree interpolation property.*

Lemma 3 *LRA interpolants computed by decomposing Farkas interpolation algorithm with gradual decomposition defined in [Asadi et al., 2020a] have tree interpolation property.*

The following lemma considers the tree interpolation property of interpolants generated from the same resolution proof in *EUF*, proven in [Christ and Hoenicke, 2016].

Lemma 4 *EUF interpolation algorithm [Alt, Hyvärinen, Asadi and Sharygina, 2017] guarantees the tree interpolation property.*

In a purely propositional setting, we rely on the results from [Gurfinkel et al., 2013; Sery et al., 2012b] which proves the correctness of the propositional tree interpolation algorithms, stated in the following lemma.

Lemma 5 *Propositional interpolation algorithm introduced by Pudlák guarantees the tree interpolation property.*

The proposed solution relies on the above lemmas for the correctness of the incremental verification algorithm in the context of bounded model checking. However, the correctness of the algorithm is not solely restricted to the preservation of the tree interpolation property *by construction*. Instead the correctness of the algorithm can be preserved by checking for the tree interpolation property *on-the-fly*. This means that the algorithm can still produce a correct output even if the tree interpolation property is passed by performing real-time checks in verification run.

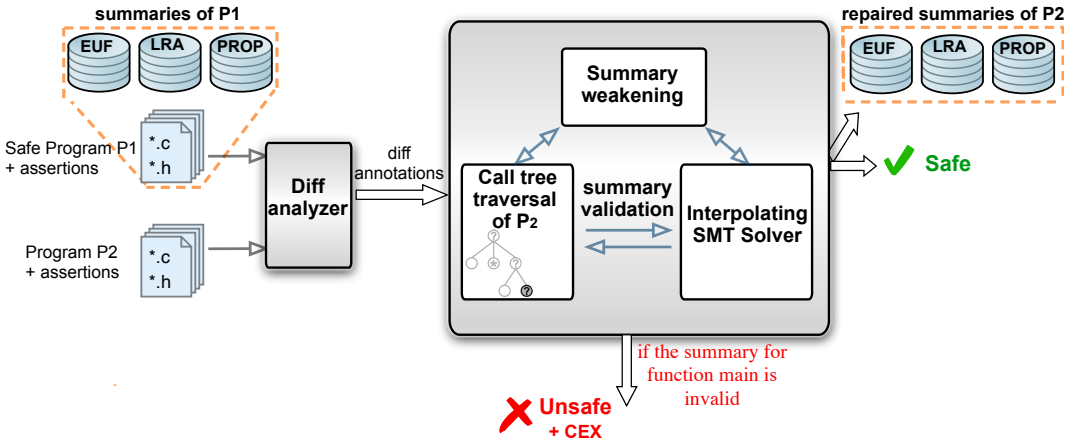


Figure 4.2. Overview of the UpProver architecture. UpProver operates at one particular level of precision at each run.

since the algorithm allows checks for the tree interpolation property on-the-fly.

It is worth mentioning that our incremental algorithm with theory $\mathcal{T}$ gives the same result as the verification from scratch used in the bootstrapping with the theory $\mathcal{T}$. Although automatically identifying a proper level of encoding is non-trivial (and not a subject of this thesis), our approach allows various encoding options to the user. In case the algorithm is instantiated with less-precise theory (e.g., *EUF*), if a bug is reported, it might be due to the abstract theory usage, and it is recommended to repeat the verification with a more precise theory (and accordingly, more precise summaries).

4.4 Tool architecture and implementation

This section describes the implementation of our algorithms in the UpProver tool which is a bounded model checker written in C++. UpProver concentrates on incremental verification of program versions written in C. After each successful verification run, it maintains a database of function summaries to store its outputs, which become available as inputs for verification of each subsequent program version. For bootstrapping verification UpProver uses HiFrog standalone bounded model checker. For satisfiability checks and interpolation UpProver uses SMT solver OpenSMT. For pre-processing as a front-end, UpProver uses the framework from CProver 5.11[2] to symbolic encoding of C by transforming

[2] http://www.cprover.org/

C program to a monolithic unrolled BMC representation that we use as a basis for producing the final partitioned logical formula.

The architecture of UPPROVER tool is depicted in Figure 4.2. UPPROVER implements proposed algorithms by maintaining three levels of precision—*LRA*, *EUF*, and purely propositional logic (*Prop*)—to check the validity of pre-computed summaries. The rest of this section describes UPPROVER's key components in more detail.

Difference analyzer. UPPROVER performs source code differencing at the level of SSA forms for both the old and the new program to identify a set of functions with code changes. It annotates the lines of code changed between P_1 and P_2. This defines the scope of summary validations. The user may choose an inexpensive syntax-level difference or a more expensive and precise semantic-level difference that compares programs after some normalization and translation to an intermediate representation [Fedyukovich et al., 2013]. The functions that have been identified as changed are stored in set Δ in Algorithm 4.

Call tree traversal. The call tree traversal guides the check of pre-computed summaries for the modified functions in bottom-up order. It exploits the SMT solver to perform summary validation. When necessary it performs an upwards refinement to identify parent functions to be rechecked using SMT solver or performs summary refinement to refine the imprecise summaries in the subtree.

Summary repair. Summaries of P_1 (of the selected level of precision) are taken as input and used in the incremental summary validation when necessary. The tool iteratively checks if the summaries are valid for P_2 and repairs them on demand, possibly by iterative weakening and then strengthening using interpolation over the refined summaries.

SMT solving and interpolation engine. For checking BMC queries and computing interpolants, UPPROVER interacts with the SMT solver OPENSMT. The solver produces a quantifier-free first-order interpolant as a combination of interpolants from resolution refutations [Alt et al., 2016], proofs obtained from a run of a congruence closure algorithm in EUF [Alt, Hyvärinen, Asadi and Sharygina, 2017], Farkas coefficients obtained from the Simplex algorithm in LRA [Blicha et al., 2019], and Decomposed Farkas interpolation in LRA [Blicha et al., 2019].

Summary storage. UPPROVER takes summaries σ_1 of P_1 as input and outputs summaries σ_2 of P_2. The user defines the precision of σ_1, and it uniquely determines the precision of σ_2. In the best-case scenario, the tool validates σ_1 and copies it to σ_2. When some of the summaries require repair, the tool produces new interpolants from the successful validity checks of the parent functions and stores them as the corresponding summaries in σ_2 (while all other summaries are again copied from σ_1). No summaries are generated when the tool returns

Unsafe.

4.5 Experimental evaluation

To evaluate the proposed algorithm, we aimed to answer the following research questions:

RQ 1 Is the use of SMT instead of SAT in incremental verification efficient for real-world programs?

RQ 2 Is reusing function summaries beneficial in incremental verification?

RQ 3 How does the proposed approach compare with other incremental verifiers?

Benchmarks and setup.[3] We chose 2670 revision pairs of Linux kernel device drivers from [Beyer et al., 2013]. The benchmarks were chosen so that they are parsable by CPROVER5.11, and contain at least one safety property (code assertion). The crafted benchmarks mainly stress-test our implementation and include function additions/deletions, signature changes, semantic/syntactic changes in function-bodies, etc. In addition, we included 240 tricky hand-crafted smaller programs. The crafted benchmarks mainly stress-test our implementation and have changes such as function addition/deletion, signature change, semantic/syntactic change in function-bodies, etc. On average, the benchmarks have 16'000 LOC, the longest ones reaching almost 71'000 LOC. For each run, we set a memory limit of 10 GB and a CPU time limit of 900 seconds. The experiments were run on a CentOS 7.5 x86_64 system with two Intel Xeon E5-2650 CPUs, clocked at 2.30 GHz, and 20 (2 x 10) cores. UPPROVER is available as open-source software. Technical information about the setup of the tool can be found at `http://verify.inf.usi.ch/upprover`.

4.5.1 Demonstrating usefulness of different theories

To answer RQ 1, we compare the run time of incremental verification using different encodings. Each point in Figure 4.3 corresponds to an incremental verification run of a single benchmark (changed program P_2). Figure 4.3a and 4.3b

[3] The experimentation data including benchmarks, evaluation results, and the source code repositories are available online at `http://verify.inf.usi.ch/upprover`

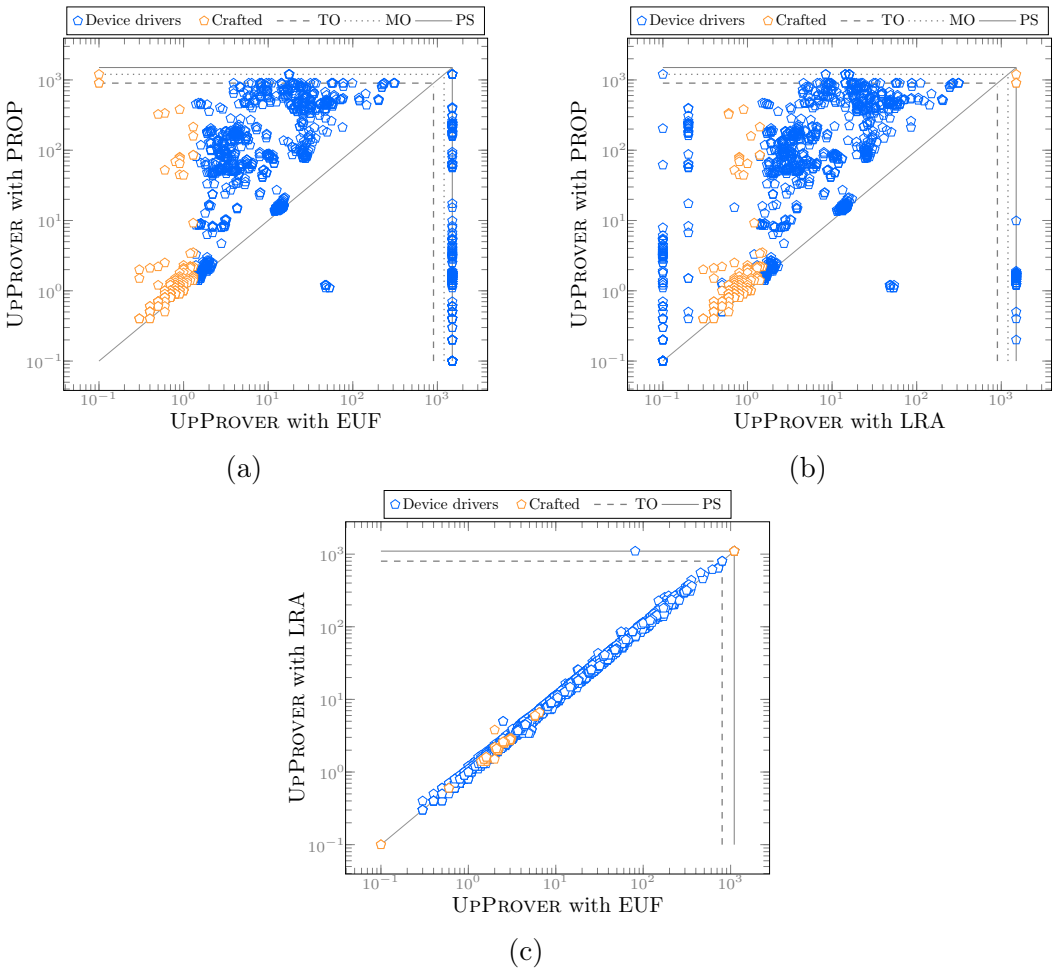


Figure 4.3. Demonstrating the impact of theory encoding by comparing timings of *LRA*/*EUF* encodings in UpProver vs. *Prop* encoding. The inner lines TO and MO refer to the time and memory limit. The outer lines PS refer to the results that are potentially spurious due to the use of abstract theory.

compare the *EUF*/*LRA*-based encodings in UpProver against the *Prop*-based encoding in UpProver[4]. Almost universally, whenever run time exceeds one second, it is an order of magnitude faster to verify with *LRA* and *EUF* than with *Prop*.

In addition, a large number of benchmarks on the top horizontal lines suggests that it is possible to solve many more instances with *LRA*/*EUF*-based en-

[4]The *Prop*-based summary reuse in UpProver depicted in Figure 4.4c uses the same algorithm from its predecessor eVolCheck but has been significantly optimized compared to its earlier version. Thus *Prop*-based UpProver can be seen as representative of eVolCheck.

coding than with *Prop*-based encoding. However, the loss of precision is seen on the benchmarks on the vertical line labeled potentially spurious (PS), indicating if the verification result using *LRA*/*EUF* is unsafe, the result might be spurious because of abstraction. Since UPPROVER only operates at one particular level of precision at each run, once the tool reports *Unsafe* in *EUF*/*LRA* it is recommended to confirm it by a stronger theory encoding.[5]

The results for benchmarks show the trade-off between the precision and run time of incremental verification. In fact the theories are complementary. This can be contrasted to the plot in Figure 4.3c where we extracted benchmarks that have successful bootstrapping phase in both *LRA* and *EUF* and ended up with 2516 versions out of which 50% are strictly faster in *EUF* and 30% are strictly faster in *LRA*. The time overhead observed in *LRA* compared to *EUF* is due to the more expensive decision procedure.

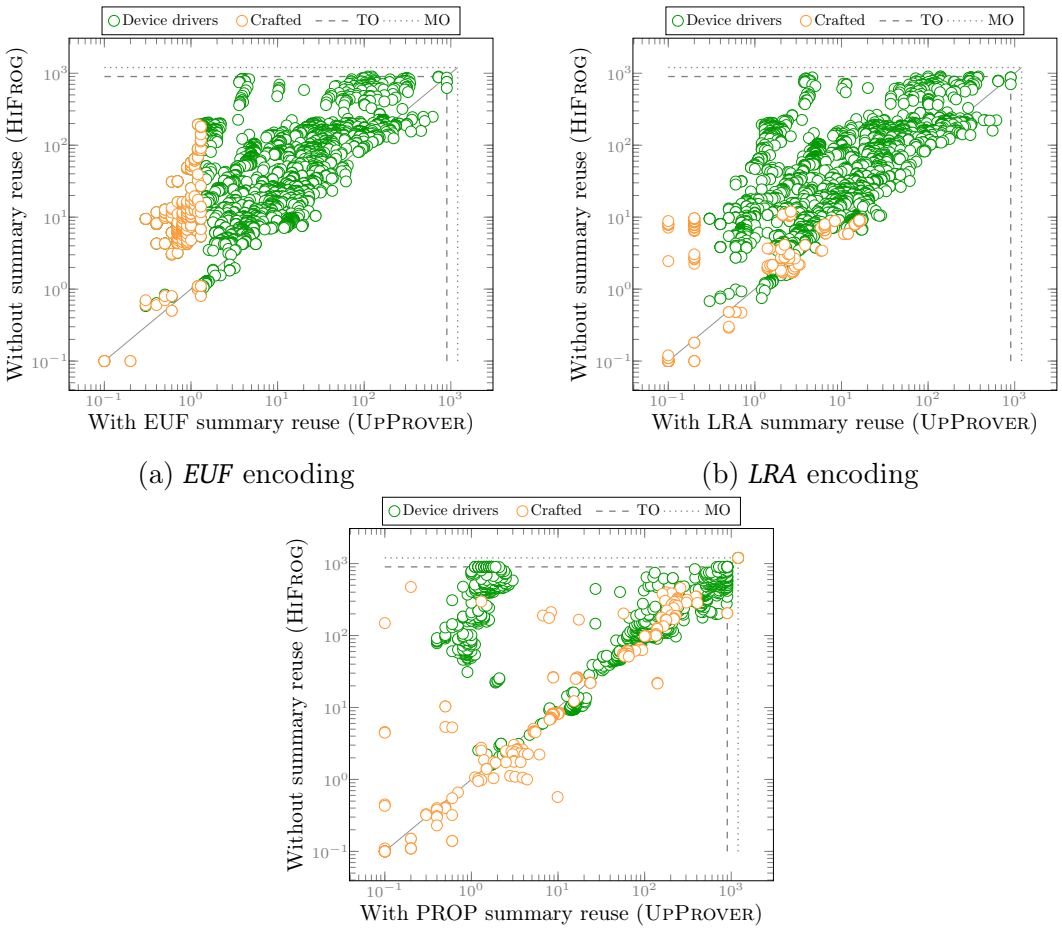
4.5.2 Demonstrating the effect of summary reuse

To answer RQ 2 we demonstrate how summary reuse and summary repair benefit incremental verification. To this end we first compare the performance of the tool with and without summary reuse, and then we show how summary repair results in more summaries while the overhead of producing more summaries is negligible.

4.5.2.1 Incremental BMC vs monolithic BMC

The purpose of this section is to compare verification time of *reusing* the summary against not reusing it. As opposed to summary-based incremental checking in UPPROVER that maintains and reuses over-approximating summaries of the functions across program versions, a standalone BMC tool, e.g., HIFROG and CBMC, creates a monolithic BMC formula and solves it as a standalone run without reusing information from previous runs of other versions. In this section, we compare UPPROVER with HIFROG as a representative sample of non-incremental BMC tool. This choice is made because both tools use the same infrastructure from CPROVER v5.11 to transform C program to obtain a basic unrolled BMC representation that UPPROVER uses as a basis for producing the final logical formula. Since UPPROVER and HIFROG share the same parser and the same SMT solver OPENSMT the comparison is not affected by unrelated implementation differences.

[5] The work in [Asadi et al., 2018] demonstrated the relation among some theories of SMT from the perspective of over-approximation.



(a) *EUF* encoding (b) *LRA* encoding

(c) *Prop* encoding

Figure 4.4. Incremental verification time of UpProver versus non-incremental verification time of HiFrog on (a) EUF, (b) LRA, and (c) PROP encoding.

The plots in Figure 4.4 compare UpProver against HiFrog (non-incremental) with three encodings *EUF*, *LRA*, and *Prop*. Each point in each plot corresponds to verification run of a changed program, with the running time of UpProver when *reusing* the summary of the first version on x-axis, and the running time of HiFrog without reuse on y-axis. The plots demonstrate the performance gains of incremental verification with reusing function summaries against not reusing it. A large amount of points on the upper triangle lets us conclude that UpProver is an order of magnitude faster than the corresponding non-incremental verification for most benchmarks.

Table 4.1 provides more details on each encoding that would clarify the scatter plots further. We use acronyms P_1 and P_2 for two versions of a program. The

Table 4.1. Number of benchmarks solved by each encoding in UpProver.

Results	EUF		LRA		PROP	
	P_1	P_2	P_1	P_2	P_1	P_2
Safe	2529	2514	2686	2663	1249	1186
Unsafe	268*	11*	95*	20*	31	15
Time Out (TO)	113	4	129	3	1536	37
Memory Out (MO)	0	0	0	0	94	11
Uniquely verified	-	11	-	41	-	4
No summary (*No-Sum*)	-	381	-	224	-	1661
Total program versions	2910					

row *Safe* indicates the number of programs reported safe by each encoding. In total out of 2910 benchmarks, UPPROVER with *LRA* verified safe the largest amount of P_2, i.e., 92% and with *EUF* and *Prop* verified 85% and 40% respectively. The row *Unsafe* indicates the number of programs reported unsafe by each encoding. The unsafe results might be spurious when theory encodings were used (indicated by an asterisk). In other words, for *EUF*/*LRA*, some of the *Unsafe* results might be a real error, but some are definitely false alarms. The row TO shows that while UPPROVER with *Prop* times out in 53% of the benchmarks, for *LRA* and *EUF* this happens for less than 1%. The row MO shows that with *Prop* encoding, UPPROVER exceeds the memory limit in 105 benchmarks of P_1 and P_2, for *LRA* and *EUF* this does not occur.

The row *uniquely verified* programs indicates how many P_2 can be incrementally verified safe in each encoding exclusively. The count of the uniquely verified using *LRA* is comparable to other encodings where 41 instances are not solved by any other encoding. Even though, *LRA* solves the most safe program versions, there are several benchmarks that can be uniquely verified by *EUF* (11 instances) and by *Prop* (4 instances). These distinctly verified programs in each encoding can be included in a portfolio.

The row *No-Sum* represents the cases where there are no possibility to perform incremental verification because no function summaries were produced in the bootstrapping phase. This can happen when the bootstrapping verification of P_1 results in *Unsafe*, TO, or MO. The *Prop* encoding results in the highest rate of *No-Sum*, i.e., 57% (1661) which is the summation of Time Out, Memory Out, and Unsafe results of bootstrapping of P_1. This asserts that using the rigid approach

of bit-blasting for majority of our real-world benchmarks obstructs the incremental verification. On the contrary, *LRA* and *EUF* encodings have a relatively small rate of *No-Sum*.

It is worth noting that we compared the results of UPPROVER with the expected results of SVCOMP, since most benchmark names indicate the expected result. Out of 2910 pairs of benchmarks, 2794 pairs had both versions classified as safe. However, for the remaining 116 benchmarks, no expected result was provided and they were marked as unknown, thus we are unable to obtain precise numbers for *Unsafe* benchmarks. For *Safe* benchmarks, we never encountered any disagreement with the expected results. This indicates a high degree of accuracy for UPPROVER in verifying safe benchmarks with theories. However, for *EUF*/*LRA*, *Unsafe* results might be either real error or false alarms. Due to these unknown benchmarks, we are unable to report the exact number of false alarms for *Unsafe* benchmarks in theory and we mark them with asterisk.

The overall findings from our experiments show evidence for the following key points: precision and performance-gain present a trade-off. UPPROVER with *EUF* and *LRA* have a better performance compared to the bit precise *Prop* encoding and are crucial for scalability. At the same time, there is a small number of benchmarks that require *Prop*. Despite the fact that bit-blasted models are more expensive to check than the *EUF* and *LRA* models, we find it surprising that the light-weight encodings succeed so often. In practice, the encodings complement each other, and the results imply an approach where the user gradually tries different precisions until one is found that suits the programs at hand.

4.5.2.2 Number of repaired summaries

In this section, we measure the number of repaired summaries that are generated by two out of the box *LRA* interpolation algorithms. Recall the two phases of the repair in the proposed algorithm: once an existing summary is marked invalid for a changed function f, Algorithm 5 first *weakens* the summary by removing broken conjuncts of the summary. In case the weakened summary is not strong enough, Algorithm 4 (line 17) *strengthens* the weakened summary by recomputing interpolants for f and conjoining with the weakened summaries. We shortlisted 43 pairs of C programs whose summaries was repaired at least once during incremental verification.

Figure 4.5 depicts the count of two types of repair in *LRA*. We use acronyms $\mathcal{W}$ for weakening, Itp^D for decomposing Farkas interpolation algorithm, and Itp^F for Farkas interpolation algorithm. We ran Algorithm 1 with *LRA* encoding over the shortlisted programs and generated 3837 *LRA* summaries in total, out of which

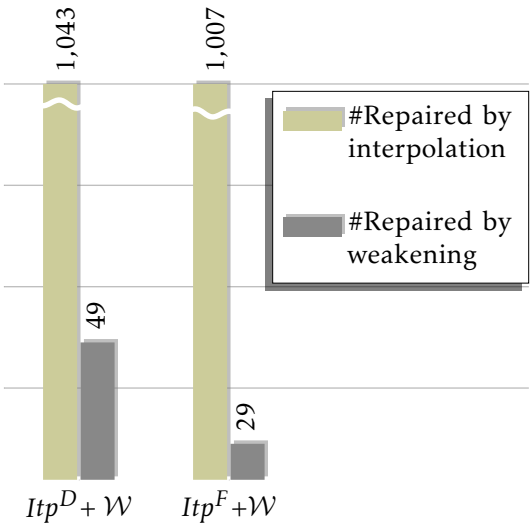


Figure 4.5. Number of repaired summaries in LRA.

1043 summaries were strengthened by Itp^D interpolation and 49 summaries were weakened by $\mathcal{W}$. The remaining summaries were either used without any repair, or unused at all due to their corresponding functions did not have change, thus no summary validation performed. Similarly, 1007 and 29 of summaries were repaired by Itp^F and by $\mathcal{W}$ respectively.

We can also view the result of this experiment from a different perspective. It can be seen as a way to test how good the interpolants are and how beneficial is the summary weakening. In the summary validation phase in UPPROVER, the more general interpolants are, the more likely they contain changes of the functions in the second version. Experimenting with Itp^D and Itp^F algorithms for producing *LRA* interpolants, shows that almost always they were as good as they could obtain more summaries with $\mathcal{W}$ technique. Observing that 49 summaries out of 3837 could be weakened further, implies that pre-computed interpolants are already as weak as possible but strong enough to be safe.

Now that we have an estimate of the number of function summaries repaired by $\mathcal{W}$, the question becomes how much overhead time this approach adds up to the overall process of UPPROVER.

4.5.2.3 Overhead of summary repair

In the following, we study the overhead of weakening ($\mathcal{W}$) process in the verification time. Figure 4.6 compares the runtime of UPPROVER when reusing sum-

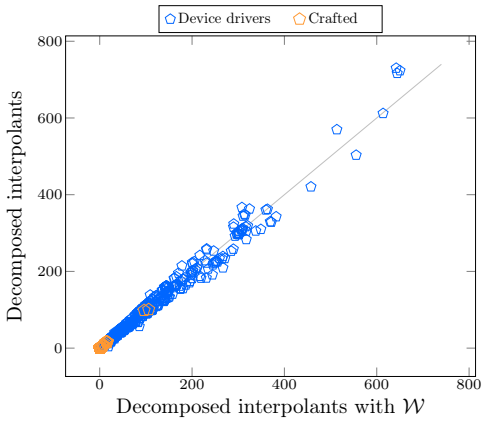


Figure 4.6. Incremental verification time of UpProver with *LRA* decomposed interpolants with and without weakening ($\mathcal{W}$).

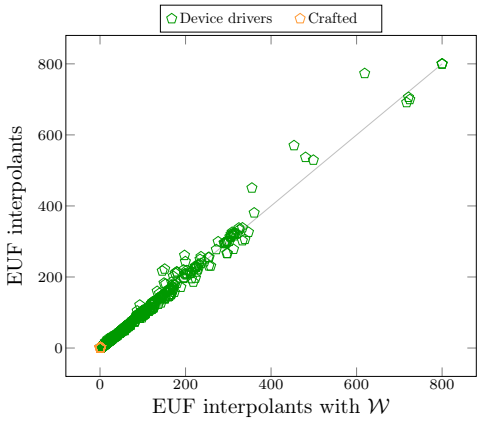


Figure 4.7. Incremental verification time of UpProver with *EUF* with and without weakening ($\mathcal{W}$).

maries generated by Itp^D with and without $\mathcal{W}$[6]. In 60% of the benchmarks Itp^D with $\mathcal{W}$ outperform or equal to Itp^D. In 32% strictly Itp^D with $\mathcal{W}$ faster than Itp^D, whereas in 41% Itp^D is strictly faster than Itp^D with $\mathcal{W}$. This reveals that in most of the cases not only weakening of summaries did not introduce considerable overhead, but also sometimes outperform the cases without $\mathcal{W}$. For instance, the cases that are above 200 seconds, 29 benchmarks with Itp^D and $\mathcal{W}$ strictly outperform Itp^D, suggesting that the Itp^D is not undesirable.

Overall, the results imply that summaries repaired by $\mathcal{W}$ and strengthened by Itp^D are beneficial in a sense that leads to more summaries in the end, and even shows speed-up in some benchmarks compared to disabling $\mathcal{W}$, thus did not affect the overall performance.

In Figure 4.7 we compared the runtime of UpProver in theory of *EUF* with and without $\mathcal{W}$. Concretely, at the area of around 350 to 800 seconds we observe 6 points above diagonal line confirms out performance of $\mathcal{W}$, whereas 3 points below diagonal shows that pure *EUF* without $\mathcal{W}$ performs better.

Table 4.2 gives further details on 14 representative pairs of benchmarks whose change type are substantially different and whose summaries had a chance of being repaired at least once. The table consists of four configurations in *LRA*. Each row refers to a pair (P_1, P_2) of programs. The columns highlighted in gray color refers to enabling *summary weakening* feature in the algorithm. The columns highlighted in blue color refers to disabling *summary weakening* feature in which

[6]The results with Itp^F is similar.

Table 4.2. Detailed verification results for four setups in *LRA*

Bootstrapping			Diff			Farkas			Farkas with $\mathcal{W}$				Decomposed Farkas			Decomposed Farkas with $\mathcal{W}$			
program pair	interpolation time (s)	initial summary #	preserved function #	changed (Δ) #	diff time (s)	validation check #	repaired by itp #	incremental time (s)	validation check #	repaired by $\mathcal{W}$ #	repaired by itp #	incremental time (s)	validation check #	repaired by itp #	incremental time (s)	validation check #	repaired by $\mathcal{W}$ #	repaired by itp #	incremental time (s)
1	0.1	19	23	1	0.1	2	1	1.8	6	1	0	2.2	2	1	1.8	6	1	0	2
2	0.1	19	22	2	0.1	2	1	1.9	5	1	3	2.3	2	1	1.9	5	1	3	2.2
3	0.1	19	23	1	0.1	2	3	2.8	5	1	4	4	2	3	2.8	5	1	4	3.8
4	0.1	16	18	1	0.1	2	1	3.2	4	1	0	**3**	2	1	3.1	4	1	0	**2.9**
5	0.1	5	14	5	0.1	5	2	3	8	1	0	3	5	2	3	8	1	0	**2.8**
6	0.1	5	14	5	0.1	5	2	3	8	1	0	3	5	2	3.2	8	1	0	**2.7**
7	0.1	4	3	4	0.1	5	4	0.2	5	0	4	0.2	5	4	0.2	7	1	3	0.2
8	3.9	370	947	159	0.2	33	50	42	33	0	50	**41**	33	50	41	34	1	49	**36**
9	1.7	448	1401	665	1.5	72	313	161	72	0	313	**160**	72	311	186	78	1	306	**166**
10	1.4	184	634	39	0.1	21	2	42.7	27	1	0	**39**	21	2	43	27	1	0	43
11	18.8	765	1265	787	0.5	214	268	147	225	2	268	**146**	214	268	144	225	2	268	144
12	3.9	325	763	131	0.2	47	87	114	52	1	88	114	47	87	113	51	1	88	**112**
13	8.4	474	1300	52	0.5	27	64	64	37	1	66	77	27	64	57	37	1	66	86
14	4.4	412	1060	63	0.4	34	53	65	36	1	54	**58**	34	53	59	36	1	54	66

the summaries are repaired only by re-computation through interpolation. The column *interpolation time* shows the time for generating all summaries after successful bootstrapping of P_1 and *initial summary* the number of function summaries in P_1 which are non-trivial, i.e., are not simply *true* formula.The column *preserved* refers to the number of functions that stayed the same in P_2 and Δ to the number of changed functions in P_2. The column *diff time* shows the time taken by difference-checker to identify changes between P_1 and P_2. The column *validation check* refers to iterative validation checks of summaries for checking the containment of summaries. The columns *repaired by itp* and *repaired by* $\mathcal{W}$ indicate the number of newly established summaries by re-computation through interpolation and weakening respectively.

Overall, the numbers in the column *validation check* are higher when $\mathcal{W}$ is used since the algorithm has to iterate more to find a subset of conjuncts in the summary formula. Nevertheless, there are benchmarks, highlighted in bold, that show that the performance improves with the use of $\mathcal{W}$. This happens because the weakened summaries can contain more changes and thus be more suitable for incremental verification. Interestingly, the columns *repaired by itp* and *repaired by* $\mathcal{W}$ show that whenever $\mathcal{W}$ is used, more summary formulas are produced. This is desirable for incremental verification. We see from the experiments that the increase in the number of summary formulas results both directly from weakening the summaries and indirectly because each successful validation check generates new interpolants.

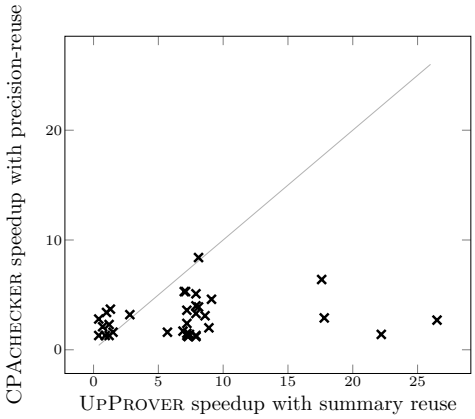


Figure 4.8. Speedup in UpProver with *LRA* summary reuse vs. speedup in CPAchecker with precision reuse.

4.5.3 Comparison of UpProver and CPAchecker

To answer RQ 3, we compare UpProver with a widely-used tool CPAchecker which is able to perform incremental verification by reusing abstraction precisions. It is an orthogonal technique to ours, i.e., it is an unbounded verifier and aims at finding loop invariants. Thus, comparing running times does not make sense since running times in UpProver crucially depend on the chosen bound.[7] Instead, we focus on comparing the speedups obtained with the two techniques since the change of a bound affects a speedup less.

Here we report the results only on device driver instances which both tools could handle. Out of 250 device drivers categories given in `https://www.sosy-lab.org/research/cpa-reuse/predicate.html`, we selected 34 categories which are suitable for UpProver.[8] These categories contain in total 903 verification tasks.

Figure 4.8 shows the comparison of speedup in UpProver and CPAchecker. A large amount of points on the lower triangle reveals that summary reuse in UpProver achieves superior speedup than the precision reuse in CPAchecker. The average speedup in UpProver with *LRA* summary-reuse is 7.3 with a standard deviation of 6 and in CPAchecker the average speedup is 2.9 with a standard deviation of 1.7. UpProver reported 4 slowdowns among 34 categories, whereas

[7] For instance, the average running times in CPAchecker is 285.3 seconds and in UpProver with *LRA* is 13.4 seconds for chosen bound 5. For other bounds UpProver would have different average running times.

[8] The reported version of UpProver is restricted by its dependency on the CProver5.11 framework which impedes its frontend from processing some benchmarks.

this was not the case for CPACHECKER.

4.6 Related work

The problem of incremental verification is not as studied as model checking of standalone programs. There are still several techniques and tools [Conway et al., 2005; Sery et al., 2012b; Beyer et al., 2013; Trostanetski et al., 2017; He et al., 2018; Rothenberg et al., 2018; Beyer et al., 2020] which the central incentive behind these lines of work is the ability to reuse intermediate results that were costly computed in previous verification runs, thus achieving performance speedup in the verification of later revisions compared to verification of programs in isolation. Works in this area vary based on the underlying non-incremental verification approach used, which defines what information to be reused and how efficiently so. Various information has been proposed for reuse, including state-space graphs [Lauterburg et al., 2008], constraint solving results [Visser et al., 2012], and automata-based trace abstraction [Rothenberg et al., 2018]. However, these groups of techniques are orthogonal to the proposed approach as we store and reuse the interpolation-based function summaries in the context of BMC for verifying revisions of programs. Moreover, apart from pure reusing the previous computations, the proposed technique repairs the already generated summaries to increase the chance of reusability.

Another approach towards efficiently verifying evolving programs, which is the one we compare in this thesis, is based on the reuse of previously abstraction precision in predicate abstraction CPACHECKER [Beyer et al., 2013]. Apart from the inherent difference that CPACHECKER is an unbounded verifier and UPPROVER is a bounded model checker, we differ from this in that we base the proposed approach on proof-based computing of interpolants and repairing them on the fly, therefore in some sense are able to store more information from the previous runs.

Other techniques for incremental verification of program revisions include reusing inductive invariants in Constrained Horn Clause across programs by guessing syntactically matching variable names [Fedyukovich et al., 2014, 2016]. However, these techniques can be applied only for programs sharing the same loop structure. In contrast, the proposed approach is applicable for all sorts of program changes in a bounded model. However, when the changes are substantial, there would not be much summary reuse even with the summary repair.

Other techniques for verifying program versions are based on *relational verification* (also known as regression verification or equivalence checking) which are

used to prove equivalence of closely related program versions. To tackle the problem of formally verifying all program revisions various techniques and tools have been proposed for the last two decades [Hardin et al., 1996; Godlin and Strichman, 2009]. Existing relational verification approaches leverage the similarities between two programs so that they verify the first revision, and then prove that every pair of successive revisions is equivalent [Lahiri et al., 2013; Pick et al., 2018; Shemer et al., 2019; Mordvinov and Fedyukovich, 2019; Felsing et al., 2014]. Since checking exact equivalence is hard to fulfill and not always practical, there is a group of techniques that check for partial equivalence between pairs of procedures [Yang et al., 2014; Lahiri et al., 2012; Godlin and Strichman, 2009] or check conditional equivalence under certain input constraints [Lahiri et al., 2013]. Despite the evident success, these techniques are sound but not complete.

The work in [Godefroid et al., 2011] investigates the effects of code changes on function summaries used in dynamic test generation. This approach is also known as white-box fuzzing which includes running a program while simultaneously symbolically executing the program to collect constraints on inputs. The aim of [Godefroid et al., 2011] is to discover summaries that were affected by the modifications and cannot be reused in the new program version. Since this approach relies on testing, it suffers from the problem of path explosion, i.e., all program paths are not covered. However, this work is orthogonal to the proposed approach as we construct and repair function summaries in a symbolical way, thus the proposed approach allows encoding of all paths of an unrolled program into a single formula.

A group of related work includes techniques using interpolation-based function summaries (such as [McMillan, 2010; Albarghouthi et al., 2012b; Sery et al., 2011]) for the standalone programs. Although these works do not support program versions, we believe that the proposed incremental algorithm may be instantiated in their context similar to how we instantiated it in the context of HiFrog. The bootstrapping phase of the proposed solution of this chapter is built on top of HiFrog, an approach for constructing and reusing interpolation-based function summaries in the context of Bounded Model Checking. In later work [Asadi et al., 2018] we propose to use function summaries more efficiently by lifting function summaries into various SMT levels, thus information obtained from one level of abstraction could be reused at a different level of abstraction.

The work we find most closely related to ours is eVolCheck, the predecessor of UpProver, which works only at the propositional level and uses the function summaries only in a bit-precise encoding. Consequently, despite being an incremental approach, eVolCheck is computationally expensive in many cases in

practice. Whereas, the proposed approach allows flexibility in balancing between verification performance and precision through both program encoding and the choice of summarization algorithms. As a result of the high-level encodings, UPPROVER summaries serve as human-readable *certificates of correctness* expressing function specifications.

4.7 Synopsis

We addressed the problem of verifying a large number of programs, in particular, when they are closely related. To avoid expensive full re-verification of each program version and repeating a significant amount of work over and over again, our proposed algorithm operates incrementally by attempting to maximally reuse the results from any previous computations.

In this chapter, we exploited an SMT-based family of summaries that condenses the relevant information from a previous verification run to localize and speed up the checks of new program versions. In this approach, the problem of determining whether a newly changed program still meets a safety property reduces to the problem of validating the family of summaries for the new program. As a result, in practice the verification is localized to the changed parts of the system, resulting in significant run time improvements. The key contribution of this work lies in enabling this flexibility by SMT encoding and exploiting the SMT summarization. Having SMT encoding allows for a lot of flexibility when reusing and repairing the summaries leading to the optimization of the whole process which was not possible in the previous SAT-based approach. To achieve incrementality, the proposed algorithm constructs and reuses SMT-based function summaries to over-approximate program functions. It also provides an innovative capability of repairing previously computed summaries by means of iterative weakening and strengthening procedures. Moreover, it offers an efficient way of building formulas and refining them on-the-fly. Through extensive experimentation, we demonstrate that the proposed approach advances the state of the art in incremental verification of program revisions and is significantly more efficient than its predecessor EVOLCHECK and non-incremental BMC approach.

4.8 Limitation and Future work

While the results of the incremental verification of program revisions are highly encouraging, there are limitations on the usefulness of the results in today's soft-

ware engineering practices, e.g., for substantial program changes such as exchanging libraries or data structures with non-local changes. A restriction of UPPROVER algorithm is that the considered program changes should be small, such that the properties of the earlier program version mostly carry over to the new version. Another shortcomings of this approach in practice is that it only considers somewhat small changes. More research is needed in this area, and therefore, it would be interesting to investigate incremental verification of program versions without restricting our approach to specific local changes. For instance, in the case of structural changes, such as merging two function calls into a single call, an interesting idea would be to glue both of the old summaries and the code between their bodies, and eliminate local variables using quantifier elimination techniques which should not appear in the merged summary.

As future work, one can investigate incremental verification of program versions without bounds, i.e., unbounded incremental verification of program revisions. This requires to reduce the problem of determining whether a changed program still meets the safety property, w.r.t. which the invariants were created, to the problem of adapting these invariants on the changed program. Thus the key idea will be lifting safe inductive invariants across program modifications. In case the complete invariant lifting is not possible, one can leverage the state-of-the-art invariant synthesizers to construct invariants for the missing parts of the invariant. In the potential future work one can perform integrating a portfolio of well-known invariant synthesis algorithms into our incremental verification framework and systematically evaluate their impact on incremental verification of different tasks. This work can be based on Constrained Horn Clauses to take full advantage of concisely expressing the verification problem.

Another potential avenue for enhancing the refinement process is the development of new heuristics for candidate selection. This thesis focuses on localized refinement of theories via the technique of theory refinement, wherein a single counterexample is utilized to determine the refinement. However, it would be worthwhile to explore a proof-based heuristic for refining the theories, which could potentially yield more relevant refinements. In situations where a larger pool of counterexamples is available for refinement, utilizing a single counterexample may prove difficult for generalization purposes. Hence, exploring alternative refinement techniques could prove fruitful for achieving more effective results. This will be a topic of interest for future research.

Chapter 5

Model checking by theory transformation

As motivated before, one of the major issue with the SMT-based program verification is that as the level of abstraction increases, missing important details of the program model becomes problematic. Precision is traded for performance by increasing the abstraction level of the model.

The previous chapters proposed an approach for SMT-based BMC. However, the main issue with SMT encoding of programs is that the light-weight theories are often imprecise. This means if a program is encoded in SMT, it may not be a ready-to-use solution for verification; it would require various (sometimes major) tuning to be reliable. Exploring such trade-offs between precision and the right level of abstraction is a challenge in program verification.

This chapter addresses this problem with an incremental verification approach that alternates precision of the program modules on demand. The idea is to model a program using the lightest possible (i.e., less expensive) theories that suffice to verify the desired property. This chapter proposes a new abstraction refinement approach to support the proposed integrated method and remove possible spurious behaviors introduced by the consequent lost of precision. Finally, in order to make the full advantage of already generated abstract models this chapter proposes a technique for tuning the abstractions via transformation of one theory to another one. Designing the *theory interface* enables migrating information among formulas in different theories.

To make the verification scalable, the proposed technique employs safe over-approximations for the program based on both function summaries and light-weight SMT theories. An over-approximating *function summary* enables reuse of information among verification runs. If during verification it turns out that the

precision is too low, the proposed approach lazily strengthens all affected summaries or the theory through an iterative refinement procedure. The resulting summarization framework provides a natural and light-weight approach for carrying information between different theories. To the best of our knowledge this is the first integrated system for SMT-based model checking in which a sequence of safety properties is verified incrementally. With keeping the ultimate goal in mind, incremental verification scalable to large-scale programs, we break down the contributions of this chapter into several parts:

- A novel approach to incremental verification that lazily identifies, among several suitable candidates, the lightest level of encoding for each given property.
- A theory interface for exchanging function summaries among formulas in different theories.
- An algorithm to leverage both function summaries and the overall precision of the program encoding that in practice demonstrates a competitive performance on a range of large-scale programs where the state-of-the-art model checker CBMC runs out of time or memory.

5.1 Overview of the technique

In this section, we give a high-level overview of the proposed approach. As programs become larger and more complex, they need more elaborated specifications to be verified for safety. Specifications with *multiple properties* are expensive to check as a significant amount of work is repeated over and over again. To overcome this issue, a verifier needs to operate incrementally. That is, the results obtained while verifying different properties should be reused to avoid wasting resources. When a specification involves certain amount of closely related properties, the incremental approaches are capable of avoiding verifying each property from scratch, and instead, they automatically identify and focus on small "deltas" in the verification conditions (see the solution presented in Chapter 4).

Verification approaches based on SMT represent a program together with a specification in first-order logic. Often the specification is naturally expressible as a set of individual properties. Given a specification consisting of multiple properties, each property requires its own encoding that is precise enough to show the absence of spurious counterexamples to the property. In a real sense, this means

that each formula requires its own *theory*: for example, some properties might be provable with a lightweight and inexpensive encoding, such as the one to the theory of equality and uninterpreted functions (EUF), while other properties might require expensive bit-precise reasoning. Identifying automatically which theory is suitable for verifying each property is challenging. In the incremental verification setting, maintaining such a framework gives new challenges.

Summaries are constructed using Craig interpolation after a successful verification run for one property and used as a light-weight replacement of the precise encoding of the corresponding functions while verifying other properties. In this study, we propose an algorithm that effectively *incorporates different theories* for incremental verification of multiple properties via creation, reuse, and refinement of function summaries.

Overal, the proposed algorithm works as follows. Given a program, a sequence of properties to verify, and an initially empty set of function summaries in several available theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, the algorithm encodes the program and the current property using the least precise theory $\mathcal{T}_1$ and the least precise summaries available. In case the algorithm finds a proof, the result is sound since we guarantee that both the theories and the summaries always over-approximate the concrete program. The proposed algorithm starts with imprecise encodings since, if sufficient for proving a property, it lowers the cost of summarization and results in more compact summaries. If no proof is found, the algorithm increases the precision lazily. Assume that the problem is currently encoded using the theory $\mathcal{T}_i$. In the phase called *local refinement*, the algorithm sequentially adds summaries translated from theories $\mathcal{T}_j$, $j \neq i$ to $\mathcal{T}_i$ and checks if the property in this encoding is provable. The algorithm enters the second phase, *global refinement*, where the problem is encoded in a more precise theory $\mathcal{T}_{i+1}$, only when all summaries are already tried on theory $\mathcal{T}_i$. Then the algorithm returns to the local refinement again. Similarly to [Sery et al., 2011], the proposed algorithm is capable of generating new function summaries and identifying actual bugs. The proposed refinement procedure is driven by a counterexample-guided analysis that distinguishes spurious counterexamples from the real ones.

5.2 Motivating example

Figure 5.1 (left) shows a C program with a function call containing non-linear operations and two user-defined assertions. The (simplified) encoding of the C program to an SMT formula is shown in Figure 5.1 (right). The resulting formula consists of five parts: a conjunct representing function `main`, two equiva-

```
int a, b, c;

void func() {
 c = b;
 if (a > 0) a = b;
 int m = 0;
 for (int i = 0; i < 100; i++)
 m += a*b;
 b = m;
}

int main() {
 a = nondet(); b = nondet();
 if (a <= 0) return -1;
 func();
 assert(a == c);
 if (a > 0) {
  func();
  if (c > 10) assert(a > 7);
 }
 return 0;
}
```

Program in C with non-linear arithmetic

```
//encoding of 1st call of 'func'
c_1 = b_0 ∧
a_1 = ite(a_0 > 0, b_0, a_0) ∧
m_0 = 0 ∧
L_UNW_1 ∧
b_1 = m_10 ∧

//encoding of 2nd call of 'func'
c_2 = b_1 ∧
a_2 = ite(a_1 > 0, b_1, a_1) ∧
m_11 = 0 ∧
L_UNW_2 ∧
b_2 = m_21 ∧

//encoding of function 'main'
a_0 > 0 ∧ a_1 > 0 ∧ c_2 > 10

/negation of 1st assertion
¬(a_1 = c_1)

//negation of 2nd assertion
¬(a_2 > 7)
```

Modular encoding into SMT formula suitable for summarization

Figure 5.1. Program in C with non-linear arithmetic.

lent (modulo renaming) conjuncts representing calls of *func*, and two conjuncts representing the negated assertions. As customary in BMC, each program variable has its indexed copies (induced by the single static assignment form). The formulas L_UNW$_i$, $i \in \{1,2\}$, represent a loop unwinding.

The proposed approach can verify the two assertions in the code Figure 5.1 incrementally. It is not hard to see that the program is safe with respect to both assertions. However, verification of this program using bit-precise encoding is expensive.

The proposed algorithm tries the less precise but easier to solve theory of EUF as the level of abstraction first, leading to successful verification of the first assertion almost immediately. Moreover, it generates and stores a summary for function `func`. To verify the second assertion, reasoning over theory of LRA is necessary. It cannot be verified at the same level of abstraction as the first assertion, so the algorithm moves to the more precise encoding using linear arithmetic. The proposed algorithm presented later in the chapter enables to translate the

summary for `func` from EUF to LRA and to reuse it to successfully verify the second assertion.

5.3 Theory-based model refinement

This section presents a general framework that allows a translation back and forth among theories of SMT with different level of precision.

This study views the problem of bounded model checking of C programs as a decision problem which is (i) decidable, and (ii) not based on Nelson-Oppen theory combination [Nelson and Oppen, 1979]. This chapter therefore concentrates in the proposed framework on four theories of interest: the quantifier-free theories of *EUF*, *LRA*, non-linear real arithmetic (NRA), and bit-vectors (*BV*)[1]. As a result, we obtain a decision procedure that has a relatively low complexity. The proposed framework, called *theory interface*, provides a common place from which the theories are instantiated, and to which they can also be converted back. This theory interface is not aimed to be passed to an SMT solver, but instead provides an infrastructure through which an instance from one theory can be converted to an instance from another theory.

The transformation from a theory $\mathcal{T}$ to the theory interface and back can be expressed in the theory-specific instantiations of the following rules, where $[\phi]^{\mathcal{T}}$ denotes that the expression ϕ is encoded using theory $\mathcal{T}$:

$$\frac{[f(\mathbf{t})]^{\mathcal{T}}}{f([\mathbf{t}]^{\mathcal{T}})} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ in } \mathcal{T} \qquad\qquad \frac{[f(\mathbf{t})]^{\mathcal{T}}}{v_{f(\mathbf{t})}} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ not in } \mathcal{T} \tag{5.1}$$

We use the notation $f(\mathbf{t})$ to abbreviate $f(t_1, \ldots, t_n)$, and $f([\mathbf{t}]^{\mathcal{T}})$ to abbreviate $f([t_1]^{\mathcal{T}}, \ldots, [t_n]^{\mathcal{T}})$. Above we write $f([\mathbf{t}]^{\mathcal{T}})$ *in* $\mathcal{T}$, if $f \in \Sigma$ and there is a derivation recursively using the rules (5.1) such that $f([\mathbf{t}]^{\mathcal{T}})$ is expressible in $\mathcal{T}$. This chapter denotes by $v_{f(\mathbf{t})}$ a variable that is unique to the expression $f(\mathbf{t})$. For example, the expression $f(x, x)$ is not expressible in the theory of linear real arithmetic if f is the multiplication operation and x is a variable, and therefore the result of applying the rules (5.1) is v_{x*x}. To simplify slightly the notation, we define a bijection $\mathcal{M}$ that maps terms $f(\mathbf{t})$ to the variables $v_{f(\mathbf{t})}$. For completeness, this

[1] For the signature of bit-vectors, we use a modification presented in [Hyvärinen et al., 2017] that preserves the high-level programming language structures to facilitate the proofs of over-approximation.

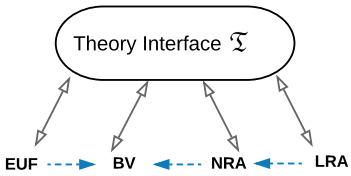


Figure 5.2. Theory interface between EUF, LRA, NRA, and BV. The horizontal arrows demonstrate the relation among these theories from the perspective of over-approximation. This relation is a part of the contribution of this study.

chapter presents the three rules for transforming non-atomic formulas

$$\frac{[\phi_1 \wedge \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \wedge [\phi_2]^{\mathcal{T}}} \qquad \frac{[\phi_1 \vee \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \vee [\phi_2]^{\mathcal{T}}} \qquad \frac{[\neg\phi]^{\mathcal{T}}}{\neg[\phi]^{\mathcal{T}}} \tag{5.2}$$

that are independent of a theory and thus common to all transformations.

5.3.1 Theory Interface

A *theory interface* T^{uni} is a general representation for formulas that we use for transformation among theories. Figure 5.2 outlines a communication among our four theories of interest. Because this chapter aims at using from early on a light-weight theory that suffices for reasoning, over-approximation among theories is at the core of speeding up the solving procedure. In the rest of this section, we formally define a theory interface and establish a relation among theories in a sound way.

Definition 9 (Theory interface T^{uni}) *Let $\mathcal{S}_\Sigma$ be a set of* formulas *and $\mathcal{R}_\Sigma$ be a set of* terms *(recall Section 2.4). Given a sequence of theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ with signatures $\Sigma_1, \ldots, \Sigma_n$ respectively, a theory interface T^{uni} is a tuple $\langle \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$ where $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \cup \cdots \cup \Sigma_n$, and each $\mathcal{M}_i$ is a* bijective *mapping $\mathcal{M}_i : (\mathcal{S}_\Sigma \cup \mathcal{R}_\Sigma) \setminus (\mathcal{S}_{\Sigma_i} \cup \mathcal{R}_{\Sigma_i}) \to \mathcal{X}_i$ where $\{\mathcal{X}_i\}_{0<i\leq n}$ are pairwise disjoint sets of unique variables not used anywhere else.*

Intuitively, $\mathcal{M}_i$ replaces the formulas and terms that are not expressible in theory $\mathcal{T}_i$ by unique fresh variables. Note that for every $\mathcal{T}_i$, $\mathcal{S}_{\Sigma_i} \subseteq \mathcal{S}_\Sigma$ and $\mathcal{R}_{\Sigma_i} \subseteq \mathcal{R}_\Sigma$.

The projection of T^{uni} to one of the theories $\mathcal{T}_i$ is done by the following rules. First, if $f(\mathbf{t}) \in \mathcal{S}_{\Sigma_i}$ (i.e., is expressible in theory $\mathcal{T}_i$), then it is projected to $\mathcal{T}_i$

without changes. Second, if $f(\mathbf{t}) \notin \mathcal{S}_{\Sigma_i}$ (i.e., is not expressible in theory $\mathcal{T}_i$), then we replace it by a fresh symbol $\mathcal{M}_i(f(\mathbf{t})) \stackrel{\text{def}}{=} v_{f(\mathbf{t})} \in \mathcal{X}_i$. For transformation in the opposite direction, i.e., $\mathcal{T}_i$ to T^{uni}, we define the inverse function $\mathcal{M}_i^{-1}$ as $\mathcal{M}_i^{-1} : v_{f(\mathbf{t})} \mapsto f(\mathbf{t})$ for $v_{f(\mathbf{t})}$ in the range of $\mathcal{M}_i$.

In the following, we develop a set of translation functions to different theories and build the over-approximation relation among these translation functions. Given a formula ϕ in theory interface T^{uni} and an arbitrary theory $\mathcal{T}$, we write $Tr_{\mathcal{T}}(\phi)$ for the translation from ϕ to $\mathcal{T}$.

Definition 10 (over-approximation) *Let ϕ be a formula in T^{uni}, and $\mathcal{T}_1$ and $\mathcal{T}_2$ two arbitrary theories. The two translation functions, $Tr_{\mathcal{T}_1}(\phi)$ and $Tr_{\mathcal{T}_2}(\phi)$ convert the original formula ϕ into $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We say that $\mathcal{T}_1$ over-approximates $\mathcal{T}_2$ if*

$$Tr_{\mathcal{T}_1}(\phi) \models \bot \text{ implies } Tr_{\mathcal{T}_2}(\phi) \models \bot \tag{5.3}$$

This section gives the specifics for the theories *EUF* and *LRA*, but in the interest of space, provide the rules of transformation from theory interface to *BV* and NRA in Appendix A. To establish the over-approximation relation, we assume in this chapter that the programs being verified admit no overflows or underflows, and that their semantics can be exactly captured by *BV*.

Definition 11 (Theory of EUF) *Let $\mathcal{X}$ be a set of variables and $\mathcal{F}$ be a set of function symbols with arities. An Equality logic formula with uninterpreted functions (EUF) is defined by the grammar*

$$\begin{aligned}
trm &::= const \\
&\quad | \, var \\
&\quad | \, f(trm, \ldots, trm) \qquad \textit{where } f \textit{ is uninterpreted} \\
fla &::= Bvar \\
&\quad | \, p(trm, \ldots, trm) \qquad \textit{where } p \textit{ is uninterpreted} \\
&\quad | \, trm = trm \mid trm \neq trm \mid \top \mid \bot \mid \neg fla \\
&\quad | \, fla \wedge fla \mid fla \vee fla \mid
\end{aligned}$$

where fla is a quantifier-free formula, $var \in \mathcal{X}$, $f \in \mathcal{F}$, and $const \in \mathcal{C}$. With the exception of equality and disequality ($=, \neq$), function and predicate symbols are treated as uninterpreted.

Semantically, EUF has the axioms of reflexivity, symmetry and transitivity for the symbol of equality, and congruence axiom for function and predicate symbols

$(\mathbf{x} = \mathbf{y}) \rightarrow (f(\mathbf{x}) = f(\mathbf{y}))$ and $(\mathbf{x} = \mathbf{y}) \rightarrow (p(\mathbf{x}) \leftrightarrow p(\mathbf{y}))$ where $\mathbf{x} = \mathbf{y}$ is an abbreviation for $(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$ and f and p are function and predicate symbols, respectively, of arity n.

Definition 12 (Theory of LRA) *A quantifier-free formula in the language of theory of Linear Real Arithmetic (LRA) is defined by the following grammar:*

$$
\begin{aligned}
trm ::= & \; const \\
& \mid var \\
& \mid const * var \\
& \mid f(trm, \ldots, trm) \qquad \textit{where } f \in \{+\} \\
fla ::= & \; Bvar \\
& \mid p(trm, \ldots, trm) \qquad \textit{where } p \in \{\leq, <\} \\
& \mid \top \mid \bot \mid \neg fla \\
& \mid fla \wedge fla \mid fla \vee fla \mid
\end{aligned}
$$

where var are variables, and const is a rational number.

5.3.2 Encoding of theory interface into specific theories

Light-weight theories help removing overly complex or irrelevant details from the encoding of a program whenever possible. We define the following rules for the theory-specific part of the transformation from T^{uni} to *EUF*:

$$
\begin{aligned}
&\frac{[v]^{EUF}}{v} \;\; v \text{ is a variable or a constant} \\
&\frac{[t_1 = t_2]^{EUF}}{[t_1]^{EUF} = [t_2]^{EUF}} \\
&\frac{[t_1 \bowtie t_2]^{EUF}}{(\neg[t_1 = t_2]^{EUF}) \wedge ([t_1]^{EUF} \bowtie [t_2]^{EUF})} \;\; \bowtie \in \{>, <\} \\
&\frac{[f(\mathbf{t})]^{EUF}}{f([\mathbf{t}]^{EUF})} \;\; \text{otherwise}
\end{aligned}
\tag{5.4}
$$

Note that in the third rule of (5.4), if the function symbol $>$ or $<$ is applied over the terms of theory interface, it can be simply translated into a disequality in *EUF*. All the other cases in the signature of theory interface which cannot be applied in the first three rules such as $\{\leq, \geq, \ldots\}$ are handled by the fourth rule.

Theorem 2 *For every* $\phi \in T^{uni}$, $Tr_{EUF}(\phi) \models \bot$ *implies* $Tr_{BV}(\phi) \models \bot$.

Proof 3 *We show that every model in BV can be translated to a model in EUF. Assume that there is a satisfying assignment in BV, such that* $a = b$ *holds for two bitvectors* a *and* b. *This can be trivially translated to an equality* $a = b$ *in EUF. In case of equality of two function applications* $f(a) = f(b)$, *we utilize the congruence rule in EUF, assuming that each function in BV is implemented as a deterministic circuit.*

We define the following rules to transform T^{uni} to *LRA* [2]:

$$\frac{[t_1 = t_2]^{LRA}}{([t_1]^{LRA} \leq [t_2]^{LRA}) \wedge ([t_2]^{LRA} \leq [t_1]^{LRA})} \quad (5.5.1)$$

$$\frac{[v]^{LRA}}{v} \quad v \text{ is a variable or an integer constant} \quad (5.5.2)$$

$$\frac{[t_1 + t_2]^{LRA}}{[t_1]^{LRA} + [t_2]^{LRA}} \quad (5.5.3)$$

$$\frac{[-t_1]^{LRA}}{(-1) * [t_1]^{LRA}} \quad (5.5.4) \qquad (5.5)$$

$$\frac{[t_1 * t_2]^{LRA}}{[t_1]^{LRA} * [t_2]^{LRA}} \quad t_1 \text{ or } t_2 \text{ is an integer constant} \quad (5.5.5)$$

$$\frac{[f(\mathbf{t})]^{LRA}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise} \quad (5.5.6)$$

The rule (5.5.6) uniquely associates the expression with a fresh variable. Essentially this rule is used for over-approximation of all the expressions that cannot be expressed in sufficient precision in *LRA*. The rules (5.5.2) and (5.5.5) operate only on integer constants in order to preserve the soundness of translation between *LRA* and *BV*. Example 3 illustrates this case in detail.

Example 3 (Over-approximation of BV by RA) *Consider the following excerpt of a program written in C:* `int x = 1; int y = 0.5 * x; assert ( y == 0 );` *Let* $\phi \stackrel{\text{def}}{=} x = 1 \wedge 0.5 * x = 0$ *represent the corresponding SMT representation. Following the semantics of C, a bit-precise encoding of* ϕ *is satisfiable since* $0.5 * 1$

[2] We assume that before undergoing a transformation, a preprocessing is done for the sake of normalization, e.g., $-1 * 2 * x$ is normalized to $-2 * x$.

is truncated to 0. However, an LRA-encoding of ϕ is unsatisfiable. According to Definition 10, this means that LRA does not *over-approximate BV. In order to get that over-approximating behavior, we impose restrictions on LRA rules (5.5.2) and (5.5.5) and apply rule (5.5.6) when these restrictions are not met. The translation applied to ϕ results in $x = 1 \wedge v_{0.5*x} = 0$ which is satisfiable in LRA. The same restrictions are imposed in* NRA *(see Appendix A).*

Theorem 3 *For every $\phi \in T^{uni}$, $Tr_{LRA}(\phi) \models \bot$ implies $Tr_{BV}(\phi) \models \bot$.*

Proof 4 *We show that every model in BV can be translated to a model in LRA. Assume that there are no overflows or underflows in BV. This guarantees that the models of all arithmetic operations in BV are also models in LRA.*

5.3.3 Decoding theories to the Theory Interface

The previous section describes the instantiation from the theory interface to a specific theory of interest. This section presents the inverse, that is, transforming from a theory to the theory interface. Such steps are necessary in order to build the over-approximation relation among theories. The key insight is to use the mapping $\mathcal{M}^{-1}$. The transformation from *EUF* to T^{uni} is defined by the following rules:

$$\frac{[t_1 = t_2]^{T^{uni}}}{[t_1]^{T^{uni}} = [t_2]^{T^{uni}}}$$

$$\frac{[v]^{T^{uni}}}{v} \quad v \text{ is a variable or a constant} \tag{5.6}$$

$$\frac{[f(\mathbf{t})]^{T^{uni}}}{f([\mathbf{t}]^{T^{uni}})}$$

Similarly, the rules for transforming from *LRA* to theory interface T^{uni} are as follows:

$$\frac{([t_1]^{T^{uni}} \leq [t_2]^{T^{uni}}) \wedge ([t_2]^{T^{uni}} \leq [t_1]^{T^{uni}})]}{[t_1]^{T^{uni}} = [t_2]^{T^{uni}}}$$

$$\frac{[t_1 \bowtie t_2]^{T^{uni}}}{[t_1]^{T^{uni}} \bowtie [t_2]^{T^{uni}}} \text{ is a function or predicate symbol in } LRA \tag{5.7}$$

$$\frac{[v]^{T^{uni}}}{v} \; v \text{ is a variable or a constant, } v \notin dom(\mathcal{M}^{-1})$$

$$\frac{[v]^{T^{uni}}}{\mathcal{M}^{-1}(v)} \; v \in dom(\mathcal{M}^{-1})$$

Determining satisfiability in an over-approximative theory does not guarantee that the formula is satisfiable in a more precise theory, since the satisfiability might have been introduced by the abstraction. In such cases the strength of the formula must be enhanced through techniques such as refinement. In the next section, we discuss how to use the theory-based model refinement idea in a model checking algorithm.

5.4 Summary and theory-aware model checking

The proposed novel approach to incremental bounded model checking is presented in Algorithm 7. It takes as input a program with a sequence $\langle Q_1, \ldots, Q_m \rangle$ of safety assertions that are to be verified, and a sequence of theories $\langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle$, such that for each i and j, $i < j$, $\mathcal{T}_j$ is *not* an over-approximation of $\mathcal{T}_j$.[3] For simplicity, we assume that all assertions are located in the entry function (i.e., $\hat{main}$), but our implementation does not have this restriction. We refer to $\sigma_{\mathcal{T}_j}(\hat{f})$ as to a summary for function f which is encoded in theory $\mathcal{T}_j$. Note that the function summary is initialized with the weakest possible summary, namely *true*. The algorithm searches for a first assertion which does not hold and then terminates with the **Unsafe** result. If no such assertion is found, the algorithm terminates with the **Safe** result.

Algorithm 7 maintains a set of mappings for each function call and each theory to a summary formula that over-approximates the behavior of the source

[3] In our implementation, we chose $\mathcal{T}_1 = EUF$, $\mathcal{T}_2 = LRA$, and $\mathcal{T}_3 = BV$.

Input: Program P with function calls $\hat{F}$, sequence of theories $\langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle$; sequence of safety assertions $\langle Q_1, \ldots, Q_m \rangle$
Output: Verification result: {**Safe**, **Unsafe**}

```
1 for each T_j do
2     for each f̂ ∈ F̂ do σ_{T_j}(f̂) ← true;
3 for each Q_i do
4     for each T_j do
5         ⟨result, σ_{T_j}⟩ ← SUMREF(P, T_j, ⟨σ_{T_1}, ..., σ_{T_n}⟩, Q_i);
6         if result = Safe then break;
7         if j = n then return Unsafe;
8 return Safe;
```

Algorithm 7: VERIFY($P, \langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle, \langle Q_1, \ldots, Q_m \rangle$)

function and is expressible in the theory. These summary formulas are initially true, but are refined after a verification run of each assertion Q_i. Importantly, they are reused by a verification run of the next assertion Q_{i+1}.

An algorithm for verifying an assertion Q with function summaries is shown in Algorithm 8. It starts by encoding the entry function in a given theory $\mathcal{T}$ and conjoins it with the negation of encoding of Q in $\mathcal{T}$. If this formula φ is unsatisfiable, then Q holds, manifesting the weakest possible summary *true* was adequate for all nested function calls from $\hat{main}$. Otherwise, the proposed algorithm starts gradually strengthening the formula φ by adding summaries of the function calls responsible for the satisfiability of φ. We rely on a method described in [Sery et al., 2011] to get models of satisfiable formulas and identifying the "reason" for their satisfiability.

The new contribution of the proposed approach is a method to refine summaries based on lazy enumeration of available theories. In particular, Algorithm 8 maintains a level of precision for each function call. In each round of refinement, if a function call $\hat{f}$ requires strengthening, its level of precision is increased by one, and a summary of that level, if available, is conjoined to φ. The key ingredient here is the set of translation rules, described in the previous section, that allow effectively reusing formulas among theories. Note that the translation process is not direct, but operates via a theory interface (omitted from the pseudocode in order to save space).

In order to prove the soundness of Algorithm 7, we need to show that a summary in one theory can be reused in another theory. In other words, the correct-

Input: Program $P = (F, f_{main})$ with function calls $\hat{F}$, theory $\mathcal{T}$; sequence $\langle \sigma_{\mathcal{T}_1}, \ldots, \sigma_{\mathcal{T}_n} \rangle$ of mappings of function calls to their summaries; Q: safety assertion to verify
Output: Verification result: {**Safe**, **Unsafe**}, updated $\sigma_{\mathcal{T}}$
Data: φ: BMC formula, $WL \subseteq \hat{F}$, Pr: precision mapping for function calls, CE: counterexample

1 $\varphi \leftarrow \text{ENCODE}_{\mathcal{T}}(\hat{main}) \wedge \neg\text{ENCODE}_{\mathcal{T}}(Q)$;
2 **for each** $\hat{f} \in \hat{F}$ **do**
3 $\quad Pr(\hat{f}) \leftarrow 0$
4 **while** $true$ **do**
5 $\quad \langle result, CE \rangle \leftarrow \text{SOLVE}(\varphi)$;
6 $\quad$ **if** $result = \text{SAT}$ **then**
7 $\quad\quad WL \leftarrow \text{GETCALLSWITHWEAKSUMMS}(CE)$;
8 $\quad\quad$ **if** $WL = \varnothing$ **then return Unsafe**;
9 $\quad\quad$ **for each** $\hat{f} \in WL$ **do**
10 $\quad\quad\quad$ **if** $Pr(\hat{f}) < n$ **then**
11 $\quad\quad\quad\quad Pr(\hat{f}) \leftarrow Pr(\hat{f}) + 1$;
12 $\quad\quad\quad\quad \psi \leftarrow \sigma_{\mathcal{T}_{Pr(\hat{f})}}(\hat{f})$;
13 $\quad\quad\quad\quad \varphi \leftarrow \varphi \wedge \text{TRANSLATE}_{\mathcal{T}}(\psi)$;
14 $\quad\quad\quad$ **else**
15 $\quad\quad\quad\quad \varphi \leftarrow \varphi \wedge \text{ENCODE}_{\mathcal{T}}(\hat{f})$;
16 $\quad$ **else**
17 $\quad\quad$ **for each** $\hat{f} \in \hat{F}$ **do**
18 $\quad\quad\quad \sigma_{\mathcal{T}}(\hat{f}) \leftarrow \sigma_{\mathcal{T}}(\hat{f}) \wedge \text{GETITP}_{\mathcal{T}}(\varphi, \hat{f})$;
19 $\quad\quad$ **return** $\langle$**Safe**, $\sigma_{\mathcal{T}}\rangle$;

Algorithm 8: $\text{SUMREF}(P, \mathcal{T}, \langle \sigma_{\mathcal{T}_1}, \ldots, \sigma_{\mathcal{T}_n} \rangle, Q)$

ness of Algorithm 7 depends on the correctness of transferral of summaries from one theory to another theory. To this end, we connect the over-approximations via function summarization with the over-approximations via less precise theory. The following theorem captures formally the correctness of summary transformation through theory interface.

Theorem 4 *Let f be a function, $f^{\mathcal{T}}_{sum}$ be a summary of f obtained from $f^{\mathcal{T}}_{precise}$, and $f^{\mathcal{T}'}_{sum}$ be a translation of $f^{\mathcal{T}}_{sum}$ to theory $\mathcal{T}'$. Then $f^{\mathcal{T}'}_{sum}$ is also a summary of f.*

Proof 5 *First, notice that by translating back $f^{\mathcal{T}}_{sum}$ to the theory interface using the*

rules in (5.1) we obtain an over-approximating representation f_{sum} of $f_{precise}$. This follows from the properties of the translation. Next, by translating f_{sum} to theory $\mathcal{T}'$ using rules in (5.1) we obtain an over-approximating formula $f_{sum}^{\mathcal{T}'}$ of f_{sum}. Finally, by transitivity $f_{sum}^{\mathcal{T}'}$ over-approximates $f_{precise}$, and hence $f_{sum}^{\mathcal{T}'}$ is a summary of f as stated in the theorem.

Note that the fact that $f_{sum}^{\mathcal{T}'}$ is a summary of f is sufficient for correctness of using $f_{sum}^{\mathcal{T}'}$ instead of $f_{precise}^{\mathcal{T}'}$ in next verification tasks in case of unsatisfiability results. It is not required that $f_{sum}^{\mathcal{T}'}$ over-approximates $f_{precise}^{\mathcal{T}'}$. In case of over-approximating theory $\mathcal{T}'$ it may happen that the full encoding of a function f, $f_{precise}^{\mathcal{T}'}$, is not sufficient to prove a property while a summary obtained from a different theory might be enough.

5.5 Implementation and evaluation

We have implemented the proposed summary and theory refinement algorithm on top of HiFrog, presented in detail in Chapter 3. As a backend, HiFrog uses the SMT solver OpenSMT which is equipped with a flexible interpolation framework for *EUF* and *LRA* for computing function summaries. Technical information about the setup of the tool can be found at `http://verify.inf.usi.ch/sum-theoref`.

With the reported experiments, the goal is to understand how bounded model checking can benefit from using over-approximative techniques based on function summaries obtained from SMT theories. We therefore compare our implementation against CBMC v5.8 [Kroening and Tautschnig, 2014], the most efficient bounded model checker based on the results of Competition on Software Verification SV-COMP. Compared to an earlier version of HiFrog presented in Chapter 3, this chapter presents 1) automating the theory refinement which previously required manual intervention and 2) transferring the summaries among theories, which previously was not supported at all. In the following, we also give an explicit experimental comparison against our earlier version to highlight the usefulness of the proposed algorithm.

We instantiated the summary and theory refinement framework as described by Algorithm 7 and Algorithm 8 with three theories: *EUF*, *LRA* and *BV* (using a standard encoding to propositional logic). In the global refinement phase of Algorithm 7, the program is first encoded in *EUF*. In case of unsuccessful verification with *EUF*, the entire program is encoded in *LRA*. Where the verification with *LRA* fails, the entire program falls back on bit-blasting. In the local refinement

phase, in each of these stages, summaries of functions are used when available and are refined on demand. After a successful verification run, summaries are constructed in the current theory and become available for verification of the subsequent assertions. Using the framework described in Section 5.3, they are translated to different theories on-demand.

Currently in our implementation, only *EUF* and *LRA* theories may exchange summaries. However, before the precise bit-blasting of the entire program, we can bit-blast the more abstract *EUF* and *LRA* summaries. While this feature is currently under development, we believe that it will lead to smaller and more compact proofs and thus improve efficiency of the entire tool. Similarly, the inverse direction of extracting high-level information from bit-precise summaries are challenging and remains a possible future work.

5.5.1 Results

For benchmarking we used an Ubuntu 16.04 Linux system with two Intel Xeon E5620 CPUs clocked at 2.40GHz. We limit the memory consumption to 2 Gigabytes and the CPU time to 200 seconds per process.

We chose 109 C programs from the `ldv` category of SV-COMP that either CBMC or HiFrog could solve within our time and memory limits. The choice of the `ldv` benchmarks in this work is justified because they exercise the proposed algorithm in an interesting way due to containing many assertions and functions, and being relatively large. We excluded programs where CBMC reported an internal error. In addition, we included 31 tricky hand-crafted smaller programs to stress-test our implementation. On average, the benchmarks have 10'000 lines of code, the longest ones reaching to 35'000 lines of code.

In total, our benchmark set contains 140 C programs and 500 assertions (verification tasks) placed inside these programs. 215 of these assertions were recognized as unreachable statements by the entry function in the C program. We excluded them from our study and focused on those tasks that require a full solving procedure. This narrowed down our set to 285 verification tasks.

In the following, we provide more details on statistics we collected after the extensive evaluation of the proposed algorithm against CBMC and three individual verification approaches from the older version of HiFrog, namely pure *EUF*, *LRA*, *BV*. Table 5.1 gives statistics on our benchmark set. The column **Solved** indicates the number of benchmarks which were solved by each tool within the time and memory limits. In total HiFrog solved 24 more benchmarks than

Table 5.1. HiFrog against CBMC, and the original version of HiFrog with respect to pure *EUF*, *LRA*, and *BV* solving, where **#sv** is the number of benchmarks from SV-COMP, and **#craft** is the number of our tricky hand-crafted benchmarks.

Tools	**Solved**		**Timeouts**		**Memory outs**		**Unknown**	
	#sv	#craft	#sv	craft	#sv	#craft	#sv	#craft
HiFrog	**67**	**31**	32	**0**	**10**	0	-	-
CBMC	63	11	**28**	20	18	0	-	-
EUF only	49	0	38	0	**10**	0	12	31
LRA only	48	1	40	0	11	0	10	30
BV only	43	4	33	4	33	23	-	-

CBMC[4]. Among 98 benchmarks for which HiFrog succeeded to return an answer within the time and memory limits, 24 benchmarks were *unsafe* and 74 benchmarks were *safe*. Interestingly, the average running time for unsafe benchmarks was longer (78 s) than the one for safe ones (48 s). This can be explained by our observation that in the unsafe cases, an iterative refinement of all the summaries was required to confirm the validity of the counterexample. However, in the safe cases, HiFrog was comparable to CBMC.

As can be seen from the column **Timeouts**, CBMC performed better than HiFrog on SV-COMP benchmarks, but it failed on almost 60% of our crafted benchmarks. As can be seen from the column **Memory outs**, HiFrog solved eight more SV-COMP benchmarks, on which CBMC immediately exceeded the memory limits. Overall, the experiments show that HiFrog is able to solve more benchmarks, and both times out and runs out of memory less often than CBMC.

Figure 5.3 gives a scatter plot representing a more detailed performance comparison of HiFrog and CBMC. Each cross in the figure stands for a single benchmark with the running time of HiFrog on the x-axis, and the running time of CBMC on the y-axis. The crosses on the outer lines correspond to executions that exceeded the memory limit of 2GB, and the crosses on the inner lines correspond to executions that exceeded the time limit of 200 s. A large amount of crosses on the top horizontal lines lets us conclude that HiFrog is able to solve benchmarks which are challenging for CBMC. Furthermore, the solving is relatively fast in these cases.

[4]Since many of our benchmarks include non-linear arithmetic, we also tried CBMC with the experimental `-refine` option. This did not significantly change the results, and therefore we report here the results obtained with the default options of CBMC.

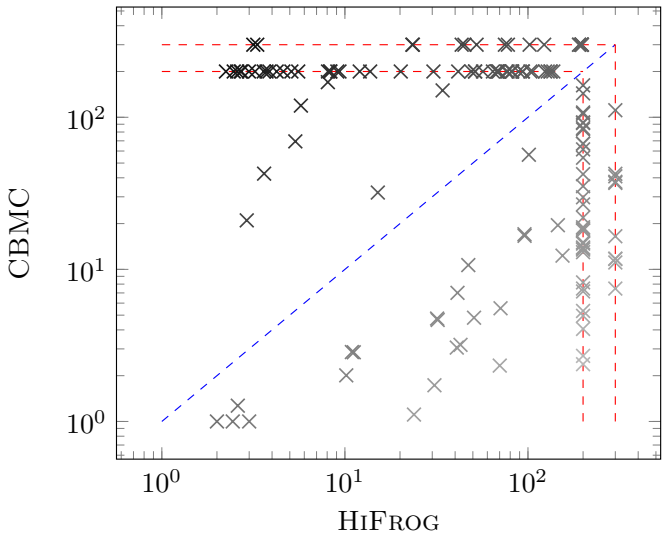


Figure 5.3. HiFrog vs CBMC. The outer horizontal and vertical lines refer to memory limit of 2GB, and the inner lines refer to timeout at 200 s.

The last three rows in Table 5.1 explain how the proposed novel algorithm in HiFrog performs compared to the earlier version of HiFrog, in which summary reuse was naïve and manual with respect to successive assertions. Because this functionality was not directly available in the older HiFrog, we prepared a set of helper scripts so that the older HiFrog could process assertions one after the other with possible re-use of the summaries. As expected, *EUF* and *LRA* had a large number of unsafe results, 43 and 40 respectively. We marked such results as *unknown* since due to the abstract nature of *EUF* and *LRA* the results are possibly spurious and thus cannot be trusted. By comparison, all unsafe results returned by the proposed new algorithm correspond to actual bugs. Verifying with *BV* revealed that a large number of benchmarks (56 instances) exceeded the memory limit, manifesting the cost of bit-blasting, which is avoided in the proposed new approach whenever possible.

In conclusion, we find it encouraging that the techniques described in this chapter provide such an impressive performance increase in the proposed model checking procedure. Considering both the effectiveness and the downsides of the proposed approach, in overall the evaluation results show a significant positive impact on the effectiveness and efficiency of verification of large-scale and multi-property benchmarks. Although we acknowledge that these initial results obtained with the 140 instances might not be enough to draw a decisive conclusion, the results do justify future efforts into extending the benchmarking, among

others, to large-scale instances with multiple user-defined assertions.

5.6 Related Work

As discussed in the previous chapters, in particular Section 3.6.1 and Section 3.6.2, both interpolants and function summaries are heavily used in model checking techniques. First we discuss the difference of the proposed technique of this chapter compared to the Chapter 3. Then we discuss related work on abstraction refinement techniques and using different levels of abstraction in model checking.

While the model checking framework presented in Chapter 3 supports different levels of SMT abstraction, information obtained from one level of SMT abstraction can only be reused at the same level of abstraction. The proposed approach in this chapter has no such limitation and is able to strengthen function summaries by converting from the current level of abstraction into a different level of abstraction.

In general, the choice of finer granularity for abstraction refinement can have both positive and negative impacts. That is, finding a weaker abstraction might make the proof easier (or not) but it could lead to a larger number of refinements that ultimately slow the proof. The concept of theory transformation is focused on optimizing the utilization of resources (‘good‘ summaries) by maximizing the potential for reuse, while concurrently minimizing the frequency of calling upon the SMT solver.

A substantial amount of work has been done in the area of abstraction refinement. The technique known as *localization reduction* [Kurshan, 1994] originated as a verification algorithm for timed automata, utilizing successive approximations [Alur et al., 1995]. Its purpose is to offer an iterative design abstraction algorithm, with respect to a specific property to be verified. The original algorithm was first integrated into the COSPAN model checker in 1992. Later, the concept of counterexample-guided refinement from localization reduction for COSPAN was expanded to apply to a broader framework through the use of CEGAR approach. CEGAR used symbolic simulation of the abstract counterexample to determine whether it is spurious and to generate a refined abstraction function. The initial implementation of CEGAR was based on the symbolic model checker NuSMV [Cimatti et al., 2002] and geared to finite state models. Since then several effective methods utilizing abstraction refinement have been developed [Chauhan et al., 2002; Clarke, Gupta and Strichman, 2004; Jain et al., 2007; Seghir et al., 2009; Heizmann et al., 2009; Lahtinen et al., 2015; Iosif and

Xu, 2018; Zhang et al., 2020]. Nevertheless, identifying the appropriate level of abstraction granularity continues to be a an active research topic in abstraction-based methodologies.

A group of related approach for abstraction refinement is PDR-based tools that have been integrated with different abstraction techniques [Fan et al., 2016; Nguyen et al., 2008; Bayless et al., 2013]. There are techniques which use word-level information for creating abstractions [Lee and Sakallah, 2014; Ho et al., 2017]. In the context of solving constrained Horn clauses with interpolation, the tool DUALITY [McMillan and Rybalchenko, 2013] generalizes IMPACT algorithm to incrementally unroll a program and solves the corresponding CHC's with interpolation until it produces valid inductive invariants. Duality computes function summaries using tree interpolants and uses them as abstractions of functions with refinement. Other PDR-based tools such as SPACER [Komuravelli et al., 2013] effectively uses interpolants as abstractions combines proof-based techniques with CEGAR and maintains both an overapproximation and an underapproximation of the input program. However, these research areas are different than the proposed theory transformation: the context of our approach is bounded model checking while the above mentioned techniques are unbounded verifiers. The distinguishing feature of our approach, in addition to the flexible SMT-level summarizations it supports, is the theory interface it provides, which enables the transformation of encoding from one layer to another level of encoding. This capability allows for maximal reuse of the already generated function summaries.

The work [Kroening et al., 2004] presents a new abstraction-based framework for deciding satisfiability of quantifier-free Presburger arithmetic formulas and later an adaptation to bit-vector formulas was presented in the approach [Bryant et al., 2007] . The proof-based abstraction-refinement approach to model checking in McMillan and Amla [2003] use a related technique to accelerate model checking algorithms over finite Kripke structures. More precisely, they invoke a bounded model checker to determine the state variables that need to be visible in order to create a 'good' abstraction for the next iteration of model checking.

The idea of using an abstract description of the bit-precise level of encoding has been tried with success in verification of hardware designs [Andraus et al., 2008] as well as software He and Rakamarić [2017]; Armando et al. [2009]. The approaches use different level of encoding for different parts of the problem; these approaches typically start with uninterpreted functions and gradually refine to bit-level precision to rule out spurious counterexamples when necessary, while mixing different levels of encoding to verify a single property. Unlike these approaches, we do not mix different levels of encoding and only shift to more

precise encoding globally, when the previous level of abstraction is insufficient. A single level of encoding allows us to extract useful information in form of function summaries from successful verification runs and reuse that information in the next verification run.

An outstanding advantage of the SMT technology is that it can employ decision procedures not only in isolation, but also in combination. For instance, once can speed up solving non-linear arithmetic formulas by first checking linear abstractions using more efficient decision procedures, before applying heavier procedures [Cimatti, Griggio, Irfan, Roveri and Sebastiani, 2017]. Additionally, theories can also be combined already in the input language of SMT solvers. For example, deductive program verification techniques generate verification conditions, which might refer to arrays, bit-vectors as well as integers; in such cases, dedicated SMT solvers can apply several decision procedures for different theories in combination [Sebastiani, 2007b; Bonacina et al., 2019; Sebastiani, 2007a; Bruttomesso et al., 2009]. However, these line of research remains orthogonal to the proposed theory transformation.

5.7 Synopsis

This chapter introduced a concept called *theory interface* to map function summaries in one theory to another one. The key idea is to exploit both the function summaries and the overall precision of the program encoding lazily. The theory interface enables the exchange of function summaries among formulas in different theories and avoids an expensive theory combination. Searching for higher precision theory is performed lazily in the sense that available summaries are tried out first by translating them into the current precision. This chapter proposed a framework to establish a set of over-approximation relations as well as translation rules among SMT theories, in particular for *EUF*, *LRA*, and *BV*. The over-approximation relations guarantee that when a program is proven to be safe with less precise theories, it is also safe with respect to precise program semantics. The translation rules allow re-using function summaries among different theories, which is essential to the laziness provided by the framework.

The idea is evaluated by implementing the algorithm which performs both *local refinement* and *global refinement* on demand. This is implemented on top of the HiFrog tool. The model checker lazily chooses the appropriate theory to precisely reason about program properties in the context of bounded model checking using function summaries. An extensive evaluation on a range of large-scale benchmarks taken from SV-COMP demonstrates the effectiveness of the

proposed algorithm in practice. The results show that in comparison to a state-of-the-art model checker CBMC, the proposed tool can solve more instances within the same limits on time and memory. From the extensive evaluation, we learned the use of summary refinement by theory transformation instead of the naive summary refinement (described in Section 3.3.2) helps the SMT solver to check the smallest possible formula. As a result, the burden of proof on the solver is reduced and leads to an efficient solving process.

5.8 Limitation and future work

We would like to point the reader to some limitations of the work and some possible directions for future research.

The framework described in this chapter is coupled with certain theories. For example, the theories in scope are limited to *EUF*, *LRA*, and *BV*. Although these theories are generally sufficient to reason about several realistic programs, it would be interesting to show that over-approximation relations can be easily derived for other theories such as the theory of arrays and LIA. Another limitation is the lack of inverse direction of extracting high-level information from bit-precise summaries, i.e., conversion of bit-blasted summary to high-level summary. This can be interesting for future work.

The technique of theory and summary transformation in this chapter was applied successfully to the domain of symbolic model checking, and it would be interesting to apply the idea to software verification based on IC3, where the correctness of unbounded programs is reduced to finding general proofs for a sequence of verification conditions that are generated on-the-fly. Another possible direction for the future would be studying the applicability of this approach to other areas of program verification, such as upgrade checking, which considers a task of verification of somewhat related programs against the same property (cf. Chapter 4) (as opposed to verification of the same program against somewhat related properties, as in the context of this chapter).

Chapter 6

Theory-aware abstraction refinement

Finding the right abstraction is a key to further extension of the applicability of formal methods to real problems of software and hardware engineering. The SMT reasoning approach is based on modeling the software and its specifications in propositional logic, while expressing domain-specific knowledge with first-order theories connected to the logic through equalities. Once a satisfying assignment is found for the propositional model, its consistency is queried as equalities from the theory solvers, which, in case of inconsistency, provide an explanation as a propositional clause. Successful verification of software relies on finding a model that is expressive enough to capture software behavior relevant to correctness, while sufficiently high-level to prevent reasoning from becoming prohibitively expensive. Since in general more precise theories are both more expensive computationally and potentially distracting for the automatic reasoning, finding such a balance is a non-trivial task.

An interesting observation is that if a property being verified is proved using one of the light-weight theories the proof holds also for the exact encoding of the program. However, the loss of precision can sometimes produce spurious counterexamples due to the over-approximating encoding. To mitigate such spurious behaviors, an iterative refinement process is employed alongside the abstraction, which involves refining the abstract model and performing repeated checks. In this chapter we address this problem, and propose a general framework for building abstraction for model checkers that involves different theories of SMT based on the precision hierarchy among SMT. Moreover, as an integral part of our abstraction framework we proposed a rigorous refinement strategy to tune the abstraction. Finally, we plan to develop a methodology to identify the critical parts of the program to guide our abstraction refinement framework to reduce the overhead of refinement procedure.

The proposed algorithms in the previous chapters attempted to strengthen the connection between two research areas of model checking and SAT-modulo-theories (SMT). Towards the seamless connection between decision procedures and modeling, this chapter introduces *theory refinement*, a counterexample-guided abstraction refinement (CEGAR) approach for modeling software modularly using theories that are partially ordered with respect to their precision. The main contribution of this chapter is the process of gradually encoding a program using the most precise theory only for a critical subset of all program statements, while keeping lower precision for the rest of the statements. Note that the granularity of such refinement is in the level of *program statements* while the refinement procedure in Chapter 5 was on the function level. The critical subset of theories is identified based on counterexamples, and theories of different precision are bound to each other through special identities. We study several automatic heuristics for guiding the encoding and provide also a manual encoding option. We apply theory refinement on verification of safety properties of software through bounded model checking. However, we believe that the technique is applicable in most verification techniques where higher level information is available on the problem structure. This includes model checking and upgrade checking, k-induction [McMillan, 2005], the IC3 algorithm [Bradley, 2011], and generation of inductive invariants [Gurfinkel et al., 2014]. We show that the modular composition of the theories preferring lower precision can be used to both obtain speed-up in solving and identifying statements whose precise semantics do not affect the program safety, providing the model checker with cleaner proofs.

Many SMT solvers use over-approximation through theories as a means of speeding up solving. For instance [Bruttomesso et al., 2007; Hadarean et al., 2014; Brummayer and Biere, 2009b] organizes the theory solvers into layers that solve problems represented in *BV*. The query is first given to fast and less precise theory solvers, and only passed on to the exact solver if previous layers fail to show unsatisfiability. In contrast to low-level SMT solving, this work studies how to automatically identify statements whose exact semantics can be ignored in model checking. This shift of view point has several advantages: (i) the approach can be used both to obtain speed-up in solving, and as a means for synthesis and finding fix-points for transition relations; (ii) the guidance from the source code allows the use of more powerful heuristics for choosing which statements should remain abstract; and (iii) the refinement takes place on the level of the program, not at the level of the theory query, an approach potentially more natural from the point of view of the semantics of the program.

This chapter presents theory refinement with two new theories called *unin-*

terpreted functions for programs (UFP) and *bit vectors for programs* (BVP) that are based on the theories of quantifier-free uninterpreted functions with equality (*EUF*), and bit vectors (*BV*), respectively. The two theories were chosen since they represent two natural extremes in precision and are commonly used in the layered solver approach (see, e.g., [Hadarean et al., 2014]). In addition to the functionality of *EUF*, UFP provides interpretations for constants, conversion of abstract values to concrete values, and commutativity for uninterpreted functions when applicable. The key difference in BVP compared to *BV* is that BVP is capable of directly injecting non-clausal refinements, modeling the program statements bit-precisely, to the inherent Boolean structure maintained in the SMT solver.

The theory refinement algorithm has been validated through its implementation in HIFROG. We report promising results both with respect to speed and the amount of refined program statements on both instances from a software verification competition and our own regression test suite. The experiments demonstrate that the approach has a potential of several orders of magnitude of improvement over the approach based solely on flattened bit-vectors, as implemented in the state-of-the-art tool CBMC and in our own tool.

6.1 Preliminaries

Let P be a loop-free program represented as a transition system, and t a *safety property*, that is, a logical formula over the variables of P. We are interested in determining whether all reachable states of P satisfy t. Given a program P and a safety property t, the task of a model checker is to find a counterexample, that is, an execution of P that does not satisfy t, or prove the absence of counterexamples on P. In the bounded, symbolic model checking approach followed in this chapter the model checker encodes P into a logical formula, conjoins it with the negation of t, and checks the satisfiability of the encoding using an SMT solver. If the encoding is unsatisfiable, the program is safe, and we say that t holds in P. Otherwise, the satisfying assignment the SMT solver found is used to build a counterexample.

A *sort* is a set of constants. For example the Boolean sort $\mathbb{B} = \{\top, \bot\}$ consists of the Boolean constants, true and false. Given a set of sorts $\{T_0, \ldots, T_n\}$, a *function* $op : T_1 \times \ldots \times T_n \to T_0$ maps a (possibly empty) sequence of constants $v_1, \ldots, v_n$ such that $v_i \in T_i$ to a *return value* $v_0 \in T_0$. Functions mapping empty sequences are *variables*, and a *term* is either a constant, a variable, or an application of a function $op(t_1, \ldots, t_n)$ where t_i are, recursively, terms with a return value in the sort T_i. In most cases in this chapter we use the usual infix nota-

Table 6.1. The functions used in the encoding we consider. Note that unsigned and signed sum coincide.

Functions		**Descriptions**
	Logical functions	
`&&`, `||`	$Sb \times Sb \to Sb$	Logical and, or
`!`	$Sb \to Sb$	Logical not
	Non-logical functions	
`+`, $*_u$, $*_s$, $/_u$, $/_s$	$Sz \times Sz \to Sz$	Sum, unsigned and signed product and division
$\%_u$, $\%_s$	$Sz \times Sz \to Sz$	Unsigned and signed remainder
$\ll, \gg_a, \gg_l$	$Sz \times Sz \to Sz$	left shift, arithmetic and logical right shift
`&`, `|`, `^`	$Sz \times Sz \to Sz$	Bitwise and, or, exclusive or
$\sim$:	$Sz \to Sz$	bitwise complement
$\leq_s$, $\leq_u$, $<_s$, $<_u$, $\geq_s$, $\geq_u$, $>_s$, $>_u$	$Sz \times Sz \to Sb$	Signed and unsigned less than or equal to and greater than or equal to

tion together with parentheses to express the well-known arithmetic and logical functions.

6.2 Combination of theories in theory refinement

This section fixes a notation for describing instances of the safety problem using SMT, and provides two communicating theories for solving the safety problem. The goal of the presentation is to clarify how the modeling works in the SMT framework, placing particular emphasis to the use of symbols and their semantic.

In modeling programs we consider sets of quantifier-free symbolic statements of the form $x = t$, where x is a variable, and t is a term. This form essentially corresponds to the Single static assignment (SSA) form [Cytron et al., 1989] for loop-free programs. The symbolic statements are defined over a sort of bounded integers Sz and a Boolean sort $Sb = \{\top_l, \bot_l\}$; we distinguish between these sorts and, for instance, the sorts of integers $\mathbb{Z}$ and Booleans $\mathbb{B}$ to clarify the difference

$$
\begin{array}{ll}
 & \left(c^b =_{BVz} \left((a^b \,\%_u\, 2^b) + (b^b \,\%_u\, 2^b)\right) \%_u\, 2^b\right)_1 \wedge \\
c = \left((a \,\%_u\, 2) + (b \,\%_u\, 2)\right) \%_u\, 2 & \left((c')^b =_{BVz} (a^b + b^b) \%_u\, 2^b\right)_1 \wedge \\
c' = (a + b) \%_u\, 2 & \left(d^u = f^u *_u e^u *_u c^u\right) \wedge \\
d = f *_u e *_u c & \left((d')^u = e^u *_u f^u *_u (c')^u\right) \wedge \\
d' = e *_u f *_u c' & \left(c^u = (c')^u\right) \leftrightarrow \left(\left(c_1^b \leftrightarrow (c')_1^b\right) \wedge \ldots \wedge \left(c_{bw}^b \leftrightarrow (c')_{bw}^b\right)\right)
\end{array}
$$

Figure 6.1. (Left) a sequence of statements and (right) the corresponding encoding in combined UFP and BVP (to be described in Sect. 6.2.3). On the left all the variables are of sort Sz, and e and f are unbound.

between this *symbolic encoding* (hence the S) and the representation used by an SMT solver. Table 6.1 lists the non-variable functions we consider in our encoding. Note that unlike some programming languages, including C and C++, we do not allow the encodings to interpret terms from Sz as terms from Sb or vice versa. We distinguish between the functions defined over the sort Sb and those defined over Sz, calling the former logical functions and the latter non-logical functions. The control-flow structures, such as `if-then-elses`, are encoded using the functions `!`, `||`, and `&&`. For the purpose of this presentation we assume that the encodings do not contain arrays and pointers.[1] Figure 6.1 (*left*) shows an example sequence of statements that we will use as a running example in the discussion of this section.

6.2.1 Bit Vectors for programs

Our theory of bit vectors for programs (BVP) has a single sort BVz^{bw} containing the integers representable in $bw \in \mathbb{N}$ bits. When the bit-width of the sort is clear from the context we simply write BVz for the sort. Each BVP term t of sort BVz^{bw} is associated with the bits $t_1, \ldots, t_{bw}$ which are variables from the sort $\mathbb{B}$. The bits t_1 and t_{bw} are called, respectively, the *least significant bit* and the *most significant bit* of t.

The BVP theory has two special constants 1^b and 0^b. For the constant 0^b, $0_i^b = \bot$, $1 \leq i \leq bw$. For the constant 1^b, $1_1^b = \top$ and $1_i^b = \bot$ for $2 \leq i \leq bw$. The equality of BVP is $=_{BVz}$: $BVz \times BVz \to BVz$. The interpretation of the equality is that if $x =_{BVz} y$ holds, then the value of the equality term is 1^b and otherwise

[1] We do support these in our implementation, but their results are treated nondeterministically, that is, as unbound variables from Sz.

0^b. Finally, BVP has the functions defined in Table 6.1 with all sorts replaced by the sort *BVz*. For a term t, the Boolean functions determining the bits t_i are computed through propositional flattening (see, e.g., [Kroening and Strichman, 2016]).

We encode a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$ in BVP as follows. Each statement $x_i = t_i$ is converted to $|x_i|^b =_{BVz} |t_i|^b$, where the operator $|\cdot|^b$ is defined for a symbolic term t recursively:

$$|t|^b \stackrel{\text{def}}{=} \begin{cases} x^b & \text{if } t \doteq x \text{ is a variable or a constant} \\ |x|^b \bowtie |y|^b & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a binary function,} \\ \circ|x|^b & \text{if } t \doteq \circ x \text{ where } \circ \text{ is a unary function} \end{cases} \tag{6.1}$$

where $a \doteq b$ denotes that the term a matches the form of b. Conjunction of the least significant bits of encoded statements in P defines its BVP-encoding $[P]^b$:

$$[P]^b \stackrel{\text{def}}{=} (|x_1|^b =_{BVz} |t_1|^b)_1 \wedge \ldots \wedge (|x_n|^b =_{BVz} |t_n|^b)_1 \tag{6.2}$$

We say that a safety property t holds in program P if and only if $[P]^b \wedge \neg[t]^b_1$ is unsatisfiable. Based on the definition we can see that the symbolic encoding in Figure 6.1 satisfies the safety property $(d = d')$ due to properties of modular arithmetic. The BVP encoding is often inefficient due to the quadratic growth of the formula with respect to *bw*. However, in many cases, the bit-precise encoding of statements (e.g., $*_u$ in Figure 6.1) are irrelevant to the safety property, and can therefore be over-approximated. This motivates the use of less precise but more efficiently solvable encodings such as those based on uninterpreted functions.

6.2.2 Uninterpreted functions for programs

The logic UFP (Uninterpreted Functions for Programs) is the standard logic of quantifier-free uninterpreted functions having the Boolean sort $\mathbb{B}$, the standard Boolean functions $op : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B}$ where *op* is an operator such as $\vee, \wedge$, and $\neg$, and an unbounded number of variables. In addition the logic is augmented with

- a sort *UFPn* of real or integer numbers;
- the functions listed in Table 6.1 treated as uninterpreted functions with the sorts *UFPn* and $\mathbb{B}$ instead of *Sz* and *Sb* respectively;
- commutativity of the functions `+`, $*_u$, $*_s$, `&`, and `|`; and

- the concept of constants beyond the Boolean $\top$ and $\bot$.

As usual, UFP also contains the equality function $=_S: T \times T \to \mathbb{B}$ for all sorts T. As in the symbolic encoding, also in UFP we differentiate between two types of functions: those with a return sort $\mathbb{B}$, and those with a return sort *UFPn*.

Given a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$, we denote its encoding in UFP by $[P]^u \stackrel{\text{def}}{=} ([x_1]^u =_{T_1} [t_1]^u) \wedge \ldots \wedge ([x_n]^u =_{T_n} [t_n]^u)$, where T_i is either *UFPn* or $\mathbb{B}$ depending on the related sort. The encoding operator $[\cdot]^u$ is defined as follows for a term t:

$$[t]^u \stackrel{\text{def}}{=} \begin{cases} x^u & \text{if } t \doteq x \text{ is a variable or a constant} \\ [x]^u \wedge [y]^u & \text{if } t \doteq x \,\&\&\, y \\ [x]^u \vee [y]^u & \text{if } t \doteq x \mid\mid y \\ \neg[x]^u & \text{if } t \doteq\ !\, x \\ [x]^u \bowtie [y]^u & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a non-logical function.} \end{cases} \tag{6.3}$$

We distinguish between the notions of program safety in UFP and in BVP. In particular, we say that a safety property t holds in program P in UFP if and only if $[P]^u \wedge \neg[t]^u$ is unsatisfiable.

The program in Figure 6.1 is safe with respect to the safety property $!(c = c') \mid\mid (d = d')$ in UFP and therefore also in BVP. However, it is not safe in UFP with respect to the safety property $d = d'$ that is safe in BVP. For checking safety of programs in UFP we use a theory solver implementing a congruence closure algorithm [Detlefs et al., 2005] that is modified to support constants and commutativity. The modifications are described in more detail in Sec. 6.5.1.

In our experiments presented in Chapter 3 we showed that safety of many programs can be established by interpreting the arithmetic functions as uninterpreted functions. In the next subsection we describe how the UFP logic and the BVP logic can be combined.

6.2.3 Combination of UFP and BVP

We present the theory refinement approach using a seamless integration of the UFP and BVP encoding, and therefore require a form of theory combination. However, unlike in conventional theory combination on bit vectors (see, e.g., [Hadarean et al., 2014]), we do not need to consider bit-vectors as theories, but instead they are embedded directly to the Boolean structure of the SMT solver. The two theories UFP and BVP are combined using a *binding formula* defined as follows.

Figure 6.2. A symbolic encoding of a program and the corresponding SMT formula. In the schematic example most of the program is encoded using UFP, while certain critical parts are encoded in BVP and made to communicate with the UFP encoding using the binding formula F_B.

Definition 13 *Given a symbolic statement t, let $[t]^u$ and $[t]^b$ be its* UFP *and* BVP*-encodings respectively. If both $[t]^u$ and $[t]^b$ appear together in a formula, we say that t is* bound. *Let B be the set of all bound statements. The* binding formula *for B (denoted F_B) is defined as*

$$F_B \stackrel{\text{def}}{=} \bigwedge_{t,t' \in B} \left([t]^u = [t']^u\right) \leftrightarrow \left(([t]^b_1 \leftrightarrow [t']^b_1) \wedge \ldots \wedge ([t]^b_{bw} \leftrightarrow [t']^b_{bw})\right) \tag{6.4}$$

Intuitively, the combination of the theories UFP and BVP with F_B allow us to express an over-approximation of the symbolic encoding of a program. This is stated more formally in the following theorem.

Theorem 5 *Let P be a program. Then $[P]^b \wedge F_B \models [P]^u$.*

Proof 6 *(*sketch*) By simulation of executions in* BVP*: if there exist values $v^b_1, \ldots, v^b_n$ for the variables $x^b_1, \ldots, x^b_n$ in a term $[a = t]^b$ then the same values $v^u_1, \ldots, v^u_n$ satisfy the corresponding equality $[a]^u = [t]^u$.*

Figure 6.2 shows the combined UFP and BVP encoding schematically. The symbolic encoding of a program is partitioned by the model checker into three parts: the UFP encoding, the BVP encoding, and the binding formula F_B. The conjunction of these is solved by the SMT solver. Figure 6.1 (*right*) describes a combination encoding of UFP and BVP together with the necessary binding formula for the running example.

6.3 Overview and motivating examples

This section demonstrates with examples the necessity for mix theories in the program encoding and solving the corresponding formula.

The great advantage of encoding program with *EUF* theory is that it allows a model checker to abstract away complicated operations and solve them efficiently. However, as expected due to the abstraction spurious counterexamples may be produced. The result of experimentation in Chapter 3 confirmed that

verification with the *EUF* in HIFROG had a 45 percent SAT result which might be potentially spurious. To cope with spurious the results, such light-weight theories need to be refined to the more precise encoding like *BV* or propositional on demand.

We discuss two C-code examples to explain the idea of partial theory refinement procedure of *EUF* formula with bit-precise formula. For both examples, once they are encoded with *EUF* solely, the solver reports an spurious model. However, as it is possible to solve correctly the first example with propositional logic, it is not the case for the second example, that cannot be solved with reasonable time and space resources. One way to solve the formula of the second example is by using theory refinement. The goal of theory refinement idea is to keep the program encoding as much as possible in *EUF* theory instead of propositional.

Example 4 *Figure 6.3 shows a C program with modulo and multiplication operators, motivated by a case where a modulo operator is refactored and the equality of the old and the new code needs to be certified. Once encoding this program with EUF, it spuriously generates a counterexample. Because unlike propositional logic, which has a precise encoding for modulo operator, EUF encodes the modulo operator as an uninterpreted symbol. Thus, EUF cannot express the fact that the expression* $(f\ \%\ x)$ *is equivalent to the expression* $((f\ \%\ y)\ \%\ x)$.

```
1 int main()
2 {
3   unsigned  e,f,x,y;
4   x = 2;
5   y = 4;
6   unsigned d1 = e * ( f % x );
7   unsigned d2 = e * ( ( f % y ) % x );
8   assert(d1 == d2);
9 }
```

Figure 6.3. Example written in C with multiplication and modulo operators.

The idea of theory refinement suggests to replace the *EUF* encoding for the expression $(f\ \%\ x)$ and $((f\ \%\ y)\ \%\ x)$ in Figure 6.3 with the bit-precise encoding. The rest of the terms are kept in *EUF* level.

The second example in Figure 6.4 is the case where propositional logic fails to verify the C code as a result of time-out. The assignment of variables a and b

are only for the sake of the example. More meaningful benchmarks are part of the experimental results in section 6.6.

```
1  int main()
2  {
3   unsigned a;
4   unsigned b;
5   unsigned c1 = (((a % 2) + (b % 2))) % 2;
6   unsigned c2 = (a + b) % 2;
7   unsigned e, f;
8   unsigned d1 = e * f * c1;
9   unsigned d2 = e * f * c2;
10  assert(d1 == d2);
11 }
```

Figure 6.4. A C example with addition, multiplication, and modulo operators.

Example 5 *Consider the C code example in Figure 6.4 where the equation* $\texttt{c1} = \texttt{c2}$ *is equivalent in Modular arithmetic. The non-linear multiplications expressions assign to* d1 *and* d2 *can be treated as uninterpreted functions with respect to the assertion. The bit-precise BMC encoding of this problem is hard to solve within the limited time and space. The EUF encoding is solved in few seconds, however the verification fails with a spurious counterexample for similar reasons as in Figure 6.3's examples. Theory refinement suggest to encode* c1 *and* c2 *in bit-precise level and glue it with the rest of program which is encoded in EUF.*

6.4 Counterexample-guided theory refinement

This section provides an algorithm for verifying safety of programs by gradually refining the *precision* ρ of the symbolic encoding from UFP to BVP in parts where satisfying truth assignments show that it is necessary for soundness. Algorithm 9 describes the high-level idea. The algorithm takes as input a symbolically encoded problem P and a safety property t, and returns either **Safe**, if t holds in P, or **Unsafe** with a bit-precise counterexample if t does not hold in P. During the execution the algorithm picks statements $s \in P \cup \{t\}$ and refines their approximations in ρ until $\rho[s]$ is equivalent to $[s]^b$. Based on ρ, the algorithm constructs the binding formula F_B sufficient to connect the UFP and BVP terms.

input : $P = \{(x_1 = t_1), \ldots, (x_n = t_n)\}$: a program, and t: a safety property
output: $\langle$**Safe**, $\bot\rangle$ or $\langle$**Unsafe**, $CE^b\rangle$

```
1  For all 1 ≤ i ≤ n initialize ρ[x_i = t_i] ← [x_i = t_i]^u
2  ρ[t] ← [t]^u
3  F_B ← ⊤
4  while true do
5      Query ← ρ[x_1 = t_1] ∧ ... ∧ ρ[x_n = t_n] ∧ ¬ρ[t] ∧ F_B
6      ⟨result, CE⟩ ← CheckSAT(Query) // Get a model in combined EUF
        and BV
7      if result is UNSAT then
8          return ⟨Safe, ⊥⟩
9      CE^b ← getValues(CE)                          // Get a model in BV
10     foreach s ∈ P ∪ {t} s.t. ρ[s] ⊭ [s]^b do
11         ⟨result, _⟩ ← CheckSAT([s]^b ∧ CE^b)   // Consult the BV solver
12         if result is UNSAT then
13             ρ[s] ← refine^s(ρ[s])      // Refine abstraction on the
                Main Solver
14             F_B ← computeBinding(ρ)
15             break
16     if No s was refined at line 13 then
17         return ⟨Unsafe, CE^b⟩
```

Algorithm 9: Counterexample-guided Theory Refinement Algorithm

The safety of the program is tested at lines 5–8 using the current precision ρ and the binding formula. If the check succeeds, the algorithm terminates at line 8. Otherwise, a satisfying truth assignment is extracted at line 9 and then used to refine ρ at lines 10–15.

The need for refinement is checked for every statement s with a precision $\rho[s]$ not equivalent to $[s]^b$. If the truth assignment CE^b is inconsistent with $[s]^b$ then $\rho[s]$ is refined to block the truth assignment. If at least one such replacement happens in the current iteration, the execution proceeds to line 5. In practice it is a good idea to refine several statements based on a single counterexample, as discussed in Sec. 6.6. If no refinement is done, the truth assignment corresponds to a counterexample and the algorithm terminates at line 17.

The algorithm uses four sub-procedures `CheckSAT`, `getValues`, `refine`^{s}, and `computeBinding`. `CheckSAT`(F) determines the satisfiability of a formula F, and

`getValues`(CE) computes a BVP encoding of CE through substituting the abstract values from UFP with concrete BVP values. $\texttt{refine}^s(F)$ refines the statement s with respect to the previous precision F, and `computeBinding`(ρ) computes the binding formula using Definition 13. Below we give a definition for the `refine` procedure, while the other procedures will be discussed in more detail in Sec. 6.5.3.

Definition 14 *The procedure* $\texttt{refine}^s(F)$ *returns an iterative refinement of the statement s of the symbolic encoding with respect to F, such that (i)* $\texttt{refine}^s(F) \models F$*, and (ii)* $\texttt{refine}^s$ *has a fix-point that is equivalent to* $[s]^b$ *and reachable in a finite number of applications of* $\texttt{refine}^s$*.*

While in the implementation discussed in Sect. 6.5 we use $\texttt{refine}^s(F) = [s]^b \wedge [s]^u$, we want to point out the possibility of using interpolation-based methods (see, e.g., [Alt et al., 2016]) for the refinement.

Theorem 6 *Algorithm 9 terminates in a finite number of steps.*

Proof 7 *Assume that Algorithm 9 does not terminate. Then there is a term in* $P \cup \{t\}$ *that can be refined an unbounded number of times before the fix-point equivalent to* $[s]^b$ *is reached, which contradicts Definition 14.*

Theorem 7 *Algorithm 9 returns* **Unsafe** *if and only if the symbolic encoding P has an execution violating the safety property t.*

Proof 8 *The algorithm maintains the invariants*

$$\begin{array}{ll} \text{Inv1} & [x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \\ \text{Inv2} & [t]^b \models \rho[t] \end{array} \tag{6.5}$$

at line 13 by Definition 14 and Th. 5. Assume that the algorithm returns **Unsafe** *but there is no execution violating the safety property t. Then there is a truth assignment* σ *such that* $\rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \wedge F_B$ *is true and* $\rho[t]$ *is false. The truth assignment* σ *must also satisfy* $[x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b$*. By* Inv2*, if* $\rho[t]$ *is false also* $[t]^b$ *is false, hence contradicting the unsafety of* (P, t)*. Now assume the algorithm returns* **Safe** *but there is an execution of P violating t. Then there is a truth assignment satisfying* $[P]^b \wedge \neg[t]^b$*. Since by Th. 5 both* $[P]^b \wedge F_B \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n]$ *and* $\neg[t]^b \wedge F_B \models \neg\rho[t]$*, also the query on line 5 is satisfiable, contradicting the assumption.*

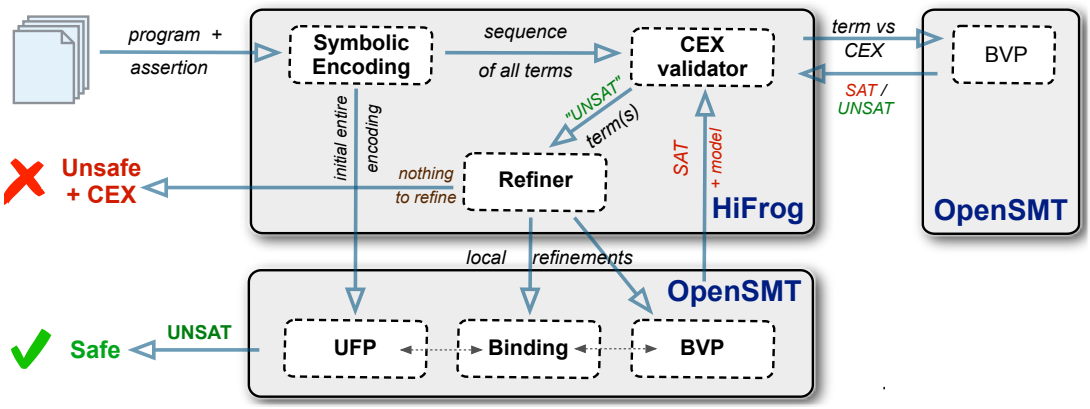


Figure 6.5. The SMT-based model checking framework implementing a theory refinement approach used in the experiments.

6.5 Implementation of theory refinement algorithm

This section describes the prototype implementation of the theory refinement algorithm. The algorithm has been implemented on the SMT solver OpenSMT and the bounded model checker HiFrog. The overview of implementation including the three main components and interactions between them is depicted in Figure 6.5.

6.5.1 The Solver for UFP

The UFP theory solver is based on the co-operation between a congruence closure algorithm, which maintains sets of equivalence classes and inequalities between the classes, and a SAT solver, which enforces a propositional structure describing the relations between the equalities. We refer the reader to [Detlefs et al., 2005] for the full description of the *egraph* algorithm that the UFP solver bases on.

Constants. The original egraph algorithm does not support constants other than the Boolean $\top$ and $\bot$, but constants play often an important role in our benchmarks. The egraph algorithm can represent an inequality between two terms t_1, t_2 by asserting explicitly the inequality $t_1 \neq t_2$ over these terms. This representation grows quadratically in the number of constants and therefore is not scalable. We adopt a different strategy for representing the inequalities between constants. An equivalence class in the egraph algorithm is represented by a linked list binding together the terms in the same class. Each class is represented by a canonical term from the linked list. In the original algorithm of [Detlefs

et al., 2005], when two equivalence classes a and b are joined, the canonical term of the new class $a \cup b$ is the representative of whichever class a or b contains more terms. This is done to allow efficient joining and splitting in the backtracking search driven by the SMT solver. In our implementation the representative of a class a is always a constant if a contains a constant. The implicit inequality between constants is then implemented by a check that the respective equivalence classes are not both represented by a constant term. This approach fits naturally into the egraph algorithm and explanation generation. In the experiments we observed no noticeable slowdown compared to the original approach.

Values. Algorithm 9 requires concrete values from the UFP theory to construct a counterexample candidate. In general the values for UFP are obtained by assigning a running number for each equivalence class that the egraph algorithm maintains. However, there are two special cases for the values. First, if the equivalence class contains a constant, the value is that of the constant. Second, a pre-processing step in the SMT solver removes terms that only appear on clauses that are true by construction. Since these terms can have any value, we indicate this with a special flag.

Commutativity. The commutativity of the functions $Co = \{+, *_u, *_s, \&, |\}$ is implemented by conjoining the set $\{\circ(a,b) \leftrightarrow \circ(b,a) \mid \circ \in Co, \circ(a,b) \text{ in } P\}$ to the instance $[P]^u$ being solved. A similar approach is followed, for instance, in [Cimatti, Irfan, Griggio, Roveri and Sebastiani, 2017].

6.5.2 The Solver for BVP

The BVP theory is solved through propositional flattening [Kroening and Strichman, 2016]. The solver supports the operations listed in Table 6.1, and allows the use of arbitrary bit-widths.[2] Based on an extensive testing the implementation is robust, but still prototypical in the sense that we implement no sophisticated pre-processing techniques that are available in many other bit-vector solvers (see, e.g., [Bruttomesso et al., 2007]).

Unlike many other SMT solvers (see, e.g., [Hadarean et al., 2014]), we do not implement the bit-vector solver as a separate SAT solver working on the flattening and driven by the main SAT solver. Instead, we flatten the problem directly to the main SAT solver. This has several advantages: we avoid the overhead of duplicate solver instantiation, and we enable the solver to potentially learn much more

[2] The shift operations $\ll, \gg_a, \gg_l$ assume a bit-width that is a power of two.

intricate relationships between the flattened formula and the formula in UFP. However, an in-depth analysis of the implications of this design is beyond the scope of this chapter.

6.5.3 Theory Refinement in Model Checking

We integrated Algorithm 9 into the bounded model checker HiFrog for C programs. HiFrog obtains first the symbolic encoding of the program P and a safety property t through a sequence of pre-processing steps, builds then the UFP formula, and finally gradually transforms parts of the UFP formula into BVP based on truth assignments until the safety is determined. We follow the approach where safety properties are expressed as assertions in the C code. The architecture is depicted in Figure 6.5. HiFrog maintains two SMT solvers during the execution and which are represented by the `CheckSAT` calls in Algorithm 9: the *main solver* for checking the satisfiability query constructed at line 5 (shown on the bottom of Figure 6.5) and the *refinement solver* for checking the spuriousness of each counterexample at line 9 (shown on the right of Figure 6.5). This choice was taken so that the expensive calls on the main solver would not be slowed down by unnecessary clauses at the refinement solver.

The counterexamples are flattened to propositional logic through the call to `getValues` by mapping the values in UFP to a unique bit-vector constant of the given bit width bw. At this stage of the development we ignore the case where the UFP solver gives more equivalence classes than what is representable in bw bits, since this limitation did not affect our results. Note that this work considers only fixed bit-width, namely 32 bits, thus exponential case split on all the possible partitions of the nodes in the egraph would not occur.

The binding formula (see Definition 13) is updated whenever a statement $x = t$ is refined. This is done by first constructing the BVP formulas $[x]^b$ and $[t]^b$, and then adding the missing equalities to F_B with the call to `computeBinding`.

6.6 Experimental results

This section evaluates the theory-refinement mode of HiFrog on C programs mostly coming from the software model checking competition (SV-COMP). The benchmarks were split into the *safe* (128 instances) and *unsafe* (30 instances) sets, indicating whether the bad behavior is reachable or not. Among safe instances, 17 require refinements.

For benchmarking we used Ubuntu 14.04 Linux system with two Intel Xeon

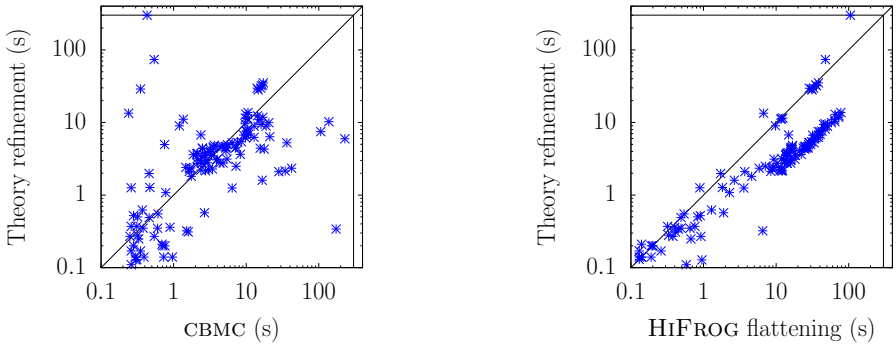


Figure 6.6. Timings of CBMC (left) and HiFrog's flattening (right) against HiFrog's theory refinement for the safe instances.

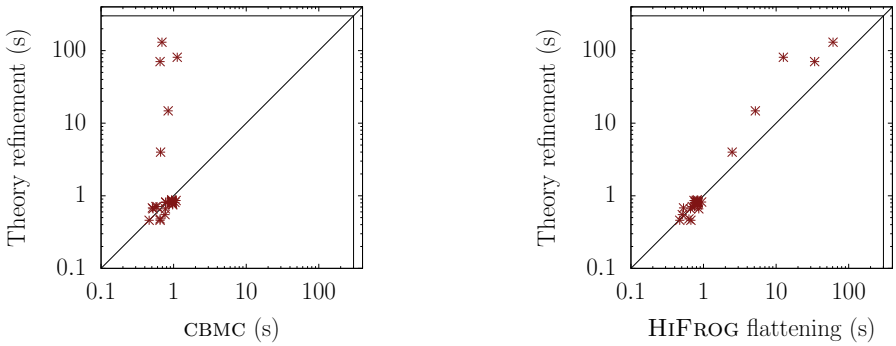


Figure 6.7. Timings of CBMC (left) and HiFrog's flattening (right) against HiFrog's theory refinement for the unsafe instances.

E5620 CPUs clocked at 2.40GHz and 12 Gigabyte memory limit per process using a timeout of 300 seconds CPU time. The model checker was compiled with the GNU C++ compiler and the O3 optimization level.

Figure 6.6 shows the verification results on safe properties. We compared (Figure 6.6, *left*) the HiFrog's theory-refinement mode against CBMC version 5.7, the winner of the software model checking competition falsification track in 2017

In 101 cases, HiFrog was either as fast or faster than CBMC, sometimes by orders of magnitude. Furthermore, HiFrog's theory refinement mode is compared against HiFrog's propositional flattening (Figure 6.6, *right*), hence ensuring that the only difference in the solvers is in how the symbolic encoding is presented to the SMT solver. In 115 cases, the theory refinement was either as fast or faster than flattening in determining safety, providing a more convincing evidence that the theory refinement approach works well in practice.

Table 6.2. Comparison of the heuristics against Min on instances requiring refinement.

	H0	H1	H2	H3	H4	H5	H6	H7	*Min*
#solved	**17**	16	**17**	**17**	**17**	**17**	**17**	**17**	17
#ref	660	2218	1250	1250	**533**	2266	1442	1831	162
time (s)	538	223	257	317	**123**	166	147	158	46.2

The verification results of unsafe benchmarks are shown in Figure 6.7. In five cases, bug detection by HiFrog was slower than the one by CBMC since HiFrog required iterative refining of all the expressions to confirm the validity of the counterexample. However, in the remaining cases, HiFrog was comparable to CBMC.

6.6.1 Experiments on Refinement Heuristic

Algorithm 9 does not address which exact statement should be refined based on a counterexample on Line 10 in case there are several possibilities. However this selection affects the run time of the model checking and is therefore of practical interest. We consider the following three features while building a refinement heuristic:

- Traversal order: the algorithm can proceed either by choosing from P the first statement (*forward order*) or the last statement (*backward order*) satisfying the condition on Line 10.
- All statements falsified by the counterexample are refined simultaneously (*simultaneous refinement*).
- All statements that depend on refined statements are refined simultaneously (*dependency refinement*).

The heuristics are as follows: H0 – Forward order; H1 – Backward order; H2 – Forward order with simultaneous refinement; H3 – Backward order with simultaneous refinement; H4 – Forward order with dependency refinement; H5 – Backward order with dependency refinement; H6 – Forward order with simultaneous and dependency refinement; and H7 – Backward order with simultaneous and dependency refinement. Based on the experimentation, the fastest solver on average results from using Forward order with dependency refinement. This

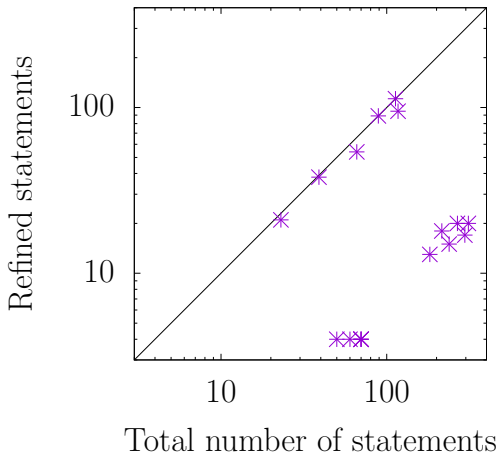


Figure 6.8. The number of refined statements using the Min heuristic with respect to the total number of statements.

is the heuristic we use in the results on Figs. 6.6-6.7. We briefly report on the results of the heuristics in Table 6.2 over the 17 instances of our total benchmark set where statements were refined. This benchmark set contains three crafted instances and the rest from the *bitvector* category of SV-COMP. The row labeled *#solved* reports how many instances the heuristic could solve before the time-out, *#ref* reports how many statements in total had to be refined over the set, and *time* reports the total run time. As a reference the table also reports results on the heuristic *Min* that requires no run time and computes a minimum set of refinements required to prove the property.

Finally, Figure 6.8 shows the reduction in the number of refined statements when using the *Min* heuristic on the 17 instances. As expected, the performance of the heuristic depends on the instance, but when effective, dramatically reduces the amount of flattened statements.

6.7 Related Work

Solving bit-vector problems with layers of theory solvers is introduced in [Brutomesso et al., 2007] and further developed in [Hadarean et al., 2014]. While we work directly on software verification instead of bit-vectors, our approach is related, as we also use hierarchy of solvers combined with rewriting techniques. However, we work explicitly on the modeling language by automatically adjust-

ing the precision to be different in different parts of the problem, and adding additional constraints that seams these parts together. In [Brummayer and Biere, 2009b] a CEGAR based approach is used for solving problems involving arrays by transforming an abstract representation into clauses. We differ from this approach in that we integrate the system on the theory solver level, employing in the experiments the congruence closure algorithm together with a propositional solver. To the best of our knowledge, no existing approach uses this level of granularity in the modeling. Furthermore, we use counterexamples that are checked against the bit-precise implementation, and this way can avoid refinement of program parts that would need to be refined in approaches based on layered theory solvers.

Exploiting simultaneously several theories for one verification goal is not new. For example, [Gurfinkel et al., 2014] presents a system for synthesizing safe bit-precise inductive invariants for software. Compared to this thesis, the refinement direction is inverted: the software is first flattened, and in case of a time-out, converted to a domain-specific theory. Furthermore, we integrate seamlessly the theories UFP and BVP into an SMT solver whereas [Gurfinkel et al., 2014] considers real arithmetic.

Uninterpreted functions have been used together with the bit-precise encoding for verifying the equivalence of Verilog designs in [Ho et al., 2016; Brady et al., 2011]. The approach uses machine learning to identify sub-components that can likely be abstracted. In contrast, our emphasis is on software verification and integration to the SMT solver.

Other abstraction refinement technique, specifically in the context of hardware verification, have also been explored in [Andraus et al., 2008]. Similarly their technique employs different abstraction level to obtain a compact representation of the design being checked compared to the original design. Unlike the abstraction refinement technique we present in this thesis, their technique uses universal facts as lemmas extracted from the concrete model to refute spurious counterexamples. Our technique obtains a model in combined EUF and BV and gradually concretizes the model to BV model only for the parts that precision is necessary and keeps the model as much as possible in EUF.

A related approach [Kutsuna et al., 2016] constructs test cases for scientific software by computing difference constraints from non-linear mathematical functions. This approach can be viewed as a special case of the framework we present in this chapter; the formulas we derive can also be used for generating test cases, although this is not the focus of this chapter. Similarly, [Cimatti, Irfan, Griggio, Roveri and Sebastiani, 2017] combines linear real arithmetic and equality of uninterpreted functions (*EUF*) for the SMT encoding of the program.

The algorithm initially uses *EUF* to abstract non-linear operators, and then uses the monotonicity and the multiplication checks to identify spurious counterexample thus avoiding simulation and code execution. Both checks might result in a refinement formula, which is added then to the current SMT encoding. Unlike ours, their approach cannot be applied as such for bit-precise reasoning. Our work in Chapter 3 reports very positive results on using the theory of EUF and LRA for encoding model checking problems. The work presented in this chapter which explores the possibilities in much more depth and rigor is motivated by this early result.

Another program-based refinement approach was proposed in [Katz et al., 2015], where compositional program is approximated with a program-specific theory of transition systems. Our approach is orthogonal to this, as we are able to handle programs in a more general way through the eventual flattening, while the theory of transition systems could likely be integrated as an additional theory.

In the domain of predicate abstraction, the work Beyer et al. [2015] uses a combination of both value and predicate analysis and proposes several heuristics for *refinement selection* to improve the effectiveness and efficiency of CEGAR-based analyses. The idea of refinement selection has been implemented in the open-source software-verification framework CPACHECKER. While this idea is restricted to predicate abstraction and considers program with unbounded loops, our approach considers only loop-free programs. Our methodology can systematically build the most light-weight abstract model by adjusting the precision to be different in different parts of the problem.

Finally, the CEGAR paradigm is extensively used in software verification. It is integrated into SLAM model checker used in Microsoft for verification of device drivers [Ball and Rajamani, 2002], but the approach does not use the fine-grained precision tuning we apply through theories. CEGAR is used in [Brummayer and Biere, 2009b] for solving problems involving arrays by transforming an abstract representation into clauses. We differ from this approach in that we integrate the system on the theory solver level, employing in the experiments the congruence closure algorithm together with a propositional solver.

6.8 Synopsis

This chapter presented a new approach for abstraction refinement in software verification by focusing on mixing different theories of SMT within the same verification task. The proposed approach introduces iterative *theory refinement* and supports solving of formulas of combined theories in the SMT solver, where

the binding to the theory is maintained by a series of identities in the original formula. The main contribution of this chapter is the gradual encoding process that uses the most precise theory only for a subset of all program statements, while handling the rest of the statements by using the less precise theories. This subset of the statements could either be identified by checking spurious counterexamples or simply specified by the user. The proposed framework can be extended by sets of theories with a partial order of refinement defined among them. In this chapter, we demonstrated the framework on the UFP theory with the partial refinement to the BVP theory. The theory refinement approach has been integrated into HIFROG. We studied various refinement strategies and compared them with a strategy computed offline, as well as with the propositional logic encoding known as flattening or bit-blasting. Improvement is seen both in the running time and in the size of the resulting formula, demonstrating that the spurious counterexamples are usually eliminated by refining a small number of statements in the formula.

6.9 Limitation and future work

A low number of abstraction refinement iterations is fundamental for the success of the CEGAR loop and, consequently to overall verification effort. In fact, the number of refinements required to verify the property grows with the complexity of a system. For this reason, it is of paramount importance to avoid as many redundant iterations as possible: even a single saved iteration can result in a substantial time-saving for large systems.

In the course of this chapter, we gained insights on how to use theories of *EUF* and *BV* in theory refinement and benefit from the best of both worlds. It would be interesting to extend theory refinement to exploit mixing more theories such as arithmetic theories and arrays. To this end, one has to define a partial order among these theories based on the level of abstraction/refinement that they provide. This will further improve the automatic refinement based on an analysis of the counterexamples using approaches such as interpolation. Moreover one also can develop more sophisticated heuristics and strategies for refinement.

Theory refinement algorithm is limited in that it cannot leverage the interpolation based summarization techniques presented in the previous chapters for further optimizations. A compelling avenue for future research would involve exploring the integration of function summarization with theory refinement. By combining these two approaches, it may be possible to further enhance the effectiveness of software verification methods. Function summarization involves sim-

plifying complex functions into more manageable abstractions, thereby reducing the overall computational cost of analyzing the program. By incorporating this technique into theory refinement algorithms, it may be possible to refine theories more efficiently and accurately, leading to faster and more precise results. Such an approach could have a significant impact on the field of software verification and could pave the way for new, more efficient verification methods.

Chapter 7

Contribution Summary

This research thesis aimed to improve the scalability and flexibility of software model checking. The study identified several challenges that hinder the broader application of symbolic model checking techniques in software verification, such as the expensive bit-precise reasoning, the need for effective ways to reuse computation history, and the determination of the appropriate level of abstraction for efficient symbolic model checking. These challenges are linked to the complexity problems inherent in the model checking paradigm. To overcome these challenges, the study introduced the concept of on-demand usage of SMT technology, which enhances the efficiency of symbolic model checking. Ultimately, the proposed approach can facilitate the wider adoption of symbolic model checking and improve the verification of software systems.

This thesis has proposed a range of techniques that have been combined in a novel way. Specifically, we have developed sound algorithms in bounded model checking that utilize SMT-based interpolation-based function summarization to reuse invested efforts between verification runs in an incremental manner. Our proposed solutions are based on Satisfiability Modulo Theories, which serve as a logical representation of programs. However, despite SMT reasoning being one of the most successful approaches to verifying software in a scalable way, it offers limited direct support for adapting the constraint language to the task at hand and requires various tuning to be reliable. Therefore, the main focus of this work has been to examine the balance between the precision of program encoding and the efficiency provided by SMT, aiming to identify an optimal level of abstraction that can facilitate effective symbolic model checking. In the following, we summarize the contribution of each chapter in more details.

In Chapter 3, we proposed an incremental verification approach whose functionality is interleaved with SMT reasoning for various computational tasks. Our

solution utilized SMT solving and SMT interpolating procedures for program abstraction and verification. The expressiveness of SMT logics allowed us for the development of symbolic model checking, which circumvents the need to expand the bit-precise encoding of models during verification. We presented a framework for generating and reusing SMT-based function summaries. The proposed framework is a fully-featured function-summarization-based model checker which enabled more efficient and scalable methods for verifying the correctness of programs. We validated our solution by developing HIFROG, a new SMT-based model checking framework for verifying a sequence of properties in a single program. We demonstrated that the use of SMT and SMT-based function summaries can scale up model checking to verify larger individual C programs with multiple properties compared to the use of rigid bit-precise encoding. HIFROG also serves as a foundation for validating the proposed approaches in the other chapters of the thesis.

In Chapter 4 we tackled the issue of verifying numerous programs that are closely connected. To prevent the need for costly full re-verification of each version and repeating a significant amount of work repeatedly, we presented an incremental algorithm that aims to maximize the reuse of prior computations. Our approach leverages an SMT-based family of summaries to compress the pertinent information from a previous verification run, enabling faster checks of new program versions. This approach simplifies the task of verifying if a modified program still satisfies a safety property by validating a set of summaries for the new program. This localization of the verification process to the altered parts of the program leads to substantial reductions in verification run time, as it eliminates the need to recheck the entire program. The proposed verification framework also provides an innovative capability of repairing previously computed summaries by means of iterative weakening and strengthening procedures. Moreover, it offers an efficient way of building formulas and refining them on-the-fly. The effectiveness of the proposed approach has been validated through the development of a new SMT-based model checking frameworks, UPPROVER, which have been successfully integrated into the interpolating SMT solver OpenSMT. Through extensive experimentation, we demonstrate that UPPROVER is significantly more efficient than its predecessor EVOLCHECK and non-incremental BMC approach.

In Chapter 5 a new concept called "theory interface" was presented, which facilitates the mapping of function summaries from one theory to another. By leveraging function summaries and the program's overall precision in a lazy manner, the theory interface allows for the sharing of function summaries across formulas in different theories, without the need for costly theory combination. The search

for a higher precision theory is performed lazily, meaning that existing summaries are attempted first by translating them into the current precision. In this chapter, a framework was introduced to create a series of over-approximation relations and translation rules between various SMT theories, specifically for *EUF*, *LRA*, and *BV*. These over-approximation relations ensure that a program proven to be safe with less precise theories remains safe according to the precise program semantics. The translation rules enable the reuse of function summaries across different theories, which is crucial to the framework's lazy approach. We evaluated the proposed idea by implementing an algorithm that performs both local and global refinement on demand, built on top of the HIFROG tool. We selectively choose the appropriate theory to accurately reason about program properties in the context of bounded model checking using function summaries. The proposed algorithm is extensively evaluated on a variety of large-scale benchmarks taken from SV-COMP, demonstrating its practical effectiveness. The results show that compared to a state-of-the-art model checker CBMC, our proposed framework can solve more instances. Our extensive evaluation reveals that using summary refinement via theory transformation instead of the naive summary refinement can assist the SMT solver in checking the smallest possible formula and improving program efficiency.

Chapter 6 presented a new approach for abstraction refinement in software verification with SMT solvers. The work was motivated by the observation that if a property being verified is proved using one of the light-weight theories the proof holds also for the exact BMC encoding of the program. However, the loss of precision can sometimes produce spurious counterexamples due to the over-approximating encoding. In order to overcome these spurious behaviors, a refinement process was employed alongside the abstraction. This process involves enhancing the abstract model and conducting repeated checks in an iterative manner. The proposed approach has introduced iterative *theory refinement* and supported solving of formulas of combined theories in the SMT solver, where the binding to the theory was maintained by a series of identities in the original formula. The main contribution of this chapter was the gradual encoding process that uses the most precise theory only for a subset of all program statements, while handling the rest of the statements by using the less precise theories. This subset of the statements could either be identified by checking spurious counterexamples or simply specified by the user. In this chapter, we demonstrated the framework on the UFP theory with the partial refinement to the BVP theory. Our study involved examining different refinement strategies and evaluating their effectiveness, comparing them to an offline-computed strategy as well as to the propositional logic encoding method known as flattening or bit-blasting. The

theory refinement algorithm has been validated through its implementation in HIFROG. The experimentation showed an improvement both in the running time and in the size of the resulting formula, demonstrating that the spurious counterexamples are usually eliminated by refining a small number of statements in the formula.

As a wrap up on future work of this thesis, we summarize some of the potential future directions that have been discussed at the end of each chapter. One interesting direction would be to have an unbounded proof by basing the approach on top of CHCs. Furthermore, supporting nontrivial programs that use arrays, heap data structures, and dynamic memory allocation would also be interesting with the help of additional theories of SMT. Since the idea of theory refinement does not use summaries, incorporating interpolation with the idea of theory refinement would be a promising idea. Another promising avenue for improving the refinement process involves developing new heuristics for candidate selection. While this thesis primarily focuses on localized refinement of theories using the technique of theory refinement, which utilizes a single counterexample to determine whether refinement is necessary, it would be beneficial to explore proof-based heuristics for refining the theories. Such heuristics have the potential to yield more relevant refinements. In situations where a larger pool of counterexamples is available for refinement, the generalization process may be easier compared to using a single data point. Hence, exploring alternative refinement techniques could prove fruitful for achieving more effective results. These will be a topic of interest for future research.

Appendix A

Transformation rules for BV and NRA

We define here a particular class of quantifier free theory of bit-vectors (*BV*) which is based on the work in Chapter 6 ([Hyvärinen et al., 2017]). In that paper the presented theory called BVP (Bit Vectors for Programs) which was an augmented version of the theory of bit-vectors. For abbreviation we use in this work the BV notation. In order to be applicable in our framework, BV should comply with the restriction that no overflows are allowed.

Based on SMT-LIB2 standard the signature in BV is as follows: $\Sigma^{BV} = \{+, *, bvand, bvor, bvudiv, bvurem, bvshl, bvlshr, bvnot, bvneg\}$. We consider the predicate symbols as $\mathcal{P} = \{>, <, \leq, \geq\}$. Note that for the addition and multiplication we use the same notation i.e., "+" and "*" throughout the work to highlight the fact that the syntax are in common with our theory of interest. Therefore syntactically they can be used in the transformation rules. However, the task of interpretation of each function symbol must be delegated to the corresponding theory solver.

In the following the rules for translation from T^{uni} to *BV* are as follows:

$$\frac{[t_1 = t_2]^{BV}}{[t_1]^{BV} = [t_2]^{BV}}$$

$$\frac{[v]^{BV}}{v} \qquad v \text{ is a variable or a constant}$$

$$\frac{[t_1 \bowtie t_2]^{BV}}{[t_1]^{BV} \bowtie [t_2]^{BV}} \qquad \bowtie \text{ is a predicate symbol or binary function symbol in } BV, \; i.e., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}$$

$$\frac{[\triangle t_1]^{BV}}{\triangle [t_1]^{BV}} \qquad \triangle \text{ is a unary function symbol in } BV, \; i.e., \triangle \in \{bvnot, bvneg\}$$

$$\frac{[f(\mathbf{t})]^{BV}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise}$$

(A.1)

The rules for transforming from BV to theory interface T^{uni} are as follows:

$$\frac{[t_1 = t_2]^{T^{uni}}}{[t_1]^{T^{uni}} = [t_2]^{T^{uni}}}$$

$$\frac{[t_1 \bowtie t_2]^{T^{uni}}}{[t_1]^{T^{uni}} \bowtie [t_2]^{T^{uni}}} \qquad \bowtie \text{ is a binary function symbol in } BV, \; e.g., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}$$

$$\frac{[\triangle t_1]^{T^{uni}}}{\triangle [t_1]^{T^{uni}}} \qquad \triangle \text{ is a unary function symbol in } BV, \triangle \in \{bvnot, bvneg\}$$

$$\frac{[v]^{T^{uni}}}{v} \qquad v \text{ is a variable or a constant}, v \notin dom(\mathcal{M}^{-1})$$

$$\frac{[v]^{T^{uni}}}{\mathcal{M}^{-1}(v)} \qquad v \in dom(\mathcal{M}^{-1})$$

(A.2)

In the following we present the translation rules from the NRA to T^{uni} and vice versa which are listed in (A.3) and (A.4), respectively. The rules for transforming

from T^{uni} to *NRA* are as follows:

$$\frac{[t_1 = t_2]^{NRA}}{([t_1]^{NRA} \leq [t_2]^{NRA}) \wedge ([t_2]^{NRA} \leq [t_2]^{NRA})}$$

$$\frac{[v]^{NRA}}{v} \; v \text{ is a variable or a constant} \tag{A.3}$$

$$\frac{[t_1 \bowtie t_2]^{NRA}}{[t_1]^{NRA} \bowtie [t_2]^{NRA}} \; \bowtie \text{ is a function symbol in } NRA, e.g., \bowtie \in \{+, -, *\}$$

Likewise the LRA rules, the rules for transforming from *NRA* to T^{uni} are as follows:

$$\frac{([t_1]^{T^{uni}} \leq [t_2]^{T^{uni}}) \wedge ([t_2]^{T^{uni}} \leq [t_1]^{T^{uni}})]}{[t_1]^{T^{uni}} = [t_2]^{T^{uni}}}$$

$$\frac{[v]^{T^{uni}}}{v} \; v \text{ is a variable or a constant} \tag{A.4}$$

$$\frac{[f(\mathbf{t})]^{T^{uni}}}{f([\mathbf{t}]^{T^{uni}})}$$

Appendix B

List of Publications

B.1 List of publications related to this thesis

B.1.1 Journals

Asadi, S., Blicha, M., Hyvarinen, A., Fedyukovich, G., and Sharygina, N. SMT-based verification of program changes through summary repair. Journal of Formal Methods in System Design. Submitted in 2021, Accepted with minor revision in April 2023.

B.1.2 Main conference proceedings

1. **Asadi, S.**, Blicha, M., Hyvarinen, A., Fedyukovich, G., and Sharygina, N. (2020b). Incremental verification by SMT-based summary repair. In Formal Methods in Computer Aided Design, (FMCAD 2020) 2020 - pages 77–82. IEEE. (link)

2. **Asadi, S.**, Blicha, M., Hyvarinen, A., Fedyukovich, G., and Sharygina, N. (2020a). Farkas-based tree interpolation. In 27th International Static Analysis Symposium, (SAS 2020) - Chicago, USA, volume 12389 of LNCS, pages 357–379. Springer. (link)

3. **Asadi, S.**, Blicha, M.,Fedyukovich, G., Hyvarinen, A., Even-Mendoza, K., Sharygina, N. ,and Chockler, H. (2018). Function summarization modulo theories. In 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2018) - Awassa, Ethiopia, volume 57 of EPiC Series in Computing, pages 56–75. EasyChair. (link)

4. Alt, L., **Asadi, S.**, Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvarinen, A., and Sharygina, N. (2017). HiFrog: SMT-based function summarization for software verification. In 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2017) - Uppsala, Sweden, volume 10206 of LNCS, pages 207–213. Springer. (link)

5. Hyvarinen, A., **Asadi, S.**, Even-Mendoza, K., Fedyukovich, G., Chockler, H., and Sharygina, N. (2017). Theory refinement for program verification. In 20th International Conference Theory and Applications of Satisfiability Testing, (SAT 2017) - Melbourne, Australia, volume 10491 of LNCS, pages 347–363. Springer. (link)

B.1.3 Poster presentation

Asadi, S., HiFrog: Interpolation-based Software Verification using Theory Refinement, FMCAD student forum poster presentation, 2017. URL: `www.cs.utexas.edu/users/hunt/FMCAD/FMCAD17/student-forum/`

B.2 Other collaborative publications

1. Alt, L., Hyvarinen, A., **Asadi, S.**, and Sharygina, N. (2017). Duality-based interpolation for quantifier-free equalities and uninterpreted functions. In Formal Methods in Computer Aided Design, (FMCAD 2017) - Vienna, Austria, pages 39–46. IEEE. (link)

2. Even-Mendoza, K., **Asadi, S.**, Hyvarinen, A., Chockler, H., and Sharygina, N. (2018). Lattice-based refinement in bounded model checking. In 10th International Conference on Verified Software. Theories, Tools, and Experiments, (VSTTE 2018) - Oxford, UK, volume 11294 of LNCS, pages 50– 68. Springer. (link)

3. Marescotti, M., Blicha, M., Hyvarinen, A., **Asadi, S.**, and Sharygina, N. (2018). Computing exact worst-case gas consumption for smart contracts. In 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, (ISoLA 2018) - Limassol, Cyprus, volume 11247 of LNCS, pages 450–465. Springer. (link)

Bibliography

Albarghouthi, A., Gurfinkel, A. and Chechik, M. [2012a]. From under-approximations to over-approximations and back, *18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, Vol. 7214 of *LNCS*, Springer, pp. 157–172.

Albarghouthi, A., Gurfinkel, A. and Chechik, M. [2012b]. Whale: An interpolation-based algorithm for inter-procedural verification, *13th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*, Vol. 7148 of *LNCS*, Springer, Heidelberg, pp. 39–55.

Albarghouthi, A. and McMillan, K. L. [2013]. Beautiful interpolants, *25th International Conference on Computer Aided Verification, CAV*, pp. 313–329.

Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S. and Sharygina, N. [2012]. Lazy abstraction with interpolants for arrays, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18*, Vol. 7180, Springer, pp. 46–61.

Alt, L., Asadi, S., Chockler, H., Mendoza, K. E., Fedyukovich, G., Hyvärinen, A. and Sharygina, N. [2017]. HiFrog: SMT-based function summarization for software verification, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Vol. 10206, Springer, Heidelberg, pp. 207–213.

Alt, L., Fedyukovich, G., Hyvärinen, A. and Sharygina, N. [2016]. A proof-sensitive approach for small propositional interpolants, *VSTTE 2015*, Vol. 9593, Springer, Berlin, Heidelberg, pp. 1–18.

Alt, L., Hyvärinen, A., Asadi, S. and Sharygina, N. [2017]. Duality-based interpolation for quantifier-free equalities and uninterpreted functions, *2017 Formal Methods in Computer Aided Design, FMCAD 2017*, FMCAD Inc, Austin, Texas, pp. 39–46.

Alt, L., Hyvärinen, A. E. J. and Sharygina, N. [2017]. LRA interpolants from no man's land, *Haifa Verification Conference, HVC*, Vol. 10629, Springer, Heidelberg, pp. 195–210.

Alt, L. and Reitwießner, C. [2018]. Smt-based verification of solidity smart contracts, *in* T. Margaria and B. Steffen (eds), *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, Vol. 11247 of *LNCS*, Springer, pp. 376–388.

Alur, R., Itai, A., Kurshan, R. P. and Yannakakis, M. [1995]. Timing verification by successive approximation, *Inf. Comput.* **118**(1): 142–157.

Andraus, Z. S., Liffiton, M. H. and Sakallah, K. A. [2008]. Reveal: A formal verification tool for verilog designs, *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 343–352.

Armando, A., Mantovani, J. and Platania, L. [2009]. Bounded model checking of software using SMT solvers instead of SAT solvers, *International Journal on Software Tools for Technology Transfer* **11**(1): 69–83.

Asadi, S., Blicha, M., Fedyukovich, G., Hyvärinen, A. E. J., Even-Mendoza, K., Sharygina, N. and Chockler, H. [2018]. Function summarization modulo theories, *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR-22*, Vol. 57, EasyChair, England & Wales, pp. 56–75.

Asadi, S., Blicha, M., Hyvärinen, A., Fedyukovich, G. and Sharygina, N. [2020a]. Farkas-based tree interpolation, *International Symposium on Static Analysis, SAS*, Springer, Heidelberg.

Asadi, S., Blicha, M., Hyvärinen, A., Fedyukovich, G. and Sharygina, N. [2020b]. Incremental verification by SMT-based summary repair, *Formal Methods in Computer Aided Design, FMCAD 2020*, IEEE, New York.

Asadi, S., Blicha, M., Hyvärinen, A., Fedyukovich, G. and Sharygina, N. [2023]. SMT-based verification of program changes through summary repair, *Journal of Formal Methods in System Design* .

Asadzade, M., Blicha, M., Hyvärinen, A. E. and Sharygina, N. [2021]. The opensmt solver in SMT-COMP 2021, *16th International Satisfiability Modulo Theories Competition (SMT-COMP 2021)*.

Babic, D. and Hu, A. J. [2008]. Calysto: scalable and precise extended static checking, *Int. Conference on Software Engineering (ICSE '08)*, pp. 211–220.

Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K. S., Rungta, N., Tkachuk, O. and Varming, C. [2018]. Semantic-based automated reasoning for AWS access policies using SMT, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, IEEE, pp. 1–9.

Balarin, F. and Sangiovanni-Vincentelli, A. L. [1993]. An iterative approach to language containment, *5th International Conference on Computer Aided Verification*, Springer-Verlag, London, UK, UK, pp. 29–40.

Ball, T., Levin, V. and Rajamani, S. K. [2011]. A decade of software model checking with SLAM, *Commun. ACM* **54**(7): 68–76.

Ball, T. and Rajamani, S. K. [2000]. Bebop: A symbolic model checker for boolean programs, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, Vol. 1885 of *LNCS*, Springer, pp. 113–130.

Ball, T. and Rajamani, S. K. [2002]. The SLAM Project: Debugging System Software via Static Analysis, *POPL*, pp. 1–3.

Barbosa, H., , Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Noetzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C. and Zohar, Y. [2022]. cvc5: A versatile and industrial-strength smt solver, *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. to appear.

Barrett, C., Fontaine, P. and Tinelli, C. [2021]. The Satisfiability Modulo Theories Library (SMT-LIB), https://smtlib.cs.uiowa.edu/.

Barrett, C. W., Sebastiani, R., Seshia, S. A. and Tinelli, C. [2009]. Satisfiability modulo theories, *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 825–885.

Bartocci, E., Beyer, D., Black, P. E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T. and Yamada, A. [2019]. Toolympics 2019: An overview of competitions in formal methods, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019*, Vol. 11429 of *LNCS*, Springer, pp. 3–24.

Basler, G., Kroening, D. and Weissenbacher, G. [2007]. SAT-Based Summarization for Boolean Programs, *Model Checking Software, 14th International SPIN Workshop*, pp. 131–148.

Bayless, S., Val, C. G., Ball, T., Hoos, H. H. and Hu, A. J. [2013]. Efficient modular SAT solving for IC3, *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, IEEE, pp. 149–156.

Beyer, D. and Dangl, M. [2019]. Software verification with PDR: implementation and empirical evaluation of the state of the art, *CoRR* **abs/1908.06271**.

Beyer, D., Dangl, M., Dietsch, D. and Heizmann, M. [2016]. Correctness witnesses: exchanging verification results between verifiers, *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, ACM, pp. 326–337.

Beyer, D., Jakobs, M. and Lemberger, T. [2020]. Difference verification with conditions, *SEFM 2020*, Vol. 12310, Springer, Cham, pp. 133–154.

Beyer, D. and Keremoglu, M. E. [2011]. CPAchecker: A tool for configurable software verification, *International Conference on Computer Aided Verification, CAV*, Vol. 6806, Springer, pp. 184–190.

Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A. and Wendler, P. [2013]. Precision reuse for efficient regression verification, *ESEC/FSE 2013*, ACM, New York, pp. 389–399.

Beyer, D., Löwe, S. and Wendler, P. [2015]. Refinement selection, *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, Vol. 9232 of *LNCS*, Springer, pp. 20–38.

Beyer, D. and Podelski, A. [2022]. Software model checking: 20 years and beyond, *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, Vol. 13660 of *LNCS*, Springer, pp. 554–582.

Biere, A., Cimatti, A., Clarke, E. M. and Zhu, Y. [1999]. Symbolic model checking without BDDs, *TACAS 1999*, Vol. 1579, Springer, Heidelberg, pp. 193–207.

Biere, A., Heule, M., van Maaren, H. and Walsh, T. [2009]. *Handbook of Satisfiability*, Vol. 85, IOS Press.

Bjørner, N. [2018]. Z3 and SMT in industrial r&d, *22nd International Symposium on Formal Methods, FM 2018*, Vol. 10951 of *LNCS*, Springer, pp. 675–678.

Bjørner, N. S., McMillan, K. L. and Rybalchenko, A. [2012]. Program verification as satisfiability modulo theories, *10th International Workshop on Satisfiability Modulo Theories, SMT*, Vol. 20 of *EPiC Series in Computing*, EasyChair, pp. 3–11.

Blanc, R., Gupta, A., Kovács, L. and Kragl, B. [2013]. Tree interpolation in Vampire, *LPAR 2013*, Vol. 8312, Springer, pp. 173–181.

Blicha, M., Hyvärinen, A., Kofron, J. and Sharygina, N. [2019]. Decomposing Farkas interpolants, *TACAS 2019*, Vol. 11427, Springer, Heidelberg, pp. 3–20.

Böhme, S., Moskal, M., Schulte, W. and Wolff, B. [2010]. Hol-boogie - an interactive prover-backend for the verifying C compiler, *J. Autom. Reason.* **44**(1-2): 111–144.

Bonacina, M. P., Fontaine, P., Ringeissen, C. and Tinelli, C. [2019]. Theory combination: Beyond equality sharing, *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*, Vol. 11560 of *LNCS*, Springer, pp. 57–89.

Bradley, A. R. [2011]. SAT-based model checking without unrolling, *12th International Conference Verification, Model Checking, and Abstract Interpretation - , VMCAI*, Vol. 6538, Springer, pp. 70–87.

Brady, B. A., Bryant, R. E. and Seshia, S. A. [2011]. Learning conditional abstractions, *International Conference on Formal Methods in Computer-Aided Design, FMCAD*, FMCAD Inc., pp. 116–124.

Brummayer, R. and Biere, A. [2009a]. Boolector: An efficient SMT solver for bit-vectors and arrays, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS*, Vol. 5505 of *LNCS*, Springer, pp. 174–177.

Brummayer, R. and Biere, A. [2009b]. Lemmas on demand for the extensional theory of arrays, *J. Satisf. Boolean Model. Comput.* **6**(1-3): 165–201.

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A. and Sebastiani, R. [2007]. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems, *International Conference on Computer Aided Verification, CAV*, Vol. 4590 of *LNCS*, Springer, Heidelberg, pp. 547 – 560.

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A. and Sebastiani, R. [2009]. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis, *Ann. Math. Artif. Intell.* **55**(1-2): 63–99.

Bruttomesso, R., Pek, E., Sharygina, N. and Tsitovich, A. [2010]. The OpenSMT solver, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Vol. 6015, Springer, pp. 150 – 153.

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

Bryant, R. E., Kroening, D., Ouaknine, J., Seshia, S. A., Strichman, O. and Brady, B. [2007]. Deciding Bit-Vector Arithmetic with Abstraction , *TACAS'07*, Vol. 4424, Springer, pp. 358–372.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L. and Hwang, L. J. [1990]. Symbolic model checking: 10ˆ20 states and beyond, *Fifth annual symposium on Logic in Computer Science (LICS '90)*, pp. 428–439.

C. [2011]. Information technology – programming languages, *Standard*, International Organization for Standardization, Geneva, Switzerland.

Cabodi, G., Murciano, M., Nocco, S. and Quer, S. [2006]. Stepping forward with interpolants in unbounded model checking, *ICCAD'06*, ACM, pp. 772–778.

Cabodi, G., Palena, M. and Pasini, P. [2014]. Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking, *FMCAD'14*, IEEE, pp. 43–50.

Champion, A., Kobayashi, N. and Sato, R. [2018]. Hoice: An ice-based non-linear horn clause solver, *16th Asian Symposium on Programming Languages and Systems, APLAS*, Vol. 11275 of *LNCS*, Springer, pp. 146–156.

Champion, A., Mebsout, A., Sticksel, C. and Tinelli, C. [2016]. The kind 2 model checker, *Computer Aided Verification - 28th International Conference, CAV*, Vol. 9780 of *LNCS*, Springer, pp. 510–517.

Chauhan, P., Clarke, E. M., Kukula, J. H., Sapra, S., Veith, H. and Wang, D. [2002]. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis, *FMCAD*, Vol. 2517, Springer, Heidelberg, pp. 33–51.

Christ, J. and Hoenicke, J. [2016]. Proof tree preserving tree interpolation, *J. Autom. Reasoning* **57**(1): 67–95.

Christ, J., Hoenicke, J. and Nutz, A. [2012]. Smtinterpol: An interpolating SMT solver, *19th International Workshop on Model Checking Software, SPIN*, Vol. 7385, Springer, pp. 248–254.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A. [2002]. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking, *International Conference on Computer Aided Verification, CAV*, Vol. 2404, Springer.

Cimatti, A., Griggio, A., Irfan, A., Roveri, M. and Sebastiani, R. [2017]. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF, *in* A. Legay and T. Margaria (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS*, Vol. 10205 of *LNCS*, pp. 58–75.

Cimatti, A., Griggio, A., Schaafsma, B. J. and Sebastiani, R. [2013]. The mathsat5 SMT solver, *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, Vol. 7795 of *LNCS*, Springer, Heidelberg, pp. 93–107.

Cimatti, A., Griggio, A. and Sebastiani, R. [2008]. Efficient Interpolant Generation in Satisfiability Modulo Theories, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, pp. 397–412.

Cimatti, A., Irfan, A., Griggio, A., Roveri, M. and Sebastiani, R. [2017]. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF, *Proc. TACAS 2017*, Vol. 10205 of *LNCS*, pp. 58 – 75.

Cimatti, A., Mover, S. and Tonetta, S. [2012]. Smt-based verification of hybrid systems, *in* J. Hoffmann and B. Selman (eds), *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, AAAI Press.
URL: *http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5072*

Clarke, E., Grumberg, O., Jha, S., Lu, Y. and Veith, H. [2003]. Counterexample-guided abstraction refinement for symbolic model checking, *J. ACM* **50**(5): 752–794.

Clarke, E., Kroening, D. and Lerda, F. [2004]. A tool for checking ANSI-C programs, *TACAS 2004*, Vol. 2988, Springer, Berlin, Heidelberg, pp. 168–176.

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

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. [2000]. Counterexample-guided abstraction refinement, *12th International Conference on Computer Aided Verification, CAV*, pp. 154–169.

Clarke, E. M., Gupta, A. and Strichman, O. [2004]. SAT-based-guided abstraction refinement, *IEEE Trans. on CAD of Integrated Circuits and Systems* **23**(7): 1113–1123.

Colón, M. and Uribe, T. E. [1998]. Generating finite-state abstractions of reactive systems using decision procedures, *in* A. J. Hu and M. Y. Vardi (eds), *International Conference on Computer Aided Verification, CAV*, Vol. 1427 of *LNCS*, Springer, pp. 293–304.

Conway, C. L., Namjoshi, K. S., Dams, D. and Edwards, S. A. [2005]. Incremental algorithms for inter-procedural analysis of safety properties, *CAV 2005*, Vol. 3576, Springer, Heidelberg, pp. 449–461.

Cook, B., Döbel, B., Kroening, D., Manthey, N., Pohlack, M., Polgreen, E., Tautschnig, M. and Wieczorkiewicz, P. [2020]. Using model checking tools to triage the severity of security bugs in the xen hypervisor, *Formal Methods in Computer Aided Design, FMCAD*, IEEE, pp. 185–193.

Cook, B., Kroening, D. and Sharygina, N. [2005]. COGENT: Accurate Theorem Proving for Program Verification, *International Conference on Computer Aided Verification, CAV*, Vol. 3576, pp. 296–300.

Cordeiro, L. C. and de Lima Filho, E. B. [2016]. Smt-based context-bounded model checking for embedded systems: Challenges and future trends, *ACM SIGSOFT Software Engineering Notes* **41**(3): 1–6.

Cordeiro, L. C., Fischer, B. and Marques-Silva, J. [2012]. Smt-based bounded model checking for embedded ANSI-C software, *IEEE Trans. Software Eng.* **38**(4): 957–974.

Cousot, P. and Cousot, R. [1977]. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in* R. M. Graham, M. A. Harrison and R. Sethi (eds), *the Fourth ACM Symposium on Principles of Programming Languages*, ACM, New York, pp. 238–252.

Craig, W. [1957]. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory, *J. Symb. Log.* **22**(3): 269–285.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, F. [1989]. An efficient method of computing static single assignment form, *POPL 1989*, ACM, pp. 25–35.

de Moura, L. M. and Bjørner, N. [2008]. Z3: an efficient SMT solver, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, Vol. 4963, Springer, pp. 337–340.

de Moura, L. M. and Bjørner, N. [2009]. Satisfiability modulo theories: An appetizer, *SBMF 2009*, Vol. 5902 of *LNCS*, Springer, Heidelberg, pp. 23–36.

de Moura, L. M. and Bjørner, N. [2011]. Satisfiability modulo theories: introduction and applications, *Commun. ACM* **54**(9): 69–77.

Decision Procedure Toolkit [2007]. `https://dpt.sourceforge.net/`. Accessed: Feb 2022.

Detlefs, D., Nelson, G. and Saxe, J. B. [2005]. Simplify: A theorem prover for program checking, *Journal of the ACM* **52**(3): 365–473.

Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A. and Podelski, A. [2019]. Ultimate treeautomizer, *the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR*, Vol. 296 of *EPTCS*, pp. 42–47.

Dill, D. L., Grieskamp, W., Park, J., Qadeer, S., Xu, M. and Zhong, J. E. [2021]. Fast and reliable formal verification of smart contracts with the move prover, *CoRR* **abs/2110.08362**.

Donaldson, A. F., Kroening, D. and Rümmer, P. [2010]. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS*, Vol. 6015 of *LNCS*, Springer, pp. 280–295.

D'Silva, V., Kroening, D., Purandare, M. and Weissenbacher, G. [2010]. Interpolant strength, *VMCAI 2010*, Vol. 5944, Springer, pp. 129–145.

Dutertre, B. [2014]. Yices 2.2, *Computer Aided Verification - 26th International Conference, CAV 2014*, Vol. 8559 of *LNCS*, Springer, pp. 737–744.

Dutertre, B. and de Moura, L. M. [2006]. A fast linear-arithmetic solver for DPLL(T), *International Conference on Computer Aided Verification, CAV*, Vol. 4144 of *LNCS*, Springer, pp. 81–94.

Eén, N., Mischenko, A. and Brayton, R. K. [2011]. Efficient implementation of property directed reachability, *FMCAD 2011*, FMCAD Inc., pp. 125–134.

Engler, D. R. and Ashcraft, K. [2003]. Racerx: effective, static detection of race conditions and deadlocks, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP*, ACM, pp. 237–252.

Ernst, M. D., Cockrell, J., Griswold, W. G. and Notkin, D. [2001]. Dynamically discovering likely program invariants to support program evolution, *IEEE Trans. Software Eng.* **27**(2): 99–123.
URL: *https://doi.org/10.1109/32.908957*

Esen, Z. and Rümmer, P. [2021]. A theory of heap for constrained horn clauses (extended technical report), *CoRR* **abs/2104.04224**.

Falke, S., Merz, F. and Sinz, C. [2013]. The bounded model checker LLBMC, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2013*, IEEE, pp. 706–709.

Fan, K., Yang, M. and Huang, C. [2016]. Automatic abstraction refinement of TR for PDR, *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*, IEEE, pp. 121–126.

Fedyukovich, G. and Bodík, R. [2018]. Accelerating Syntax-Guided Invariant Synthesis, *TACAS, 2018*, Vol. 10805, Springer, Heidelberg, pp. 251–269.

Fedyukovich, G., D'Iddio, A. C., Hyvärinen, A. E. J. and Sharygina, N. [2015]. Symbolic detection of assertion dependencies for bounded model checking, *in* A. Egyed and I. Schaefer (eds), *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015*, Vol. 9033, Springer, pp. 186–201.

Fedyukovich, G., Gurfinkel, A. and Sharygina, N. [2014]. Incremental verification of compiler optimizations, *NFM 2014*, pp. 300–306.

Fedyukovich, G., Gurfinkel, A. and Sharygina, N. [2016]. Property directed equivalence via abstract simulation, *International Conference on Computer Aided Verification, CAV*, Vol. 9780, Part II, Springer, Cham, pp. 433–453.

Fedyukovich, G., Kaufman, S. J. and Bodík, R. [2017]. Sampling invariants from frequency distributions, *Formal Methods in Computer Aided Design, FMCAD*, IEEE, pp. 100–107.

Fedyukovich, G., Sery, O. and Sharygina, N. [2013]. eVolCheck: Incremental upgrade checker for C, *TACAS 2013*, Vol. 7795, Springer, Heidelberg, pp. 292–307.

Fedyukovich, G., Sery, O. and Sharygina, N. [2017]. Flexible SAT-based framework for incremental bounded upgrade checking, *STTT* **19**(5): 517–534.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P. and Ulbrich, M. [2014]. Automating regression verification, *ASE 2014*, ACM, New York, pp. 349–360.

Flanagan, C. and Leino, K. R. M. [2001]. Houdini, an annotation assistant for esc/java, *FME 2001*, Vol. 2021, Springer, Heidelberg, pp. 500–517.

Freeman, T. S. and Pfenning, F. [1991]. Refinement types for ML, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, ACM, pp. 268–277.

Gacek, A., Backes, J., Whalen, M., Wagner, L. G. and Ghassabani, E. [2018]. The jkind model checker, *Computer Aided Verification - 30th International Conference, CAV*, Vol. 10982 of *LNCS*, Springer, pp. 20–27.

Gadelha, M. Y. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B. and Nicole, D. A. [2018]. ESBMC 5.0: an industrial-strength C model checker, *the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*, ACM, pp. 888–891.

Ganai, M. K. and Gupta, A. [2006]. Accelerating high-level bounded model checking, *2006 International Conference on Computer-Aided Design, ICCAD 2006, San Jose, CA, USA, November 5-9, 2006*, ACM, pp. 794–801.

Godefroid, P. [2007]. Compositional dynamic test generation, *the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, pp. 47–54.

Godefroid, P., Lahiri, S. K. and Rubio-González, C. [2011]. Statically validating must summaries for incremental compositional dynamic test generation, *International Symposium on Static Analysis, SAS*, Vol. 6887, Springer, Heidelberg, pp. 112–128.

Godlin, B. and Strichman, O. [2009]. Regression verification, *the 46th Design Automation Conference, DAC*, ACM, New York, pp. 466–471.

Graf, S. and Saïdi, H. [1997]. Construction of abstract state graphs with PVS, *9th International Conference on Computer Aided Verification, CAV*, Vol. 1254 of *LNCS*, Springer, pp. 72–83.

Grebenshchikov, S., Lopes, N. P., Popeea, C. and Rybalchenko, A. [2012]. Synthesizing software verifiers from proof rules, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, ACM, New York, pp. 405–416.

Griggio, A., Le, T. T. H. and Sebastiani, R. [2011]. Efficient interpolant generation in satisfiability modulo linear integer arithmetic, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS*, pp. 143–157.

Grishchenko, I., Maffei, M. and Schneidewind, C. [2018]. Foundations and tools for the static analysis of ethereum smart contracts, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, Vol. 10981, Springer, pp. 51–78.

Güdemann, M. [2022]. Smt-based verification of concurrent critical system, *in* C. Wressnegger, D. Reinhardt, T. Barber, B. C. Witt, D. Arp and Z. Á. Mann (eds), *Sicherheit, Schutz und Zuverlässigkeit: Konferenzband der 11. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), Sicherheit 2022*, Vol. P-323 of *LNI*, Gesellschaft für Informatik, Bonn, pp. 67–82.

Gupta, A., Popeea, C. and Rybalchenko, A. [2011]. Solving recursion-free Horn clauses over LI+UIF, *Programming Languages and Systems - 9th Asian Symposium, APLAS*, Vol. 7078, Springer, pp. 188–203.

Gurfinkel, A., Belov, A. and Marques-Silva, J. [2014]. Synthesizing safe bit-precise invariants, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Vol. 8413, Springer, pp. 93–108.

Gurfinkel, A. and Bjørner, N. [2019]. The science, art, and magic of constrained horn clauses, *21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*, IEEE, pp. 6–10.

Gurfinkel, A., Chaki, S. and Sapra, S. [2011]. Efficient predicate abstraction of program summaries, *NASA Formal Methods - 3rd International Symposium, NFM*, Vol. 6617 of *LNCS*, Springer, pp. 131–145.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. [2015]. The SeaHorn verification framework, *27th International Conference on Computer Aided Verification, CAV*, Vol. 9206, Springer, Heidelberg, pp. 343–361.

Gurfinkel, A., Rollini, S. F. and Sharygina, N. [2013]. Interpolation properties and SAT-based model checking, *ATVA 2013*, pp. 255–271.

Hadarean, L., Bansal, K., Jovanović, D., Barret, C. and Tinelli, C. [2014]. A tale of two solvers: Eager and lazy approaches to bit-vectors, *International Conference on Computer Aided Verification, CAV*, Springer, Heidelberg, pp. 680 – 695.

Hardin, R. H., Kurshan, R. P., McMillan, K. L., Reeds, J. A. and Sloane, N. J. A. [1996]. Efficient regression verification, *Proc. WODES'96*, IEEE, Proc., pp. 147–150.

Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S. T. V. and Zill, B. [2015]. Ironfleet: proving practical distributed systems correct, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*, ACM, pp. 1–17.

He, F., Yu, Q. and Cai, L. [2018]. When regression verification meets CEGAR, *CoRR* **abs/1806.04829**.

He, S. and Rakamarić, Z. [2017]. Counterexample-guided bit-precision selection, *Asian Symposium on Programming Languages and Systems*, Springer, pp. 534–553.

Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T. and Podelski, A. [2018]. Ultimate automizer and the search for perfect interpolants - (competition contribution), *Proc. TACAS 2018*, pp. 447–451.

Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C. and Podelski, A. [2013]. Ultimate automizer with smtinterpol - (competition contribution), *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS*, Vol. 7795 of *LNCS*, Springer, pp. 641–643.

Heizmann, M., Hoenicke, J. and Podelski, A. [2009]. Refinement of trace abstraction, *in* J. Palsberg and Z. Su (eds), *Static Analysis, 16th International Symposium, SAS*, Vol. 5673, Springer, pp. 69–85.

Heizmann, M., Hoenicke, J. and Podelski, A. [2010a]. Nested interpolants, *POPL 2010*, ACM, New York, pp. 471–482.

Heizmann, M., Hoenicke, J. and Podelski, A. [2010b]. Nested interpolants, *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, New York, pp. 471–482.

Henzinger, T., Jhala, R., Majumdar, R. and McMillan, K. [2004]. Abstractions from Proofs, *POPL*.

Ho, Y., Mishchenko, A. and Brayton, R. K. [2017]. Property directed reachability with word-level abstraction, *in* D. Stewart and G. Weissenbacher (eds), *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, IEEE, pp. 132–139.

Ho, Y.-S., Chauhan, P., Roy, P., Mishchenko, A. and Brayton, R. [2016]. Efficient uninterpreted function abstraction and refinement for word-level model checking, *Proc. FMCAD 2016*, ACM, pp. 65–72.

Hoare, C. [1971]. Procedures and parameters: An axiomatic approach, *Symposium on Semantics of Algorithmic Languages* pp. 102–116.

Hoare, C. A. R. [1969]. An axiomatic basis for computer programming, *Commun. ACM* **12**(10): 576–580.

Huang, X., Kwiatkowska, M., Wang, S. and Wu, M. [2016]. Safety verification of deep neural networks, *CoRR* **abs/1610.06940**.

Hyvärinen, A. E. J., Asadi, S., Mendoza, K. E., Fedyukovich, G., H.Chockler and Sharygina, N. [2017]. Theory refinement for program verification, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference*, Vol. 10491, Springer, Heidelberg, pp. 347–363.

Hyvärinen, A., Marescotti, M., Alt, L. and Sharygina, N. [2016]. OpenSMT2: An SMT solver for multi-core and cloud computing, *SAT 2016*, Vol. 9710, Springer, Heidelberg, pp. 547–553.

Iosif, R. and Xu, X. [2018]. Abstraction refinement for emptiness checking of alternating data automata, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018*, pp. 93–111.

Jain, H., Kroening, D., Sharygina, N. and Clarke, E. [2007]. VCEGAR: Verilog CounterExample Guided Abstraction Refinement, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*.

Jhala, R. and McMillan, K. L. [2007]. Array abstractions from proofs, *International Conference on Computer Aided Verification, CAV*, Vol. 4590, Springer, pp. 193–206.

Kahsai, T., Garoche, P., Tinelli, C. and Whalen, M. [2012]. Incremental verification with mode variable invariants in state machines, *4th International Symposium on NASA Formal Methods NFM*, Vol. 7226 of *LNCS*, Springer, pp. 388–402.

Kahsai, T., Rümmer, P., Sanchez, H. and Schäf, M. [2016]. Jayhorn: A framework for verifying java programs, *in* S. Chaudhuri and A. Farzan (eds), *28th International Conference on Computer Aided Verification, CAV*, Vol. 9779, Springer, Heidelberg, pp. 352–358.

Kapur, D., Majumdar, R. and Zarba, C. G. [2006]. Interpolation for data structures, *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006*, ACM, pp. 105–116.

Katz, G., Barrett, C. and Harel, D. [2015]. Theory-aided model checking of concurrent transition systems, *Proc. FMCAD 2015*, IEEE, pp. 81–88.

Komuravelli, A., Gurfinkel, A. and Chaki, S. [2016]. Smt-based model checking for recursive programs, *Formal Methods Syst. Des.* **48**(3): 175–205.

Komuravelli, A., Gurfinkel, A., Chaki, S. and Clarke, E. M. [2013]. Automatic abstraction in SMT-based unbounded software model checking, *International Conference on Computer Aided Verification, CAV*, Vol. 8044 of *LNCS*, Springer, Berlin, Heidelberg, pp. 846–862.

Kroening, D., Ouaknine, J., Seshia, S. A. and Strichman, O. [2004]. Abstraction-based satisfiability solving of presburger arithmetic, *International Conference*

on Computer Aided Verification, CAV, Vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 308–320.

Kroening, D., Schrammel, P. and Tautschnig, M. [2023]. CBMC: the C bounded model checker, *CoRR* **abs/2302.02384**.

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

Kroening, D. and Tautschnig, M. [2014]. CBMC - C bounded model checker - (competition contribution), *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014,*, Vol. 8413 of *LNCS*, Springer, pp. 389–391.

Kurshan, R. P. [1994]. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*, Princeton University Press, Princeton, NJ, USA.

Kutsuna, T., Ishii, Y. and Yamamoto, A. [2016]. Abstraction and refinement of mathematical functions toward SMT-based test-case generation, *International Journal on Software Tools for Technology Transfer* **18**(1): 109–120.

Lahiri, S. K., Hawblitzel, C., Kawaguchi, M. and Rebêlo, H. [2012]. SYMDIFF: A language-agnostic semantic diff tool for imperative programs, *International Conference on Computer Aided Verification, CAV*, Vol. 7358, Springer, Heidelberg, pp. 712–717.

Lahiri, S. K., McMillan, K. L., Sharma, R. and Hawblitzel, C. [2013]. Differential assertion checking, *ESEC/FSE 2013*, ACM, New York, pp. 345–355.

Lahtinen, J., Kuismin, T. and Heljanko, K. [2015]. Verifying large modular systems using iterative abstraction refinement, *Reliab. Eng. Syst. Saf.* **139**: 120–130.

Lauterburg, S., Sobeih, A., Marinov, D. and Viswanathan, M. [2008]. Incremental state-space exploration for programs with dynamically allocated data, *ICSE 2008*, ACM, New York, pp. 291–300.

Lee, S. and Sakallah, K. A. [2014]. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction, *International Conference on Computer Aided Verification, CAV*, Vol. 8559 of *LNCS*, Springer, pp. 849–865.

Lei, S., Cheng, M. and Jiang, J. [2019]. Tactics for proving separation logic assertion in coq proof assistant, *ICVISP 2019: 3rd International Conference on Vision, Image and Signal Processing, Vancouver, BC, Canada, August 26-28, 2019*, ACM, pp. 99:1–99:5.

Leino, K. R. M. and Wüstholz, V. [2015]. Fine-grained caching of verification results, *International Conference on Computer Aided Verification, CAV*, Vol. 9206 of *LNCS*, pp. 380–397.

Li, G. and Gopalakrishnan, G. [2010]. Scalable smt-based verification of GPU kernel functions, *in* G. Roman and A. van der Hoek (eds), *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, ACM, pp. 187–196. **URL:** *https://doi.org/10.1145/1882291.1882320*

Loos, R. and Weispfenning, V. [1993]. Applying linear quantifier elimination, *Comput. J.* **36**(5): 450–462.

Marescotti, M., Gurfinkel, A., Hyvärinen, A. E. J. and Sharygina, N. [2017]. Designing parallel pdr, *FMCAD 2017*, IEEE, pp. 156–163.

Marescotti, M., Hyvärinen, A. E. J. and Sharygina, N. [2018]. SMTS: Distributed, visualized constraint solving, *LPAR 2018*.

Matsushita, Y., Tsukada, T. and Kobayashi, N. [2021]. Rusthorn: Chc-based verification for rust programs, *ACM Trans. Program. Lang. Syst.* **43**(4): 15:1–15:54.

McMillan, K. L. [1993]. *Symbolic model checking*, Kluwer Academic Publishers, Norwell, MA, USA.

McMillan, K. L. [2003a]. Interpolation and SAT-based model checking, *15th International Conference on Computer Aided Verification, CAV*, Vol. 2725, Springer, pp. 1–13.

McMillan, K. L. [2003b]. Interpolation and SAT-based model checking, *International Conference on Computer Aided Verification, CAV*, Vol. 2725, Springer, pp. 1–13.

McMillan, K. L. [2004]. An interpolating theorem prover, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Vol. 2988, Springer, pp. 16–30.

McMillan, K. L. [2005]. An interpolating theorem prover, *Theor. Comput. Sci.* **345**(1): 101–121.

McMillan, K. L. [2006]. Lazy abstraction with interpolants, *18th International Conference on Computer Aided Verification CAV*, Vol. 4144 of *LNCS*, Springer, pp. 123–136.

McMillan, K. L. [2010]. Lazy annotation for program testing and verification, *International Conference on Computer Aided Verification, CAV*, Vol. 6174 of *LNCS*, Springer, Heidelberg, pp. 104–118.

McMillan, K. L. [2014]. Lazy annotation revisited, *International Conference on Computer Aided Verification, CAV*, Vol. 8559 of *LNCS*, Springer, pp. 243–259.

McMillan, K. L. and Amla, N. [2003]. Automatic abstraction without counterexamples, *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*.

McMillan, K. L. and Rybalchenko, A. [2013]. Solving constrained horn clauses using interpolation, *Technical report*, MSR-TR-2013-6.

Mentel, L., Scheibler, K., Winterer, F., Becker, B. and Teige, T. [2021]. Benchmarking smt solvers on automotive code, *24th workshop methods and description languages for modeling and verification of circuits and systems MBMV*, pp. 1–10.

Mordvinov, D. and Fedyukovich, G. [2019]. Property directed inference of relational invariants, *FMCAD 2019*, IEEE, San Jose, pp. 152–160.

Müller, P., Schwerhoff, M. and Summers, A. J. [2016]. Viper: A verification infrastructure for permission-based reasoning, *VMCAI*, Vol. 9583 of *LNCS*, Springer, pp. 41–62.

Nelson, G. and Oppen, D. C. [1979]. Simplification by Cooperating Decision Procedures, *ACM Transactions on Programming Languages and Systems* **1**(2): 245–57.

Nelson, G. and Oppen, D. C. [1980]. Fast decision procedures based on congruence closure, *Journal of ACM* **27**(2): 356–364.

Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X. [2017]. Hyperkernel: Push-button verification of an OS kernel, *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, pp. 252–269.

Nguyen, M. D., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D. and Kunz, W. [2008]. Unbounded protocol compliance verification using interval property checking with invariants, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **27**(11): 2068–2082.

Nieuwenhuis, R. and Oliveras, A. [2005]. Proof-producing congruence closure, *RTA 2005*, pp. 453–468.

Nigam, V. and Talcott, C. L. [2022]. Automating safety proofs about cyber-physical systems using rewriting modulo SMT, *in* K. Bae (ed.), *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers*, Vol. 13252 of *LNCS*, Springer, pp. 212–229.

O'Hearn, P. W., Reynolds, J. C. and Yang, H. [2001]. Local reasoning about programs that alter data structures, *in* L. Fribourg (ed.), *15th International Workshop Computer Science Logic, CSL - 10th Annual Conference of the EACSL*, Vol. 2142 of *LNCS*, Springer, pp. 1–19.

Padon, O., McMillan, K. L., Panda, A., Sagiv, M. and Shoham, S. [2016]. Ivy: safety verification by interactive generalization, *the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, ACM, pp. 614–630.

Pick, L., Fedyukovich, G. and Gupta, A. [2018]. Exploiting synchrony and symmetry in relational verification, *International Conference on Computer Aided Verification, CAV*, Vol. 10981, Springer, Cham, pp. 164–182.

Pólrola, A., Cybula, P. and Meski, A. [2014]. Smt-based reachability checking for bounded time petri nets, *Fundam. Informaticae* **135**(4): 467–482.

Pudlák, P. [1997]. Lower bounds for resolution and cutting plane proofs and monotone computations, *J. Symb. Log.* **62**(3): 981–998.

Qadeer, S., Rajamani, S. K. and Rehof, J. [2004]. Summarizing procedures in concurrent programs, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, pp. 245–255.

Queille, J. and Sifakis, J. [1982]. Specification and verification of concurrent systems in CESAR, *5th International Symposium on Programming*, pp. 337–351.

Raghunathan, D., Beckett, R., Gupta, A. and Walker, D. [2022]. ACORN: network control plane abstraction using route nondeterminism, *CoRR* **abs/2206.02100**.

Ramalho, M., Freitas, M., Sousa, F. R. M., Marques, H., Cordeiro, L. C. and Fischer, B. [2013]. Smt-based bounded model checking of C++ programs, *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, ECBS*, IEEE Computer Society, pp. 147–156.

Reps, T. W., Horwitz, S. and Sagiv, S. [1995]. Precise interprocedural dataflow analysis via graph reachability, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pp. 49–61.

Rollini, S. F., Alt, L., Fedyukovich, G., Hyvärinen, A. and Sharygina, N. [2013]. PeRIPLO: A framework for producing effective interpolants in SAT-based software verification, *LPAR 2013*, Vol. 8312, Springer, Heidelberg, pp. 683–693.

Rothenberg, B., Dietsch, D. and Heizmann, M. [2018]. Incremental verification using trace abstraction, *Static Analysis - 25th International Symposium, SAS 2018*, pp. 364–382.

Rümmer, P., Hojjat, H. and Kuncak, V. [2013]. Disjunctive interpolants for Horn-clause verification, *International Conference on Computer Aided Verification, CAV*, Vol. 8044, Springer, pp. 347–363.

Rybalchenko, A. and Sofronie-Stokkermans, V. [2007]. Constraint solving for interpolation, *8th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*, Vol. 4349, Springer, pp. 346–362.

Schrijver, A. [1999]. *Theory of linear and integer programming*, Wiley-Interscience series in discrete mathematics and optimization, Wiley, Sons.

Sebastiani, R. [2007a]. From KSAT to delayed theory combination: Exploiting DPLL outside the SAT domain, *Frontiers of Combining Systems, 6th International Symposium, FroCoS*, Vol. 4720 of *LNCS*, Springer, pp. 28–46.

Sebastiani, R. [2007b]. Lazy satisfiability modulo theories, *Journal on Satisfiability, Boolean Modeling and Computation* **3**(3-4): 141–224.

Seghir, M. N., Podelski, A. and Wies, T. [2009]. Abstraction refinement for quantified array assertions, *International Symposium on Static Analysis, SAS*, Vol. 5673, Springer, pp. 3–18.

Sery, O., Fedyukovich, G. and Sharygina, N. [2011]. Interpolation-based function summaries in bounded model checking, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC*, Vol. 7261 of *LNCS*, Springer, pp. 160–175.

Sery, O., Fedyukovich, G. and Sharygina, N. [2012a]. FunFrog: Bounded model checking with interpolation-based function summarization, *ATVA 2012*, Vol. 7561, Springer, Heidelberg, pp. 203–207.

Sery, O., Fedyukovich, G. and Sharygina, N. [2012b]. Incremental upgrade checking by means of interpolation-based function summaries, *FMCAD 2012*, IEEE, New York, pp. 114 – 121.

Sheeran, M., Singh, S. and Stålmarck, G. [2000]. Checking safety properties using induction and a sat-solver, *Third International Conference on Formal Methods in Computer-Aided Design*.

Shemer, R., Gurfinkel, A., Shoham, S. and Vizel, Y. [2019]. Property directed self composition, *International Conference on Computer Aided Verification, CAV*, Vol. 11561, Springer, Heidelberg, pp. 161–179.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A. and Saraswat, V. A. [2006]. Combinatorial sketching for finite programs, *the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, ACM, New York, pp. 404–415.

Szymoniak, S., Siedlecka-Lamch, O., Zbrzezny, A. M., Zbrzezny, A. and Kurkowski, M. [2021]. SAT and smt-based verification of security protocols including time aspects, *Sensors* **21**(9): 3055.

Trostanetski, A., Grumberg, O. and Kroening, D. [2017]. Modular demand-driven analysis of semantic difference for program versions, *International Symposium on Static Analysis, SAS*, Vol. 10422, Springer, Cham, pp. 405–427.

Vick, C. and McMillan, K. L. [2023]. Synthesizing history and prophecy variables for symbolic model checking, *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI*, Vol. 13881 of *LNCS*, Springer, pp. 320–340.

Visser, W., Geldenhuys, J. and Dwyer, M. B. [2012]. Green: reducing, reusing and recycling constraints in program analysis, *SIGSOFT/FSE 2012*, ACM, New York, p. 58.

Vizel, Y. and Grumberg, O. [2009]. Interpolation-sequence based model checking, *FMCAD'09*, IEEE, pp. 1–8.

Vizel, Y. and Gurfinkel, A. [2014]. Interpolating property directed reachability, *26th International Conference Computer Aided Verification, CAV*, Vol. 8559 of *LNCS*, Springer, Heidelberg, pp. 260–276.

Vizel, Y., Gurfinkel, A. and Malik, S. [2015]. Fast interpolating BMC, *International Conference on Computer Aided Verification, CAV*, pp. 641–657.

W., F. R. [1967]. Assigning meanings to programs, *Symposium on Applied Mathematics* **19**: 19–32.

Xie, Y. and Aiken, A. [2005a]. Saturn: A SAT-Based Tool for Bug Detection, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pp. 139–143.

Xie, Y. and Aiken, A. [2005b]. Scalable error detection using boolean satisfiability, *the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, ACM, pp. 351–363.

Xu, L. [2008]. Smt-based bounded model checking for real-time systems (short paper), *in* H. Zhu (ed.), *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK*, IEEE Computer Society, pp. 120–125.

Yang, G., Khurshid, S., Person, S. and Rungta, N. [2014]. Property differencing for incremental checking, *ICSE 2014*, ACM, New York, pp. 1059–1070.

Zhang, H., Yang, W., Fedyukovich, G., Gupta, A. and Malik, S. [2020]. Synthesizing environment invariants for modular hardware verification, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI*, Vol. 11990 of *Lecture Notes in Computer Science*, Springer, pp. 202–225.

Zhu, H., Magill, S. and Jagannathan, S. [2018]. A data-driven CHC solver, *the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, ACM, New York, pp. 707–721.